Tailoring Configuration to User's Tasks under Uncertainty

Vahe V Poladyan

CMU-CS-08-121

April 28, 2008

Computer Science Department

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Thesis Committee:
David Garlan and Mary Shaw (Co-Chairs)
Mahadev Satyanarayanan
Ashish Arora (Heinz School of Public Policy, Carnegie Mellon)

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

This research was funded in part by the National Science Foundation Grants ITR-0086003, CCR-0205266, CCF-0438929, CNS-0613823, by the Sloan Software Industry Center at Carnegie Mellon, by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298 and by DARPA grant N66001-99-2-8918. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the US government or any other entity.

Abstract

The expansion of computing infrastructure has opened the possibility of a world in which users can compute everywhere. Despite such advances, computing resources are often scarce and changing, limiting a user's ability to take advantage of the applications and devices, and requiring changes to the application runtime settings.

Currently, the burden of managing the computing environment (devices, applications, and resources) falls on the user. A user must manually start applications and adjust their settings according to the available resources. Assigning such chores of configuration to the user has a number of disadvantages. First, it consumes user's precious cognitive resources. Second, effectively managing the environment requires skills that a typical user might not have. Third, even with adequate low-level expertise, managing the environment optimally (or even adequately) can be difficult.

Ideally, the computing needs of a user are seamlessly matched with the capabilities of the environment: devices, applications, and available resources. The user should enjoy the best possible application quality, without worrying about managing the low-level computing mechanisms.

In this dissertation, we describe a novel approach that substantially automates the control of the configuration of the environment for a user's task: finding and starting applications, configuring their runtime settings, and allocating possibly limited resources. Our approach simultaneously satisfies two important requirements: utility and practicality. Utility ensures that configuration decisions take into account user's preferences for specific applications and quality of service. Practicality ensures that configuration has low runtime overhead in terms of the latency of configuration decisions and its usage of resources.

First, we model configuration analytically as a problem of optimizing user's utility based on three inputs: (1) user's preferences, (2) application capability, and (3) resource availability. Formally, automating the control of the configuration requires solving an optimization problem, and then using the optimization solution to control the environment.

Next, we design a software infrastructure that is based on the analytical model. The infrastructure implements efficient algorithms to solve the problem of configuration, eliminating the need for manual configuration. We validate our approach using experiments and simulation, demonstrating that the infrastructure satisfies the requirements of utility and practicality while substantially automating configuration.

Acknowledgements

Only a few days removed from completing this dissertation, I wish to acknowledge those whose contributions have been instrumental in shaping my path at Carnegie Mellon as well as those who helped add some color to my journey.

Mary Shaw and David Garlan, for showing me how to ask focused questions, how to carve a problem of the right size, how to reason more clearly, and how to formulate my thoughts more precisely. I am much indebted to both Mary and David for giving me the freedom to choose my research and for helping me mature as a researcher.

Satya, for challenging me to add the theme of uncertainty to this thesis and for helping me to see the important parts of the solution.

Ashish Arora, for guiding me on the subjects of utility and economics, and for thorough discussions on analytical aspects of my work.

The faculty and staff at the School of Computer Science for cultivating a supportive and collaborative environment to help graduate students succeed.

Joao, for his trailblazing efforts in the area of task-oriented computing and for contributing to the results in this thesis.

Bradley, for his selfless contributions to the design and development of the research artifacts in this thesis, and for diligently organizing ABLE group meetings.

DMC, for being a great roommate throughout most of my graduate studies.

Stefan, for being my fellow salsaholic.

My colleagues Jernej, Monica, Lucian, Vince, Mugizi, George, Owen, for being great friends above all.

Heather, for her friendship and love.

My parents, for encouraging me to enter graduate school, and for their support and love.

Astghik, my little sister, my little sister, for her love, support, and friendship.

Table of Contents

1 INTRODUCTION 11

1.1 PROBLEM AND REQUIREMENTS 12
1.2 THESIS 13
1.3 TECHNICAL CHALLENGES 14
1.4 CLAIMS AND VALIDATION STRATEGY 15
1.5 DISSERTATION ROADMAP 17

2 RELATED WORK 19

2.1 USING UTILITY AND PREFERENCES TO GUIDE ENGINEERING DECISIONS 19
2.2 TASK ORIENTED COMPUTING 20
2.3 MULTI-FIDELITY ADAPTATION: MECHANISMS FOR CONFIGURATION 20
2.4 RESOURCE SUPPLY AND DEMAND PREDICTION 21
2.5 RESOURCE ALLOCATION MODELS AND SYSTEMS 22
2.6 HOME AUTOMATION SYSTEMS 23
2.7 AUTONOMIC COMPUTING AND SELF-* SYSTEMS 24
2.8 ANTICIPATORY APPROACHES 25
2.9 DYNAMIC PROGRAMMING AND MARKOV DECISIONS PROCESSES 25

3 BACKGROUND AND SETTING 27

3.1 RELEVANT TERMS 27
3.2 PROBLEM SETTING 30
3.2.1 Target Users 30
3.2.2 Computing Tasks of the Targeted Users 30
3.2.3 The Computing Environment 31
3.2.4 The Properties of the Resources 32
3.3 CHAPTER SUMMARY 34

4 MODELING THE CONFIGURATION PROBLEM 35

4.1 PROBLEM STRUCTURE: OVERVIEW 35
4.2 BUILDING BLOCKS: SPACES OF THE PROBLEM 36
4.2.1 Utility 36
4.2.2 Capability 37
4.2.3 Resource 38
4.3 PROBLEM INPUTS 38
4.3.1 User Preferences 38
4.3.2 Supplier Profiles 40
4.3.3 Resource Availability 41
4.4 DECISION VARIABLES 41
4.4.1 Task Layout: Suite of Suppliers and Expected Utility 41
4.4.2 Runtime Settings: Selected Quality-Resource Pair from Supplier Profiles 42
4.4.3 Resource Allocation 42
4.5 OPTIMIZATION PROBLEM 42
4.6 DYNAMIC BEHAVIOR AND CONFIGURATION STRATEGIES 42
4.7 CHAPTER SUMMARY 43

5 REACTIVE MODEL OF CONFIGURATION 45

5.1 SPACES 45
5.1.1 Utility 45

5.1.2 Capability ... *45*
5.1.3 Resource ... *46*
5.2 INPUTS ... 46
5.2.1 User Preferences ... *46*
5.2.2 Available Suppliers and their Profiles ... *48*
5.2.3 Resource Availability ... *49*
5.3 DECISION VARIABLES ... 49
5.3.1 Task Layout: Suite of Suppliers and Expected Utility ... *49*
5.3.2 Runtime Settings: Selected Quality-Resource Pair from Supplier Profiles ... *49*
5.3.3 Resource Allocation ... *49*
5.4 OPTIMIZATION PROBLEM ... 50
5.5 DYNAMIC BEHAVIOR AND RECONFIGURATION ... 51
5.6 LIMITATIONS OF THE REACTIVE CONFIGURATION STRATEGY AND MODEL ... 51
5.7 CHAPTER SUMMARY ... 52

6 PREDICTION AND UNCERTAINTY ... 55

6.1 SOURCES OF PREDICTIVE INFORMATION ... 56
6.1.1 Sources of Information for Predicting Users Tasks ... *56*
6.1.2 Sources of Information for Predicting Resource Demand by Applications ... *57*
6.1.3 Sources of Information for Predicting Resource Availability ... *57*
6.2 UNCERTAINTY IN PREDICTION ... 58
6.2.1 Two Types of Uncertainty ... *58*
6.2.2 Uncertainty When Predicting a Sequence of Future Values of a Variable ... *58*
6.2.3 Predicting Likely Paths of Future Values of Variable ... *59*
6.3 OPERATIONAL CONSIDERATIONS FOR A PREDICTION INTERFACE ... 62
6.4 DISCRETE TIME MODEL ... 63
6.5 PREDICTING USER'S TASKS ... 63
6.5.1 A Motivating Example ... *63*
6.5.2 Building Blocks of the Task Prediction Model ... *64*
6.5.3 Task Request Prediction Model ... *65*
6.5.4 Information Arrival ... *66*
6.6 PREDICTING APPLICATION AND HARDWARE RESOURCE DEMAND ... 66
6.7 PREDICTING RESOURCE AVAILABILITY ... 67
6.7.1 Time Series Background ... *67*
6.7.2 Resource Predictor Types ... *68*
6.7.3 Combining Predictors: Operations and Algorithm ... *73*
6.7.4 Identifying Appropriate Predictor Types for Each Resource ... *75*
6.8 CHAPTER SUMMARY ... 77

7 ANTICIPATORY MODEL OF CONFIGURATION ... 79

7.1 SPACES ... 79
7.1.1 Utility Space ... *79*
7.1.2 Capability Space ... *80*
7.1.3 Resource Space ... *80*
7.2 INPUTS ... 81
7.2.1 User Preferences and Predictions of Tasks ... *81*
7.2.2 Supplier Profiles: Quality-Resource Mappings ... *81*
7.2.3 Resource Availability ... *82*
7.3 DECISION VARIABLES ... 82
7.4 OPTIMIZATION PROBLEM ... 83

7.5 DYNAMIC BEHAVIOR AND RECONFIGURATION 84
7.6 CHAPTER SUMMARY 84

8 AN ARCHITECTURE FOR AUTOMATIC CONFIGURATION 87

8.1 THE AURA SOFTWARE ARCHITECTURE 88
8.2 THE DESIGN OF THE ENVIRONMENT MANAGER 91
8.3 THE DESIGN OF THE PREDICTION FRAMEWORK (RESOURCE MANAGER) 94
8.4 AGGREGATE PREDICTION PROTOCOL: AN API FOR RESOURCE PREDICTIONS 98
8.5 DISCUSSION OF KEY DESIGN DECISIONS AND THEIR ALTERNATIVES 102
8.6 MAPPING THE ARCHITECTURE TO A CONTROL-THEORETIC FRAMEWORK 104
8.7 CHAPTER SUMMARY 105

9 ALGORITHMS FOR AUTOMATIC CONFIGURATION 107

9.1 STAGING THE PROBLEM 107
9.2 STAGE 1 109
9.2.1 Sub-problem: calculating maximum QoS utility for a supplier assignment 111
9.2.2 Calculating maximum overall utility by Iterating over Supplier Assignments 115
9.3 STAGE 2: ANTICIPATORY STRATEGY WITH PERFECT PREDICTIONS 117
9.4 STAGE 3.1: RESOURCE PREDICTIONS WITH UNCERTAINTY 121
9.5 STAGE 3.2: TASK PREDICTIONS WITH UNCERTAINTY 125
9.6 STAGE 3.3: NON-PERISHABLE, NON-RENEWABLE RESOURCES 127
9.6.1 Background Research on Battery Drain 128
9.6.2 Modeling Battery Drain for Configuration Purposes 129
9.6.3 Expressing the Intertemporal Substitution Possibilities of Battery Energy 131
9.7 STAGE 4: COMBINING PREDICTIONS WITH UNCERTAINTY 133
9.7.1 Combining Two Resource Predictions 133
9.7.2 Combining two Task Predictions 134
9.7.3 Combining Resource Predictions with Task Predictions 136
9.8 STAGE 5: PREDICTIONS OF RESOURCES, TASKS, AND BATTERY 136
9.9 CHAPTER SUMMARY 136

10 THESIS VALIDATION 139

10.1 AUTOMATING THE CONTROL OF THE CONFIGURATION 139
10.1.1 Phase 1: Acquire Knowledge Needed to Perform Configuration 141
10.1.2 Phase 2: Define a Task in Terms of Services and Preferences 142
10.1.3 Phases 3 and 5: Find and Start Applications, Stop Applications 142
10.1.4 Phase 4: Configure / Reconfiguration Application Runtime Settings 143
10.1.5 Summary of the Results 143
10.2 SATISFYING UTILITY AND PRACTICALITY (REACTIVE MODEL AND STRATEGY) 144
10.2.1 Experimental Setup 145
10.2.2 Addressing Practicality 146
10.2.3 Addressing Utility 148
10.3 CLAIMS ABOUT UNCERTAINTY 149
10.3.1 Predictions with Uncertainty can be Encoded and Transmitted Efficiently 149
10.3.2 Utility and practicality In the face of Uncertainty 151
10.4 SUMMARY: DEMONSTRATING THESIS 158

11 DISCUSSION, FUTURE WORK, AND CONCLUSION 159

11.1 DESIGN DECISIONS MADE AND ALTERNATIVES CONSIDERED 159
11.1.1 Model of Preferences 159
11.1.2 Do Preferences Make a Difference in Practice? 161

11.1.3 Comparing Anticipatory and Reactive Configuration Strategies 162
11.1.4 Accuracy of Supplier Profiles 164
11.1.5 Cost of Disrupting the User 166
11.1.6 Applying the Prediction Framework Outside of the Current Context 167
11.2 FUTURE WORK 168
11.2.1 Ensuring the Privacy of User's Tasks 168
11.2.2 Defining Environment Scope and Environment Boundaries 169
11.2.3 Configuration There and Then: Combining Motion Profiles 171
11.3 CONTRIBUTIONS 172
11.3.1 Contributions of the Approach 172
11.3.2 Analytical Contributions 172
11.3.3 Architectural and Implementation Artifacts 173
11.4 CONCLUSION 174

1 Introduction

In the past decade or so, computing infrastructure has grown rapidly: the number of computing devices around us has increased and the geographical coverage of connectivity has expanded to include venues outside of the office or home. This advancement in computing technology has enabled lay users to take their tasks everywhere, from airports to coffee shops and outdoor areas. Yet computing resources, both ambient and on-board, still lag the resource demand of widely used applications.

The proliferation of the computing infrastructure has opened the possibility of a world in which users seamlessly take advantage of the available computing devices around them. Ideally, the computing needs of a user are seamlessly matched with applications and resources. The user enjoys the best possible experience in terms of application quality, without worrying about managing the necessary low-level computing mechanisms to make all this happen.

For example, consider Jane, a movie critic who is writing a review for a newly released film. For this task, she needs a video player, a text editor, a browser, network connectivity, one or more devices for computation, and a display. Jane first starts the task on her office computer, using wired infrastructure, works for two hours, and then suspends the task in order to catch a flight. She resumes the task in the airport, using her laptop and limited wireless bandwidth to watch selected episodes and take notes just before her departure. Later, while in flight, she puts the finishing touches on the report, completing the task. As Jane might have other tasks in progress, she chooses to switch between them as needed.

Currently, however, the burden of managing devices and applications falls mostly on the user. In the example, Jane has to manually perform steps such as finding and starting necessary applications, and choosing the quality of the media stream. She might also have to adjust the power settings on the laptop to allow the battery to last longer and later change the settings of the media player and the browser to accommodate dropping bandwidth. Today, a typical computer user must perform many such tedious steps in the course of the day as she resumes a task, modifies it, or switches between tasks.

There are significant problems with assigning such management activities to the user. First, it consumes precious and limited cognitive resources from the user. Second, managing the computing environment requires a skill set that a typical user might not possess or might not want to learn. As a result, many users may struggle and fail to do a good job of managing the environment or simply give up and live with unsatisfying configurations. Third, even with adequate low-level expertise, managing the devices and applications optimally, or even well, can be difficult. The mobility of the user, the number of concurrent applications in the task, the total number of available devices and applications, and resource fluctuations complicate the problem further.

The third set of issues makes the burden of managing the computer environment particularly complex and leaves the user prone to making potentially bad choices. For example, a user equipped with a PDA might choose a local speech recognition program fearing that bandwidth is insufficient to use a remote recognition engine. This decision might be based on limited knowledge of the available resources, the resource requirements of the different applications, or the ca-

pabilities of the PDA itself. Equipped with better information, the user might have chosen to use the remote engine, which delivers better quality of translation without sacrificing latency.

Unfortunately, no existing approach has fully addressed the problem of user-level management of devices and applications. Research in two closely related areas has solved important parts of the problem. However, a complete solution to the configuration problem is still lacking.

Research in multi-fidelity and reduced-fidelity systems has provided an important building block towards solving the configuration problem. Such systems implement adaptive mechanisms for reduced fidelity operation of applications and devices. For example, many video players have mechanisms to show the same video clip at different quality levels by using multiple streams encoded at different bit-rates. Lower bit-rate streams have either lower frame rate, lower resolution, or both. This allows the user to watch video when the bandwidth is low, or when it drops. While multi-fidelity computation enables trade-offs between output quality and resource consumption, it still lacks in two important respects. First, when the user cares about multiple dimensions of task quality, e.g. frame rate and resolution, a multi-fidelity application can not determine which dimension should have priority. I.e., when bandwidth is not sufficient to run the best possible video stream, the application can not decide to lower the frame rate or the resolution without additional guidance from the user. And second, managing a suite of simultaneously executing multi-fidelity applications requires both low level knowledge and considerable cognitive effort.

Second, task-oriented computing addresses the problem of eliciting task needs. In his thesis [52], Sousa defines a task description language for describing the requirements of everyday user tasks, implements a software infrastructure for *eliciting* those from the user via intuitive graphical user interfaces. This approach contributes towards automating task management by allowing the user to express task requirements in platform and application-independent terms. But this approach falls short of demonstrating how the requirements of the user can be optimally mapped to specific applications in the environment in the face of current resource constraints.

An additional software layer is necessary to perform an intermediate step, which I call configuration. Configuration maps the description of a user's task to a choice of applications and desired control settings for each application. Once the desired quality settings are determined, they can be used to guide the runtime behavior of the multi-fidelity applications to meet user's goals for quality of service under various resource conditions.

Ideally, the burden of configuration should be borne not by the user, but by software specifically designed for that purpose. This thesis describes an approach and software infrastructure that implements such capability. The software infrastructure we have implemented coordinates the various pieces of the computing environment and automates many of the chores that the users perform manually today.

1.1 Problem and Requirements

The problem solved in this thesis is that of *automating the configuration* of the computing environment: finding an appropriate suite of applications and devices to satisfy the user's tasks, allocating resources among applications, and dynamically tuning the runtime settings of applications. Our solution monitors the environment, including the resources, and adjusts the configuration over time in order to continually provide the user with optimal or near-optimal quality of service for the tasks at hand.

But what are some of the desired requirements for a system that automates configuration? To be usable in practice, a good solution must make configuration decisions that are simultaneously: (a) tailored to the individual preferences of a user and her tasks, and (b) maximize or nearly maximize the expected utility of the user over some temporal horizon. The preferences should reflect user's familiarity and affinity for specific applications as well as describe the relative importance of the different dimensions of output quality to the user. We call this requirement **optimizing the utility**.

Second, automatic configuration should add little perceptible delay to the user compared to what is being automated, such as the startup time of applications. The latency of automatic configuration, measured from the time a user issues a request until the system satisfies the request, should not exceed 1 second. Furthermore, the latency should grow sub-linearly as a function of the number of applications in the user's tasks. As a comparison, starting up applications and setting their state manually take at least 3-5 seconds per application. Furthermore, the resource footprint of the software infrastructure should not to exceed 2% of the available resources such as CPU, memory, battery and bandwidth. We call this requirement **practicality**.

We target computing environments that are dynamically changing. The availability of resources changes over time. There is an element of uncertainty, both in the level of available resources in the future as well as in the length of time that a user might need to complete a task. Uncertainty makes the planning and decision making difficult. The third requirement for a system that automates configuration is to quantify the uncertainty in the inputs and factors that affect configuration decisions, and to ensure that the requirements of utility and practicality continue to be satisfied in the face of uncertainty. We call this requirement optimally **coping with uncertainty**.

1.2 Thesis

In this dissertation we demonstrate that:

> ***The configuration of the computing environment can be substantially automated by tailoring it to the needs and preferences of an individual user for each task, while simultaneously satisfying the requirements of optimizing utility and practicality while coping with uncertainty.***

To demonstrate this thesis, we have an approach with both analytical and runtime components. First, we define an *analytical model* that formalizes the notion of utility for user's tasks and expresses automatic configuration as a mathematical problem of maximizing the *expected utility of the user* from the running state of the environment under the constraints of the computing environment.

The analytical model provides a carefully crafted structure for the problem, allowing efficient runtime configuration algorithms to search the problem space for good solutions. We use that structure to define two concrete configuration strategies: reactive and anticipatory. These strategies differ in two important respects: (1) the amount of input information available and (2) the temporal horizon of the decisions. These strategies also differ in how they treat uncertainty in the available information. The reactive approach ignores any uncertainty associated with future events; the anticipatory approach explicitly quantifies the uncertainty of future events, and copes with uncertainty by planning for future changes.

Using the analytical model as a formal basis, we have designed and implemented a software infrastructure for automatic configuration. Our contributions to the infrastructure are threefold: (1) a central component that makes near optimal configuration decisions, (2) a prediction framework that provides resource prediction on demand, and (3) a programming interface between the centralized decision maker and the prediction framework. The central decision making component leverages the structure of the analytical models to implement efficient and near-optimal configuration algorithms. The resource prediction framework quantifies the future level of resource availability by combining predictive information from multiple sources.

This infrastructure complements existing software such as Prism, the task management infrastructure implemented by Sousa in his thesis, and leverages fidelity-aware, adaptive applications using a standardized supplier interface.

Automating configuration reduces the overhead that users spend on various configuration chores. By tailoring the configuration to the specific needs and preferences for each task, the infrastructure improves the overall quality of the user's task. And lastly, by leveraging predictions about future changes (e.g., changes in resource availability and changes in user's intent) over time, our solution makes configuration decisions and adjustments in anticipation of changes and eliminates the need for potentially disruptive configuration actions while the task is in progress.

1.3 Technical Challenges

The primary technical challenge of the thesis is simultaneously satisfying the requirements of optimizing utility and practicality under uncertainty. As noted, practicality requires the solution to be fast and efficient while utility requires the solution to be close to optimal. Unfortunately, these are in conflict because selecting an optimal solution is computationally complex. Even a simple version of the configuration problem is NP-complete [34,46]. Moreover, even approximate solutions must deal with three sources of complexity: a large space of configuration alternatives, the interdependence of decisions over time, and uncertainty.

To illustrate, consider a simple version of the configuration problem in which we are only interested in a short period of time. During this time, the state of the world does not change. In this setting, the problem of configuration reduces to finding appropriate applications and allocating available resources to those applications. With even a handful of applications available for each type of application in the task (e.g., three different video players, three different browsers, and four different text editors) and a few dozen different quality settings for each application, the configuration space of a task can have hundreds of thousands of alternatives [48].

Next, consider making configuration decisions over multiple periods of time. In this setting, configuration decisions need to be made not only for what the user wants now, but also for what he might want in the future, e.g., for duration of the task. This consideration affects the allocation of resources such as battery in configuration decisions, because the limited supply of battery may be exhausted before the user decides to suspend the task. Furthermore, configuration decisions need to consider possible future changes in availability of resources such as bandwidth. Such changes might require costly reconfigurations in the future and reduce the utility to the user. Multiple decisions made over time have a dependency, and ensuring optimality becomes much more complex in the face of the dependencies.

Uncertainty exacerbates the complexity of the problem. On the one hand, ignoring uncertainty can simplify the problem of configuration. On the other hand, uncertainty can not be ignored without foregoing the benefit of predictions about future changes in resources and user's needs. One of the technical challenges of this thesis is to quantify uncertainty and investigate how the magnitude of uncertainty affects configuration decisions. In particular, we demonstrate that when uncertainty is small, predictions provide a non-trivial utility benefit to the user.

1.4 Claims and Validation Strategy

This thesis makes the following claims: (1) the control of the configuration can be substantially automated, (2) configuration can be tailored to individual user preferences so that the requirements of optimizing utility and practicality are simultaneously satisfied, and (3) our approach works under uncertainty.

The first claim is that the configuration of the computing environment can be substantially automated. To support this claim, we have implemented software infrastructure that eliminates certain class of overhead activities that users currently must perform manually. These activities include: (1) starting and stopping applications, (2) manipulating application runtime settings. We have identified two dimensions of user overhead: what the user needs to know to perform configuration and what input actions the user needs to execute to make configuration happen. Knowledge refers to the awareness and understanding of correctly manipulating application settings via a user interface. Input actions refer to the interactive commands such as mouse clicks and keyboard strokes that a user must perform to accomplish configuration activities. We demonstrate the reduction in the knowledge about configuration by counting the number of interfaces (menu items, control widgets, pup-up windows) that the user must access across all available applications in order to adjust settings for the applications in a representative task. We demonstrate the reduction of input actions by comparing the number of input events that a user must execute with and without automatic configuration software.

The second claim of the thesis is that configuration can be tailored to an individual user's preferences for each task, while simultaneously satisfying the requirements of utility and practicality. In support of this claim, we have constructed an analytical model with a specific class of preference functions. These functions allow expressing preferences with respect to multiple objectives as long as these objectives satisfy certain independence assumptions. We argue that the model expresses the problem in a useful way while keeping the configuration problem tractable in (e.g., by creating a special structure in the analytical problem so that it can be tackled by algorithms efficiently). We demonstrate that the software for automatic configuration can maximize the expected utility of the user for each task according to tailored preferences. To support the claim about practicality, we demonstrate that the configuration software has low latency and low resource footprint. We measure the latency of configuration actions and the resource usage of the software experimentally. As a benchmark for success, the latency should be comparable or lower than manual startup of applications, while resource usage should not exceed a small percentage of the total available resources.

And lastly, we make two claims about uncertainty. In our work, we address two types of uncertainty. The first type of uncertainty arises as a result of noisy measurement of inputs such as resource availability. The second type of uncertainty arises when predicting future outcomes, e.g., changes in resource availability and user's intent.

Our first claim about uncertainty is that predictions with uncertainty can be encoded and transmitted efficiently at runtime by designing an application programming interface (API) specifically for that purpose. Our second claim is that the runtime infrastructure simultaneously satisfies the requirements of utility and practicality in the face of uncertainty in the predictions.[1]

Table 1 provides a summary of the thesis claims.

Table 1: A concise summary of the thesis claims.

Claim	Claim Element	Demonstration of the Claim	Criteria
1	The control of the configuration can be substantially automated	User's overhead (number of input actions and knowledge about configuration activities) is reduced	By counting input events (mouse clicks/keyboard strokes). By qualitative analysis of the knowledge required of the user to execute configuration actions
2.1	Approach satisfies the requirement of utility	Algorithms make configuration decisions that nearly maximize the expected utility using tailored preferences for each task	By comparing the utility of configuration decisions made by software to utility of randomly selected configurations
2.2	Approach satisfies the requirement of practicality	For practicality, software has low overhead (latency and resource footprint)	Latency of each configuration decision is under 1,000 ms. Software uses 1-2% of CPU and 5-10% of RAM.
3.1	API for predictions is efficient an flexible	API allows efficient encoding and transmission of predictive uncertainty; API provides an explicit cost-accuracy tradeoff	By analyzing the resource cost of producing and transmitting predictions
3.2	Utility and practicality are satisfied in the face of uncertainty	Same as in 2.1, but with the addition of predictive uncertainty	Same as in 2.1, but with the addition of predictive uncertainty
3.3		Same as in 2.2, with the addition of predictive uncertainty	Same as in 2.2, but with the addition of predictive uncertainty

[1] While our approach specifically addresses two types of uncertainty, it does not address other sources of uncertainty. Also, our approach does not separately treat the two types of uncertainty. Instead, we describe and handle the cumulative uncertainty, which may be attributable to one or both types.

1.5 Dissertation Roadmap

The dissertation document is organized as follows.

Chapter 2 reviews major categories of related work, helping place our work at the intersection of pervasive computing, fidelity-aware adaptation, task-oriented computing, resource prediction, and adaptive systems.

Chapter 3 introduces the key terms used throughout the thesis, and provides the background setting for our work.

Chapters 4-6 describe the the analytical models of configuration.

Chapter 4 provides an overview of the analytical models. In this chapter, we build a conceptual structure of the problem of configuration, separating the different concerns of the problem into three spaces: utility, capability, and resource. We then describe: (a) user preferences, which establish relationships between the capability space and utility space, (b) application profiles, which establish a relationship between the resource space and the capability space, and (c) resource availability, which provide constraints. We define configuration as an optimization problem involving user preferences, application profiles, and resource availability. We also describe and contrast two configuration strategies: the reactive and anticipatory, which differ by the amount of information they use to make decisions.

In Chapter 5, we build on the concepts introduced in Chapter 4, and provide the formal expression of the reactive model of configuration.

In Chapter 6, we first motivate the need for prediction, discuss the role of uncertainty, and bring up key operational and design considerations for integrating prediction into the analytical model. Next, it we describe a prediction model for user tasks. Lastly, we describe a prediction model for resource availability.

In Chapter 7, we use the predictive models from Chapter 6 to provide the formal expression of the analytical model of configuration.

In Chapter 8, we describe the architecture of the infrastructure for automatic configuration. We place our work in the context of the Aura Architecture, highlighting the contributions of thesis. The contributions are threefold: (1) the design of the Environment Manager, (2) the design of the resource prediction framework, and (3) the design of the resource prediction API.

In Chapter 9, we describe the design of the algorithms for automatic configurations. For tractability, we stage the configuration problems into incrementally difficult instances. We first tackle the problem of reactive configuration. Next, we tackle the simplest version of the anticipatory configuration problem. In this version, predictions of tasks and resources are perfect and contain now uncertainty. We then gradually relax the assumption about perfect predictions, solving the more general problem instances with imperfect predictions.

Chapter 10 demonstrates how our approach supports the thesis of this dissertation. We present the results of experiments, and demonstrate how the results support the separate claims of the thesis.

In Chapter 11, we summarize the results and the contributions of the thesis. We discuss important analytical and engineering problems we tackled, issues that were left unsolved and how they may be tackled by future work.

2 Related Work

2.1 Using Utility and Preferences to Guide Engineering Decisions

In this thesis, we make use of linearly-additive preference functions to determine how to best allocation resources. This approach is widely used in microeconomics and decision theory to help users make decisions or explain the decision that the users make. Preference functions are an important tool when deciding among alternatives with multiple dimensions or attributes.

A number of results in microeconomics have established sufficient conditions under which the use of linear-additive preference functions is justified. These conditions are: transitivity, preferential independence, tradeoff independence, and difference independence. Proving that the three independence conditions are satisfied for the various dimensions of application QoS in the context of pervasive computing might is difficult. However, we argue that it is plausible that the user views the different dimensions of quality as independent, e.g., frame update rate and the audio quality of a video stream. Well-known results in microeconomics state that even if independence among the dimensions does not hold, a linearly additive model provides sufficiently close approximations to the actual preferences [62].

In the past decade, multi-dimensional utility functions have been successfully applied to software engineering problems [16]. These problems cover a wide range of domains, anywhere from software project management to mobile computing.

The Security Attribute Evaluation Method (SAEM) helps IT security managers optimally invest a limited budget into a set of counter-measures and reduce the effect of security attacks [4]. SAEM uses a linear utility function elicited from security managers to assess the relative importance of different costs of successful attacks such as loss of revenue, impact on company reputation, productivity loss. SAEM then recommends an optimal bundle of security counter-measures that is with in a given budget [4].

The Cost-Benefit Analysis Measure (CBAM) [36] is an enhancement of the Architecture Tradeoff Analysis Method (ATAM). As part of CBAM, the preferences of multiple stakeholders are elicited and reconciled, helping make design and architecture decisions in software projects. CBAM uses linear weighed-sum utility functions to evaluate the architectural alternatives according to their properties and quality attributes.

SAEM, CBAM, and the current thesis share a property: they all use multi-dimensional preference functions to help make an engineering decision that maximizes or nearly maximizes the expected utility of one or more stakeholders with respect to a number of attributes. SAEM and CBAM help make decisions in the software design process and affect organizations, where there are multiple stakeholders with different objectives. This thesis makes runtime decisions in a pervasive computing setting, where the stakeholder is a lay user.

CARISMA [7] is a software middleware for mobile computing that allows dynamic resolution of resource allocation among conflicting needs based on preferences aggregation, specifically

sealed-bid Vickrey auctions. Like our work, CARIMSA is a runtime system. The difference between our work and CARISMA is the problem being solved. CARISMA applies microeconomics and game theory to make runtime decisions about allocating scarce resources among multiple users. Our work, on the other hand, helps resolve resource scarcity in a one-user setting. Although different applications compete for scarce resources, these applications are running on behalf of one user. Thus, our problem has no game theoretic aspects.

2.2 Task Oriented Computing

As noted in the introduction, the work in this thesis fits into a broader research agenda for task management and configuration. To make automatic configuration possible, our work dove-tails with research in two related areas. The first of these areas is concerned with eliciting the requirements and preferences from the user for each task (the second area is on multi-fidelity applications and mechanisms, and is described in section 2.3).

In [52] Sousa defined a lightweight task description language for describing user tasks and described the design and the implementation of Prism, a graphical tool that allows users to express task requirements and preferences.

Our work and Sousa's are complementary. As argued in the introduction, to automate the control of the configuration, simply eliciting and describing user's tasks are not sufficient to automate the configuration. A coordination infrastructure is necessary to take task requirements elicited from the user and map to the capabilities of the environment. Our work provides the necessary decision-making and control infrastructure required to start from a task's description and control adaptive applications in a manner that maximize the expected utility to a user. In Chapter 8, we provide an overview of the architecture of Aura, and articulate the relationship between Sousa's work and our work in terms of concrete technical artifacts in the aura software.

The Intelligent Desktop Assistant project at Oregon State University [56,57] has developed Task Tracer, a prototype for task-oriented personal computing. The overarching goal of the Task Tracer is to automate some routine chores that users perform on desktop computers. For example, the Task Tracer prototype remembers and learns where the user likes to save certain types files, what sequence of keystrokes the user performs repeatedly, etc. The Task Tracer is specialized around monitoring the user and gradually learning and eliminating routine chores of configuration. The level of task abstraction that Task Tracer operates is much lower than the level of abstraction that Sousa defined. The extent of automation in the Task Tracer is limited to recovering the contextual information about the task, so that user's cognitive overhead is reduced. Furthermore, Task Tracer is not concerned with managing computational resources, but rather recall and recovery of task's contextual information.

2.3 Multi-fidelity Adaptation: Mechanisms for Configuration

To make automatic configuration possible, our thesis relies on the multi-fidelity, resource-aware applications. Multi-fidelity adaptation systems and applications provide an important enabling mechanism for automatic configuration. Such mechanisms reduce the resource requirement of applications by reducing the amount of computation, the fidelity of the data such as video and images, or by off-loading computation from local devices with limited resources to to remote servers with more ample resources. The objective of multi-fidelity mechanisms is to ensure ade-

quate quality of service despite shortage of computational resources such as CPU, bandwidth, and battery. Successful multi-fidelity adaptation has been demonstrated for a variety of applications. For example, in streaming media applications, reduced fidelity is possible by encoding the same video in different frame rate, frame size, and audio fidelities. In browser applications, reduced fidelity is possible by reducing the downloaded content and by lengthening the download latency of web pages. In natural language processing applications (speech recognition, speech-to-text), reduced fidelity is possible by reducing the size of the vocabulary or by reducing the amount of computation. In virtual reality applications, reduced fidelity is possible by reducing the number of triangles rendered.

Some notable systems in the area of multi-fidelity adaptation include GRACE [63], Odyssey [38,42], Puppeteer[33], and Spectra[17,18]. Multi-fidelity features are found in a number of commercial and open-source applications as well. For example, widely available media players, web browsers, and speech recognizers provide mechanisms for multiple fidelity operation.

As the research in multi-fidelity adaptation has matured, many of the multi-fidelity features have been implemented in widely used off-the-shelf applications, e.g., video players, browsers, speech recognizers.

In our work, multi-fidelity applications and mechanisms create alternative ways to realize the delivery of the service of the application. Our thesis provides an approach to map user's high level requirements and goals to specific application settings, thereby choosing among the alternative fidelity modes of the applications by controlling their settings automatically.

2.4 Resource Supply and Demand Prediction

Resource prediction [14,23,24,38,39,40,61] has been a subject of ongoing research in a variety of domains. The term, resource prediction, is widely used and heavily overloaded in computer science research literature. First, resource prediction may refer to estimating the aggregate supply (the available level) of a resource in the near future, e.g., in the next 10 seconds or in the next 2 minutes. Second, prediction may refer to the prediction of the current level of the aggregate resource demand by all users, processes, and applications that require that resource. The predictions of the aggregate supply and demand are related, because having measured or estimated one of those variables can be used to infer the other. And third, resource demand prediction for a specific application may refer to the prediction of how much resource an application will consume in a particular setting. In our work, we are interested in the first and third meanings of the word prediction, i.e., the prediction of the aggregate supply of the resource and the prediction of individual application demand.

Notable research in aggregate resource supply prediction includes the Network Weather System [60] and the RPS toolkit [15]. Both systems make predictions for the aggregate supply of a shared resource, e.g., network bandwidth between two links or CPU load on a shared server. The Network Weather Service provides aggregate resource demand and availability prediction in a system of networked computers that are available for remote computation. Computers equipped with NWS predictors [40] can be queried for their resource load and availability using an internet browser or a programmatic interface. NWSLite is a lightweight implementation of NWS targeted for mobile devices [24]. It has been used to estimate application resource demand and available resource supply on mobile devices. RPS is a library of statistical tools and models that can be used to analyze historical resource traces for the purposes of prediction. Several studies have

used the RPS toolkit or similar tools for off-line predictions [23,49]. The Remos network monitoring and prediction tool [15] is based on the RPS tool and allows prediction of bandwidth online. There is additional evidence (see, for example [49]) suggesting that RPS-like tools can be used for online prediction that works in near real-time. Our model of resource prediction in section 6.7 is motivated by and consistent with the body of research available in aggregate supply prediction. In our work, we use autoregressive-moving average (ARMA) time series models which have been shown both NWS and RPS research to be good predictors of resource availability.

Predicting the resource demand of individual applications is another important building block for our work. The main approach for estimating the resource demand of an application is based on historical profiling of application. Historical profiling consists of the following steps: (1) execute an application in different operating modes while profiling its resource usage, (2) collecting and summarizing the traces of execution, (3) using those traces as a basis for prediction in the future. A number of systems have successfully used this mechanism for resource demand prediction. In some cases, the prediction requires knowledge about the task of the user, specifically, the data that the application is processing (e.g., [39]). In other cases, past history alone is sufficient for resource demand prediction (e.g., [24]). Generalizing, we can state that in the presence of sufficient number of estimators, resource demand of an application can be predicted fairly accurately based on historical profiles of executing the application in different modes.

And lastly, while predicting resource demand is an important building block towards controlling the runtime state of an application into a desired range, it falls short of providing a complete solution to automating configuration because additional information is needed about the user's task and the best way to control multiple concurrent applications. Our approach provides the missing piece.

2.5 Resource Allocation Models and Systems

Part of the intellectual contribution in our approach is to decide how to best allocate scarce resources among multiple concurrent applications that demand those resources. The resource allocation approach we use in this thesis falls under the category of reservation-based approaches. Using estimates for both the level of the available resources and the resource requirements of the applications, our approach decides how to allocate resources among the applications, and instructs the applications not to exceed those allocation limits, effectively reserving portions of each resource for each application.

A large number of resource management systems have used a reservations-based approach to resource allocation. From analytical point of view, Q-RAM system [34] is the closest to ours. Q-RAM is intended for real-time systems such as radars and performs well with dozens of resources and hundreds of services. The objective of Q-RAM is to admit applications (in Q-RAM, these are called tasks) and allocate resources among the admitted applications to maximize the utility of the system. Q-RAM uses resource demand profiles of each application and allocates resources among the admitted applications based on these profiles. The resource allocation problem in Q-RAM is a one-period, multiple-dimension, multiple-choice knapsack problem. The resource allocation problems in our system are similar to Q-RAM's. However, the problem we solve in this thesis has additional challenges that Q-RAM does not address: (1) selecting among alternative applications, (2) handling resources such as battery which have intertemporal substitution possibilities, and (3) making resource allocation decisions that are optimal over time. We borrow two

algorithms for resource allocation from Q-RAM. These algorithms are used in solving an important sub-problem in stage 1 of our configuration problems.

There alternatives to a reservation-based allocation. For example, in the Nemesis operating system [41], resource allocation mechanism is based on congestion prices. For each resource that is utilized near capacity, a congestion price is determined based on the demand for that resource from all applications. Applications are then charged the congestion price. At the same time, the applications are credited based on the level of quality of service they provide. If an application is being charged more in resource usage then being credited for quality of service, then the application reduces its quality of service. In this manner, applications are rewarded for finding an operating point where credits offset charges. Less efficient applications will be forced to scale down their resource usage, while more efficient applications will be able to increase their resource consumption. This is akin to a market for resources, where higher bidders are able to take the resources, ensuring that supply-demand imbalances eventually disappear. Unlike our approach, the approach in the Nemesis system does not require a centralized resource arbiter (but does require centralized resource accounting for each resource). However, compared to a reservation-based resource allocation, a congestion based approach has two potential drawbacks: (1) the system may take a long time to reach equilibrium and (2) even when equilibrium is reached, the system might be at a local maximum, i.e., there are no guaranteed that the equilibrium is a global maximum. More importantly, Nemesis and other decentralized systems have no planning for the future, i.e., they are reactive.

Dynamic resolution of resource allocation policy conflicts involving multiple mobile users is addressed in CARISMA [7] using sealed bid auctions. While our configuration mechanism shares utility-theoretic concepts with CARISMA, the problem solved in our work is different. In that work, the objective is to select among a handful of policies so as to maximize an objective function of multiple users with competing objectives. In our work, the objective is to choose among possibly thousands of configurations so as to maximize the objective function of one user and determine optimal resource allocation among applications. As such, our work has no game-theoretic aspects, but faces a harder computational problem, namely a resource allocation and planning problem with a large space of alternatives.

2.6 Home Automation Systems

As homes become increasingly rich in devices and appliances, home automation systems have become an important topic of research. Several research projects have addresses the automation of home devices and appliances. The eHome Systems project [44,45] addresses device interoperability, installation time configuration, and deployment automation, with the focus of reducing the costs due to home automation product and service vendors. eHome's three-phase software process model: Specification, Configuration, and Deployment (SCD), logically parallels the task description and automatic configuration steps in the Aura Architecture, which is umbrella infrastructure within which our work fits. The eHome Configurator tool leverages the configurable features of the eHome platform and allows vendor technicians to easily tailor the installation to the needs of the client. Unlike the eHome project, that targets installation time and deployment configuration automation by vendors of software, our work automates the control of the configuration of everyday user applications on mobile devices that can be used both at home as well as outside of the home.

The Open Home Framework (OHF) developed by the Korean ETRI institute [28] focuses on hardware, software, and protocol interoperability and integration issues. Having collaborated with the ETRI institute, we have discovered that the features offered by the automatic configuration software described in this thesis are complementary to those provided by the OHF Home Server. Specifically, our work can be used to allow the end users to configure tasks according to a user's needs and preferences, while the services offered by the OHF can be used as available suppliers.

The uDesign [55] is an end-user design tool for notification and monitoring tasks. Using uDesign, a lay user can assemble the components required for home monitoring and notification tasks from available services. For example, an integrated home security monitoring task can be assembled using one or more sensors, video camera, and a notification device. Our work and uDesign differ in two respects: (1) they target different types of tasks and (2) resource monitoring and allocation is not a key concern for uDesign. The uDesign system architecture is based on the Aura Architecture and leverages our infrastructure for automatic configuration at the level of finding and interconnecting suppliers.

2.7 Autonomic Computing and Self-* systems

In the past decade, there has been growing interest in engineering systems that are resilient to failures and require less manual oversight. Systems that provide the ability to automate some aspect of repair, adaptation, or control are called self-* systems.[2] Our work fits under the broad umbrella of self-* systems.

There are two broad approaches to engineering self-* systems: self-adaptive and self-organizing. These approaches differ in the way that a desired outcome is achieved in a system. In self-adaptive systems, there is a global objective function and a global entity that coordinates all the distributed parts of a system to meet the global objective. The system is guided to the desired state in a top-down manner, using the objective function. The global entity has access to all available information and typically maintains a global model or state of the system. Based on the distinction between self-adaptive and self-organizing systems, the infrastructure for automatic configuration is a self-adaptive system. In our work, the Environment Manager has the role of the global coordinator, the overall utility of the task is the global objective, and the desired state is a configuration that maximizes utility.

Self-organizing systems, on the other hand, use locally optimal strategies to achieve a globally desired state. There is no global system model or a global coordinator. The available information and decision-making are distributed across many nodes in the system. Self-organizing systems are typically inspired by biological [27], social [21], or other natural phenomena. By using the behavior of ants, migratory birds, or some other natural or social structure observed in the physical sciences, the designers of self-organizing systems implement local policies, actions, and

[2] The asterisk in the term, self-*, is a wildcard for a desired property, such as configuring, healing, or adapting.

mechanisms that when used across the entire system, collectively yield a globally optimal outcome or state.

The main advantage of self-organizing systems over self-adaptive systems is in scalability. Self-organizing systems can typically scale to very large sizes, because computation is distributed among a large number of independent nodes, and there is no global coordination required. As a result, the communication costs are greatly reduces, and the computation cost of node is typically fixed even as the size of the system grows. In self-adaptive systems, the global coordinator can become a bottleneck as the system size increases. Typically, to ensure that the coordinator is not slowing the entire system, efficient algorithms are required.

The advantages of the self-adaptive systems over the self-organizing systems are several: (1) the strategy of maximizing the system's utility is typically easier to implement in the presence of one global objective, (2) a self-organizing system can easily reach a local maximum and not improve any further, while a self-adaptive system has an explicit model of optimality and typically does not get stuck at a local optimum, and (3) the time it takes for a self-organizing system to reach an optimal or near-optimal operating point may be long. Also, the design of a self-organizing system requires inventing or borrowing from another scientific discipline one or more local policies that ensure a globally optimal behavior. Such policies may take considerable amount of cognitive resources and trial-and-error. The majority of self-organizing systems have implemented policies observed elsewhere.

2.8 Anticipatory Approaches

One of the differentiators of our work from prior adaptive and self-adaptive systems is the anticipatory strategy of configuration. The anticipatory strategy uses predictions of the future values of input variables to make forward-looking decisions of resource allocation and application selection. Forward-looking approaches have been proposed and used in other domains. For example, the online stochastic combinatorial optimization (OSCU) approach is similar to our anticipatory strategy [3,29]. In OCSU, various combinatorial optimization problems such as optimal vehicle dispatch and network packet routing are solved by leveraging probabilistic priors of the future values of problem inputs. There is equivalence between the algorithms for automatic configuration in this thesis and the algorithms described in [29]. The equivalent of our Reactive, Perfect, and Anticipatory are called respectively Local, Offline, and Expectation algorithms. While our work shares theoretical foundations with OCSU, our work clearly differs from OSCU because of the problem domain.

Our work is similar to the Active Virtual Network Management Prediction System (AVNMP), which aims to use simulation models running head of real time to predict resource demand among network nodes. Such predictions can be used to allocate network capacity in anticipation of demand increase, and to ensure adequate quality of service to different network flows [20]. While our work shares conceptual foundation with Active Networks, the two are in entirely different domains.

2.9 Dynamic Programming and Markov Decisions Processes

Our anticipatory algorithms described in Chapter 9 are based on the dynamic programming method. In this context, programming refers to a tabular method, not to writing computer pro-

gram code. Dynamic programming method (see chapter 16 in [10]) solves problems by combining the solutions to subproblems. Each sub-problem is solved once, and its answer is stored in a tabular structure. The algorithm can be implemented either as a series of nested loops, or as a recursion.

There is a different, but equivalent formulation of dynamic programming that is often used to compute the optimal behavior of a dynamic system. In this formulation, Markov Decision Processes (MDPs) are used to describe the system [2].

An MDP is a tuple $(S, A, P_a(s,s'), R(s))$, where

- S is a set of states that the system can be in,
- A is a set of actions that can be invoked upon the system,
- $P_a(s,s')$ is a transition probability matrix, that states that probability of transitioning into state s' when action a is invoked while the system is in state s,
- $R(s)$ is the immediate reward (or expected immediate reward) received after the system into state s.

The objective is to maximize a cumulative function of the reward, e.g.,

$$\sum_{t=0}^{\infty} \chi^t R(s_t)$$

where γ is the discount rate, $0 < \chi \leq 1$.

A problem expressed as an MDP can be solved using dynamic programming. The solution to an MDP is expressed as an optimal policy π, which specifies the best possible action, a, for each state s.

There are similarities between MDPs and the problem of automating the configuration solved in this thesis. In particular, we can express a coarse approximation of the problem of anticipatory configuration as an MDP. The states of the system are described by the past values resources, past prediction errors, and the currently running supplier assignment. (The number of the past values of the resources and prediction errors required in the state description depend on the auto-regressive and moving average orders of the distinguished linear recent history predictor.) The set of actions is the set of supplier assignments. Transition probabilities are given by the resource predictors.

However, there are important differences between MDPs and the problem of configuration. Specifically, absolute time and resources such as battery make the problem of configuration non-Markovian. Thus, the MDP formulation is not sufficient to describe the problem of configuration.

Despite the differences, we use the dynamic programming method to solve the problem of configuration. Briefly, our solution simulates resource paths forward up to future time that equals the current time plus a fixed decision horizon, DH. Then, the problem is solved using dynamic programming by tabulating backwards in time (see sections 9.4and 9.5 in Chapter 9).

3 Background and Setting

In this chapter we define the terms used throughout the thesis and describe the relationship among them[3]. We use the example about the movie critic from the introduction to provide the reader with an intuition behind the terms.

3.1 Relevant Terms

The set of devices, applications, and resources that are accessible to a user at a particular ***location*** constitutes the computing ***environment*** for the user. For a movie critic working in the airport, the computing environment is his laptop, all the applications available on the laptop, and the wireless network in the airport lounge, if available.

Applications and devices in an environment provide ***services***, such as: playing a video clip or editing a text document. A service is an abstract description of the capabilities of an application or a device. Associated with a service is a ***type***, which consists of a literal name, and zero or more ***quality dimensions***. A quality dimension describes an aspect of the service perceptible to the user. For example, an application that can play back a video stream offers a service of type "*play video*". "*Frame size*", "*frame rate*", and "*audio*" are all quality dimensions of a "*play video*" service.

A quality of service dimension has the following properties:

- it is perceptible by the user,
- it is objectively measurable, e.g., by programmatic means provided by the application,
- the level of quality has an impact on user's satisfaction; typically, the higher the quality of service, the more satisfied the user is,
- it can be controlled at runtime either programmatically, manually, or both,
- there is a correlation between the level of service and resource demand (typically, better service requires more resources),
- all applications that provide a particular service type share the same set of quality dimensions.

For example, all "*play video*" applications have a user-perceptible quality that identifies how many frames are displayed per second, "*frame rate*". "*Frame rate*" can be measured at runtime. The higher the frame rate, the smoother the frame transitions, and typically the more satisfied the user is. The level of frame rate can be controlled at runtime by using video streams encoded at

[3] The terminology has been established as part of the design of the Aura architecture for ubiquitous computing at Carnegie Mellon University. Sousa introduced the majority of the terms in [54]. We have extended his terminology for our thesis.

different bit-rates or using scene smoothing techniques. Watching the video at a frame rate requires streams of higher bit-rate, which in turn demands higher bandwidth and CPU utilization.

The relationship between services and applications is "many-to-many". First, in a given environment there may be multiple applications that provide the same service. For example, Windows Media Player, Real Player, and QuickTime applications provide a "play video" service. When playing video, all three applications have three measurable quality dimensions: "*frame size*", "*frame rate*", and "*audio*". Second, an application may provide multiple services. For example, Windows Media Player provides both "*play video*" and "*play audio*" services. As another example, Microsoft PowerPoint provides "*edit slides*" and "*show slides*" services.

We use the term ***supplier*** to uniquely identify an instance of an application running on a specific device. For example, the instance of the Windows Media Player running on the movie critic's laptop is a supplier of a "*play video*" service. This definition allows us to treat the installations of the same application code on two different hardware devices as different suppliers. Associated with the supplier is the list of the services supported by the application.

Resources are the means available in the environment to support the capabilities of the suppliers. Suppliers consume resources and provide services. The resources available to the movie critic at the airport lounge are the CPU, battery, and the wireless network in the airport[4]. From a user's point of view, the availability of resources such as bandwidth is important only to the extent that it impacts the quality of service delivered by the applications. However, the remaining supply of a resource such as battery might be valuable to the user. For the purpose of this thesis, we will consider the following four types of resources: CPU, bandwidth, and battery. Jointly, these resources exhibit an interesting mix of properties [47].

A supplier consumes resources to provide service. A supplier, by virtue of being an instance of an application providing service, has a number of settings, which directly control the runtime level of quality of service delivered by the supplier. For each service that a supplier provides, it maintains a profile of resource consumption. This profile correlates the runtime level of quality of service to the required level of resources. The level of resources required by the supplier can be indirectly inferred (e.g., by profiling the supplier under different runtime settings), and controlled (e.g., by instructing the supplier to consume no more than a specified amount of a certain resource).

In his thesis [54], Sousa introduces the notion of task-oriented computing. He argues that users execute a suite of related applications simultaneously in order to accomplish their computing goals. Because these applications are almost always started and stopped together, Sousa groups them together as a logical set. For the purpose of this thesis, a ***task*** is a set of related services that the user needs to use simultaneously in order to accomplish a goal. The ***description*** of the task specifies the services needed for the task and ***preferences***, which are numerical weights and mathematical functions that capture user's preferred trade-offs for the different user-perceptible aspects of all the services in the task. For example, movie critic's task of writing a review requires three services: "*play video*", "*edit text*", and "*browse web*".

---

[4] The environment has other computational resources, e.g., disk space. However, we don't include any other resources in the analysis of this thesis.

Tasks have states: pending, active, and closed. When a user *creates* a task, it is in a ***pending*** state. When the user decides to work on a pending task, he *resumes* it, making the task ***active***. The user can have multiple active tasks. The user can *suspend* an active task, making the task pending again. Before the task is eventually *completed*, the user may resume and subsequently suspend it several times. Once the user has met the goal for which the task was created, he can *close* it, making the task ***closed***. After the task is closed, it can not be resumed again.

Let's consider an example. In order to start working on the movie review, the critic first has to create a task for that purpose. Once the task is created, it is in the pending state. The critic can then resume the task, making it active. Before boarding the plane, the critic can suspend the task, making it pending again. In the plane, the critic can resume the task. Once he finishes working on the task, he can mark the task closed.

When used as a noun, a ***configuration*** refers to a suite of suppliers, their runtime settings, and a resource allocation among the suppliers. When used as a verb, the act of configuration refers to selecting a configuration, i.e., finding a suite of suppliers and determining their runtime operating parameters in response to a task request. ***Instantiating*** a configuration refers to the act of executing the suppliers in the configuration (e.g., by starting them) and ***configuring*** them, i.e., setting their runtime operating settings. A supplier is ***assigned*** to a service in a task when the said supplier is chosen as part of the configuration for the task. A ***supplier assignment*** is the entire suite of applications in a configuration. A ***reconfiguration*** is the act of changing the configuration when the task is in progress. A reconfiguration might change the supplier assignment (e.g., by switching one or more suppliers by alternatives), change the runtime state of any of the suppliers in the assignment, or both.

Windows Media Player, Microsoft Word, and Firefox Browser suppliers together with their runtime settings are one configuration for the task of reviewing a movie. The relevant runtime settings of the Media Player specify the current values of each of the quality of service dimensions: "*frame rate*", "*frame size*", and "*audio*". The settings of the Firefox Browser specify the values of "richness" and "latency" quality dimension of loading pages.

Notice that the definition of a task does not specify suppliers. Instead, a task definition contains services and preference functions. The definition of a service and the criteria for identifying quality dimensions of a service ensure that two different suppliers of the same service type can be interchangeable for the purpose of configuration (because two suppliers that offer the same service share a service type and a common set of quality dimensions). For example, Firefox and Internet Explorer browsers are interchangeable for browsing purposes. An astute reader will rightfully argue that two different applications may not always be interchangeable from the user's point of view. Indeed, a simple editor such as Notepad might not be a good substitute for a sophisticated editor such as Microsoft Word. We will see how careful modeling of user preferences addresses this concern in the detailed description of the analytical model of configuration.

Automatic configuration is part of a larger and broader-scope infrastructure that provides automatic task management. In his thesis [54], Sousa described two separate functional concerns for an automatic task management infrastructure: (1) to define *what* a lay user needs for her everyday tasks in terms of services and their output quality, and (2) to determine *how* best to marshal the applications and resources in a given environment for the best possible experience to the user. This separation exploits the task abstraction, and paves the way for separately solving the two problems. Sousa provided a solution for the first problem. His solution implements a runtime

tool for eliciting the needs and preferences of each task from the user, and expressing those using a machine readable task description. In this thesis, we address the second problem.

3.2 Problem Setting

3.2.1 Target Users

In this thesis, we target users with a wide range of computer skills and expertise: anywhere from novice to savvy. While a percentage of the targeted users may be experts in the functional aspects of the specific applications they use for day-to-day tasks, we assume they are not experts in the underlying computing and network infrastructure.

We assume that the user utilizes both personal devices as well as any surrounding infrastructure available to him at a particular location. This infrastructure may include hardware, displays, network, etc. A task may require more than one sitting to complete; therefore, the user may work on the same task over many days.

We target users that are transient but not continuously mobile. Over the course of their productive day, these users prefer to work in different locations such as an office, lecture room, home, coffee shop kiosk, airport, client site, or a hotel. Typically, a user spends a meaningful period of time working in one location, and then moves to a different location as needed. A user does not work on tasks continuously, preferring to take intermissions for a number of reasons, e.g. outside interruptions, commitments that don't require computing infrastructure, or circumstances such as unavailability of devices or resources. Although we don't target users who work on their computing tasks while moving continuously, we do mention about that possibility in the future works section.

Targeted users may carry with them hardware devices such as laptops, PDAs, smartphones. However, we make the reasonable assumption that the user is also willing to utilize nearby hardware and infrastructure and nearby or remote compute servers.

3.2.2 Computing Tasks of the Targeted Users

The tasks we target require simultaneous use of multiple applications that are intensive in one or more computational resources. These everyday user tasks can be satisfied by widely deployed application products from a large number of vendors.

We assume the availability of a complementary supporting infrastructure that allows the user to describe his task as well as some additional information regarding the user's preference for various perceptible dimensions of the applications in the task, e.g., Prism [54]. This task description and elicitation infrastructure works in concert with the configuration provided by this thesis.

We do not exhaustively list the entire range of possible applications that the user might need. However, the research in this thesis is particularly applicable to applications that are adaptable to resource conditions. In the literature, such applications have been referred to by a number of descriptions: fidelity-aware, resource-aware, adaptive, or adaptable. Adaptation mechanisms are either bundled with the application by the vendor or added as a feature at a later time. Several research projects have addressed the problem [13,39] of adding adaptation mechanisms to existing applications. Applications that we target in our work include but are not limited to the following: spreadsheet editing; slide editing and presentation; document editing; web browsing,

chatting; videoconferencing; listening to audio; watching video; reading news and email; editing photos; updating web-pages; creating reports; text and speech translation.

A task may require multiple activations before completion. For example, a report for a project might require several days to write and the user may work on this report multiple times in various locations. Other tasks may be repeatable at known days and times of the week. A user's tasks may have invocation times and durations with predictable patterns. For example, using calendar information, task history, or direct input from the user, it is possible to elicit a sequence of tasks and their durations that the user will request over a foreseeable horizon. The analytical model in this model leverages any predictability in the user's task usage patterns to user's benefit.

3.2.3 The Computing Environment

We expect that the targeted users will choose to carry out their computing tasks in different locations, taking advantage of the computing capabilities available to them at a given location. Due to the proliferation of computing devices and resources, it is reasonable to expect that different environments will offer a different mix of devices, applications, and resources, i.e. lack homogeneity. The configuration infrastructure we propose is capable of supporting user's tasks in such environments despite the *heterogeneity* across environments.

Another challenging characteristic of the environments we target is limited and fluctuating resources. *Resource scarcity* has a number of root causes. First, environments where the available level of resources is shared among a large number of anonymous users might experience resource scarcity. This is common in environments where the resources are free at the margin, e.g. a free or almost free wireless network in public spaces, public computer terminals, etc. Another contributor of resource scarcity is miniaturization of hardware: devices with smaller form factor have a comparative disadvantage in resources relative to desktop or laptop computers. And third, newer versions of platform and application software are typically designed to consume more resources than their older predecessors. Combined together, these factors contribute to the scarcity of resource availability relative to application resource demand. Resource scarcity presents many challenges for the users we target. In this thesis, we explicitly address that problem by mitigating scarcity in the following resources: computing cycles (CPU), network bandwidth (upstream and downstream), physical memory (RAM), and battery energy. The solution in this thesis leverages existing research such as reduced fidelity operation and resource-aware adaptation.

Changing environment introduces additional challenges. The set of available applications and devices may change over time. The level of available resources may also change over time. When the changes reduce the number of available applications, devices, or resource levels, a user's task may be impacted negatively. When the changes are additions or improvements, there may be additional opportunities to improve the utility of user's task. In this thesis, we address both scenarios.

Another challenge that we address in this thesis is the large number of alternative ways to realize a task. These alternative realizations are made possible by the availability of alternative applications for a specific part of the task and by the large number of alternative runtime configuration settings of many of the applications. For example, either Internet Explorer or Mozilla Firefox can be used where a browser is required. As another example, a video clip can be provided in different frame update rates, different pixel sizes and resolutions, different audio qualities, different encoding schemes. The Cartesian product of such possibilities creates a large number of alternative realizations of the same service by a particular application. Furthermore, the number of different applications that can stream video contributes to the alternative realizations of the task. The number of possible combinations of applications and configuration choices provide alterna-

tive realizations of a user's task. This is an important enabling mechanism when resources can be scarce or changing and when different environments offer different capabilities. Alternative task realizations allow the user to work under a large number of resource variations and environment conditions.

However, the large number of alternative realizations comes at a cost to the user: the growing complexity of the environment and its management. This thesis addresses two important challenges: the *selection* of the best possible realization of a user's tasks given the large number of alternatives and the *dynamic control* of the realized task over long periods of time.

3.2.4 The Properties of the Resources

In this thesis, we consider four types of resources: CPU cycles, network bandwidth, battery, and memory. In a position paper [47], we argued that resources have differentiating characteristics that must be accounted for when making allocation decisions. Table 2 summarizes the properties of the four resources. The last two columns have been added to our model since the publication of the original paper [47].

Table 2: The properties of the resources.

		Properties of Resources						
		Unit	Scale	Divisibility /Granularity	Perishable	Rival	Shared	Renewable
Resource	CPU cycles	MHz	Ratio	Dense-discrete	Y	Y	Depends	Automatic
	Bandwidth	Mbps	Ratio	Dense-discrete	Y	Y	Yes	Automatic
	Battery	wattHour	Ratio	Dense-discrete	N	Y	No	If charged
	Memory	MB	Ratio	Dense-discrete	Y	Y	Yes	Automatic

The *measurement scale* describes the kind of scale that is appropriate for measuring the resource. All four resources we consider have *ratio* scale. A ratio scale is an *interval scale* with the additional property that distances are stated with respect to a rational zero [31]. A ratio scale, just like an integer scale, preserves the sizes of differences.

The *divisibility* (*granularity)* property describes the density of the underlying space of the resource. There are three possible values: continuous, dense-discrete, and sparse-discrete. Intuitively, this property indicates in what increments the resource can be allocated. Resources that have continuous granularity can be allocated in infinitesimally small portions. Resources that

have discrete granularity can only be allocated in multiples of some minimum amount. All four considered resources are dense-discrete[5]. In this thesis, we allocate resources in discrete units.

The *perishable* property describes whether an unused amount of resource will be lost if not used. Battery is not perishable, while CPU and bandwidth are perishable. Notice that a non-perishable resource has intertemporal fungibility, while a perishable resource does not. Memory is not perishable either, because unused memory does not get lost.

The *rival* property describes whether the consumption of a unit of resource by one entity precludes the consumption of the same unit by any other entity. To help the user appreciate this property, consider sunshine as a resource. On a sunny day, the consumption (or enjoyment) of sunshine by one person does not preclude that by another person. For rival resources, the available supply of the resources creates a limit on the aggregate consumption of that resource by all involved entities. All four resources we consider are rival.

The *renewal* property describes how new amounts of resource are created. In case of CPU and bandwidth, the resource is automatically renewed. In case of memory, unused portions of the resources are In case of battery, the resource is replenished only if it is being charged from the wall.

The *shared* property indicates if the resource is shared among many users. Network bandwidth is shared among many users. On the other hand, the resources of a personal computing device, e.g., the CPU, the battery and memory, are typically not shared. The shared attribute has implications on modeling the future available level of the resource.

In his thesis [37], Narayanan proposed taxonomy of computing resources that classifies each resource into one of three basic categories: *time-shared*, *space-shared*, *cache*, and *exhaustible*. He states that those three categories are not exhaustive; however, they are sufficient to describe the resources considered in his thesis. CPU and bandwidth are time-shared resources, while battery is an exhaustible resource. Memory is a space is a space-shared resource. Narayanan notes that time-shared and space-shared resources are renewable: time-shared resources such as CPU and bandwidth are renewed periodically, while space-shared resources such as disk space are reclaimed when applications release them after use.

The two models of the resources (ours and Narayanan's) are complementary and consistent with each other. The time-shared property of the two resources: CPU and bandwidth is important to consider when scheduling the resource at a very small granularity of time, e.g., microseconds or milliseconds. Time-shared property is important when there are deadline constraints in a system and latency of computations is important. Narayanan's system performs fine-grain management of the resources, while the system in our thesis leaves such management to either the operating system or a fidelity-aware middleware. For the purpose of our thesis, the renewable and perishable properties of CPU and network bandwidth are more important.

---

[5] Some might argue that all four resources are continuous. However, for the purpose of this thesis, the distinction does not matter. A resource that has continuous granularity can be allocated in infinitesimally small portions, and can safely be treated as a dense-discrete resource.

Narayanan's model classifies battery energy as exhaustible. As he notes, the amount of resource is fixed and depletes over time. To replenish the resource, the battery must be charged using wall electricity. Our model reflects these properties of battery and is consistent with Narayanan's.

3.3 Chapter Summary

In this chapter we have introduced terms that are used throughout the thesis. We have described the separation of the problem of automatic task management and configuration into two high level goals: (1) what is needed in terms of services and their quality for each task and (2) how to marshal the applications and resources in the environment to best provide user's needs.

We have described the characteristics of the users, applications, and the resources. By doing so, we have set the stage for rigorously describing the problem in terms of an analytical model.

4 Modeling the Configuration Problem

The analytical models in this thesis capture the dimensions, inputs, and constraints of the problem of automatic configuration. Recall from Chapter 3, that a configuration is a suite of suppliers, their runtime settings, and resource allocation among them. The problem of configuration is to find a sequence of near-optimal configurations over a period of time such that the expected utility of that sequence to the user is nearly optimal according to the preferences of the user.

In this chapter, we provide an overview to structuring the problem of configuration. Throughout this chapter, we make use of a running example involving Jane, the movie critic. Jane is working on a review for a newly released movie for her employer, a local newspaper. To write the review, Jane has defined a task with three services: play video, browse web, and edit text.

4.1 Problem Structure: Overview

The key insight to structuring the problem of configuration is to separate what the user needs in terms of abstract capabilities in the environment from the mechanisms that are required in order to provide those capabilities. To that end, we separate the relevant concerns of the problem into three spaces:

- user's utility,
- abstract capability,
- resource.

Utility is a measure of user's satisfaction with respect to the possible configurations of the environment. The user cares about the level of capabilities that a configuration offers, e.g., how good the video playback is, or how fast the pages are loaded in a browser. We use preferences to evaluate the capabilities of configurations from the user's point of view. Preferences are mathematical functions that map from the capability space to the utility space.

While the user cares about capabilities offered by a configuration, the infrastructure making configuration decisions needs to compute the amount of resources required to provide those capabilities. In our model, supplier profiles provide part of the necessary information. Informally, supplier profiles express the efficiency of the suppliers when converting resources into different levels of capability. Formally, the supplier profiles are relation between the capability space and the resource space.

In addition to the resource-efficiency of the suppliers, the infrastructure must also ensure that the resources available in the environment are sufficient to provide the desired level of capability. So the infrastructure must know the level of available resources in order to decide how to allocate the resources among the suppliers in a configuration. Resource availability provides this information.

To summarize, our model has three inputs:

- user's preferences,
- supplier profiles,
- resource availability.

The user's preferences, respectively the supplier profiles, are mappings between capability space and the utility space, respectively between the resource space and the capability space. The third input, resource availability, is a constraint in the resource space.

The analytical approach in this thesis is based on optimizing the match between the preferences of a user and the capabilities of the available suppliers under the constraints of the available resources. Finding such a match corresponds to finding a configuration that maximizes the utility of a user for a set of tasks, given the profiles of available suppliers, and the level of available resources. A configuration has three parts:

- the suite of suppliers and expected utility from those suppliers,
- runtime setting for each supplier,
- resource allocation for each supplier.

In the remaining sections of this chapter, we discuss the structure of the configuration problem in more detail. Section 4.2 describes the three spaces that the problem is organized into. Section 4.3 describes the input variables of the problem. Section 4.4 describes the decisions variables of the problem.

4.2 Building Blocks: Spaces of the Problem

4.2.1 Utility

Utility is widely used in microeconomic theory to express the satisfaction of a user with respect to alternative outcomes. In this thesis, utility is a measure of user's satisfaction with respect to the current configuration of the environment. Configurations with higher utility are preferred by the user.

We distinguish between *instantaneous* from *accrued* utility. Instantaneous utility (IU) is a real number in the [0,1] interval and captures a user's satisfaction with respect to the current configuration over a short, fixed-length period of time. An IU close to 0 corresponds to a configuration of an environment that is totally unacceptable to the user while an IU close to 1 corresponds to a configuration that is perfectly optimal from the user's point of view. Further improvements to the task do not provide perceived benefits to the user utility.

Accrued utility (AU) allows reasoning about a user's satisfaction with respect to the state of the environment over a long period of time. In this work, accrued utility over a period of time is the integral of instantaneous utility with respect to time. Just like instantaneous utility, accrued utility is also measured in positive real numbers; and higher accrued utility translates to better user ex-

perience when the length of time under consideration is kept constant. Accrued utility is capped by a value that is proportionate to the length of time for which accrued utility is calculated.

4.2.2 Capability

We first describe the capability space of a single service. Recall, from section 3.1, that a service is an abstract description of application capability. A service is defined by its type, which consists of a literal name and one or more quality of service (QoS) attributes. Each QoS attribute is defined by a literal name, a unit, and a range of values, either continuous or discrete.

The capability space of a service is a multidimensional vector space. The number of dimensions in this space is one more than the number of QoS attributes of the service. The additional dimension expresses all additional features of a service that are not captured by any of the QoS dimensions. In our model, we use the name of the supplier to capture all those attributes. This is a reasonable way to capture all features that differ from one application to another. The possible values along this dimension include the names of all known suppliers of the service.

For example, a "*play video*" service defines the abstract capability of watching a video. This service has 3 QoS attributes: "*frame rate*", "*frame size*", and "*audio*". "*Frame rate*" captures the update rate of the video playback in units of frames per second (fps). "*Frame rate*" typically ranges from 0 to 30 fps (although in practice we don't see frame rates of lower than 10, because usually are unhappy w/ anything less than that). "*Frame size*" captures the number of pixels in each frame and usually ranges from a few thousand to a few million pixels. "Audio" describes the quality of the audio in bits per second (bps), and can range from very poor (12 Kbps) to CD-quality (384 Kpbs). The fourth dimension in the capability space captures the choice of the application. Possible values include: "Windows Media Player", "RealOne Player", "QuickTime Player", and "Java Media Player". To capture the possibility that a different video player might be available in some environments, we allow a special value called "Other" in this dimension.

Here is one point in the capability space of the "*play video*" service:

- ("Windows Media Player"; 12 fps; 256,000 pixels; 128 Kbps).

The entire capability space of the "*play video*" service is quite large and includes all combinations of possible values in each dimension of the space.

We define the capability space of a task by using the capability space of a service as a building block. The capability space of a task is the Cartesian product of the capability spaces of the services that make up that task. Jane's task has three services: (1) "*play video*", (2) "*browse web*", and (3) "*edit text*". A "*play video*" service has 3 QoS attributes, and a 4-dimensional capability space. A "*browse web*" service has 2 QoS attributes ("*latency*" and "*content*") and a 3-dimensional capability space. "*latency*" is measured in seconds(s) and can range from 0 to infinity. "*content*" is a discrete attributes, and can take two values: "*images*" and "*noImages*". An "*edit text*" service has no QoS attributes and a 1-dimensional capability space. Therefore, the task has an 8-dimensional capability space.

Here the description of one point in the capability space of that task (we deliberately describe the parts of the point corresponding to each service on a different line for readability):

- ("Windows Media Player"; 12 fps; 256,000 pixels; 128 Kbps),

- ("Internet Explorer"; 5s; "images"),

- ("Windows Word").

4.2.3 Resource

The multidimensional resource space allows reasoning about resource availability, demand, and resource arithmetic. For the purpose of this thesis, we consider 4 types of resources: network bandwidth, battery, and CPU. However, there may be multiple instances of the same resource in an environment. The resource space captures each resource instance. Formally, the resource space in an environment is a multidimensional space that is the Cartesian product of the resource instances available in the environment. The unit and range of each resource is determined by the type of that resource. We use real positive numbers to capture resource availability and demand.

The points in the resource space are vectors. Here is one point in the resource space of an environment that has 3 resource instances: the CPU, network bandwidth, and memory:

- (600 MHz; 500 Kbps, 50,000 KB).

4.3 Problem Inputs

4.3.1 User Preferences

User preferences are a collection of real-valued weights and functions that evaluate how good a configuration is from the point of view of the user. Formally, preferences map from the capability space into the utility space. Therefore, the domain of the preferences is the capability space, and the range is the utility space.

This utility provides a measure of the usefulness of the points in the capability space to the user. Although the capability space is potentially large, it follows a certain structure as described in 4.2.2. User preferences are designed to take advantage of such structure. The preferences can evaluate *any* point in the capability space, i.e., they provide a complete map from the capability space into the user space. In practice, though, the infrastructure only needs to evaluate a small portion of the points in the capability space.

User preferences capture two concepts, corresponding to two different kinds of dimensions in the capability space. First, *QoS preferences* express a user's utility for the levels of service in each individual dimension of quality of service (QoS) and tradeoffs among these dimensions. Second, *supplier affinity preferences* express user's familiarity, expertise and preference for specific suppliers that can supply services in a task.

In addition to capturing preferences for the choice of suppliers and their quality of service, we are also interested in modeling the effect of unwanted changes. Supplier *switching penalties* express the negative effect of disruptions from replacing already running suppliers with alternatives. Such switches might be necessary in order to adjust to changes in the environment conditions, e.g., a drop in the available level of resources or simply desirable in order to provide better utility.

Our model of preferences is *linearly-additive*. Recall that points in the capability space of the task are multidimensional vectors. To calculate the overall utility of a point in the capability

space of the task, we first apply the utility function for each component to calculate component-wise utility, and then combine the utility of each component using weighted addition. The weights applied to the utility of each dimension specify how important that dimension is to the user when in comparison to all the other dimensions in the task. The weights are normalized so that they sum to 1. This ensures that the maximum possible instantaneous utility is 1.

To illustrate supplier preferences, let's refer again to Fred's example. For taking notes, Jane's first choice is "Microsoft Word", which is somewhat preferred over "Notepad" or "Emacs". Jane is unwilling to use the "*vi*" editor at all. Although she is willing to try other text editors, she would rather use "Notepad" or "Emacs". Recall that a *supplier identifying tag*, e.g., "Microsoft Word", "Notepad", is a compact representation of application-specific features. The supplier identifying tag is the input of the supplier preference function for "*edit text*" service, as follows:

- For "Microsoft Word", utility is 1,
- For "Notepad" or "Emacs", utility is 0.7,
- For "vi", utility is 0.0.
- For "Other", utility is 0.5.

Because the choice of a text editor is not all that important, Jane assigns a weight of 0.08 to this preference. Recall that: (1) the capability space of the task has 8 total dimensions, and (2) the sum of all 8 weights must add to 1. Therefore, the average weight is 0.125, and a weight of 0.08 makes a particular dimension less important.

As an example of QoS preferences, suppose that the user is watching the video over a network link and the bandwidth suddenly drops: should the video player reduce the frame quality or frame-update rate? The answer depends on the user's preference for frame rate and frame quality in the context of the current task. For a soccer game, the user may prefer to preserve the frame-update rate at the expense of frame quality; however, if the user is watching a movie, he may prefer image quality to be preserved at the expense of frame-update rate. Preferences with respect to tradeoffs such as these are represented by indicating the acceptable levels for each *QoS dimension* of the service. In the example, the QoS preferences for the task of watching a soccer game would set a high threshold for the acceptable frame updated rate, say 25 frames per second, and a low threshold on the acceptable image quality, say 20Kbit per frame; whereas for the movie, the QoS preferences could set a high threshold for image quality, and a low one for frame update rate. Since both QoS dimensions compete for resources, such as bandwidth, by swaying the thresholds the user can direct a resource-adaptive video player to make different tradeoffs upon resource variation.

To make QoS preferences easier to elicit and process, we adopt preference functions originally proposed by Sousa [52]. Sousa makes two simplifying assumptions for modeling those. First, the preferences for each QoS dimension are modeled *independently* of each other. In other words, the preference function for each quality dimension captures the user's preferences for that dimension independently of other dimensions. This assumption allows capturing part of utility that comes from one dimension of quality separately. Second, for each continuous QoS dimension, we characterize two intervals: one where the user considers the quantity as good enough for his task, the other where the user considers the quantity as insufficient. *Sigmoid* functions characterize such intervals and provide a smooth interpolation between the limits of those intervals.

Sigmoids are easily approximated using just two points: the values corresponding to the knees of the curve; that is, the limits *good* of the good-enough interval, and *bad* of the insufficient interval. The case of when more-is-better (e.g., frame size) is just as easily captured as the case where less-is-better (e.g. latency) by flipping the order of the *good* and *bad* values. For discrete QoS dimensions, for instance web browsing "*content"*, whose values are "*image*"a nd "*noImage*", we simply use a discrete mapping (table) to the utility space.

4.3.2 Supplier Profiles

Typically, a supplier supports only a subset of the capability space corresponding to its various fidelities of output. In practice, approximating this subset using a discrete enumeration of points provides a reasonable solution, even if the corresponding capability space is conceptually continuous and infinite. For example, while it makes sense to discuss a video stream encoding of decimal frames per second, typically video streams are encoded at integer rates. Conversely, encoding frames in integer numbers per second provides sufficient granularity and does not sacrifice the precision of the analytical model. Despite discrete approximation, our approach does allow the handling of a rich capability space. For example, the capability space of a video player application can have 36 points, which is made possible by combining 3 frame rates, 4 frame sizes, and 3 audio qualities. Such a space can be made possible by encoding the same video in multiple frame rates, frame size, and audio quality, and leveraging application-specific features such as video smoothing.

A supplier profile is a set of vector pairs, where the first element is a Quality of Service (QoS) vector and the second element is the corresponding resource demand vector. This profile is a promise by the supplier that each capability point in the set can be supported at runtime by consuming a level of resources that does not exceed the specified demand level. Formally, a supplier profile is a partial relation between the resource space and the capability space a service.

For illustration, we show samples points from the supplier profile of "Windows Media Player" installed on author's laptop (for readability, we have placed a double-sided arrow between the QoS vector and the resource vector, and we have numbered each pair on the left side of the line):

(1) (25fps; 320x200 pixels; 64 Kbps) ←→ (230 Mhz, 441 Kbps, 17.25 MB)

(2) (25 fps; 640x400 pixels; 160 Kbps) ←→ (670 Mhz, 2060 Kbps, 21.37 MB)

The first capability-resource pair states that in order for "Windows Media Player" to play a video that has a frame rate of 25 fps, frame size of 320x200 (equals 64,000 pixels), and audio of 160 Kbps, the supplier requires 230 MHz of CPU, 441 Kbps bandwidth, and 17.25 MB of physical memory. Notice that the second capability point has increased resource requirement, because it has 4 times larger frame size and better audio.

For the purposes of recording the capability profile of suppliers, each application installation is profiled on each hardware device, creating a unique supplier instance with a distinct capability profile. For example, if two different computers both have a "Windows Media Player" application, then each installation of the application is represented by a distinct supplier profile. While this approach requires a unique profile for each application on each hardware device, it allows the model to encapsulate any differences between hardware platforms in unique supplier profiles.

We rely on methods from fidelity- and resource-aware system research to compute supplier profiles. There is ample evidence that historical data collection can be successfully used to build accurate application profiles (see, for example, [38] and [18]).

Multiple suppliers need to be combined in order to satisfy a task with multiple services. The profiles of multiple suppliers, one for each service in a task, can be combined to provide a relation between the resource space and capability space of the entire task.

We will use the verb, to configure a supplier, to refer to identifying and selecting one of the pairs from its profile. A configured supplier must provide a level of quality service specified by the QoS vector and consume a level of resources not to exceed the resource vector.

4.3.3 Resource Availability

The available level of resources in an environment are typically bounded by hardware, network, and energy resource limits. In our models, we need to know the available level of the resources, because all concurrently executing suppliers together must not consume more resources than the level available in the environment. We consider four types of resources: computing cycles (CPU), network bandwidth, memory, and battery. The available computing resource on a device is limited by the hardware specification of the device's CPU. The available level of bandwidth depends on the utilization of the network hops from the device to the source of data. The available level of physical memory is limited by hardware specification and memory usage by overhead processes, such as the operating system and any other supporting applications that are not part of the user's task. The available level of battery resources is limited by the battery capacity.

4.4 Decision Variables

The objective of the configuration problem is to optimize the match between the preferences of the user and capabilities of the environment by finding and instantiating a configuration that has the best expected utility for the user's task. A configuration has three parts: (1) a suite of configured suppliers, one for each service in the task, and expected utility of the task when suppliers are running, (2) the runtime setting for each supplier, i.e., a quality-resource pair that is selected from the profile of that supplier, and (3) resource allocation for each supplier. The optimization procedure must select the suppliers, select a runtime setting for each, and decide a resource allocation among those suppliers simultaneously, as one decision. The following three subsections describe the three parts of a configuration.

4.4.1 Task Layout: Suite of Suppliers and Expected Utility

A task layout specifies a suite of suppliers, one per service in the task, and the expected overall utility to the user. For example, the task layout for Fred's movie has the following suppliers: (1) "Windows Media Player" for the "*play video*" service, (2) "Internet Explorer" for the "*browse web*" service, and (3) "Microsoft Word" for the "*edit text*" service. The expected utility of this layout is 0.95, which is a close to perfect match for the user's preferences. This level of utility should be achieved if the conditions in the environment remain unchanged and suppliers behave as configured.

4.4.2 Runtime Settings: Selected Quality-Resource Pair from Supplier Profiles

To provide the user with the best possible match for his tasks, the configuration infrastructure needs to decide the runtime setting of each supplier in the task layout. The runtime setting for each supplier is a pair selected from that supplier's profile: a QoS vector and associated resource demand vector. The supplier is instructed to configure its runtime state so that: (1) the level of quality of service matches the QoS vector and (2) the level of resources consumed by the supplier does not exceed resource vector.

For example, a set-point for a Windows Media Player application might be the following pair first shown in subsection 4.3.2:

(1) (25fps; 320x200 pixels; 64 Kbps) ←→ (230 Mhz, 441 Kbps, 17.25 MB)

The infrastructure knows the supplier can provide the capability point, because it is selected from the supplier's profile. Recall that each capability-resource pair in the supplier's profile is a promise by the supplier to provide the specified level of capabilities in exchange for consuming the specified level of the resources.

The set-point instructs the supplier to play the video at 25 frames per second, 320x200 pixel resolution, and 128 Kpbs audio quality. The supplier must utilize no more than 230 MHz of the available CPU, 441 Kbps of bandwidth, and 17.25 MB of physical memory.

4.4.3 Resource Allocation

When deciding the quality set-point of each supplier in the task layout, the configuration infrastructure needs to ensure that the aggregate allocated amount of each resource across all the suppliers does not exceed the available level of the resource. This ensures that the resource allocation is feasible and the resource availability constraint is satisfied.

4.5 Optimization Problem

The problem structure we have defined provides us with a conceptual framework to reason about configuration. In this framework, we define an optimization problem with input variables and constraints that reflect: (1) a user's preferences for the different dimensions of the task, (2) the capabilities of supplies in terms of resource efficiency and level of capability, and (3) available resources. By solving that optimization problem, we determine a simultaneous choice of a suite of suppliers, their runtime settings, and allocation of resources among these suppliers that allow the suppliers to provide a level of capability in terms of the quality of service that matches user's preferences as best as possible.

4.6 Dynamic Behavior and Configuration Strategies

Configuration is a dynamic problem. A previously optimal configuration may no longer be optimal if any of the problem inputs change. The environment is dynamic: resource availability can change, existing applications may become unavailable, and new applications might become available. User may decide to change his preferences, or switch to a different task. Because input variables can change over time, the optimal solution found initially may not only cease to be optimal, it may also be rather far from it. For example, a decrease in the availability of one or

more resources might result in the failure of the resource allocation constraint. The infrastructure needs to make adjustments in order to continue providing an optimal experience to the user. One possible response to such a change might be to adjust the resource allocation among the currently running suite of suppliers. Another possible remedy is to replace one or more suppliers with better alternatives. Consider another example, when the user requests a new service for the current task. The resource allocation needs to be adjusted because a new supplier will need to be added to the current suite of running suppliers, adding to resource demand.

We propose two difference strategies of configuration: the reactive and the anticipatory. These strategies share the same conceptual framework. Specifically, they both structure the problem in terms of utility, capability, and resource spaces. They both define input variables that provide relationships between these spaces, and define the problem of configuration as an constrained optimization problem. However, they differ by the amount of input information available and how that input information is used to make decisions.

The reactive model is named so because it uses only the current snapshots of input variables to make configuration decisions and reacts to changes in input variables. In that model, only instantaneous utility matters, i.e., the system is concerned with optimizing user's experience for one optimization cycle. Optimal configuration is computed based on the instantaneous values of the inputs. The initial configuration is obtained using the initial values of the inputs. Dynamic behavior is achieved by monitoring the relevant inputs and computing the new optimal configuration using the most recent values of the inputs. As input variables change, a new instance of the optimization problem is constructed and solved. Switching penalties (see subsection 4.3.1) that are designed to reduce or eliminate the impact of disruptive changes give an advantage to the suppliers in the current configuration. Switching penalties also create dependence among multiple decisions made over time. Therefore, decisions in the past have an impact on future decisions. However, the reactive model does not consider that impact at all. As a result, several configuration decisions may often be less than optimal together when resources change over time.

The anticipatory configuration model leverages predictable patterns in the history of input variables, such as changes in resource availability or sequence of user task requests, and makes decisions based on predictions of such changes into the future. Even though these predictions are imperfect, we show that the anticipatory configuration strategy stands a good chance of improving over the reactive one, if the predictors are sufficiently accurate. Because the anticipatory model uses predictions and makes decisions with the benefit of information about future changes, the formal expression of its input and decision variables are different. Specifically, inputs of user's task requests and available level of resources are predictions. The decisions made by the anticipatory configuration model consider sequences of possible configuration actions into the future to select the best current configuration decisions.

4.7 Chapter Summary

In this chapter, we have defined a multi-dimension problem structure. This structure allows us to reason about configuration in terms of input and decision variables. Input variables are given or determined externally, while the decision variables are those that the system can control. In the context of this problem structure, in order to automate the configuration of the environment optimally, we need to find those values of the decision variables that result in the maximum value of the objective function, the utility of the user.

The conceptual structure of configuration problem in this chapter helps us define the concrete instances of the models in forth-coming chapters. In Chapter 5 we describe the reactive model and strategy. In Chapter 7 we describe the anticipatory model and strategy (Chapter 6 describes the models of prediction and uncertainty that are necessary to understand Chapter 7).

5 Reactive Model of Configuration

In this chapter we describe the notation of the reactive model of configuration and formulate an optimization problem according to the reactive strategy. Recall that the problem of configuration is to find a suite of suppliers, determine their runtime settings, and decide a resource allocation among them so that the expected utility of the user is nearly optimized according to the preferences of the user. In Chapter 4, we described an abstraction of the problem that organizes the relevant dimensions of the problem into 3 spaces: utility, capability, and resource. We then explained that the problem of configuration can be described in terms of inputs variables, decision variables, and an optimization equation. The three inputs of the configuration problem describe relationship between the three spaces. First, user preferences describe how points in the capability space map into the utility space. Second, supplier profiles describe a relation between the capability space and the resource space. And third, resource availability provides the constraints in the problem. Solving the problem of configuration entails finding a configuration that maximizes user's expected utility. The configuration is described using its three parts (the decision variables): the task layout, application runtime settings, and resource allocations. This chapter describes the notation of the problem spaces, problem inputs, and decision variables, and formulates an optimization problem according to a reactive strategy.

Throughout this chapter, we assume that the user has one task, and the composition of the task in terms of the services does not change. The user's task T is defined in terms of the services in the task: $SvcSet = \{s_1, s_2, \ldots, s_n\}$.

5.1 Spaces

5.1.1 Utility

We use instantaneous utility (IU) to measure user's happiness with respect to the running configuration. The range of IU is the [0, 1] interval of real numbers.

5.1.2 Capability

The capability space $\boldsymbol{C_s}$ corresponding to a service, s, is the Cartesian product of all individual QoS dimensions d of the service and an additional distinguished dimension that captures possible values of the identification tags for the suppliers of that service.

$$C_s \triangleq Supplier_s \otimes \left(\otimes_{d \in QoS\,\dim(s)}\, dom\,(d)\right)$$

For example, a *"play video"* service has 3 QoS dimensions: *"frame rate"*, *"frame size"*, and *"audio quality"*. The capability space of that service has 4 dimensions: three QoS dimensions and the distinguished dimension for the name of the supplier.

The joint capability space of two or more services is the Cartesian product of the capability spaces of the services. In particular, for two distinct services s and t, their combined capability space is formally expressed as:

$$C_{s \cup t} \triangleq C_s \otimes C_t$$

For a task *T* that has services $\{s_1, s_2, \ldots, s_n\}$, the capability space is formally defined as follows:

$$C_{\{s_1, s_2, \ldots, s_n\}} = \otimes_{i=1}^{n} C_{s_i}$$

For example, a *"browse web"* service has two QoS dimensions: *"latency"* and *"page richness"*. The capability space of Jane's task with three services (*"play video"*, *"browse web",* and "*edit text*") is an 8-dimensional space. Here is the formal notation for the capability space of the task, defined in terms of the Cartesian product of the capability spaces of the 3 services:

$T = \{s_1, s_2, s_2\}$

C_{s1} = *"play video"* ⊗ (*"frame rate"* ⊗ *"frame size"* ⊗ *"audio quality"*)

C_{s2} = *"browse web"* ⊗ (*"latency*" ⊗ *"page richness"*)

C_{s2} = *"edit text"*

$C_T = C_{s1} \otimes C_{s2} \otimes C_{s3}$

To fully describe a point in the capability space of this task, we need to specify a value for each of the 8 dimensions of its capability space. As an example, here is one point in the capability space of task T defined in the previous paragraph:

{("Windows Media Player"; 12 fps; 256,000 pixels; 128 Kbps), ("Internet Explorer"; 5s; "images"), ("Windows Word")}.

5.1.3 Resource

The resource space ***R*** is the Cartesian product of the individual resource dimensions *r* of the entire environment:

$$R \triangleq \otimes_{r \in RES \dim(E)} dom(r)$$

The number of dimensions in the resource space is equal to the number of resource instances. The number of resource dimensions depends on the environment and the available devices in the environment.

If ***r*** is a resource vector, $\boldsymbol{r} = (r_1, r_2, \ldots, r_k)$, then we will use the notation ***r[j]*** to refer to its *j*th component: $\boldsymbol{r}[j] = r_j$.

5.2 Inputs

5.2.1 User Preferences

The user specifies preferences in the context of each task. Formally, preferences are a collection of real-valued weights and functions. The weight assigned to each capability dimension indicates the importance of that dimension to the user. The function captures how the user values improvements along a particular capability dimension.

The domain of the preference functions of a service is the capability space of that service. The range of the preference functions is the utility space.

We differentiate two types of preferences: QoS preferences and supplier preferences. QoS preferences evaluate the utility of the level of QoS attributes. Supplier preferences evaluate the choice of a supplier.

QoS Preferences

The user specifies QoS preferences for each QoS attribute. The utility of service *s* as a function of the quality of service is given by:

$$U_{QoS}(s) \triangleq \sum_{d \in QoS\ \dim(s)} w_{S,d} f_{S,d}(q[d])$$

where for each QoS dimension *d* of service *s*:

- $f_{S,d} : dom(d) \rightarrow [0,1]$ is a function that indicates how much the user cares about improvements in the quality of service *d*,
- *q[d]* is the *d*th dimension of a capability point *q* from the capability space of *s*. The values, *q*, come from the capability profiles of suppliers that provide service *s*,
- $w_{S,d} \in [0,1]$ is a weight that reflects how much the user cares about QoS dimension *d*.

For example, a *"play video"* service has QoS dimensions *"frame rate"*. The user must specify a function, $f_{\text{"play video","f rame rate"}}$, that evaluates the utility of every possible frame rate and a real-valued weight, $w_{\text{"play video","frame rate"}}$, that specifies how much the user cares about frame rate relative to all other dimensions in the task.

Supplier Preferences and Switching Costs

The user specifies supplier preferences and switching costs. To evaluate the assignment of specific suppliers to a service, *s*, we employ a discrete preference function that assigns a score to a supplier based on its name. Also, we account for the *cost of switching* from one supplier to another at run time. Precisely, the utility of supplier assignment for a set $SvcSet = \{s_1, s_2, \ldots, s_n\}$ of requested services is:

$$U_{Supp}(SvcSet) \triangleq \sum_{i=1}^{n} x_{S_i} h_{S_i} + W_{S_i} F_{S_i}(Tag)$$

where for each service $s_i \in SvcSet$,

- $F_{Si} : Supp(S_i) \rightarrow [0,1]$ is a function that appraises the choice for the supplier for s_i based on the supplier identification tag, *Tag*,
- $W_{Si} \in [0,1]$ is a weight that reflects how much the user cares about the supplier assignment for that service.

The term $x_{Si}h_{Si}$ in the above formula expresses a penalty with respect to switching costs as follows: h_{Si} is a negative number and indicates the user's tolerance for a change in supplier assignment: a value close to 0 means that the user is fine with a change. The higher the absolute value of h_{Si}, the less happy the user will be. The weight, x_{Si}, indicates whether the change penalty should be considered or not. x_{Si}=1 if the supplier for *s* is being exchanged by virtue of dynamic

change in the environment, and x_{Si}=0 if the supplier is being newly added or replaced at the user's request. For example, during the initial configuration, there are no penalties because the user does not have to switch context from one supplier to another.

Calculating Overall Utility

Using both QoS preferences and supplier preferences, we can compute overall utility of a task *T* that has a requested set of services $SvcSet = \{s_1, s_2, \ldots, s_n\}$ as follows:

$$U_{overall}(SvcSet) = \sum_{i=1}^{n} \left(x_{S_i} h_{S_i} + W_{S_i} F_{S_i}(\bullet) + \sum_{\forall d \in QoS\,\dim(S_i)} w_{S_i,d} f_{S_i,d}(\bullet) \right)$$

Using the function of overall utility, any candidate suite of suppliers and any combination of runtime settings of these suppliers can be evaluated. Also, using the function of overall utility, two different suites of suppliers can be compared against each other relative to user's preference.

5.2.2 Available Suppliers and their Profiles

The third input in the reactive model is the set of available suppliers, $AvailableSuppliers = \{Supp_1, Supp_2, \ldots, Supp_p\}$, and a supplier profile for each. For each supplier *Supp*, the profile records the following information for each service *s* that the supplier provides:

- the supplier identification tag of the supplier, Tag_{Supp},
- the type of the service *s* that the supplier provides,
- a relation between the capability space of *s* and the resource space: $QoSprof_{Supp} : C_s \leftrightarrow R$.

Due to the design constraints of the supplier and runtime constraints of profiling, the suppliers profile has a finite number of points, even if the capability space of a service is infinite.

We represent the supplier profile as a set of vector pairs: *(q,rDemand)*. In each pair, the first vector, *q*, is a quality of service (QoS) vector that expresses the level of QoS attributes. For each valid QoS attribute dimension *d* of service *s*, we use the notation: *q[d]* to refer to the *d*th dimension of vector *q*.

The second vector, *rDemand*, is a resource vector that indicates the required level of resources for the supplier to provide a level of capability specified by *q*. The supplier profile is an explicit promise by the supplier to provide a quality of service specified by the capability point, *q*, while consuming no more than *rDemand* amount of resources.

To configure a supplier, we select one QoS-resource vector pair, *(q,rDemand)*, from its profile, $QoSprof_{Supp}$, and instruct the supplier to provide a level of quality, *q*, and consume no more than *rDemand* amount of the resources.

For each service *s*, we use the notation, *Supp(s)*, to refer to the subset of $\{Supp_1, Supp_2, \ldots, Supp_p\}$ that includes only the suppliers that provide service *s*.

5.2.3 Resource Availability

In the reactive model of configuration, resource availability is an instantaneous snapshot of current level of resources. Formally, the instantaneous resource availability is a vector in the resource space $\boldsymbol{R}$: $(rAvl_1, rAvl_2, .., rAvl_k)$. Resource availability is provided by monitoring mechanisms in the system.

5.3 Decision Variables

In the reactive model, the problem of configuration is to find a configuration that has nearly optimal expected overall instantaneous utility for a given task. A configuration has three parts:

- a task layout that includes a suite of configured supplier, one for each service in the task, and overall expected instantaneous utility from those suppliers,

- a quality-resource pair selected from the profile of each supplier,

- resource allocation.

These three parts are the decision variables in this model. Let's describe each in detail.

5.3.1 Task Layout: Suite of Suppliers and Expected Utility

Task layout is a suite of configured suppliers, one per service in the task, and the expected instantaneous utility when executing those suppliers concurrently. For a task T that has a set of services $SvcSet = \{s_1, s_2, \ldots, s_n\}$, a task layout is an assignment of one supplier to each service represented as a set of service-supplier pairs: $\{(s_1,a_1), (s_2, a_2), \ldots, (s_n, a_n)\}$ and their utility, u, that is in the range [0,1]. In the supplier assignment, each supplier a_i is from the set $Supp(s_i)$, i.e., supplier a_i can provide service s_i,

We use the notation: *TaskLayout(Conf)* to refer to the task layout of a configuration, *Conf*.

5.3.2 Runtime Settings: Selected Quality-Resource Pair from Supplier Profiles

Each supplier a_i in the task layout $\{(s_1,a_1), (s_2, a_2), \ldots, (s_n, a_n)\}$ is configured to according to the quality-resource pair $(q_{ai}, rDemand_{ai})$ selected from the supplier's profile, $QoSprof_{ai}$. We use the notation: $QRPair_i\ (Conf)$ to refer to the selected quality-resource pair for supplier a_i in a configuration, *Conf*.

5.3.3 Resource Allocation

A resource allocation is a set of resource vectors, one for each supplier in the task layout, as follows: $\{rAlloc_1, rAlloc_2, \ldots, rAlloc_n\}$, where each $rAlloc_i$ equals $rDemand_{ai}$. The aggregate resource allocation is the vector sum of those allocation vectors and no element of that vector may exceed the availability of the corresponding resource:

$$RAggAlloc(Conf) = \sum_{i=1}^{n} rAlloc_i \leq (rAvl_1, rAvl_2, ..., rAvl_k)$$

We use the notation: *RAggAlloc(Conf)* to refer to the vector sum of the resource allocation vectors in a configuration, *Conf*, and *RAggAlloc(Conf)[j]* to refer to the *j*th component in that vector.

5.4 Optimization Problem

Given a user task *T* with services $SvcSet = \{s_1, s_2, \ldots, s_n\}$, with the following three inputs:

Input 1: QoS preferences, $w_{Si,d}$, $f_{Si,d}$, supplier preferences, $W_{Si,d}$, $F_{Si,d}$, and supplier change penalties x_{Si}, h_{Si} for each service S_I (as described in section 5.2.1),

Input 2: The list of available suppliers: $AvailableSuppliers=\{Supp_1, Supp_2, \ldots, Supp_p\}$, and a supplier profile for each supplier, $QoSprof_{Suppi}$, (as described in section 5.2.2),

Input 3 The current level of available resources: $\boldsymbol{rAvl} = (rAvl_1, rAvl_2, .., rAvl_k)$ (as described in section 5.2.3),

Find a configuration *Conf* which is described by its 3 parts as follows:

Decision 1: A task layout of configured suppliers $\{(s_1,a_1), (s_2, a_2), \ldots, (s_n, a_n)\}$ that provides overall instantaneously utility *u*,

Decision 2: For each supplier a_i, a quality-resource pair $qrPair_{ai} = (q_{ai}, rDemand_{ai})$,

Decision 3: A resource allocation set: $\{rAlloc_1, rAlloc_2, \ldots, rAlloc_n\}$, one for each supplier, such that $rAlloc_i = rDemand_{ai}$.

Such that the overall instantaneous utility is maximized:

$$\underset{\substack{a_i \in Supp(S_i) \\ q_{a_i} \in QoSprof(a_i)}}{\arg\max} \sum_{i=1}^{n} \left(x_{S_i} h_{S_i} + W_{S_i} F_{S_i}(Tag(a_i)) + \sum_{\forall d \in QoS\dim(S_i)} w_{S_i,d} f_{S_i,d}(q_{a_i}[d]) \right)$$

Subject to the following constraints:

Constraint 1, the supplier profile constraint: for each supplier, a_i, the selected quality-resource pair $(q_{ai}, rDemand_{ai})$ must come from the profile of that supplier:

$$(q_{a_i}, rDemand_{a_i}) \in QoSprof(a_i), where: q_{a_i} = \otimes_{\forall d \in QoS\dim(S_i)} q_{a_i}[d]$$

Constraint 2, aggregate resource availability constraint: the sum of resources in the aggregate resource allocation among the suppliers can not exceed the available level of resources (the inequality must hold in each resource dimension):

$$\sum_{i=1}^{n} rDemand_{a_i} = \sum_{i=1}^{n} rAlloc_i \le (rAvl_1, rAvl_2, \ldots, rAvl_k)$$

Note that the expression of instantaneous utility requires that for each service a supplier be chosen if there is at least one supplier of that service available.

5.5 Dynamic Behavior and Reconfiguration

To model the dynamic behavior of the system under the reactive strategy, we take a simple approach. We assume that problem inputs can change over time, but that changes occur at equally spaced times. The changes in the inputs are monitored and recorded, providing the model with new values of the inputs. Then, using the new values of the inputs, a new instance of the optimization problem is formulated. This new problem instance is identical to the original one in structure, but potentially has different values for the various inputs. Because there is a running configuration, the change penalties might apply. Because the original optimization problem generally includes change penalties, the new instance of the optimization problem is no different than the initial instance.

The dynamic loop of the strategy works as follows:

- Monitor and record updated values of the inputs,
- Setup and solve a new optimization problem instance, calculating a new optimal configuration,
- Effect the changes in the environment, brining the environment to the state described by the new optimal configuration.
- Wait a pre-determined amount of time.

Waiting for a pre-determined amount of time between loop operations ensures that the model operates at a uniform frequency.

5.6 Limitations of the Reactive Configuration Strategy and Model

The reactive strategy provides a simple framework to model the dynamic configuration problem. But with simplicity comes a significant limitation. Each configuration decision in the reactive model is only locally optimal. Specifically, given the possible configuration alternatives, the reactive strategy selects one that optimizes the instantaneous utility to the user at that time. However, several instantaneously optimal decisions may often be sub-optimal when considered together. Let's consider a simple example. For brevity, let's assume that the user has a task with only one service, "*play Video*", and there is only one resource, network bandwidth. The task has a duration of three time units, starting at time 0, and lasting till time 3. The environment has three suppliers that can provide a *"play Video"* service: A, B, and C. The available bandwidth can be in one of three possible states: High, Medium, Low (and for the purpose of this example, codifying the range is sufficient). For each of the three suppliers and for each resource state, we calculate the maximum possible instantaneous utility when that supplier is chosen to supply the task's *"play video"* service. Because the task has only one service, this utility number is the task's overall utility, excluding any penalties that might incur when switching suppliers. For simplicity, we assume that switching suppliers incurs a penalty of -0.2.

Table 3: Maximum utility possible per supplier in each resource state. Rows entries are resource states, and column entries are suppliers.

		Suppliers		
		A	B	C
Resource Availability	High	**1**	0.95	0.7
	Medium	0.8	**0.85**	0.6
	Low	0.3	0.5	**0.55**

Let's assume that resource availability is High at time 0, drops to Low at time 1, and then increases to Medium at time 2. The user starts the task at time 0 and suspends at the end of time 2, lasting a total of 3 time periods. The reactive strategy of configuration will select supplier A at time 0, because it provides the best instantaneous utility when resource availability is High. At time 1, the reactive strategy will switch to supplier C, because C provides an instantaneous utility of 0.35 (0.55 less penalty of .2) when resource is Low, while continuing with supplier A only gives 0.3 utility. At time 2, when resource availability changes to Medium, the reactive strategy will select B which provides instantaneous utility of 0.65 (0.85 less penalty of 0.2). So the accrued utility over the duration of the task is: 1.0 + 0.35 + 0.65 = 2.0.

A system that had information about the resource changes in hindsight could have done better. Indeed, selecting supplier B when the task is resumed at time 0, and then continuing with B at times 1 and 2 would result in accrued utility of 2.3 = 0.95 + 0.85 + 0.5.

This example demonstrates why a reactive strategy of configuration that optimizes instantaneous utility may often be sub-optimal a over long of time. This deficiency is a result of two properties of reactive configuration: (1) information used for decision making does not extend into future, and (2) the planning horizon of the strategy is short and does not consider the effect of current decisions on future utility.

5.7 Chapter Summary

In this chapter we have provided the formal definition of a configuration model and strategy. This model has the following distinguishing characteristics:

- It has a short decision-making horizon,
- It uses the instantaneous utility of the task as the objective function,
- It requires and uses only the most recent values of the available level of the resources.

Based on those characteristics, the model has a number of advantages. The calculations required to solve for optimal utility are relatively fast. The uncertainty in the future values of resource

availability and changes to user task can be ignored, thereby reducing the complexity of the inputs needed as well as the calculations required.

The model also has a significant disadvantage. Because the decision making horizon is short, the utility of the configuration to the user is guaranteed to be optimal only over short periods of time. Over longer periods of time, the accrued utility of multiple decisions is likely to be less than optimal. Also, the model can not reason about the availability and optimal allocation of resources such as battery, because it has no temporal dimension.

6 Prediction and Uncertainty

Predicting resources and leveraging these predictions for resource management has gained increasing mindshare in pervasive computing research in the past decade. Satyanarayanan's paper [51] on the challenges in the field identified prediction as an important desired feature of systems that aim to provide a seamless user experience. One of the scenarios described in the paper suggests how good predictions of resource availability can help a system make decisions that greatly improve user's experience.

> *Jane, a business traveler, is at an airport waiting for her connection flight. Having edited several large documents, she would like to e-mail them. The wireless bandwidth in her current location is rather low. Aura, her intelligent assistant, calculates that completing all e-mail uploads at current bandwidth will cause her to miss her flight. Next, Aura consults the airport's network weather service for better bandwidth locations. Aura advises Jane to walk to a nearby Gate where current level of bandwidth is much higher, because the number of users connected to the network is much smaller. Aura also knows that the number of connected users is not expected to increase because there are no arriving and departing flights in the near future. Aura floats a dialog box on the laptop suggesting that Jane take a short walk to the identified location and asks Jane to prioritize her e-mail. Jane follows Aura's suggestions, and is eventually able to board her flight just on time for departure.*

In this scenario, predictions of available resources help ensure that the user's tasks are completed on time for a deadline (plane's departure time). Furthermore, the preferences of the user are taken into account when deciding how to best complete the task, sending higher priority documents first.

But what are the challenges in building a system that can provide and use such predictions?

First, predictions describe future events and the uncertainty around those events, e.g., the available level of bandwidth in the near future at user's current and nearby locations. Future events and future outcomes are uncertain; consequently, predictions of future events are imperfect. Such predictions must carefully describe the uncertainty of future outcomes. Furthermore, a predictive system needs to provide appropriate communication and coordination mechanisms to request and use imperfect predictions for important system decisions.

Second, predictions can come from multiple sources. In the Jane scenario, Aura uses predictions from recent history, flight schedule, and possibly historical observations. The level of available bandwidth and the number of users connected to the network in the immediate past are used by Aura to predict bandwidth in the immediate future at multiple access points. Furthermore, the flight schedule is used to infer that there will be no arriving or departing flights in the following 30 minutes. Aura could potentially use the on-time statistics of the flights to fine-tune that prediction further. As a result of combining predictions from these sources, Aura is able to conclude that bandwidth will continue to be good at a particular access point for the amount of time needed to upload the documents. As the example demonstrates, a well-designed resource management

system must be able to uses multiple sources of prediction and must be able to combine those predictions in a principled manner.

In this thesis, we are interested in predictions about resource availability and user tasks. For the purpose of this thesis, a prediction is an informed estimation of future random values of variables, e.g., the available level of a resource in 10 seconds or the duration of a user's task. This estimation describes the statistical properties of the variable in the future, e.g., its expected value and the standard deviation. In the first three sections of this chapter (6.1, 6.2, and 6.3) we discuss issues that affect how predictions are obtained, communicated, and used in a pervasive computing system. We use this discussion as a starting point for a more thorough technical treatment of predictions in the following sections (6.4, 6.5, 6.6, and 6.7).

6.1 Sources of Predictive Information

In this section we describe sources of predictive information. We organize the discussion around each of the three input variables in the analytical model of configuration: user's tasks, application profiles, and resource availability.

6.1.1 Sources of Information for Predicting Users Tasks

We are interested in predicting when the user will start his tasks and how long they will run. Such information can help marshal the available resources in the best possible manner, as well as plan the allocation of resources over time.

We start with the reasonable assumption that the set of all possible computing tasks of a user is known, e.g., by eliciting from the user (see [52]). Each task in the list of known tasks can be in one of two states: *active* or *pending* (see section 3.1). When the user creates the task, it is initially in a pending state. The user resumes the task, making it active. When the user is done working on the task, he suspends the task, making it pending again. Because tasks consume resources only when active, our challenge is to predict the future times when the user resumes and suspends his tasks.

Information about when the user may resume and suspend his tasks can come from a variety of sources:

- calendar and schedule,
- traces of task activation history,
- user's own prediction of future resume and suspend time,
- statistical analysis combining the any of the above three sources,

Users typically follow a daily or weekly calendar, and many of the user's tasks conform to a schedule according to the calendar. For example, certain tasks might be resumed and suspended at specific times on specific days of the week with very high probability. This information might be inferred from the calendar of the user, helping provide predictions for the future. Historical traces of resume and suspend occurrences can be used to characterize the significance of such predictions.

Eliciting task information directly from the user might provide information not found in other predictive sources, e.g., in the calendars or in the historical traces. But hints and other predictive information elicited from the user might contain estimation bias. E.g., a user might repeatedly underestimate or overestimate the duration of a task. By comparing the user input with actual data observed in the aftermath of task execution, we can identify the user's bias and characterize its statistical properties.

6.1.2 Sources of Information for Predicting Resource Demand by Applications

We are interested in predicting the resource demand of applications that are resource and fidelity-aware. For the purpose of prediction, we are interested in describing the space of possible fidelity levels of an application as well as the resource demand for each fidelity level.

Prior research (e.g., [38]) has demonstrated that the resource demand of applications can be predicted with good accuracy using historical traces of application execution. An application is instrumented to execute at various levels of fidelity. At the same time, the resource utilization of the application is measured and logged. In this manner, a correlation between the fidelity level of the application and resource usage is obtained, which can be used to predict the resource demand of the application in the future. This correlation is used to generate the QoS vs. profile relationship of the supplier.

6.1.3 Sources of Information for Predicting Resource Availability

We are interested in predicting the available level of resources such as network bandwidth, battery energy, and CPU cycles. As the Jane example demonstrated, such predictions can be successfully used to improve the experience of the user, e.g. by completing tasks on time and by providing adequate quality of service.

The resources under consideration are different in a number of ways. For the purpose of prediction, it is important to distinguish whether the resource is shared among many users or not. For example, bandwidth along a network link is typically shared by many users using that link. Predictions of the available level of bandwidth can potentially leverage this property. Battery and CPU on a particular device are not shard resources. Therefore, the available level of battery or the available level of computing cycles of a hardware device is a function of the usage pattern of that user only.

We are interested in sources of predictive information for four resources: CPU, bandwidth, physical memory, and battery. Such predictions may come from a number of sources:

- historical traces, e.g. traces of used and available levels of a resource,
- calendar and schedule,
- specifications of hardware and network.

Historical traces in turn can be separated into two broad groups: recent history and ancient history. Recent history can be a good indication of near future. For example, by proving the available level of bandwidth, e.g., by downloading some content, we can use the level of probed bandwidth as a prediction for the level of bandwidth in the near future.

Ancient history of resource availability can provide clues to periodic, repeating patterns. For example, if historical resource availability has repeating daily or weekly patterns, then such patterns can be used to make predictions of future availability. In the Jane scenario, Aura consulted the flight arrival schedule and found out that there are no arriving or departing flights in a particular part of the airport for a desired time horizon. Lack of arriving or departing flights implies that there will not be an influx of additional network users in the near future. Then Aura was able to conclude that the recent history of bandwidth will continue to be a good predictor of future bandwidth.

6.2 Uncertainty in Prediction

Predictions of future events are not perfect, because future is uncertain. Consider the task of predicting tomorrow's average temperature. Weather predictions are rarely perfect. However, even imperfect predictions can be good enough to be usable. In order for such a prediction to be usable, it is important to know how often the actual weather falls within the predicted bounds or how far the actual temperature might differ from the predicted value. In other words, it is important to know the uncertainty around the weather prediction. Similarly, a system that makes predictions of resource availability must describe the uncertainty in those predictions. The system must also cope with uncertainty when making decisions based on imperfect predictions.

In the ubiquitous computing literature, two types of uncertainty are discussed: (1) uncertainty in the measurement or estimation of an already observed event and (2) uncertainty in the predicted value of future random events. Let's further discuss these two types of uncertainty.

6.2.1 Two Types of Uncertainty

Measurement uncertainty commonly occurs in the physical sciences, e.g., when a physical phenomenon is sensed, measured or estimated. Such estimation is often noisy due to the noise in the input data or noise in the measurement itself. Examples include the voltage across two poles, the speed of a moving object, or the contextual state of a computer user. Even when measurements of the physical world are not exact, it is possible to obtain the statistical profile of the errors by sampling, averaging, and other techniques. Typically, the error of the measurement is described by a probability distribution function or by confidence intervals. The measurement error is uncertainty of the first type.

Future uncertainty arises when predicting future events. Consider the problem of forecasting the weather again. Until tomorrow arrives, tomorrow's weather is uncertain. Any forecast of the weather is likely to be imperfect. However, we can measure the error in the forecast by comparing the forecast with the actual temperature. We can then characterize the forecast error, e.g., its statistical properties such as the mean or its entire probability distribution. The error in the prediction of future events, i.e., the difference between the predicted and the realized values of the event, is uncertainty of the second type.

6.2.2 Uncertainty When Predicting a Sequence of Future Values of a Variable

In this work, our challenge lies in describing and coping with the uncertainty of the second type. We are interested in predictions of future events such as the start time of a particular user task, or the level of available bandwidth in 30 seconds.

We are predicting events that have a temporal dimension. Not only are we interested in predicting the bandwidth in the next 30 seconds, but also in the following 10 minutes, perhaps in 30 second intervals. In other words, we are interested in predicting the sequence of future values of a variable. Again, taking an analogy from weather forecasting, imagine the task of predicting the weather for the next several days, for each day. Many weather prediction services provide such a capability, allowing predictions for the weekend, for the next week, next 10 days, etc.

We need a solution for characterizing the uncertainty when predicting the sequence of future values a variable, e.g., the available level of bandwidth for next several minutes in 30 second intervals or the daily temperature for the following 7 days.

Intuitively, predicting further into the future should result in a larger prediction error, because there is more uncertainty. The challenge is in determining how the magnitude and shape of uncertainty changes over time as we predict further into the future. In order to characterize the uncertainty when predicting a sequence of values, we need a prediction model with such a capability. Ideally, the model can express the uncertainty in terms of a sequence of probability distributions as a function of the prediction horizon.

6.2.3 Predicting Likely Paths of Future Values of Variable

When making a prediction for a sequence of future values of a variable, there are two alternative ways to express such prediction. The first approach expresses predictions of future values conditional on all past observed values of the variable. In this manner, the prediction expresses possible future sequences of the variable and their probabilities. We call this approach path-dependent prediction. The second approach expresses the probability distribution of the level of resource at each future time independently of the variables realized path. We call this approach path-independent prediction

To show the difference between these two approaches, let's consider a simple experiment of tossing coins. We toss a fair coin 3 times in a row, one coin toss per second. We record the result of the coin toss and keep an accumulator variable: if the coin turns head, we increment the accumulator by one, and if is a tail, we decrement the accumulator by one. The accumulator starts at zero.

We are interested in the predictive distribution of the accumulator variable after 0, 1, 2, and 3 seconds, because we are interested in making decisions in response to the different values of the accumulator variable. We demonstrate how the two approaches can represent that distribution.

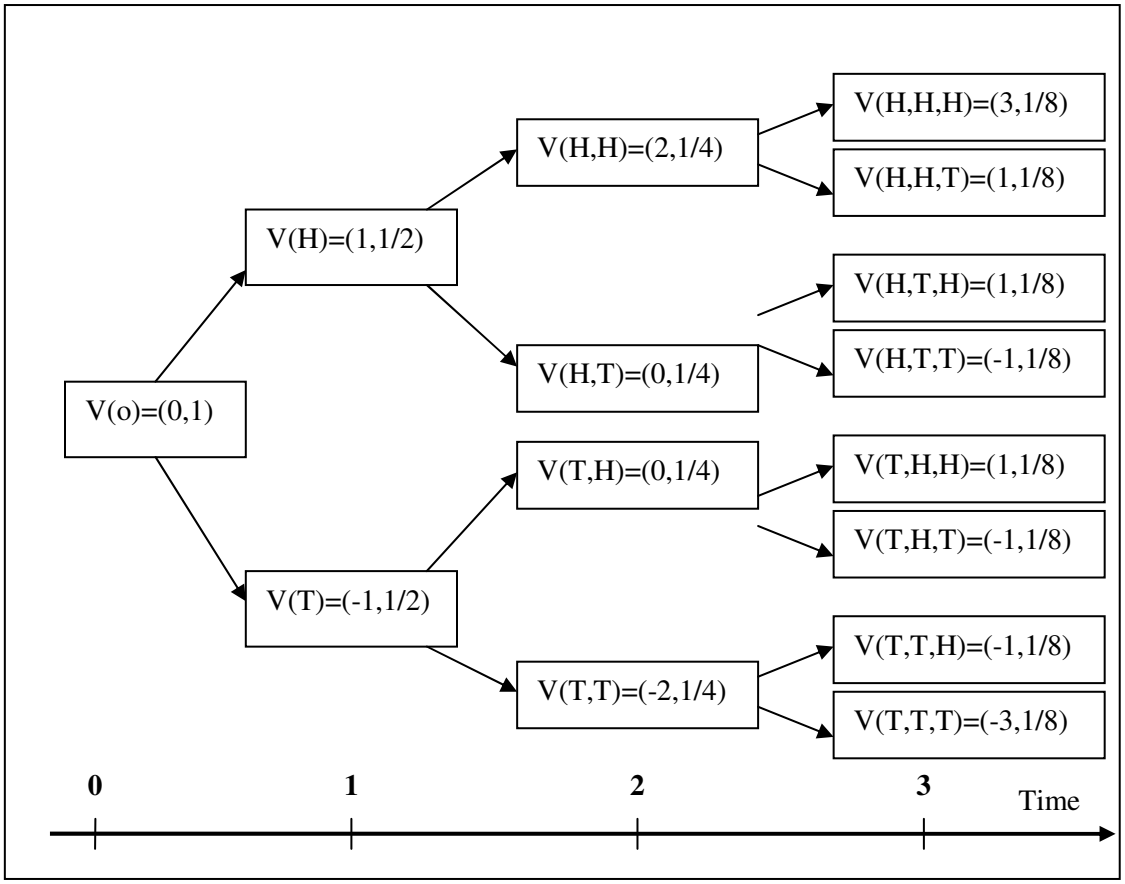


Figure 1: This figure shows a representation of the coin toss sequence that captures path-dependent prediction information. Each node in the figure records the path of a sequence of coin outcomes, the probability of reaching that node, and the value of the accumulator variable once that node is reached. For example, the node V(H,H) = (2,1/4) states that the probability of getting the coin toss sequence H,H is ¼ and the value of the accumulator at that node is 2.

Path-dependent prediction approach records likely future paths of the variable and their probabilities in a binary tree. Each node in the tree records the path of coin outcomes up to that node by the virtue of recording the parent node. Each node also records the value of the accumulator and the probability of that node. For example, the root node records the initial value of the accumulator, 0, as well as the initial probability, 1. At depth *d*, the tree has 2^d nodes and captures the possible values of the accumulator after *d* tosses. Each node of the tree can be described by the coin toss sequence that it took to get to that note; the value of the accumulator at each node is a function of that sequence. By following the edges of the tree to a particular node, we have an explicit record of the path that accumulator took to reach that node. The tree is shown in Figure 1.

Notice that multiple nodes at a particular depth can have the same accumulator value. For example, the value of the accumulator at (H,T) is the same as that at (T,H). However, these nodes are represented separately, because they record different coin toss paths.

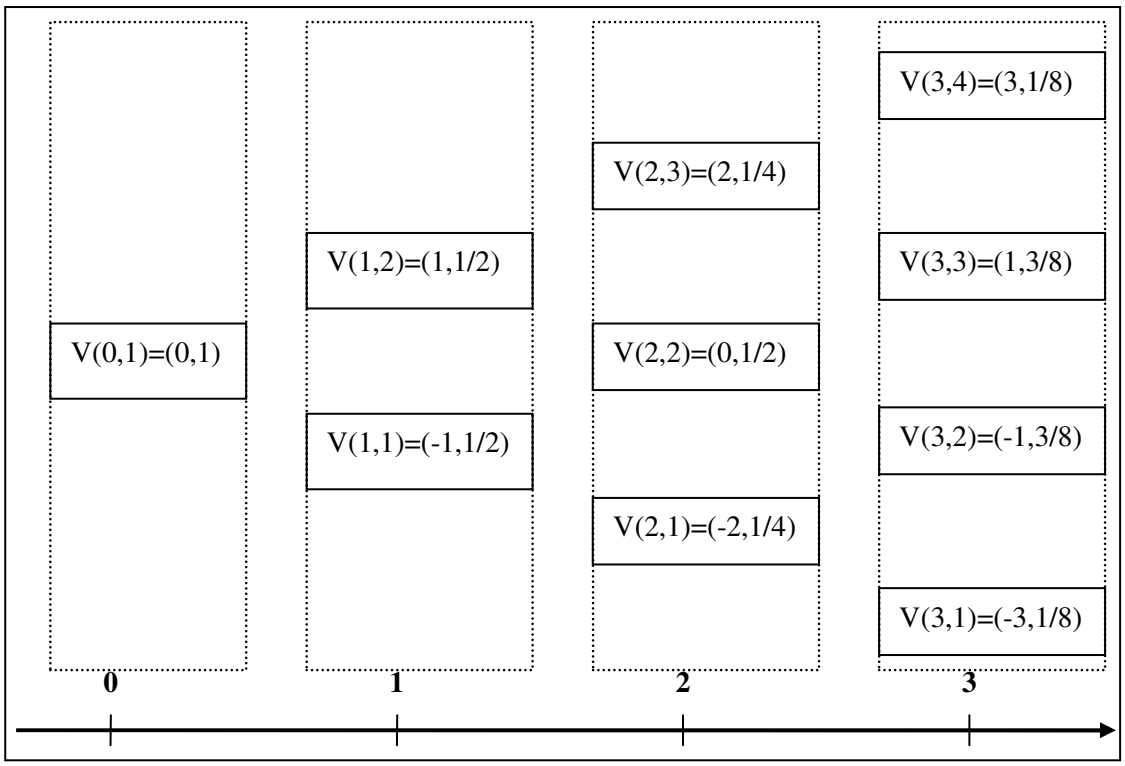


Figure 2: A representation of the accumulator values and probabilities. This representation expresses the predictive distribution of the accumulator variable over time, but records no path information. Dashed rectangles group together nodes that comprise a probability mass function. Each node records time and outcome index, e.g., V(1,1) and V(1,2) are the two possible outcomes at time 1. Lack of arrows between nodes emphasizes that there is no path-dependence in the distributions, i.e. the distributions are conditioned only on the knowledge that at time zero the value of the accumulator is zero.

The second approach, shown in Figure 2, does not record path information. Instead, the distribution of the accumulator variable at each time step is represented independently of the coin toss history. The representation in this case is a sequence of a set of related nodes, where each node is a value-probability pair. Each set of related nodes comprises the distribution (the probability mass function) of the accumulator at a specific time. For example, we know that the probability of node V(2,2) is ½, but we don't know how node V(2,2) was obtained (unless we are told about the underlying process of node generation, i.e. sequence of fair coin tosses).

To see the connection between this example and predicting resource availability, consider the way that uncertainty is represented when making predictions of a sequence of future events. Coin tosses are future events and have uncertainty. Predicting the value of the accumulator is akin to predicting the level of the resources. To predict the value of the accumulator variable, we need to describe its probability distribution. This distribution in turn describes the uncertainty in the prediction of the accumulator variable. The value of the accumulator is analogous to the level of available resource. When making predictions of resource availability, we have a choice of two models to describe the accumulation of uncertainty in predicting a sequence of resource availability values.

Notice that both representations demonstrate an important point: predictions of the accumulator further into the future have more uncertainty, as evidenced by the increasing range of the possible values of the accumulator. This is an important consideration when designing systems that use such predictions. Also, both representations capture the distribution of the accumulator variable at future times. The first representation is more general, and records sufficient information to infer the information described by the second. However, the reverse is not true.

The key point of this section is to emphasize the important difference between the two representations. The first representation records important information that the second does not: the path it took for the accumulator variable to reach each node in the tree. The second representation does not record this information. There is no way to infer, without guessing, how the nodes at time 3 could be reached from the nodes at time 2. Thus, it is important to record predictions of paths, because that information can be used for analyses when making system decisions. And this is precisely the message of this section.

6.3 Operational Considerations for a Prediction Interface

A predictive system needs to compute and communicate predictions at runtime. There are operational challenges associated with providing these functionalities. Specifically, the provider and the consumer of predictions need to coordinate on the syntax and semantics of the predictions. The two parties need to agree on a small number of important parameters that can help fully and adequately describe predictions. These parameters are:

- Prediction horizon,
- Prediction scale (or granularity),
- Prediction detail.

Prediction horizon is the future interval over which predictions are desired. For example, if we are interested in a 7-day forecast of weather, then the prediction horizon is 7 days. It is important that the provider of the predictions and the consumer of predictions agree on the value of this parameter in order to ensure that the predictions are useful to the consumer. Indeed, if a prediction consumer needs a prediction of the bandwidth for the next 10 minutes, then providing a prediction for the next 5 seconds will not be useful to the consumer at all.

Prediction scale or granularity is the time window of a single prediction in a sequence of predictions. A traveler on a week-long business trip might need a daily weather forecast with a 7-day horizon, while a mountain hiker might need an hourly forecast with a 36 hour horizon. In the first case, the scale of prediction is 24 hours, while in the second case it is 1 hour. A prediction at a daily scale will not be acceptable for the hiker.

Now consider an example from network bandwidth prediction. Imagine a system that needs predictions of bandwidth at the scale of 1 second, but the only available predictor can provide predictions at the scale of 1 minute. While a minute-scale prediction might provide an adequate prediction of 1-second bandwidth over a 1 minute period, it is entirely useless for the system. The system might be sensitive to fluctuations of bandwidth at the second scale, and might not be able to use the minute-scale predictions. In this case, the requested scale is much smaller than the provided scale, resulting in a mismatch. A mismatch might also occur when the opposite is true,

i.e., the requested scale is larger than the provided scale. Predictions at a smaller scale might have a larger prediction error than those at a larger scale, even when normalizing the scales [49]. As a result, providing a prediction at a smaller scale than desired will not be as useful because it will be less precise.

Prediction detail refers to the amount of information used to describe the probability distributions of the predictive uncertainty at runtime. On paper, predictors can be described analytically, e.g. using mathematical functions. At runtime, it may be more appropriate to use discrete data structure representations to express and communicate predictions. When converting predictions expresses using continuous functions into a discrete representation, the system providing predictions needs to make approximations, resulting in loss of detail. It is important for the prediction communication mechanism to define parameters that control the detail of approximation so that the consumers and the providers of predictions can synchronize.

Supporting these three parameters in a prediction API helps narrow the space of useful predictions from the perspective of the consumer. It is likely that the prediction provider can only support a limited range of these parameters; therefore, the API must allow for a negotiation between the consumer and the provider to agree on the acceptable values of these parameters. This is an important design consideration for the prediction API. In section 8.4, we will describe the design of the API, and demonstrate how it addresses the operational issues brought forth in this section.

6.4 Discrete Time Model

We adopt a discrete time model, in which time is divided into windows of equal length. The length of the window is set by the prediction scale (section 6.3) and determines the frequency at which the system makes configuration decisions. The length of that window, and consequently, the prediction scale, is fixed by design. We assume that changes to the input variables occur at the beginning of each window. Configuration decisions and corresponding changes are also made at beginning of a time window. Although in practice changes in the input variables might occur at any time, for the purposes of optimization and configuration decisions, we can approximate the changes to occur exactly at interval boundaries.

Throughout this section, as we express the description of the model in detail, we will use variables t, s, and DH as follows: t denotes current time, s denotes some unspecified future time, and DH denotes the decision-making horizon. Predictions beyond $t + DH$ are not necessary because they are not included in the calculations of the next immediate configuration decision.

6.5 Predicting User's Tasks

In this section, we describe a model for predicting user tasks. Recall, from section 3.1, the vocabulary of task operations: *create*, *resume*, *suspend*, and *complete*. In this section, we are concerned with predictions of the *resume* and *suspend* times of the tasks, because they have consequences on resource demand and require configuration changes.

6.5.1 A Motivating Example

Let's look an example to help motivate the model. A university professor teaches a course that meets every Monday and Wednesday from 10:30 till 11:30 am in a particular classroom. For the

course, the teacher has defined a task that has three services: *"show slides"*, *"play video"*, and *"browse web"*. This task is activated around 10:30 and is deactivated around 11:30, although the times are not exact. For this task, it is reasonable to assume that the the predictions of resume and suspend times are known with great accuracy. Also, it is reasonable to assume that the predictions don't change much as the actual start and end times approach.

6.5.2 Building Blocks of the Task Prediction Model

As the example demonstrates, we are interested in predictions of three variables: the resume time, the suspend time, and the duration, of each instantiation of a user's tasks. In an ideal world with a prediction oracle, suspend and resume times would be known exactly, and the duration could be calculated by taking the difference between suspend and resume times. In such an ideal world, knowing any two of those variables can help to calculate the third.

Even if the predictions are not exact, we can use the relationship between those three variables in the model. In fact, once the task is active, then the duration of the task is exactly equal to the suspend time minus the resume time. If a probability distribution of one of those variables were available, we could calculate the probability distribution of the other one.

We are interested in a simple model for expressing the probability distribution of the resume time, suspend time, and the duration of the task for each instantiation.

A good approximation to the probability distribution of a predicted variable is a pair of numbers; the mean and the variance. The variance of a prediction is an important indicator of how tight the realized value of the variable will fall around the predicted mean value. Small variance, e.g., on the order of several dozen seconds or single-digit minutes, indicates that the prediction is tight. Otherwise, we have a large variance.

In our thesis, we build a prediction model that differentiates the two cases: (a) small prediction variance and (b) large prediction variance.

An appropriate distribution to capture a small-variance prediction is either a one-tail or a two-tail normal (Gaussian) density function. A one-tail density function may be appropriate when modeling a deadline. In that case, the task must end by a specified time. A two-tail density function may be appropriate when modeling a scheduled end time, allowing for both early and late completions with similar likelihood.

To capture a prediction with a large variance, we use a probability mass function. A probability mass function has two advantages: (1) it can approximate any distribution of a random variable without relying on a closed-form analytical formula, and (2) it can be communicated easily between the provider of predictions and the consumer of predictions, without committing to a particular family of probability distributions.

For the remainder of this section, we consider all possible cases of predictions involving any two of the three variables: resume time, suspend time, duration, and any two of the prediction cases: small variance or not. These cases can be grouped into the following 3 main groups:

- two or three of the three variables can be predicted with a small variance,

- only one of the three variables can be predicted with a small variance,

- none of the three variables can be predicted with a small variance.

6.5.2.1 Case 1: two or three variables can be predicted with a small variance

When at least 2 of the 3 variables can be predicted with small variance, we have 3 combinatorial choices: (1) resume and suspend times, (2) resume time and duration, and (3) duration and suspend time. In the first case, we have a task with *a scheduled start* and *a schedule end* times, e.g. the task of providing the lecture. In the second case, we have a task with a *scheduled start* and a *fixed duration*. Notice that even though the start time of the task is fixed, there is still some small probability that the task starts late. In that case, fixed duration implies that the task will also end late. Compare this to the previous case, when a late start will not automatically result in a late end time, because the end time is also fixed by schedule. In the third case, we have a task with *a scheduled end* time and a *fixed duration*. Think of this task as having a deadline. Finishing well before or well after the deadline is not likely, and the duration is relatively fixed.

All of the choices under this case can be modeled similarly. The predictions of the variables can be expressed using either one-tailed or two-tailed normal density function with a small variance.

6.5.2.2 Case 2: only one of the three variables can be predicted with a small variance

This case describes situations when only one variable can be predicted with small variance. Again, we have three possible choices: (1) resume time, (2) suspend time, and (3) duration. In the first case, the resume time is scheduled, but the predictive information about the duration or the suspend time of the task is less. It is helpful to think of such tasks as completion-driven or quality-driven. Imagine a task that needs to be started at a specific time, but until some amount of progress is made or some quality is achieved, the task can not finish.

The choices that fall under this case can also be modeled similarly. The variable with a tight prediction can be modeled using either a one-tailed or a two-tailed normal density function.

6.5.2.3 Case 3: none of the three variables can be predicted with a small variance

In this case, none of the three variables about the task has tight predictions. We can capture the prediction of two of the three variables using a generic probability mass function.

6.5.3 Task Request Prediction Model

For each activation instance of a task, we predict three variables: resume time, duration, and suspend time. Before the resume event has occurred, the has predictions for two of the three variables (the third variable does not require a separate prediction, because of the relationship between the three variables). Once the task has been resumed, either the duration or the suspend time is predicted, but not both.

A prediction is a triple:

$$\{tight, pf, probability\},$$

where *tight* is a Boolean variable (T = true, and F = false), *pf* is a probability function, and *probability* is the aggregate probability of the event occurring.

When a prediction is known to be tight, i.e., *tight* = true, then we use a one-tail or a two-tail normal density function. Specifically, *pf* can be described using the following 4 parameters:

{mean, stdev, left, right},

where *mean* and *stdev* are variables that express time; *left* and *right* are Boolean variables. When expressing a prediction for the resume or suspend times, *mean* is in absolute, or calendar time; when expressing a prediction for duration, *mean* shows length of time. *Stdev* always expresses length of time. *Left* and *right* Boolean variables respectively indicate if the density function has left, right tails. The density function may have one or both tails.

When a prediction is not tight, then *pf* expresses an arbitrary probability mass function as an enumeration of *(value, probability)* tuples. For suspend and resume times, the values express absolute time. For duration, the values express time lengths.

Here is a prediction for the next resume time of the lecture task (described in section 6.5.1). This prediction is made at 10:12 am of the same day:

Resume: *{tight=T, {mean=Dec 12, '07, 10:32:30am EST, stdev=120 s, left=T,right=T}, 1.0},*

Suspend: *{tight=T, {mean=Dec 12, '07, 11:30:00am EST, stdev=120 s, left=T,right=T}, 1.0}.*

This is a scheduled task and has tightly predicted resume and suspend times. The predicted variance around the mean for both resume and suspend times is 120 seconds, and the variance is two-tailed, i.e., the task might resume earlier or later than the mean predicted time. Notice that the mean predicted time for resume is 10:32:30 am, indicating that the task starts later than the scheduled time of the lecture. This might be due to a number of reasons, e.g., the professor needs some time to start the task once the lecture has started, or the students tend to come in a little later than the start time of the lecture, thus requiring the professor to resume the task only after sufficiently many students are present in the classroom.

6.5.4 Information Arrival

In order to allow for the arrival of information into the system, we require that predictions be updated periodically. In this way, later predictions will be more accurate based on the arrival of new information. For example, once a task has been resumed, we need only a prediction for either duration or suspend time. As another example, as a task gets closer to its completion time, the prediction for duration should get better, providing a better estimate for the suspend time.

6.6 Predicting Application and Hardware Resource Demand

The approach to predicting application and hardware demand is based on historical profiling. In his thesis [37], Narayanan describes multiple approaches to application resource demand prediction. The goal of such prediction is to estimate the resource demand for each fidelity level of an application. When the space of fidelity levels of an application is discrete and sparse, then *binning* is used. For each fidelity level, an application's resource demand is logged, and the entire log is subsequently used as a lookup table to compute future demand. When the space of possible fidelity levels is large, then a complete sampling of this space is not possible, and Narayanan shows how partial sampling can be combined with statistical inference to calculate demand

across the entire space of fidelity levels of an application. Both of these techniques rely on the premise that the observed history of the applications execution profile is a sufficiently good predictor of applications future resource demand.

Narayanan demonstrates the viability of application resource prediction using a large spectrum of mobile-interactive applications. We rely on his results to perform application demand prediction for the purpose of this thesis. In Narayanan's work, the time scale of prediction, execution and control is on the order of hundreds of milliseconds. In our work, the time scale is on the order of dozens of seconds. However, we observe that the general techniques of historical profiling apply equally well. Also, due to averaging over longer periods of time, the predictions of demand are likely to be more accurate.

In addition to predicting the resource demand of applications, we are also interested in the resource usage of hardware devices. First, devices that have batteries have a limited supply of energy, and accurate prediction of their baseline energy usage must be carefully modeled. And second, power management options in many hardware devices such as laptops introduce dependencies among resources. For example, voltage-scaling of power-aware CPUs enables a power-efficient mode of operation, but slows down the effective computational power of the CPU. As another example, lower screen brightness consumes less battery, but might be undesirable for a user.

In his thesis [17], Flinn demonstrated an approach to measuring the power consumption of hardware devices such as laptops and handheld computers. First, his approach characterized the battery drain due to various hardware components in different modes. For example, Flinn's approach measures the power consumption of the display at each level of display brightness. Next, he demonstrated how to measure the power consumption due to a single fidelity operation of an application. Flinn then developed a model, which separates the power drain of a mobile computer into baseline consumption and application-specific consumption. The level of baseline consumption is the sum of the baseline consumption by the different devices and their power settings. The application-specific consumption measures the incremental battery drain that is due to the application. That incremental level of battery consumption corresponds to the level of resource usage that is needed for the supplier profile.

6.7 Predicting Resource Availability

In this section we motivate and construct the model of resource availability prediction. Section 6.7.1 contains a rudimentary introduction to time series. In section 6.7.2, we define the formal expression of the three predictor predictors that are used to make different type of predictions. In section 6.7.3, we describe how different predictors can be combined.

6.7.1 Time Series Background

Time series analysis is a commonly used statistical tool for predictions in general, and for resource availability predictions in particular. Appendix A presents an overview of time series analysis. Here is a brief recipe of time series analysis:

- Collection: collect historical series of the variable; partition the series into two halves,

- Analysis: using one half of the series, identify one or more predictive models and calculated the best-fit parameters of each model. Typically, the parameters are chosen to maximizes a measure of predictability,

- Testing: perform some testing of the model and parameters using the second half of the series. Identify and select the model(s) that perform best,

- Prediction: using the selected model, perform predictions using new data.

The first step is performed by monitoring and accumulating the values of the variable that's being predicted. The next two steps are often performed offline by a human versed in statistics (although some researchers have claimed that those steps can be performed online without any human supervision). The fourth step is done at runtime and must execute at a near real-time speed to ensure that predictions of values are available well before these values are observed.

The analysis step requires selecting a time series model and calculating the values of its parameters that provide the best fit to the data. Several research studies (see [49]) have shown that: (1) resources such as network bandwidth and server CPU load often have good predictability, and (2) relatively inexpensive time series models with autoregressive (AR) and moving average (MA) components predict do as well as more complex and computationally costly models. Based on those findings, we have decided to build our model of uncertainty around an autoregressive-moving average (ARMA) time series model.

6.7.2 Resource Predictor Types

In this section, we introduce and define three predictor types: known pattern, recent history linear, and bounding. We motivate the need for each predictor using examples. For each predictor type, we provide the formal expression of the prediction output. The goal of this section is to build a small battery of predictor types which can jointly express predictions from different sources.

6.7.2.1 Known Pattern Predictor

Identifying a known pattern component is challenging and might require domain-specific knowledge. Instead of developing a theory on how known patterns can be detected and removed, we solve a different problem that is still suitable for our purpose: (1) we illustrate examples where the use of a known pattern component is likely to improve predictability and (2) we allow the possibility of one or more known pattern predictors in the framework should resource availability analysis discover a statistically significant known pattern for any resource.

Example 1: Daily observations at a coffee shop in the corner of a busy intersection reveal some patterns. The students from the nearby university flood the shop during the intermissions, while the professionals working in the area visit the shop early in the morning, during lunchtime, and later in the evening. Based on the computer usage of different groups, the wireless bandwidth availability at the coffee shop may show repeatable patterns.

Example 2: A university department enforces a strict policy that prohibits students from surfing the web and downloading music during classes. Using class schedule, enrollment information and a map of wireless access points on campus, it might be extract known patterns in the availability of the wireless bandwidth.

Example 3: Late mornings and early afternoons on Mondays are a known period when professional employees check their online account balances from work. This pattern creates significant traffic and resource utilization spikes for the networks and resources of the employer as well as the companies and ISPs that provide and host those accounts.

Example 4: A typical laptop computer has several scheduled background tasks (backup, virus scan, spy-ware check, other self-diagnostic activity) that run daily, weekly, etc. Often the execution of these administrative tasks can not be interrupted or suspended by the user, but the time and length of execution is known exactly in advance. The resource utilization imposed by such tasks certainly has known behavior and can be predictable.

Motivated by the classical decomposition approach and the examples above, we propose a known pattern predictor type in our resource prediction framework. A particular series may have multiple known pattern components. These components might have daily, weekly, and other periods. Furthermore, there might be multiple known patterns with the same period. The prediction framework should allow for these possibilities by allowing multiple known pattern predictors.

Formally, at time t, the prediction of a known pattern predictor is a vector of non-random values indexed by time:

- $\{s_{t+1}, s_{t+2}, s_{t+3}, \ldots\}$.

The prediction may potentially extend to infinity, although in practice it should be limited by the prediction horizon. The prediction is unconditional and does not change with the arrival of new information.

This definition allows combining two known pattern predictors such that the resulting predictor is also a known pattern predictor (see section 6.7.3).

6.7.2.2 Recent History Linear Predictor

A recent history linear predictor captures patterns in the recent history of a resource. The recent history linear predictor is based on a Gaussian ARMA(p,q) model. Formally, a Gaussian ARMA(p,q) model is prediction model that predicts the next value X_{t+1} in a time series, $\{X_t\}$, using the following equation:

$$X_{t+1} = \varphi_1 X_t + \varphi_2 X_{t-1} + \ldots + \varphi_p X_{t-p+1} + \mathbf{Z_{t+1}} + \theta_1 Z_t + \theta_2 Z_{t-1} + \ldots + \theta_q Z_{t-q+1}$$

Here p is the autoregressive order, q is the moving average order, $\{X_t\}$ is the series being predicted and $\{Z_t\}$ is the series of prediction errors. $\varphi_1, \varphi_2, \ldots, \varphi_p$ are the autoregressive parameters, and $\theta_1, \theta_2, \ldots, \theta_q$ are the moving average parameters. The values of the series $\{Z_t\}$ are jointly normally distributed with a standard deviation σ, i.e. each $Z_t \sim N(\mu_{t+1}, \sigma)$. The size of the variable σ relative to the mean of the series $\{X_t\}$ shows the size of the uncertainty. The degenerate case of a linear predictor with σ parameter value equal to zero shows no uncertainty.

The purpose of the prediction analysis is to determine order parameters p, q, and solve for the autoregressive parameters φ_i and moving average parameters θ_j. Ideally, the error series $\{Z_t\}$ has zero mean and no serial correlation.

Once the values of the parameters are determined, we can use the model equation to make predictions. At time t, the prediction for X_{t+1} is a normal random variable with a known mean and standard deviation. The values of the two series, $\{X_t\}$ and $\{Z_t\}$, are non-random for all $s <= t$. The only randomness in the equation of X_{t+1} is Z_{t+1}. Because Z_{t+1} is drawn from a normal distribution with mean 0 and standard deviation, σ, we know that X_{t+1} in turn is a normal random variable with mean equal to $\varphi_1 X_t + \varphi_2 X_{t-1} + \ldots + \varphi_p X_{t-p+1} + \theta_1 Z_t + \theta_2 Z_{t-1} + \ldots + \theta_q Z_{t-q+1}$ and standard deviation, σ. For brevity, let's denote:

$$\mu_{t+1} = \varphi_1 X_t + \varphi_2 X_{t-1} + \ldots + \varphi_p X_{t-p+1} + \theta_1 Z_t + \theta_2 Z_{t-1} + \ldots + \theta_q Z_{t-q+1}.$$

Thus,

$$X_{t+1} \sim N(\mu_{t+1}, \sigma),$$

where μ_{t+1} is a function of the observed values of the time series and the error series.

Using an ARMA model, we can make multiple-step ahead predictions. To do so, we first substitute t with $t+1$ everywhere in the prediction formula. In this way, we can express the prediction of X_{t+2} conditional on the values of X_{t+1} and Z_{t+1}. Next, we can substitute the value of X_{t+1} with its predictive formula, as follows:

$$X_{t+2} = \varphi_1 X_{t+1} + \varphi_2 X_t + \ldots + \varphi_p X_{t-p+2} + \boldsymbol{Z_{t+2}} + \theta_1 \boldsymbol{Z_{t+1}} + \theta_2 Z_t + \ldots + \theta_q Z_{t-q+2} =$$

$$\varphi_1 (\varphi_1 X_t + \varphi_2 X_{t-1} + \ldots + \varphi_p X_{t-p+1} + \boldsymbol{Z_{t+1}} + \theta_1 Z_t + \theta_2 Z_{t-1} + \ldots + \theta_q Z_{t-q+1}) +$$

$$\varphi_2 X_t + \ldots + \varphi_p X_{t-p+2} + \boldsymbol{Z_{t+2}} + \theta_1 \boldsymbol{Z_{t+1}} + \theta_2 Z_t + \ldots + \theta_q Z_{t-q+2} +$$

$$(\varphi_1 + \theta_1) \boldsymbol{Z_{t+1}} + \boldsymbol{Z_{t+2}} + f(X_t, X_{t-1}, \ldots, X_{t-p+1}, Z_t\, Z_{t-1}, \ldots, X_{t-q+1})$$

The last line in the above formula expresses the prediction of X_{t+2} as a sum involving two random variables, $\boldsymbol{Z_{t+1}} + \boldsymbol{Z_{t+2}}$, and observed values of the series $\{X_t\}$ and $\{Z_t\}$, i.e., the prediction of X_{t+2} is path-dependent on past values of $\{X_t\}$ and $\{Z_t\}$. It is easy to see that the technique can generalize to an n step-ahead prediction for any integer n. Multiple step-ahead predictions made in this manner are path-recording because the prediction of X_{t+n} depends on the history of the resource as recorded by previous values of the resource.

The prediction equations of the ARMA model express a prediction as a continuous probability density function. However, our model of configuration is discrete and can not use continuous probability density functions. Therefore, we need to express predictions using a discrete representation. Furthermore, the discrete representation must be able to express path-dependant, multiple step prediction.

To obtain a discrete representation of continuous predictions, we note a well-know result from statistics: any continuous probability function can be approximated using a discrete probability mass function. In particular, for a normal (Gaussian) probability density function, there is a discrete probability mass approximation using any integer m that is greater than 2. The probability mass function is constructed using the binomial distribution.

As an example, for a few small values of m, we show the probability mass functions (each function is represented as a set of m points, where each point is a value-probability pairs):

- $m = 2$, the pairs are: (+1, ½) and (-1, ½). The mean of probability mass function is 0, the variance is 1.

- $m = 3$, the pairs are (+2, ¼), (0, ½), and (-2, ¼). The mean is 0, variance is 2.

- $m = 4$, the pairs are (+3, 1/8), (+1, 3/8), (-1, 3/8), (-3, 1/8). The mean is 0, the variance is 3.

Notice that the variance is increasing with m. To keep the variance constant at 1, we can divide the values by the square root of m. This will ensure that the variance of each probability mass functions will be 1, thus approximating the same normal distribution with mean 0 and variance 1.

Next, to approximate a normal distribution with mean zero and variance σ^2, we further scale the values in the function by multiplying by σ. To approximate a normal distribution with a non-zero mean, e.g. μ, we add μ to the values in the probability mass function.

For example, to approximate a normal distribution with mean μ and variance σ^2 using 3 points, we have the following probability mass function:

- $m = 3$, the points are as follows: (2/sqrt(2) * $\sigma + \mu$, ¼), (0 + μ, ½), and (-2/sqrt(2) * $\sigma + \mu$, ¼). By performing some simple algebra, we verify that the mean of this probability mass function is μ and the variance is σ^2, as expected.

Once we have the requisite probability mass function, it can be used to make a one step-ahead prediction. In order to express a multiple step-ahead prediction, we need to use a sequence of different probability mass function recursively, one per random variable in the prediction. An appropriate representation for a multiple step-ahead prediction is a probability mass tree. In this tree, the root represents an event with probability 1. All the children of a node jointly represent a probability mass function, i.e., their aggregate probability adds to 1.

Here is an example of n = 2 step-ahead prediction that uses $m = 3$ masses. The mean of the prediction is equal to zero ($\mu = 0$), and the variance is equal to σ^2. The tree is printed using depth-first traversal (see Figure 3 for visualization of the tree).

Step 0, root node: (0, 1).
  Step 1, node 1: (2/sqrt(2) * σ, 1/4),
    Step 2, node 1_1: (2/sqrt(2) * σ + 2/sqrt(2) * σ, 1/4),
    Step 2, node 1_2: (2/sqrt(2) * σ, 1/2),
    Step 2, node 1_3: (0, 1/4),
  Step 1, node 2: (0 + μ, 1/2)
    Step 2, node 2_1: (2/sqrt(2) * σ, 1/4),
    Step 2, node 2_2: (0, 1/2),
    Step 2, node 2_3: (-2/sqrt(2) * σ, 1/4),
  Step 1, node 3: (- 2/sqrt(2) * $\sigma + \mu$, 1/4),
    Step 2, node 2_1: (2/sqrt(2) * σ + 2/sqrt(2) * σ, 1/4),
    Step 2, node 2_2: (2/sqrt(2) * σ, 1/2),
    Step 2, node 2_3: (2/sqrt(2) * σ + 2/sqrt(2) * σ, 1/4).

The probability of each node is conditional on the parent node. For example, the probability that node 1_2 will occur conditional that node 1 has occurred is 1/2. The unconditional probability is 1/2 * 1/4 = 1/8.

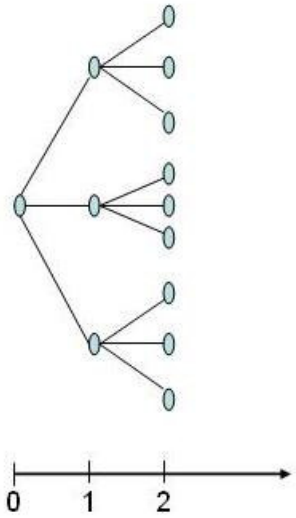


Figure 3: Prediction tree visualized.

6.7.2.3 Bounding Predictor

Linear Gaussian predictors assign non-zero probability to any interval on the real line. The probability of the realized value of the predicted variable falling into a particular interval is always non-zero. This may be a problem for predicting resource availability. In practice, there are limits on the realized value of the available level of the resources. For example, for most resources, availability can not be negative. Also, most resources are physically limited from above. Network connections have hard upper limits for bandwidth, e.g. a DSL line at home has maximum available bandwidth, a wireless network has maximum available bandwidth, etc. Available CPU level is limited from above by the hardware specification of the CPU, and available energy of a device is limited by the capacity of the batteries.

We propose a bounding predictor for the purpose of specifying upper and lower limits of resource availability. Formally, a bounding predictor is a series of min-max tuples, $\{(min, max)_t\}$, where for each t, the corresponding values min_t, max_t specify the minimum, respectively maximum, possible value of the resource being predicted.

It is possible to have more than one source of bounding prediction. For example, the available bandwidth of an 801.11g wireless access point is physically limited by 54 Mbps. However, if the access point is operating in the compatibility mode that also allows 801.11b devices, then the effective bandwidth of the access point for *all* devices will be lower. Thus, one predictor can specify a maximum value of 54 Mbps, but another predictor may specify a lower maximum, at 18 Mbps.

The prediction of a bounding predictor is a vector of scalar tuples that show the minimum and maximum possible values of the series being predicted:

- $\{(min_{t+1}, max_{t+1}), (min_{t+2}, max_{t+2}), \ldots\}$

In theory, this prediction can extend in time to infinity. However, in practice, it would be difficult to predict bounds too far into future, so predictions should be limited by a prediction horizon. Because bounds can change with the arrival of new information, the prediction of a bounding predictor might change over time.

6.7.3 Combining Predictors: Operations and Algorithm

Having built a vocabulary of predictors, we need a mechanism for combining those predictors in a principled manner. To that end, we define binary operators over predictors. Next, we describe an algorithm that takes the set of all available predictors as input and using a sequence of predictor operators, produces a final, aggregating predictor. The justification for the order of combining predictors is partly based on the time series classical decomposition technique (see Appendix A).

6.7.3.1 Predictor Operations and Aggregation Algorithm

We define the following operations on predictors:

- Binary operation *addition1* (+): takes two known pattern predictors as input, and returns a known pattern predictor,
- Binary operation *addition2* (+): takes one linear and one known pattern predictor as input, and returns a generalized predictor,
- Binary operation *bounding1* (||): takes two bounding predictors as input, and returns a bounding predictor,
- Binary operation *bounding2* (||): takes a generalized predictor and a bounding predictor, and gives generalized predictor with bounds.

Given three sets of currently available predictors: linear predictors *LP*, known pattern predictors *SP*, and bounding predictors *BP*, we combine the predictions as follows:

- First, we combine all the known pattern predictors by using the first addition operator with two predictors at a time. This eventually results in a single known pattern predictor,
- Second, we combine the distinguished linear predictor and the remaining known pattern predictor using the second addition operation, creating a generalized predictor,
- Third, we combine all the bounding predictors together, two at a time, using the bounding operator. The eventual result is a single bounding predictor,
- And fourth, we apply the remaining bounding predictor to the generalized predictor, to obtain a generalized bounding predictor. This is the aggregated predictor.

As an example, if we have one linear predictor: lp1, two known pattern predictors: sp1 and sp2, and two bounding predictors: bp1 and bp2, then we obtain a generalized bounding predictor as follows:

$$gbp = (lp1 + (sp1 + sp2)) \,||\, (bp1 \,||\, bp2)$$

An astute reader might ask: why this order of operations? The order of operations is partly dictated by the order in which the various phases of analyses were performed: first de-seasoning and

then linear analysis. Recall that during analysis, the known trend was removed first, resulting in a time series that can be analyzed using linear models.

6.7.3.2 Operational Semantics

We now present the operational semantics of the predictor calculus. For each operation in the calculus, the semantics describe how the output predictor is computed from the input predictors. We assume that the semantics of basic algebraic operations are given.

Addition 1 (takes two known pattern predictors as input, returns a third known pattern predictor)

Concatenation takes two known pattern predictors, $spT^{(1)}$ and $spT^{(2)}$, and returns a third known pattern predictor, $spT^{(3)}$). Let:

- $spT^{(1)}$ predict the following known pattern vector: $\{s^{(1)}_{t+1}, s^{(1)}_{t+2}, s^{(1)}_{t+3}, \ldots\}$,
- $spT^{(2)}$ predict the following known pattern vector: $\{s^{(2)}_{t+1}, s^{(2)}_{t+2}, s^{(2)}_{t+3}, \ldots\}$.

Then the known pattern vector predicted by $spT^{(3)}$ is $\{s^{(1)}_{t+1} + s^{(2)}_{t+1}, s^{(1)}_{t+2} + s^{(2)}_{t+2}, s^{(1)}_{t+3} + s^{(2)}_{t+3}, \ldots\}$, in other words the sum of the two vectors.

Concatenation is associative and commutative. These properties can be proven by using the associative and commutative properties of basic addition.

Addition 2 (takes a linear predictor and a known pattern predictor as input, returns a generalized predictor)

Addition takes a linear predictor, $lpT^{(1)}$, and a known pattern predictor, $spT^{(1)}$, and produces a generalized predictor, $gp^{(1)}$. Thus have we have not discussed the formal representation of a generalized predictor.

The addition shifts the mean of the predicted value of the linear predictor by the value of the known pattern predictor. Let:

- $spT^{(1)}$ predict the following known pattern vector: $\{s^{(1)}_{t+1}, s^{(1)}_{t+2}, s^{(1)}_{t+3}, \ldots\}$, and
- $lpT^{(1)}$ predict the following distribution: $X^{(1)}_{t+1} \sim N(\mu^{(1)}_{t+1}, \sigma^{(1)})$,

Then $gp^{(1)}$ predicts $X^{(2)}_{t+1} \sim N(\mu^{(1)}_{t+1} + s^{(1)}_{t+1}, \sigma^{(1)})$. Intuitively, the addition of a known pattern predictor shifts the predicted values of the linear predictor by an amount equal to the prediction of the known pattern predictor. The shift is applied after the linear predictor has generated the entire path-dependant multiple step-ahead prediction.

When the result of a linear prediction is expressed as a probability mass tree, the addition of a known pattern predictor to the linear predictor results in incrementing *all* the values in the probability mass tree at level d by amount equal to $s^{(1)}_{t+d}$.

Bounding 1 (takes two bounding predictors as input)

The first bounding operator takes two bounding predictors and produces a third one. The effect of bounding is to take the *larger* of the two predicted minimums and the *smaller* of the two predicted maximums. Let:

- $bpT^{(1)}$ predict the following bound sequence: $\{(min^{(1)}_{t+1}, max^{(1)}_{t+1}), (min^{(1)}_{t+2}, max^{(1)}_{t+2}), \ldots, \}$,
- $bpT^{(2)}$ predict the following bound seuqence: $\{(min^{(1)}_{t+1}, max^{(1)}_{t+1}), (min^{(1)}_{t+2}, max^{(1)}_{t+2}), \ldots, \}$,

Then the bound sequence predicted by $bpT^{(3)} = bpT^{(1)} \parallel bpT^{(2)}$ is as follows:

- $\{ (\max\{min^{(1)}_{t+1}, min^{(2)}_{t+1}\}, \min\{max^{(1)}_{t+1}, max^{(2)}_{t+1}\}),$

  $(\max\{min^{(1)}_{t+1}, min^{(2)}_{t+1}\}, \min\{max^{(1)}_{t+1}, max^{(2)}_{t+1}\}), \ldots, \}$,

Bounding 2 (takes a generalized predictor and a bounding predictor as input)

The second bounding operator take a generalized predictor and a bounding predictor. The effect of the bounding operator is to limit the support of the generalized predictor's probability distribution to the bounds of the bounding predictor.

Formally, if a generalized predictor *gp* has an expression {R}, and the bounding predictor *bp* has the following prediction bounds*:*

- $\{(min_{t+1}, max_{t+1}), (min_{t+2}, max_{t+2}), \ldots, \}$,

Then $gbp = gp \parallel bp$ is expressed using the following notation:

- $\{R, \{(\min_{t+1}, \max_{t+1}), (\min_{t+2}, \max_{t+2}), \ldots, \}\}$.

The bounds are appended to the expression of the generalized predictor.

6.7.4 Identifying Appropriate Predictor Types for Each Resource

In this section, we use the earlier analysis of resource properties from section 3.2.4 to identify which predictor types are appropriate to use when predicting each resource type. The shared, perishable, and renewable properties can impact some part of resource management: resource allocation, resource prediction, or both. We discuss each property individually and describe how the management of a resource is affected based on its properties. We then discuss each resource individually and identify the predictor types that are appropriate for that resource.

The shared property affects resource prediction. When predicting resources that are not shared, the availability of the resource does not depend on tasks that are outside of the control of the user. Thus, there are no *exogenous* factors that affect resource availability. Therefore, we can assume that there is no uncertainty when predicting such resources. On the other hand, for shared resources, the use of the resource by other users cause fluctuations in the future level of resource availability. Therefore, predictions are likely to have uncertainty. In our resource prediction model, we use a distinguished recent history linear predictor that is based on ARMA time series model to express the uncertainty. For network bandwidth and remote server compute cycles, which are both shared resources, the use of such a predictor is appropriate. For compute cycles

on a personal device, battery energy, and physical memory, the use of such a predictor is not necessary.

According to Table 2, the perishable and renewal properties are perfectly correlated. Battery is the only non-perishable resource and it is also the only resource that is not automatically renewable. Therefore, for the purposes of identifying appropriate predictors, we consider the perishable and renewable properties together.

The renewable and perishable properties affect the inter-temporal substitution of the resources. For renewable and perishable resources, there is no inter-temporal substitution. This simplifies resource allocation strategy because the entire available supply of the resource can be used in the present, without any impact on future resource availability. For non-perishable resources, on the other hand, the resource allocation strategy must consider how much of the available resource to allocate now and how much to save for later.

Next, we discuss each type of resource and build an argument for the types of resource predictors that are appropriate for that resource.

First, network bandwidth is a shared, perishable resource. Based on the shared property of the resource, its availability is given exogenously. Consequently, the use of a recent history predictor is appropriate. Furthermore, bandwidth availability might have repeatable patterns. Therefore, the use of one or more known pattern predictors is also appropriate. In addition, the resource is subject to absolute availability bounds. For example, the availability of network bandwidth can never be negative and maybe bounded by network protocol and network wire specifications. Thus, the use of one or more bounding predictors is appropriate. The perishable property of network bandwidth implies that there are no inter-temporal supply substitution possibilities. This simplifies the prediction and usage accounting of the resource.

Second, the availability of CPU cycles on a device that is under the exclusive control of the user does not require modeling uncertainty, because there are no exogenous factors affecting the available level of the resource. In other respects, modeling the availability of this resource is very similar to modeling server CPU. Therefore, it is appropriate to use one or more known pattern predictors and one or more bounding predictors for predicting the CPU cycles on a personal device. However, the use of a linear recent history predictor is not necessary.

Third, battery is not shared and not perishable. The *current* supply of battery is not random and can be measured precisely. The future supply of battery depends on the drain rate and the charge rate. The drain rate is affected by user's own tasks; however, there are hardware dependent maximum and minimum charge rates. Furthermore, it is reasonable to assume that between the minimum and maximum, the drain rate can vary at some constant incremental rate. The minimum drain rate, the maximum drain rate, and the incremental drain rate are part of the battery specification and can be communicated as part of resource discovery. The charge rate can take two values: (a) zero if not connected to wall electricity and (b) a constant non-zero rate when connected to wall electricity. In this thesis, we make two simplifying assumptions: (1) when the battery is connected to wall electricity, our model and algorithms will not manage battery allocation, and (2) we will not predict the future times when battery will be plugged into or out of wall electricity. With those assumptions, it suffices to model the case when the battery is not charging and there is no expectation that the battery will be charging in the future. Consequently, the current available level of battery can be expressed using one number. This number is a 0 step-ahead

prediction of available battery. This prediction can be made using a degenerate linear predictor that has no uncertainty, but provides only the current level of the resource.

Fourth, memory on a personal device is neither shared nor perishable. Because there are no exogenous factors affecting the usage of memory, we don't need to model uncertainty in the predictions of memory. Therefore, the use of the linear recent history predictor is not necessary. The use of the known pattern and bounding predictors is appropriate for the same reasons mentioned when discussing the CPU cycles on a personal device.

6.8 Chapter Summary

In this chapter, we have built the case for predictions in pervasive, resource management systems. We have motivated the importance of using both recent as well as ancient history in making predictions. We have shown how analysis of recent history can be used to obtain a probabilistic profile of the immediate future behavior of the resource. We have also argued how long-term patterns of resource behavior can be used in conjunction with recent history analysis.

We have demonstrated the importance of operational issues, such as choosing prediction scale, horizon, and detail that are appropriate for the system. Furthermore, we have described the importance of path-recording predictions and contrasted it with predictions that do not record path. We have used these considerations to build a basis for requirements of a runtime resource prediction system.

In the second part of the chapter, we have formally defined prediction models. For task request prediction, we have identified possible sources of predictions, such as user's calendar and schedule, past history of task requests, and information directly solicited from the user. We have described plausible ways to capture predictive information using a two-level hierarchy. First, we identify how accurate the prediction is based on the source. If accurate, we use one or two-sided Gaussian distribution to capture the arrival type of a task request event. Otherwise, we allow arbitrary probability mass function to describe that arrival time.

We have defined three types of resource predictors: linear-recent history, known pattern, and bounding. We have also defined operations for combining predictors, semantics for those operations, and an algorithm that uses the pool of available predictors as input and combines their predictions into a generalized, aggregated predictor.

7 Anticipatory Model of Configuration

The goal of the anticipatory model of configuration is to define the problem of configuration in a way that allows forward-looking analysis of configuration decisions. The objective of anticipatory configuration is to select a sequence of configurations so that the accrued utility of those configurations is maximized.

To help define the formal expression of the anticipatory configuration model, we use the problem structure introduced in Chapter 4. Furthermore, we build upon the abstraction and notation defined in Chapter 5. The anticipatory model of configuration makes the following incremental changes to the reactive model: (1) it uses accrued utility to make configuration decisions (in the reactive model instantaneous utility was used), (2) it introduces predictions of task requests, (3) it introduces predictions of resource availability.

In the anticipatory model of configuration, there is a temporal dimension. The definition of the spaces, input variables, and decision variables need to be modified to reflect the temporal dimension. Furthermore, the predictions are updated periodically.

To help the reader navigate through this chapter, we will occasionally refer to the following simplified example. In this example, the user has two known tasks: T_1 and T_2, and each task has two services, as follows: $T_1 = \{S_1, S_2\}$, $T_2 = \{S_3, S_4\}$. Furthermore, we'll assume that the future resume and suspend times of the tasks are known without uncertainty. Starting with some initial time t_0, we know that:

- Task T_1 will be activated at time t_1, $t_1 > t_0$,
- Task T_2 will be activated at time t_2, $t_2 > t_1$,
- Task T_1 will be deactivated at time t_3, $t_3 > t_2$,
- Task T_2 will be deactivated at time t_4, $t_4 > t_{23}$.

In other words, task T_1 is active from t_1 to t_3, and task T_1 is active from t_2 to t_4. Because the predictions are perfect, all future events occur at their predicted times.

In general, throughout this chapter, we will assume that the user has a fixed set of known tasks: $KnownTasks = \{T_1, T_2, \ldots, T_j\}$. In the example above, the set of known tasks has only two tasks: $\{T_1, T_2\}$.

7.1 Spaces

7.1.1 Utility Space

In the anticipatory model of configuration, the measure of user satisfaction is accrued utility (AU). The accrued utility over a period of time $[t_1, t_2)$ is the integral of the instantaneous utility

from t_1 to t_2. In our discrete world, this integral is equal to the sum of the instantaneous utilities from t_1 to t_2 -1:

$$AU\Big|_{t_1}^{t_2} = \sum_{i=t_1}^{t_2-1} IU_i$$

The above definition omits the details of how IU is computed. In subsection 7.2.1, we will see that the IU is calculated using the same expression of the preferences and utility from the reactive model.

7.1.2 Capability Space

For configuration purposes, we are interested in the capability space of the tasks that are active. As the user resumes or suspends tasks, the set of active tasks changes, and so does the capability space. Given the set of active tasks, the definition of the capability space is the same as that in the reactive model. Recall from subsection 5.1.2, that the capability space of a set of services is defined in terms of the capability space of each service in that set.

In the anticipatory model, the capability space changes over time depending on which tasks are active at the time. In order to identify the set of active services at a particular time, it is sufficient to capture resume and suspend operations on all known tasks up to that time.

Using the example introduced in the beginning of the chapter, we have the capability spaces of tasks T_1 and T_2 as follows;

- $T_1 = \{S_1, S_2\}$, $C_{T1} = C_{S1} \otimes C_{S2}$,
- $T_2 = \{S_3, S_4\}$, $C_{T2} = C_{S3} \otimes C_{S4}$.

The capability space of the active services changes as follows:

- From time t_0 till time t_1, there are no active tasks, so the active capability space is empty,
- From t_1 to t_2, the active capability space is $C_{T1} = C_{S1} \otimes C_{S2}$,
- From t_2 to t_3, it is $C_{T1} \otimes C_{T2} = (C_{S1} \otimes C_{S2}) \otimes (C_{S3} \otimes C_{S4})$,
- From t_3 to t_4, it is $C_{T2} = C_{S3} \otimes C_{S4}$,
- From time t_4 onwards, the capability space is empty again.

7.1.3 Resource Space

In the anticipatory model, the resource space is the same as in the reactive model. However, to record resource availability, we use time series of multi-dimensional vectors of resource availability. We use curly brackets indexed by time to denote a time series of resource availability: $\{(rAvl_1, rAvl_2, rAvl_3, \ldots, rAvl_k)\}_s$. When referring to predictions of available level of the resources for some future time s at current time cT, we use the following notation:

$$(rAvl_1, rAvl_2, rAvl_3, \ldots, rAvl_k)_{s|cT}$$

In the above expression, the conditional operator, |, indicates that the prediction for time *s* is conditioned on the available information at time *t*. We may also separately refer to the time series and predictions of each component of a resource vector, as follows:

$$\{rAvl_j\}_s \text{ and } (rAv_t)_{s|cT}$$

7.2 Inputs

7.2.1 User Preferences and Predictions of Tasks

Each task in the set of known tasks has associated preferences. Preferences are elicited from the user when the task is created. Once elicited, preferences do not change.

The formal expression of the preferences in this model is the same as that in the reactive model (see Chapter 5, section 5.2.1). In particular, the expressions for QoS Preferences, Supplier Preferences, and overall instantaneous utility still apply in this model.

In the anticipatory model, there are predictions for task request operations: the resume time, the suspend time, and the duration. These predictions follow the model in section 6.5.3.

In the example introduced earlier in this chapter, we have the following expression of the predictions for tasks T_1 and T_2.

For task T_1,

- Resume: *{tight=T, {mean=t1, stdev=0 s, left=T,right=T}, 1.0},*
- Suspend: *{tight=T, {mean=t3, stdev=0 s, left=T,right=T}, 1.0}.*

For task T_2,

- Resume: *{tight=T, {mean=t2, stdev=0 s, left=T,right=T}, 1.0},*
- Suspend: *{tight=T, {mean=t4, stdev=0 s, left=T,right=T}, 1.0}.*

In general, at current time *cT*, for each task T_i in *KnownTasks*, there is a prediction for the future resume, suspend times, and durations of that task, $TaskPred_i(cT)$. The prediction is expressed using the notation described in subsection 6.5. Task predictions are periodically updated.

7.2.2 Supplier Profiles: Quality-Resource Mappings

In this thesis, we don't model the availability of suppliers over time. For simplicity, we assume that the list of available suppliers does not change over time. A supplier profile has been defined in section 5.2.2. We continue using that definition. To recap, the second input consists of the following:

- The set of available suppliers, $AvailableSuppliers = \{Supp_1, Supp_2, \ldots, Supp_p\}$, and
- A profile for each supplier *Supp*: $QoSprof_{Supp} : C_s \leftrightarrow R$.

7.2.3 Resource Availability

For each resource in the space of resources, has an aggregated prediction. At current time *cT*, for the *i*th resource in the space of the resources, the aggregate prediction for the level of available resources is $ResPred_i(cT)$, $ResPred_i(\bullet)$ is defined per section 6.7.3. We will use a short-hand notation for the predicted resource vector for all resources in the environment, as follows: $(rAvl_1, rAvl_2, \ldots, rAvl_k)_{s|cT}$, where $s \geq cT$. For $s = cT$, the prediction is exact and shows the current resource availability. For $s > t$, the prediction is the probability distribution of a random variable.

Resource predictions are periodically updated.

7.3 Decision Variables

In the anticipatory model, the problem is to find a sequence of configurations that maximizes the accrued utility. As in the reactive model, we continue to define a configuration using its three parts. As a reminder to the reader, these three parts of a configuration, *Conf*, are: (1) the task layout, denoted as *TaskLayout(Conf)*, (2) for each supplier a a_i in that task layout, a quality-resource pair, $QRPair_i$ *(Conf)*, that must be selected from the profile of that supplier, and (3) aggregate resource allocation, *RAggAlloc(Conf)*, which is the sum of the resource allocation vectors in the configuration, *Conf*. Instead of reproducing the definition and the notation of the three parts, we refer the reader to sections 5.3.1, 5.3.2, and 5.3.3.

In the anticipatory model, we must consider the accrued utility of a sequence of future configurations, $SEQ(t, t+pt-1)= \{Conf(t), Conf(t+1), \ldots, Conf(t+pt-1)\}$. We can find out the *actual* accrued utility of a sequence of configurations only after the fact. We use the term, *ex post*, to refer to both configurations choices made and the accrued utility of those configurations after the fact.

However, when making decisions, we don't have the benefit of the outcome of the future events yet, and we can only evaluate and maximize the *expected* accrued utility by considering some or all possible sequences of future configuration. To refer to the expected value of a sequence of configuration decisions, we use the term, *ex ante*. The best that a configuration algorithm can do is to maximize *ex ante* expected utility. Therefore, we must formulate the optimization problem of configuration in the anticipatory model using *ex ante* utility.

Even though the formulation of the optimization problem must be expressed in terms of the *ex ante* expected accrued utility of a sequence of future configurations, we only need to decide and commit to the next immediate configuration decision. Therefore, the decision variables in this model are exactly the same as those in the reactive model, i.e., the three parts that make up a configuration.

We will use the shorthand notation, $IU(Conf(s), Conf(s-1))$, to express the expected instantaneous utility of the next immediate configuration choice *Conf(s)*, when switching from *Conf(s-1)*. Here, *Conf(s)* and *Conf(s-1)* take values from the space of all alternative configurations. However, *Conf(s-1)* can also take the special value of *Null* to indicate that there was no previously running configuration.

The accrued utility of a sequence of configurations, $SEQ(t, t+pt-1)= \{Conf(t), Conf(t+1), \ldots, Conf(t+pt-1)\}$, when starting from initial configuration *Conf(t-1)* is as follows:

This expression can be considered both *ex post* and *ex ante*. When considered *ex post*, the con-

$$AU\big|_{t}^{t+pt}(SEQ) = \sum_{i=0}^{pt-1} IU\,(Conf\ (t+i), Conf\ (t+i-1)$$

figuration decisions in the sequence have already been made, and the utility is actual accrued utility. When considered *ex ante*, the accrued utility is expected. In the expression for accrued utility, when starting from no configuration at all, we allow *Conf(t-1)* to take the special value, *Null,* that indicates no previously existing configuration.

7.4 Optimization Problem

In the anticipatory model, the objective of configuration is to maximize the *ex ante* expected accrued utility of a sequence of configurations over the decision horizon, *DH*. In this model, we define and solve a sequence of optimization problems instances.

Let *tInit* be some initial time, *cT* be the current time, *DH* be the duration horizon, and *T* be some final time when the user is done working. Let $KnownTasks = \{T_1, T_2, \ldots, T_j\}$ denote the set of user's known tasks.

In the problem formulation, *cT* is initialized to *tInit*, and is incremented periodically.

Given the following inputs:

- (Input 1) Preferences for each task, and predictions for the future resume, suspend times and duration for each task, $TaskPred_i\ (cT)$ (Note: preferences do not change over time, but task predictions are updated),

- (Input 2) The set of available suppliers, $AvailableSuppliers = \{Supp_1, Supp_2, \ldots, Supp_p\}$, and a profile of each supplier, $QoSprof_{Suppi}$ (Note: neither the set of suppliers nor the profiles change over time),

- (Input 3) An aggregate predictor for each resource, $ResPred_j\ (\bullet, cT)$, (Note: predictions are updated periodically),

Find:

- (Decisions 1-3) A configuration *Conf(cT)* (the three parts of the configuration are defined in section 5.3),

Such that the *ex ante* expected accrued utility of the future sequence of configuration decisions, $SEQ(cT, cT+DH) = \{Conf(cT), Conf(cT+1), \ldots, Conf(cT+DH-1)\}$ is maximized:

$$\underset{SEQ \in Set(Seq)}{\arg\max}\ E\left[AU\big|_{cT}^{cT+DH}(SEQ) = \sum_{i=0}^{pt-1} IU(Conf(cT+i), Conf(cT+i-1) \mid ct \right]$$

Subject to the following constraints:

- (Constraint 1, current aggregate resource availability constraint) The sum of the resources in the aggregate resource allocation among the suppliers of configuration *Conf(cT)* can not exceed the level of available resources:

$$RAggAlloc\ (Conf\ (cT)) \le (rAvl_1, rAvl_2, \ldots, rAvl_l)_{cT|cT}$$

For non-perishable, non-renewable resources, based on Contraint 1, we can infer the following additional constraint:

- (Constraint 2, for a non-perishable, non-renewable resource, the aggregate supply of the resource is fixed and does not increase) *Ex post*, over the interval of time *[tInit, tInit+T)*, the resource allocation among all configurations in the sequence *SEQ(tInit, tInit+T)* must satisfy the following resource availability constraints for each non-perishable, non-renewable resource RR_j:

$$\sum_{i=0}^{T} RAggAlloc\ (Conf\ (tInit + i))[\,jj\,] \leq (rAvl_{\,jj})_{tInit\,|tInit}$$

Since Constraint 2 can be inferred from Constraint 1, the former need not be included in the formulation of the problem.

As an aside note that for perishable resources the only feasibility constraint expressed in the problem is that the allocation of the resource among the suppliers of the configuration can not exceed the current available level of the resources. An optimal or near-optimal solution must also take into account the predictions of the future available levels of the resource.

7.5 Dynamic Behavior and Reconfiguration

The dynamic behavior of the anticipatory strategy is similar to that in the reactive model (see section 5.5 for the loop in the reactive strategy). The only difference is that the inputs include predictions. The dynamic control loop proceeds as follows:

- Monitor and record updated values of the inputs, including predictions,
- Setup and solve a new optimization problem instance, calculating a new optimal configuration,
- Effect the changes in the environment, brining the environment to the state described by the new optimal configuration.
- Wait a pre-determined amount of time.

Waiting for a pre-determined amount of time between loop operations ensures that the model operates at a uniform frequency.

7.6 Chapter Summary

In this chapter we have provided the formal definition of the anticipatory configuration model. This model has the following distinguishing characteristics:

- It has a long decision-making horizon, e.g., several dozen or hundred units of prediction scale,
- It captures dependencies among multiple configuration decisions,

- It leverages predictions of resource availability and task request operations.

This model has a number of advantages over its reactive counterpart. By leveraging predictions and utilizing a long decision horizon, it has the potential to provide better utility to the user. Also, this model can reason about consumption and supply of resources that are non-perishable and non-renewable, e.g., battery.

But the potential for improved decision-making of the model comes at a cost. The search space of the anticipatory configuration strategy is significantly larger than that of the reactive model because of predictive inputs and long decision making horizon. Consequently, the anticipatory strategy may have significantly larger overhead. With the anticipatory strategy, a single optimization decision is likely to be much more costly both in terms of computational resources and have higher latency.

8 An Architecture for Automatic Configuration

In this chapter, we describe the architectural design of the infrastructure for automatic configuration. The goal of the infrastructure is to automate the routine chores of managing applications, devices, and resources that lay users must perform to support their daily computing tasks. In the previous chapters, we defined two analytical models (reactive and anticipatory) that provide a structure to the problem of configuration. This structure allows reasoning about system decisions and selecting near-optimal configuration from a large number of alternatives. The infrastructure implements computations and mechanisms necessary for automation, while addressing a number of technical challenges described in Chapter 0. We organized these challenges into two groups: functional requirements and extra-functional attributes.

The infrastructure must satisfy the following functional requirements: (1) continuously providing near-optimal utility to the user, (2) monitoring changes from the user and the environment, and adapting the runtime state of the applications accordingly, and (3) leveraging predictions while dealing with uncertainty in those predictions.

First, in order to satisfy the requirement of utility, the infrastructure must continually optimize the utility of the user for each task. The infrastructure must select applications that are best suited to the user's preferences, allocate resources among those applications, and set the runtime state of the applications to best utilize the allocated resources.

Second, the infrastructure must be able to deal with changes. There are two sources of changes. The first source is the user, who can resume and suspend tasks, switch between tasks, and make incremental modifications to the running tasks. The other source of changes is the environment. The infrastructure must be able to handle fluctuations in the available level of resources as a normal situation, requiring no user intervention. The infrastructure must provide mechanisms for monitoring changes from both sources, and adapting to them as needed. To ensure a seamless experience to the user, the infrastructure must balance potentially costly reconfigurations with the quality of service of the task.

Third, uncertainty presents a key challenge in the design of the architecture. Because changes from the user or from the environment can cause potentially costly adaptations, anticipating such changes early may create opportunities for improvement (see section 5.6 for a discussion and an example showing potential improvements). The architecture must provide the mechanisms to acquire predictions and implement algorithms that use predictions to benefit the user.

In addition to the functional requirements, there are a number of extra-functional attributes that the architecture must address: (1) performance, (2) interoperability in the face of heterogeneous applications and platforms, (3) reasonable cost of incremental maintenance when new features become available in parts of the infrastructure, (4) availability of the infrastructure despite transient failures in the environment.

First, to ensure that automatic configuration is useful in practice, the infrastructure must address two performance concerns: resource overhead and configuration latency. Because the infrastruc-

ture is targeted for mobile and pervasive computing where resources are often limited, the resource overhead of the infrastructure should be small, preferably negligible. In the face of the complex calculations required to make near-optimal configuration decisions, satisfying this requirement is critical. Next, we must ensure that configuration actions such as task instantiation incur low latency. For example, when a user resumes a task, the time it takes for the infrastructure to instantiate the applications in the task should be low relative to the manual start-up of the same set of applications.

Second, the automatic configuration infrastructure must work with a variety of hardware, operating, and application platforms. Therefore, the infrastructure must interoperate with different implementation languages and programming interfaces.

Third, parts of the infrastructure may change over time, requiring modifications to the rest of the infrastructure. The design should aim to reduce the cost of potential maintenance in response to the following changes: (a) when new applications become available, (b) when new algorithms for resource allocation and decision making become available, and (c) when new resource prediction models become available. We emphasize these changes because their impact on the architecture is potentially large. For example, implementing a new resource model may require potentially costly changes elsewhere. A well designed interface can help reduce or eliminate such changes.

The fourth quality attribute is the availability of the overall infrastructure despite failures in the environment. Such failures include application crashes and drop in the available level of resources. The system should be able to recover from these failures by offering alternative configurations to the user. All such changes must be handled as part of the normal operation of the system, and require no special handling.

The rest of this chapter is organized as follows. In section 8.1, we describe the architecture of Aura, an infrastructure for pervasive and mobile computing that automates task management. Our work fits in the broader context of Aura, and provides key features to make automatic configuration possible. The following 3 sections describe the contributions of this thesis to Aura. In section 8.2, we describe the design and the internal structure of the Environment Manager, which is responsible for configuration decisions. In section 8.3, we describe the design of a resource prediction framework, which is responsible for monitoring resource availability and providing resource predictions. In section 8.4, we describe the aggregate resource prediction API, which is the communication conduit between the Environment Manager and the resource prediction framework. In section 8.5, we discuss the rationale behind the key architectural decisions and compare the merits of those decisions with alternatives.

8.1 The Aura Software Architecture

Figure 4 shows the high level components and connectors of Aura, a software infrastructure for task management and automatic configuration. The architecture has 3 layers: the Task Management layer, the Environment Management Layer, and the Environment Layer. The allocation of the architectural component into the layers is organized around the key functional roles, as each layer is implements part of the functionality to make automatic configuration possible.

The role of the Task Management layer is to elicit the requirements of a task from the user. A Task Manager is an interactive graphical application that allows the user to define tasks, search for tasks, and perform operations on tasks such as resume and suspend. To instantiate task re-

quests, the Task Manager communicates with the Environment Management layer using a representation specifically designed for that purpose. In his thesis [52], Joao Sousa described the design and the implementation of the Task Manager. His thesis also defined a lightweight, platform-independent task description language that is used to store user's tasks. The design of Prism and the task description language are outside of the scope of this thesis.

The Environment Management layer is comprised of a set of Supplier Managers and an Environment Manager (EM). In order to participate in automatic configuration, each application on each hardware device must have a Supplier Manager. The role of a Supplier Manager is to act as a proxy between a managed application and the rest of Aura by providing a uniform *monitoring and control* interface. Each Supplier Manager has an intimate knowledge about the application it manages. A Supplier Manager implements the application-specific APIs for monitoring and control, and translates application-specific information to a standard format. The Supplier Manager also maintains the application's quality-resource profile, registers the availability and the capabilities of the managed application with the Environment Manager, and processes configuration instructions from it, such activation, deactivation, and resource use adjustment. To facilitate the development of Supplier Managers, we have implemented a Supplier Manager Development Kit (SDK) in the Java and C/C++ programming languages. The SDK has been used by a number of undergraduate and Master-level students to develop about a dozen Supplier Managers. The SDK greatly reduces the development time of the Supplier Managers by providing reusable libraries for communicating with the rest of Aura.

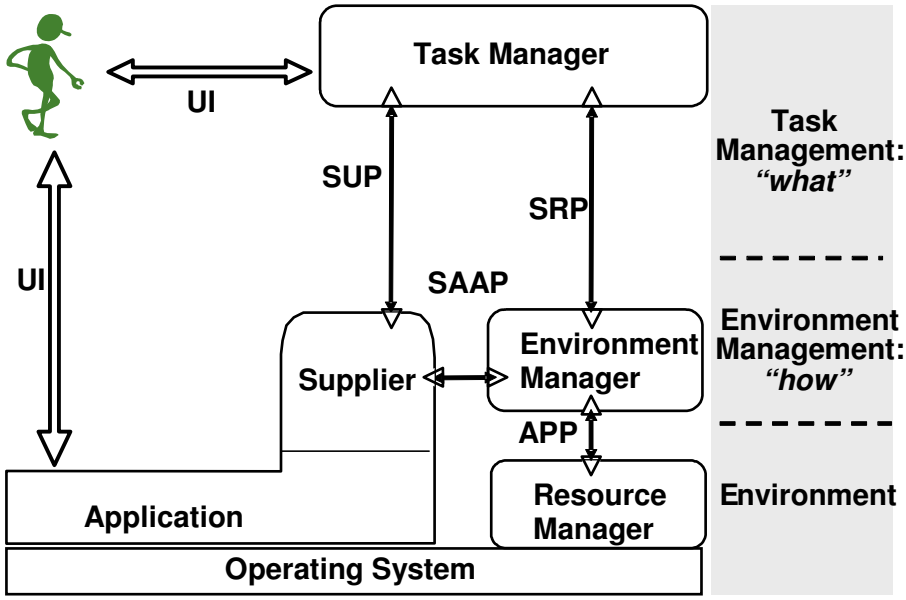


Figure 4: A component and connector view of the Aura Software Architecture.

The Environment Manager is the centerpiece of the Environment Management layer, and is responsible for coordinating the available information from three sources: the Task Manager, the Supplier Managers, and the Resource Managers. The EM implements data structures that collect available information from all three sources. This information is used by efficient algorithms that calculate which configurations of the environment provide near-optimal utility to the user. Section 8.2 provides a detailed discussion of the design of the EM.

The Environment Layer is comprised of the operating system, the network, the applications, and the set of Resource Managers (RM). The contribution of this thesis to the Environment layer is the design of the Resource Managers. An RM is responsible for monitoring the available level of a resource and providing aggregated resource predictions to the Environment Manager. The design of the Resource Manager is provided in section 8.2.

The remaining components in the architecture are provided by third parties. Aura can work with both off-the-shelf, widely deployed applications as well as research prototypes that provide fidelity and resource-aware features (for examples of systems providing such features, see, the theses of Noble [43] and DeLara [13]).

Table 4 summarizes the mission of the Aura layers and credits the authors of the various components that make up the Aura architecture.

Table 4: Summary of Aura components and their roles.

layer	*mission*	*components*	*role*	*credit*
Task Management	*what* the user needs	Task Manager	- elicit task requirements - communicate task requests to the EM - maintain task state between activations	Joao Sousa's thesis
Environment Management	*how* to best configure the environment	EM	- calculate optimal configurations - instantiate configurations - monitor inputs and adapt	This thesis (section 8.2)
		Suppliers	- control and monitor applications	Joint work (8.1)
Environment Proper	support user tasks	Resource Manager	- monitor resources, calculate predictions	This thesis (8.3)
		Applications	- provide service to the user, manage QoS	
		Operating System Network	- manage low-level resource scheduling	Vendors

The architecture also defines connectors that facilitate the communication between the Aura components:

- Service Request Protocol (SRP) between the Task Manager and the Environment Manager. Using this protocol, the Task Manager can communicate task definition and task requests to the Environment Manager. In response to the requests of the Task Manager, the Environment Manager sends task layout information: a list of the activated Suppliers and the level of the QoS these suppliers are expected to provide,

- Supplier Announcement and Activation Protocol (SAAP) between the Suppliers and the EM. A Supplier announces its availability by sending a registration message to the EM.

This message describes the QoS vs. resource profile of the application. When the Environment Manager decides to activate a Supplier, it sends an activation message with instructions about the optimal runtime state of the application. Based on the information in the activation message, the Supplier adjusts the runtime state of the application. The Supplier monitors the runtime state of the application, sending periodic status message to the EM,

- Service Use Protocol (SUP) between the Suppliers and the Task Manager. The Task Manager uses this protocol to set the task-specific parameters of the application that relate to information assets such as which files to open and where to set the cursor. Task Manager maintains this information between task instantiations,

- Aggregate Prediction Protocol (APP) for resources between the Resource Managers and the Environment Manager. The Environment Manager uses this protocol to request resource availability prediction from Resource Managers. Upon activation of a task, the Environment Manager reports the expected resource usage of the suppliers in the task to the Resource Manager. This information corresponds to the third decision variable in the analytical models of configuration.

The first three protocols: the SRP, the SUP, and the SAAP, are outside of the scope of this thesis (see [52] for a detailed description of these protocols). The Aggregate Prediction Protocol is one of main contributions of this thesis. A detailed description of that protocol is in section 8.4.

8.2 The Design of the Environment Manager

The mission of the Environment Manager is to determine *how* to best support the user's tasks. This mission can be broken down to a number of functional requirements:

- Receive and parse messages from the Task Manager, Suppliers, and the Resource Managers. Instantiate data structures based on the messages. Maintain an internal view of the runtime state of the applications, the resources, and the known tasks,

- In response to task request operations from the Task Manager, calculate near-optimal configurations for the relevant tasks. Periodically repeat the calculations for all active tasks to make necessary re-configurations to take advantage of improved conditions or to mitigate any deterioration in the environment,

- Based on the results of calculations, instantiate the chosen configurations in the environment by starting, stopping, or adjusting the runtime state of the applications in those configurations. Transmit the outcome of the instantiation to the Task Manager, so that it can maintain a consistent view of the activated components in the environment,

- Monitor the state of the environment in a continual manner including the level of available resource and the runtime state of the running applications. Based on the monitored information, adjust the configuration to maintain its optimality,

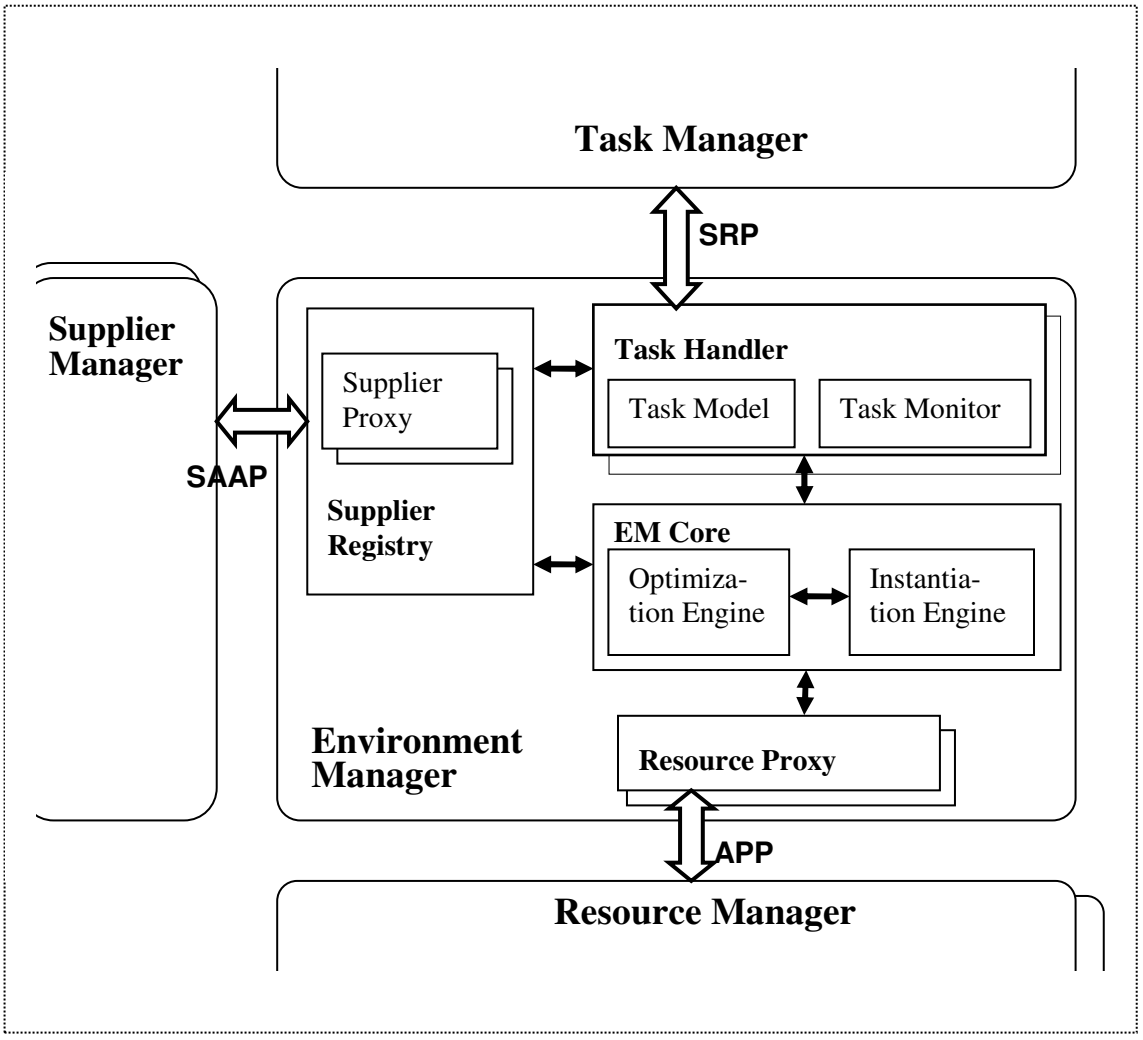


Figure 5: Internal structure of the Environment Manager.

The design of the Environment Manager must also promote all four extra-functional attributes identified in the opening section of the chapter.

Figure 5 shows a sketch of the internal structure of the Environment Manager in the context of the overall architecture. The major components in the EM are implemented as Java classes, and the interactions between them (shown as small filled arrows) are realized as Java method invocations. The large unfilled arrows show the communication between the components of the EM and the architectural-level components of Aura: the Task Manager, the Suppliers, and the Resource Managers.

The Supplier Registry processes registration messages sent by the Suppliers. Upon the receipt of a registration message, the Registry creates an instance of a Supplier Proxy. The Proxy stores the network coordinates and the quality-resource profile of the Supplier. The Registry offers a query interface, so that available suppliers can be queried by service type and a number of other parameters. Registrations are time-stamped to track expiration. If a Suppliers' registration becomes stale, the query interface omits it from the results.

For each known task in the Task Manager, the EM creates and maintains an instance of a Task Handler object. Each Task Handler object has an instance of a Task Model object. The Task Model maintains the state of the task: the services, their definitions, preferences, and the activation status. For each active service, the Task Model also records expected and actual level of QoS for monitoring purposes.

Both the Task Manager and the Environment Manager maintain views of the state of the known tasks. In order to ensure that the views maintained by them are consistent, we use a synchronization protocol. Both the TM and the EM can make incremental changes to the tasks and communicate the changes to the other. For example, if the user adds a service to a task, or resumes the task, the Task Manger communicates the changes to the EM so that the latter can take the requisite configuration actions. Similarly, when the EM switches an active supplier with an alternative, the Task Manager is notified. The synchronization protocol is part of the SRP and is described in [52].

Each Task Handler maintains an instance of a Task Monitor object that observes the runtime state of the task. Active Suppliers send status (heartbeat) messages to the EM, and EM directs this traffic to the appropriate Task Monitor. These heartbeat messages contain statistical summaries of the QoS performance of the application. A Task Monitor triggers re-configuration in two situations: (1) when a heartbeat message from a Supplier indicates that an immediate action is required, or (2) when a pre-determined amount of time has passed. The latter case ensures that the configuration in the environment reflects the latest information from all input sources, including the resource predictors.

For each Resource Manager in the environment, the EM creates and maintains a Resource Proxy object. This object maintains all available information about a resource, including the current available level of the resource as well as predictions. Predictions are updated at pre-determined frequency. A Resource Proxy uses the Aggregate Prediction Protocol to negotiate and request predictions from a Resource Manager. Section 8.4 describes the protocol in detail.

The EM Core comprises two components: the Optimization Engine and the Instantiation Engine. The Optimization Engine is a stateless object that implements optimal algorithms for configuration. It can be invoked by any Task Handler instance. When invoked, the Optimization Engine is provided with inputs from the three sources: the Task Handler, the Supplier Registry, and Resource Proxies. The result of the Optimization Engine's computation describes the incremental modifications necessary to bring the configuration of the environment to the desired state. The Instantiation Engine takes the task layout and implements the changes in the environment from the current to the desired state. The Instantiation Engine invokes start, stop, or adjust operations on the Suppliers that are part of the currently active and the newly optimal configurations. Upon the completion of instantiation actions, it prepares a Task Layout data structure and passes this structure to the Task Handler. The Task Handler uses the Task Layout object to update its internal state and prepares a message for the Task Manger. And lastly, the Instantiation Engine creates a Resource Allocation data structure which is provided to the Resource Proxies. Resource Proxies send this information to the appropriate Resource Managers.

8.3 The Design of the Prediction Framework (Resource Manager)

The design requirements of the Resource Manager are driven by the information needs of the Environment Manager. In order to make the design of the Resource Manager applicable in contexts outside of Aura and pervasive computing, we also considered the predictions needs of an adaptive system in the enterprise computing domain, Rainbow [9]. Additional information about Rainbow and the applicability of the resource prediction framework to Rainbow is found in section 11.1.6.

Based on the analyses of the prediction needs of systems such as Aura and Rainbow, we have formulated the following set of requirements for a resource prediction framework:

- The predictions provided by the prediction framework must be *consistent with existing analytical models* published in recent literature. In concrete terms, the predictions should be consistent with the resource prediction model and operational semantics defined in section 6.7,

- The predictions must be available for *multiple steps* into the future from the current time and must record *path* information. As explained in section 6.2.3, predictions that record likely future paths and their probabilities contain more information compared to predictions that are not path-dependant. Both Aura and Rainbow have use case scenarios when path-recording predictions are necessary,

- The resource prediction framework should be flexible to allow multiple sources of predictions when such sources are available. The design of the prediction framework must use all available known pattern and bounding predictors when calculating an aggregate prediction,

- The framework design must allow predictors to join the framework and to leave it dynamically, without affecting uptime of the framework,

- The framework must produce aggregated predictions on demand. The aggregated prediction must be based on the information from all available predictors at the time of prediction. The calculation of the aggregate prediction must be consistent with the operational semantics described in section 6.7.3.1.

The design of the resource manager is driven by the sources of the inputs needed and the calculations required to obtain an aggregate prediction as described in the schematics in Appendix A, Figure 24. To calculate aggregate predictions at runtime, the resource management framework requires the following inputs:

- Monitored values of the observed resource availability, $\{Y_t\}$. This information comes from resource monitors that use platform-specific APIs to obtain such information. The architecture needs to define a standard component and an interface for making the monitored values available for aggregated prediction purposes,

- The model parameters of the linear recent history predictor according to section 6.7.2.2. These values are obtained using prediction analysis done offline. The architecture should provide mechanisms for setting the parameters of the model and changing them if necessary,

- The list of available known pattern and bounding predictors and the output of their predictions (see sections 6.7.2.1 and 6.7.2.3). The list of predictors might change dynamically, as new predictors become available, or existing predictors become unavailable.

Based on the available predictors and their inputs, the prediction framework must make certain calculations. The explanation for the order of calculations and the operational semantics of the calculations is provided in section 6.7.3:

- Remove all known pattern components from the raw series, $\{Y_t\}$, to obtain series, $\{X_t\}$,
- Apply the linear recent history predictor $\{X_t\}$ in order to generate predictions of $\{X_t\}$,
- Combine all available known pattern predictors using the *addition* operator for season predictions to obtain a single known pattern prediction,
- Add the resulting known pattern prediction to the prediction of $\{X_t\}$ using the *addition* operator for a known pattern predictor and a linear recent history predictor,
- *Bound* the result using the bounding predictors to obtain an aggregated prediction.

The architecture of the framework is shown in Figure 6. For each resource instance, there is one logical instance of the framework. There are 5 component types in the architecture of the framework: (1) Aggregator, (2) Basic Predictors, (3) Monitor, (4) Controller, and (5) Consumer.

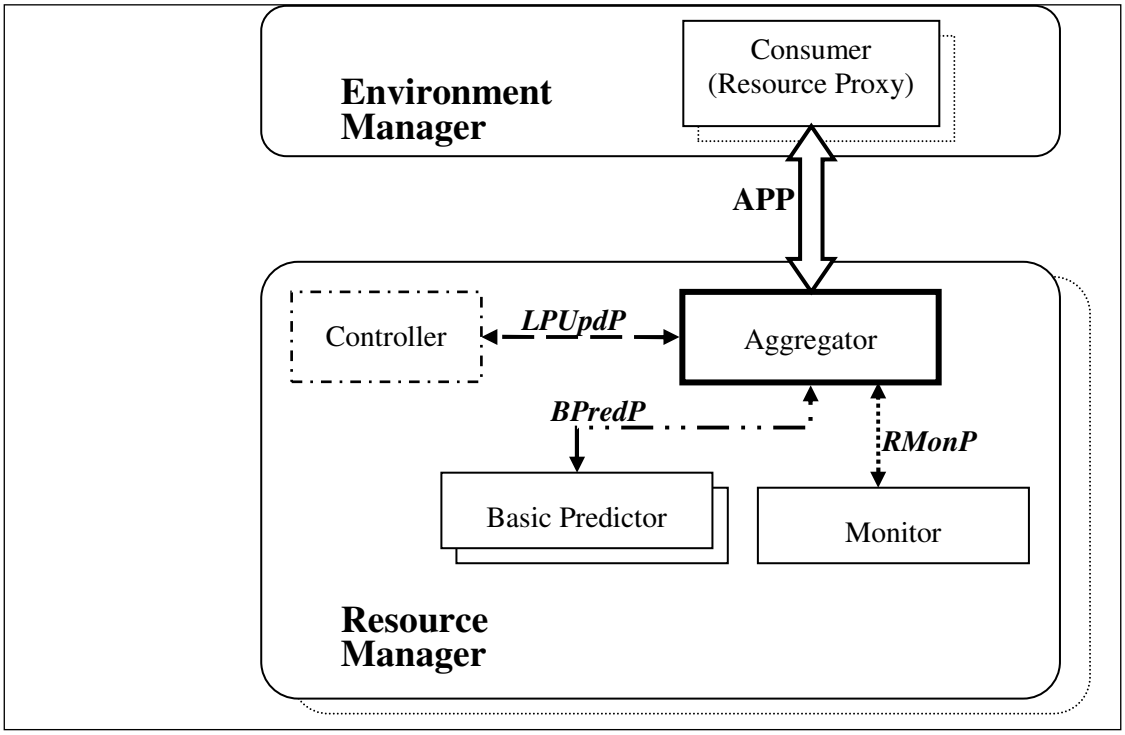


Figure 6: The internal structure of a Resource Manager. The Environment Manager is shown to place the design of the Resource Manager in the context of the Aura architecture.

The roles of the components are as follows:

- the Aggregator is the centerpiece of the prediction framework, responsible for combining information from all available Basic Predictors and calculating aggregate predictions. The Aggregator maintains an up-to-date list of currently available Basic Predictors,

- The Controller allows setting the model parameters of the linear recent history predictor in the Aggregator. The model parameters are expected to be relatively stable over time, changing only infrequently. There is one Controller resource instance,

- A Basic Predictor component implements a wrapper around either a known pattern or a bounding predictor. The design of the architecture allows for multiple Basic Predictors. Upon startup, a Basic Predictor registers with the Aggregator. As new sources of predictions become available, additional Basic Predictors can be added to the framework,

- The Monitor probes the environment for actual resource availability and provides periodic monitoring reports to the Aggregator. These monitored values correspond to the values of the series, $\{Y_t\}$. A Monitor provides a uniform interface to the Aggregator, encapsulating platform, network, and resource-specific details,

- A Consumer is the recipient and beneficiary of aggregate predictions. A Consumer is implemented by the coordinating entity of an adaptive resource management system, e.g., the Environment Manager. The prediction framework allows multiple concurrent Consumers to co-exist, each with its own aggregate prediction session.

There are four connector types in the architecture. Each connector defines a communication protocol between two components, as follows:

- Resource Monitoring Protocol (RMonP) is used between the Monitor and the Aggregator for reporting actual resource availability. The reporting of monitored values is organized around a session. The Aggregator initiates a session by sending a *startMonReports* message. The Monitor responds with periodic *monReport* messages until requested to stop,

- Basic Prediction Protocol (BPredP) is used between a Basic Predictor and the Aggregator. This protocol allows a basic predictor to registers its availability and parameters by sending a *registerPredictor* message to the Aggregator. Prediction reporting is organized around a reporting session. The Aggregator request prediction reports by sending a *startPredReports* message. The Basic Predictor acknowledges the request by sending *ackPredReports*. It then starts the periodic delivery of report messages, *predReport*, until requested to stop,

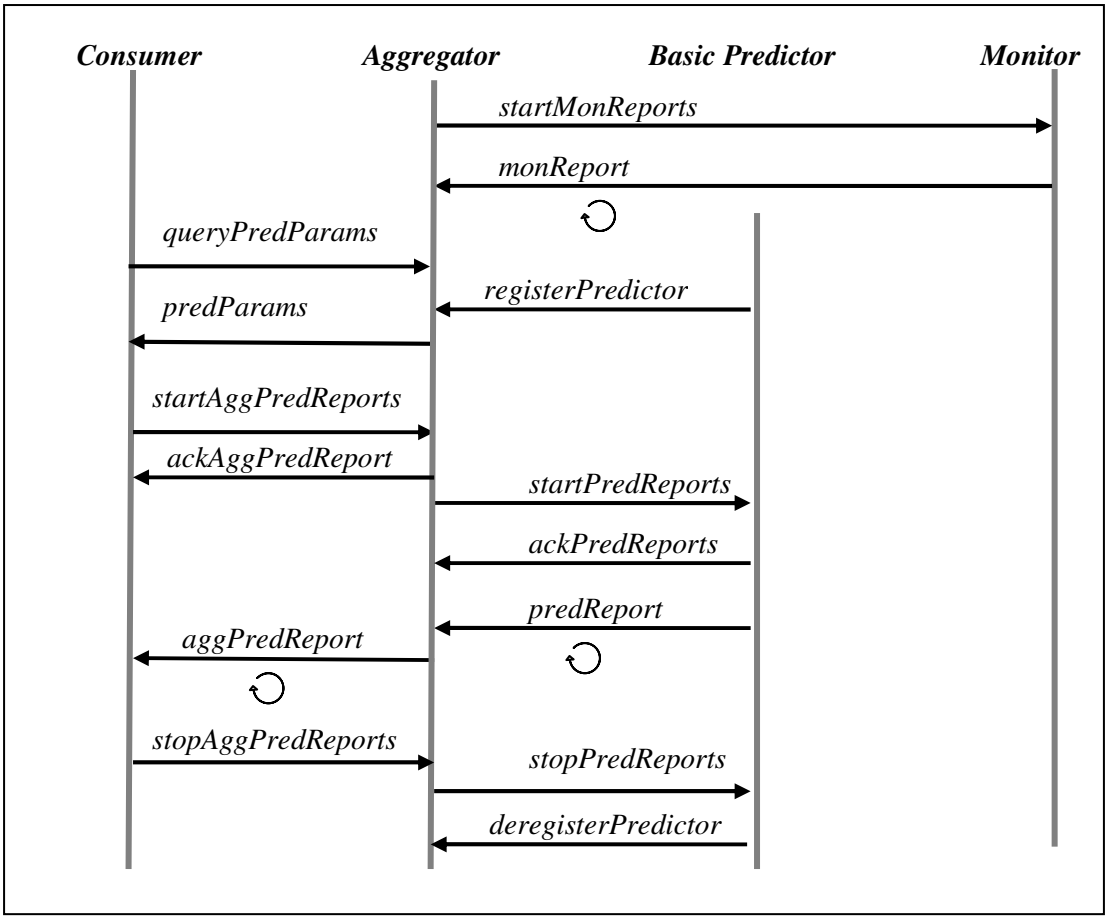


Figure 7: A sequence of messages showing the interactions between the consumer, the aggregator, the basic predictor, and the resource monitor.

- Linear Predictor Update Protocol (LPUpdP) is used by the Controller to update the model parameters of the recent history linear predictor that are housed in the Aggregator,

- Aggregate Prediction Protocol (APP) allows a Consumer to request aggregate predictions from the Aggregator. This high-level protocol is visible in the Aura architecture. The Consumer sends a *queryPredParams* message to query the range of available values of the parameters such as the prediction scale and horizon. The Aggregator responds by sending a *predictorParams* message that contains the supported range of the parameter values. The Consumer requests a reporting session by sending a message, *startAggPredReports*. The Aggregator initiates the session by sending an acknowledgement, *ackAggPredReport*. To help calculate predictions, the Aggregator uses the BPredP protocol to request basic prediction reports from all available Basic Predictors,

Figure 7 shows an event sequence diagram of the messages exchanged between the components of the framework. The use case shows the Consumer request aggregated predictions. The components in the predictor framework calculate periodic prediction reports and send to the Consumer.

8.4 Aggregate Prediction Protocol: an API for Resource Predictions

The Aggregate Prediction Protocol defines the interface between a prediction Consumer and the resource prediction framework. The functional requirements of the protocol are derived from the requirements of the architecture:

- (F-APP1) The Aggregate Prediction Protocol must provide periodic resource availability prediction reports consistent with the resource prediction model and operational semantics defined in section 6.7, upon the request of a Prediction Consumer,

- (F-APP2) The Aggregate Prediction Protocol must allow the Consumer to control the parameters of the prediction reports, as long as these parameters are in the supported range.

The design of the protocol must also promote all four extra-functional attributes identified in the beginning of the chapter.

The protocol has 4 steps: (1) query of prediction parameters, (2) establishment of a prediction reporting session, (3) reporting of predictions, and (4) termination of the reporting session.

The query step provides the consumer with an opportunity to discover the range of the prediction parameters and any dependence that might exist between them. As described in section 6.3, these parameters control the prediction scale, horizon, and detail. There is statistical evidence that the prediction scale and the standard deviation of the prediction errors are dependent (see, for example [49]). Thus, different prediction scales can have different prediction errors, and choosing the prediction scale determines the prediction error. Although there is no analytical relationship between the two, it is possible to express the relationship as an enumeration of tuples. In cases when the Consumer has the flexibility to select a prediction scale, the Consumer might decide to select a prediction scale that minimizes the standard deviation of the prediction errors.

The query step works as follows. First, the Consumer sends a *queryPredParams* message to the Aggregator, identifying the resource in question. The first three lines in Figure 8 show a sample query message. Next, the Aggregator responds with a *predictorParams* message, that contains en enumeration of dependent parameter tuples as well as the supported range of the remaining parameters. The rest of lines in Figure 8 show a sample *predictorParams* message.

In step 2, the Consumer establishes a prediction reporting session with the Aggregator by sending a *startAggPredReports* (see Figure 9). In this message, the Consumer must specify the prediction scale, horizon, and detail. To choose a desired scale and horizon, the Consumer specifies value from the allowed range for the parameters, *scale* and *horizon*, respectively. The detail of predictions is controlled using two parameters: *branchDepth* and *branchFactor*.

```
<queryPredParams>
   <predictor resourceId="192.168.1.101/willow/Network/1/IPBandwidthDown“/>
</queryPredParams >
```

```
<predictorParams>
   <predictor resourceId=”192.168.1.101/willow/Network/1/IPBandwidthDown“>
        <tuple param1=”scale” unit1=“seconds” param2=“sigma” unit2=“percent”>
           <values>
             <value>0.5 14.3</value>
             <value>1 12.1</value>
             <value>2 11.9</value>
             <value>5 11.6</value>
             <value>10 10.7</value>
             <value>15 10.3</value>
             <value>20 11.2</value>
             <value>40 11.6</value>
           <values/>
        <tuple/>
        <horizon unit="seconds" type=“range” from=“1” to=“2000”/>
        <branchDepth unit=“none” type=“range” from=“1” to=“10”/>
        <branchFactor unit=“none” type=“range” from=“1” to=“6”/>
   </predictor>
<predictorParams>
```

Figure 9: Messages for querying the range of and dependence among the aggregator's parameters.

```
<startAggPredReports>
   <predictor token="randomTok“ type="aggregate"
         resourceId="192.168.1.101/willow/Network/1/IPBandwidthDown">
                  <horizon unit="seconds" value="500"/>
                  <scale unit="seconds" value="20"/>
                  <branchDepth="2" unit=“none"/>
                  <branchFactor=“3" unit=“none"/>
   </predictor>
</startAggPredReports>
```

Figure 8: Consumer initiates a prediction reporting session by sending a startAggPredReports message.

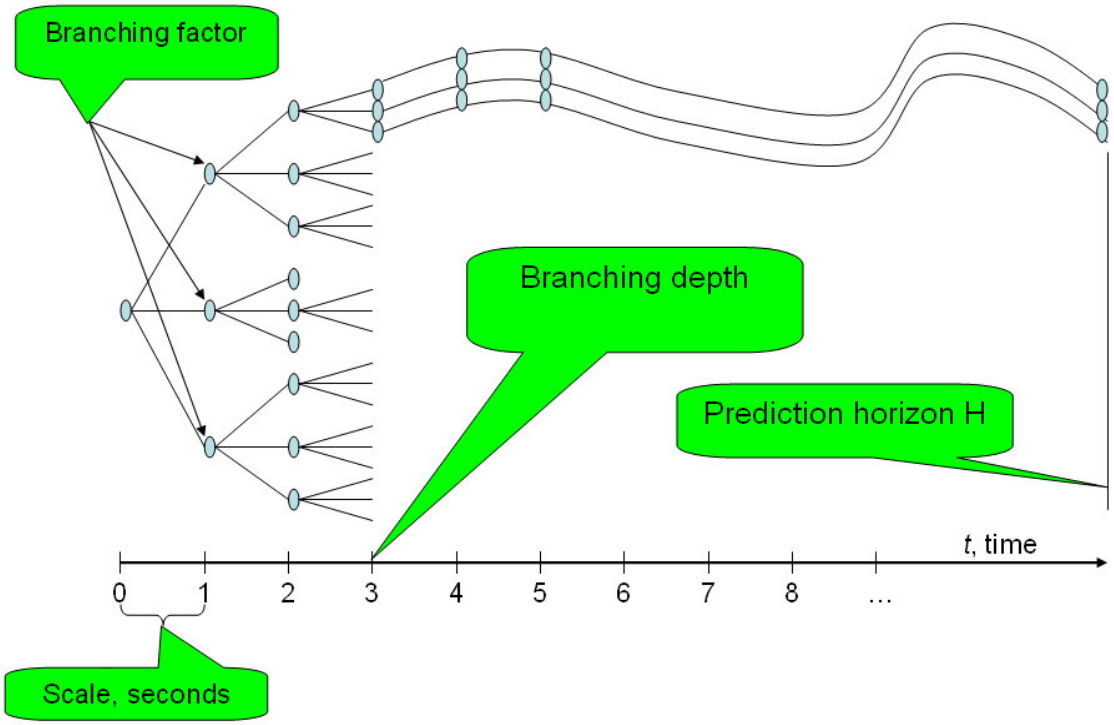


Figure 10: This is a graphical representation of an aggregate prediction tree. The hybrid tree starts with branching factor *BF*, but after depth *BD* it reverts to a line. The green callouts highlight the parameters are chosen by the consumer and that control the shape, length, and detail of the tree.

Figure 10 shows a graphical representation of aggregate prediction. The callouts in green highlight the key parameters that control the horizon, scale, and detail of predictions. Let's discuss how these parameters affect the cost of obtaining, transporting, and processing predictions. Let:

- ***S*** be the scale of prediction in seconds,
- ***H*** be the horizon of prediction in seconds. ***H*** must be divisible by ***S***,
- ***BF*** be the branching factor of the first part of the tree,
- ***D*** be the depth of that first part. ***D*** must be less than or equal ***H/S***,

Let **nWin = *H* / *S***. This is the total depth of a prediction tree. The number of points required to describe the prediction tree is the sum of the following terms:

- $1+BF^2+BF^3+ \ldots +BF^D + BF^D * (nWin - D) = (BF^{D+1} - 1) / (BF - 1) + BF^D * (nWin - D)$.

That sum is on the order of BF^D with a constant factor that is very close to *nWin - BD +1*. As *BF* and *D* increase, so does the cost of calculating, transporting, and using predictions. A Consumer

```
<ackAggPredReport>
   <predictor aggPredSessionId="uniqueSessionId1“ token="randomTok“ type="aggregate"
            resourceId="192.168.1.101/willow/Network/1/IPBandwidthDown">
         <probabilityTree>
            <!-- this shows the probabilities at each node in the tree, total of
                 1 + factor + factor^2 + … + factor^depth numbers
                   Level_0: prob0
                   Level_1: prob1, prob2, prob3
                   Level_2: prob11, prob12, prob13
                              prob21, prob22, prob23
                              prob31, prob32, prob33, etc -->
                 prob0, prob1, prob2, prob3
                 prob11, prob12, prob13, prob21, prob22, prob23, prob31, prob32, prob33
         </probabilityTree>
   </predictor>
</ackAggPredReport>
```

```
<aggPredReport><predictor …>
         <valueTree>
                   val0
                   val1, val2, val3
                   val11, val12, val13
                   val21, val22, val23
                   val31, val32, val33
                   ....
         </valueTree>
         <meanPaths>
                   val11_3, val11_4, val11_5,  ..., val11_25
                   val12_3, val12_4, val12_5,  ..., val12_25
                   val13_3, val13_4, val13_5,  ..., val13_25
                   val21_3, val21_4, val21_5,  ..., val21_25
         </meanPaths>
</predictor></aggPredReport>
```

Figure 11: Messages showing requests for aggregate predictions and reports.

can use the above calculations to select prediction parameters that achieve an acceptable cost-benefit in terms of reduce network resource usage and prediction detail.

In response to the Consumer's request to establish a reporting session, the Aggregator sends an acknowledgement, *ackAggPredReports*. This message confirms that the Aggregator and the Consumer have agreed upon the prediction parameters. Once these parameters are set, the shape of the prediction tree is fixed. Therefore, the predicted values and their probabilities can be transmitted using an array. The Aggregator can build the array from a tree structure, and the Consumer can reconstruct the tree from an array.

The *ackAggPredReports* message contains the conditional probabilities of the nodes in the prediction tree as one array. By reporting these probabilities only once, at the beginning of a session, we reduce the size of the prediction reports by a factor of two. This helps reduce network usage between the Consumer and the Aggregator.

After acknowledging the session, the Aggregator starts the periodic delivery of *aggPredReports* messages that contain prediction reports. Each prediction report contains two arrays. The first array contains the values of the earlier nodes in the prediction tree (see Figure 10). The Consumer collates the values in this array with the probabilities received in the acknowledgment message. The second array contains the values of the mean predicted paths for the later nodes in the prediction tree.

The Consumer uses the values in these arrays to reconstruct the prediction tree. Figure 11 shows sample *ackAggPredReports* and *aggPredReports* messages.

8.5 Discussion of Key Design Decisions and Their Alternatives

We faced a number of alternative choices in the course of designing the overall architecture of Aura as well as the various components. In this section we reveal the decision rationale behind key decisions.

We decided to make the Supplier a high-level architectural component type, and locate the Suppliers outside of the execution process of the Environment Manager. In an early prototype of Aura, Suppliers were implemented as dynamic link library modules and were loaded from the same process as the Environment Manager. The alternative we chose has provided a number of benefits:

- Great latitude in the deployment of suppliers, promoting distributed computing and extensibility. Suppliers do not have to be located on the same hardware as the Environment, allowing for flexibility in drawing environment boundaries. In terms of runtime extensibility, as new Suppliers can enter the Environment, register with the EM and become available to the user. In terms of maintainability, if more sophisticated discovery protocols become available, they can be implemented on top of the existing architecture much easier. We have some evidence in support of this claim. We have extended the architecture to support security, authentication, and encryption, and the decision to separate the Suppliers into a separate component has proven to be useful. The discussion of those additional features is beyond the scope of this thesis,

- Loose coupling between the Environment Manager and the applications. This prevents the Environment Manager from crashing should any applications or Suppliers crash. The availability of the system is improved,

- Inter-operability across heterogeneous implementation and runtime platforms. A Supplier provides a standard control and monitoring interface to a wide range of applications, ensuring that the architecture can work with a wide range of platforms,

- Simplified design of the Environment Manager and the Task Manager. A simpler design has helped implement many of the functional requirements of the EM and the TM with

less implementation effort. For example, the EM is able to handle a number of failure scenarios as a normal case, requiring no special handling,

In the design of the EM Core, we separated the data from the calculations. Both the Optimization Engine and the Instantiation Engine are designed as stateless objects, promoting the maintainability of the EM. When new features are needed or better algorithms become available, this separation helps reduce maintenance cost. Although never implemented, an alternative design which did not separate the data from the calculations would greatly complicate the maintenance of the Core of the EM.

In the course of the design of the Resource Manager and the Aggregate Prediction Framework, we were faced with a number of design alternatives. Although the internal structure of the resource prediction framework is tied to the prediction model defined in section 6.7.3, the aggregate prediction reporting (APP) interface is independent of the model. In order to make this possible, we introduce the Aggregator component type into the prediction framework and placed all the required calculations inside the Aggregator. The aggregate prediction interface between the Consumer and the Aggregator is independent of the prediction model. Thus, changes to the model or the resource prediction framework are less likely to affect the Consumer. An alternative approach was to pass prediction formulas to the Consumer, requiring it to perform calculations that are specific to the prediction model. This alternative had some merits, and we strongly considered it. It would have reduced the transmission cost of predictions, but at the cost of tight coupling between the Consumer and the prediction framework. Any changes to the prediction model would propagate to the Consumer, requiring potentially costly maintenance updates to Consumers.

We recognized the distinguished role of the linear history predictor, and decided to house the predictor parameters in the Aggregator while allowing changes to the parameters using the LPUpdP protocol. Our initial design provided a separate component type, Linear Predictor, for the linear history predictor and called for a separate protocol between it and the Aggregator. However, the latter alternative did not add useful functionality, while greatly increasing the complexity of the architecture and the runtime resource cost of transmitting predictions. With the chosen alternative, the calculation of aggregate predictions is only one tier removed from the Consumer, helping promote performance. Resource costs are reduced, while scalability of the resource prediction framework to multiple Consumers is increased.

Adding the query step in the Aggregate Prediction Protocol has provided a great deal of flexibility to the Consumer, by exposing prediction cost-accuracy trade-offs to the Consumer. Earlier, we considered an alternative that did not have the query step. While this alternative would reduce the communication by one round-trip message, the cost of the predictions would not be visible to the Consumer. Some readers might rightfully argue that the current query interface gives the Consumer too many choices when selecting prediction parameters. This can be remedied by providing a Consumer-side client library with a simplified query interface. Consumers that are not interested in exploring the range of the parameter space can simply use the simplified client library to select the parameter values for the prediction session.

We believe that the Aggregate Prediction Protocol strikes a fine balance between simplicity and expressiveness, exposing critical details to the Consumer without overly complicating the prediction request interface.

8.6 Mapping the Architecture to a Control-theoretic Framework

Aura is an adaptive system that uses the preferences of the user to control the environment. We demonstrate how Aura can be described as a feedback-control system. A common feedback-control system is a linear feedback system in which a control loop, including sensors, controls algorithms and actuators, is arranged in such a fashion as to try to regulate a variable at a set-point or reference value.

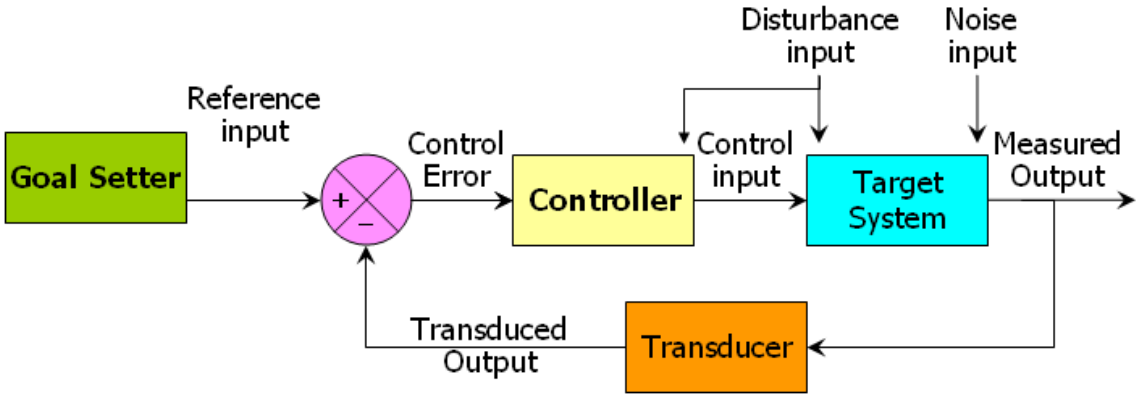


Figure 12: A Linear feedback-control system.

Figure 12 shows a typical feedback-control system. In such a system, the Goal Setter provides a goal, the Reference Input, which is fed into Controller. The Controller controls the Target System using actuators. The actuators are not shown in the diagram, but their effect corresponds to the Control Input. The state of the Target System is measured using sensors, also not shown in the diagram. The Measured Output is fed back into the system. There are two optional elements. First, the Controller can make use of Disturbance Input. Second, the Measured Output can be Transduced before feeding back into the system.

Figure 13: A mapping between the elements of Aura and a typical feedback control system.

Aura	Feedback_Control System
Task Manager	Goal Setter
Environment Manager	Controller
Suppliers	Target System Sensors and Actuators
Resource Managers	Disturbance Input Sensors
The Environment	Target System
Preferences	Reference Input
QoS Set-points	Control Input
QoS Output Monitoring	Measured Output
Resource Monitoring	Disturbance Input

There is a natural correspondence between the key elements (components, inputs and outputs) of Aura and the key elements of a feedback control system. This correspondence is show in Figure 13. Several of the Aura components perform transducing operations. For example, Suppliers monitor the application QoS and report the moving average of the QoS to the Environment Manager. Resource Managers also calculate moving averages when making predictions. Therefore, the Aura architecture can be viewed as a feedback control system.

8.7 Chapter Summary

In this chapter, we have described the architecture of Aura, an infrastructure for task management and automatic configuration. The organization of Aura into layers exploits the structure in the problem that we defined in analytical models in Chapters 5 and 7. The Task Management layer is responsible for determining what is needed for the user's tasks. The Environment Management layer is responsible for configuration decisions. The Environment layer is responsible for supports user's tasks.

This thesis makes three contributions to Aura: (1) the design of the Environment Manager, (2) design of the resource prediction framework, and (3) the design of a resource prediction API between the prediction framework and the Environment Manager. We have motivated the design of the two components and API using a number of functional requirements and extra-functional attributes. We have described the detailed design of the two components and the API. We have then provided the rationale for the key design decisions.

9 Algorithms for Automatic Configuration

In this section, we describe the optimization algorithms that comprise the core computations of the Environment Manager. These algorithms are designed to address the technical challenges described in the Introduction by exploiting the structure of the reactive and anticipatory models of configuration described in the earlier chapters.

9.1 Staging the Problem

For tractability of computation, we have organized the optimization problems into a number of stages in terms of increasing difficulty. At the initial stage, we ignore uncertainty and exclude resources such as battery. This results in an optimization problem that is easier to solve. Then, we gradually make the problem more general by allowing predictions, uncertainty, and more resources. The end goal is to formulate and solve the most general problem instance possible within the definition of the anticipatory model of configuration. To help organize the staging, we can control one or more of the following parameters and features:

- The length of the decision-making horizon,
- The amount of information available in the inputs,
- The way uncertainty in the predictions is dealt with,
- The inclusion of different resource types.

In the reactive model, the decision horizon is equal to one time step. In the anticipatory model, the decision-making horizon can be any number of time-steps. In practice, because of the increasing cost of computations required for a longer time horizon, we have to limit the maximum decision horizon to keep the problem tractable. To stage the problem, we can consider two cases of the value of the decision horizon: (a) one and (b) greater than one. When the decision making horizon is one time step, having only the current snapshots of the inputs variables is sufficient to formulate a problem. Put in a different way, having access to predictive inputs will not improve the best possible solution attainable.

On the other hand, when the decision horizon is larger than one, predictive inputs are required. Predictions must be as far advance into the future as the decision horizon. Recall that there are two sources of temporal predictions, corresponding to two of the three inputs in the models: (1) task request operations and (2) resource availability. One way to simplify the problem is to use predictions, but ignore the uncertainty in the predictions. Analytically, ignoring the uncertainty in the predictions is equivalent to assuming that the predictions are perfect. In other words, there is no uncertainty in the predictions *ex ante*, and *ex post* realized values of the inputs are exactly equal to the predicted values.

We can gradually stage the problem by choosing to ignore the uncertainty in the predictions of the two inputs. First, we can ignore the uncertainty in both task request predictions and resource

predictions, assuming predictions for both are *ex ante* perfect and *ex post* accurate. Next, we can ignore the uncertainty in the predictions of task requests, while explicitly modeling and coping with the uncertainty in the resource predictions. And lastly, we . In this way, we have three problem stages of increasing complexity.

The last consideration for staging the problem is including different types of resources in the analysis. The non-perishable, non-renewable resources like battery have a different prediction model than resource like bandwidth. The future availability of battery is affected by present consumption rate. In case of a resource such as bandwidth that relationship does not hold. In other words, the availability of the future level of non-perishable, non-renewable resources is not *exogenous*. We can simplify and stage the problem by first excluding all non-perishable, non-renewable resources from the analysis. Then, we can include all non-perishable, non-renewable resources, obtaining a new stage of the problem.

Table 5: Problem stages.

Stage	Decision-Horizon	Predictions		Resources
		Resource Availability	Task	Non-perishable
1	1	None	None	No
2	> 1	Perfect	Perfect	No
3.1	> 1	Uncertainty	Perfect	No
3.2	> 1	Perfect	Uncertainty	No
3.3	> 1	Perfect	Perfect	Yes
4	> 1	Uncertainty	Uncertainty	No
5	> 1	Uncertainty	Uncertainty	Yes

The above table shows the problem stages. For example, in stage 1, the parameters of the problem are set as follows:

- decision-making horizon is equal to 1,
- inputs are snapshots of currently known values, i.e. no predictions are available,
- no uncertainty, as there are no predictions,
- no non-perishable resources.

The stage 1 problem is the only one in the context of the reactive model of configuration. The remaining stages are in the context of the anticipatory model. The problems in stages 3.1, 3.2, and 3.3 differ from the stage 2 problem in only column. In this way, we allow complexity to be introduced into the problem one dimension at a time. In stage 4, we combine the incremental effects from stages 3.1 and 3.2. And at last, in stage 5, we combine all three effects, creating the most general problem in the context of the anticipatory configuration strategy.

9.2 Stage 1

We provide three different approaches to solving the stage 1 problem. We discuss their advantages and disadvantages, and describe how well they are suited to solving the later stage problems.

Recall the objective function defined by the reactive model from section 5.4:

$$\underset{\substack{a_i \in Supp(S_i) \\ q_{a_i} \in QoSprof(a_i)}}{\arg\max} \quad \sum_{i=1}^{n} \left(x_{S_i} h_{S_i} + W_{S_i} F_{S_i}(a_i) + \sum_{\forall d \in QoS\dim(S_i)} w_{S_i,d} f_{S_i,d}(q_{a_i}[d]) \right)$$

We can re-write the double-sum in the sum can further be re-written as follows:

$$\sum_{i=1}^{n} \left(x_{S_i} h_{S_i} + W_{S_i} F_{S_i}(a_i) \right) + \sum_{i=1}^{n} \sum_{\forall d \in QoS\dim(S_i)} w_{S_i,d} f_{S_i,d}(q_{a_i}[d])$$

We have separated the first two parts in the outer sum from the third part. The first part is the supplier change penalty, the second part is the supplier utility, and the third part is the QoS utility. Recall that a *supplier assignment* is a set of suppliers, one per service in a task. The first and second terms in the above sum depend only on the choice of the suppliers in the supplier assignment. This observation suggests an approach to finding the best configuration and the maximum possible utility. This approach would evaluate configurations based on the partial utility from the assignment of the suppliers first, and then further evaluate *some* of the configurations based on QoS utility.

Now we are ready to describe some basic routines that calculate the first two terms in the above sum. These routines are required by the algorithm for calculating the optimal configuration. It will benefit the reader to enumerate the inputs to the problem:

- Task *T*. This is the description of the task and corresponds to the first input in the reactive model of configuration,

- List <Supplier> *AvailableSuppliers*. This is the list of the available suppliers and their corresponding QoS-resource profiles, and corresponds to the second input in the reactive model of configuration,

- ResourceAvailabilityVector *res*. This is the current snapshot of the available level of the resources and corresponds to the third input in the reactive model of the configuration.

`GenAllSA`:

- **input** Task *T*, **input** List<Supplier> *AvailableSuppliers*,
- **output** SupplierAssignment[1..numSA] *AllSupplierAssignments*,

This routine is implemented as an ad-hoc combinatorial enumeration and generates the list of all supplier assignments given for a task, *T*.

`CalcSuppUtil`:

- **input** Task *T*, **input** SupplierAssignment[1..N] *allSAs*,
- **output** real[1..nSA] *arraySuppUtilPerSA*,

This routine is implemented by summing the supplier affinity utility for the suppliers in the supplier assignment. Supplier preference functions are provided by task *T*.

`CalcChPen`:

- **input** Task *T,* **input** SupplierAssignment[1..N] *allSAs*,
- **output** real[1..nSA] [1..nSA] *arrChangePenaltyPerSAPair*,

This routine calculates the change penalty in the hypothetical scenario when the configuration selected in the previous time window was based on supplier assignment SA1, and we are considering replacing it with another one, SA1. This routine can be implemented as a doubly nested loop, iterating over the list of the supplier assignments. Change penalty scores are provided according to the penalty preferences in Task *T*, and that's why *T* appears in the list of inputs.

To help analyze the running times of the above routines, we introduce the following variables.

- ***nSvc***, the number of services in the task,
- ***nAltSupp***, the average number of alternative suppliers available per service type,
- ***nSuppTotal*,** the total number of suppliers available in the environment,
- ***nSA*,** the number of supplier assignments. This number is O(***nAltSupp*^ *nSvc***).

`GenAllSA` has running time O(***nSvc * nSA***) = O(***nSvc * nAltSupp^nSvc***).

`CalcSuppUtil` has running time O(***nSvc * nSA***).

`CalcChPen` has running time O(***nSA * nSA***).

9.2.1 Sub-problem: calculating maximum QoS utility for a supplier assignment

In the expression for overall utility, the third term is the QoS utility. In order to solve the problem of maximizing overall utility under resource constraints, we need to solve the sub-problem of maximizing QoS utility for a fixed supplier assignment. We will refer to this sub-problem instance as MaxQoS.

This is a well-studied problem and is a variant of the famous Knapsack packing problem. Per our formulation, the problem is a *multi-dimensional*, *multiple-choice* knapsack problem. Multi-dimensional refers to the multiple resources, each of which provides a capacity constraint. Multiple-choice refers to the choices made available by the quality-resource tuples in the profile of each supplier. Selecting an optimal configuration requires selecting one quality set-point in the profile of each supplier. This further justifies using the label, multiple-choice, to describe the variant of the Knapsack problem we have.

Before we describe existing solutions, let's consider a contrived example with a complete description of the task, including the preference functions, as well as a complete description of the supplier profiles. This example is a slightly different version of this one introduced in section 5.1.2.

Task *T* has 2 services: $\{s_1, s_2\}$. s_1 has service type "*play video*" and 2 QoS dimensions: "*frame rate*" and "*frame size*". s_2 has service type "*show Pictures*" and 2 QoS dimensions: "*latency*" and "*picture quality*". The QoS preferences of the task are shown in Table 6.

Table 6: The description of a task, including preference functions.

Service 1, type: "*play Video*"	
QoS Dim 1	
Name:	*frame size*
Units:	pixels
Weight:	0.2
Function	f_size(460K) = 1 f_size(250K) = 0.8 f_size(64K) = 0.5
QoS Dim 2	
Name:	*frame rate*
Units:	frames per sec.
Weight	0.4
Function	f_fr(20) = 1 f_fr(10) = 0.4

Service 2, type: "*show Pictures*"	
QoS Dim 1	
Name:	*latency*
Units:	seconds
Weight:	0.3
Function	f_lat(Fast) = 1.0 f_lat(Slow) = 0.3
QoS Dim 2	
Name:	*page quality*
Units:	None
Weight:	0.1
Function	f_pq(High) = 1.0 f_pq(Med) = 0.9 f_pq(Low) = 0.5

According to the task description, the formula for QoS utility is:

$$(0.2 * F\_size(.) + 0.4 * F\_fr(.)) + (0.3 * F\_lat(.) + 0.1 * F\_pq(.)),$$

where the 4 preference functions are defined according to Table 6.

Next, assume that we have found a supplier assignment satisfies this task: $\{Supp_1, Supp_2\}$. $Supp_1$ is a video player from vendor A, and $Supp_2$ is a web-browser from vendor B. For simplicity, assume each supplier has only 6 points in its profile, shown in Table 7 (this example considers 2 resources: CPU cycles and downstream bandwidth):

Table 7: The QoS-resource profiles of two suppliers shown side-by-side.

Supplier 1, provides "*play Video*"		
Utility	QoS	Resource
0.60	(460K,20)	(75,800)
0.56	(250K,20)	(60,600)
0.50	(64K, 20)	(45,400)
0.36	(460K,10)	(50,450)
0.32	(250K,10)	(38,350)
0.26	(64K, 10)	(25,250)

Supplier 2, provides "*show Pictures*"		
Utility	QoS	Resource
0.40	(Fast, High)	(20,200)
0.39	(Fast, Med)	(18,100)
0.35	(Fast, Low)	(16,50)
0.19	(Slow, High)	(20,100)
0.18	(Slow, Med)	(18,50)
0.14	(Slow, Low)	(16,25)

The second column of each profile shows the level QoS that the supplier can provide, as a vector, while the third column shows the resource requirement for that QoS level. In the first column, we have calculated the portion of QoS utility that is due to that point. For example, the utility of the point (20K, 10) in the first suppliers profile is calculated as follows:

$$0.2 * F\_size(20\text{K}) + 0.4 * F\_fr(10) = 0.2 * 0.8 + 0.4 * 0.4 = 0.32$$

The general formula for calculating the contribution towards QoS utility of one supplier is this:

$$\sum_{d \in QoS\ \dim(S_1)} c_d F_d \left(q_{Supp_1, d} \right)$$

The following analogy might help the reader visualize the problem. Let's represent each supplier's profile as a bag of items. Each item represents a pair in the supplier's profile. Each item has an associated quality vector and a resource vector, corresponding to the QoS vector and the resource vector in the profile. Each item has a value that can be calculated by applying the preferences to the quality vector of that item (the result of this calculation is recorded in the first column of each supplier's profile in Table 7).

The problem of maximizing QoS utility under resource constraints becomes equivalent to selecting exactly one item from each bag such that:

- the sum of the values of the selected items is maximized,
- the vector sum of the resource cost of those items does not exceed resource supply in any dimension.

Table 8: This table shows all possibilities of combining one point from each supplier's profile.

(61,440)	1	2	3	4	5	6
1	**1.00**	**0.99**	**0.95**	**0.79**	**0.78**	**0.74**
2	**0.96**	**0.95**	**0.91**	**0.75**	**0.74**	**0.70**
3	**0.90**	**0.89**	**0.85**	**0.69**	**0.68**	**0.64**
4	**0.76**	**0.75**	**0.71**	**0.55**	**0.54**	**0.50**
5	**0.72**	**0.71**	**0.67**	**0.51**	**0.50**	**0.46**
6	**0.66**	**0.65**	**0.61**	**0.45**	**0.44**	**0.40**

In Table 8, we have shown all combinatorial possibilities of selecting one point from each supplier's profile. The rows are the 6 points from the profile of $Supp_1$. The problem of maximizing QoS utility requires selecting one cell in the table out of 6 * 6 = 36 possibilities. In the upper-left corner of the table we have recorded a hypothetical resource constraint: (61, 440), showing that the available supply of the first, respectively second, resource is 61, respectively 440, units. Shaded cells (when printed in color the shading is red) are not feasible, because the combined resource requirement exceeds the supply of at least one of the resources. Among the cells available, the maximum utility is 0.67, realized using the QoS points (20K, 10) from $Supp_1$ and (Fast, Low) from $Supp_2$. Thus, the best possible QoS utility is achieved when $Supp_1$ provides the video at 20K frame size and 10 frames per second rate, and $Supp_2$ provides the slide show at a Fast rate and Low picture quality.

There are two well-known solution approaches for solving the multidimensional, multiple-choice Knapsack problem [34,46]. The first approach is based on a *greedy* selection tactic and is suited for solving the problem for one resource supply vector. The second approach is based on *dynamic programming* and is better suited for solving the MaxQoS problem for all possible resource supply vectors. Neither approach is guaranteed to find the maximum possible utility, because this variant of the Knapsack problem is NP-complete [34].

Both of the approaches are described in detail in Lee's thesis [34]. The greedy selection approach is called AMRMD1 and the dynamic programming approach is called MRMD. Below, we provide the intuition behind the two algorithms, without producing any code.

In the approach using a greedy selection tactic, the quality-resource points from each supplier profile are placed in a two-dimensional space according to their value and cost. The metric for value is calculated according to the preference functions, exactly as shown in column 1 of Table 7. The metric for cost is calculated using a weighed average (e.g., quadratic) of the dimension-wise resource cost.

A convex hull of the points is calculated, providing a frontier of the most efficient points in terms of the value/cost ratio. Such a frontier is generated separately for each supplier in the assignment, using the points from the profile of each supplier.

The greedy algorithm operates by selecting certain efficient points from the convex frontiers of the different suppliers, ensuring that at least one point is selected from each supplier. Initially, the algorithm picks the point with the highest slope (value / cost ratio) from among all suppliers. The algorithm records the resource requirement, subtracting that amount from the supply. If there is a spare amount of resource remaining, the algorithm continues. The algorithm chooses the best possible "upgrade" available, substituting one of the currently selected points with the next one on the same frontier. The algorithm continues this until the resource is exhausted, or the next upgrade will cause the resource requirement to exceed the remaining supply.

The second approach solves the MaxQoS problem for all possible values of the available level of resources using dynamic programming.

First, for each resource, we determine a minimum allocation unit and create the possible values of the resources in multiples of the minimum allocation unit. For example, by choosing a minimum allocation unit of 4% for CPU, we can get 21 possible allocation levels, from 4% to 100% inclusive. For downstream bandwidth, assuming a maximum available level of bandwidth of 760 Kbps, and an allocation unit of 20 Kpbs, we get 39 possible resource levels. We create a resource sieve by combining all possible values of the two resources, giving us 21 * 39 = 819 possible resource vectors.

Next, we produce partial sets of the suppliers from the current supplier assignment. For example, when there are three suppliers in the assignment, $\{Supp_1, Supp_2, Supp_3\}$, we generate the following partial sets: $\{Supp_1\}$, $\{Supp_1, Supp_2\}$, and $\{Supp_1, Supp_2, Supp_3\}$. The dynamic programming algorithm will solve the MaxQoS problem for all possible resource states and all partial sets of the task (although the solution for the two partial sets, $\{Supp_1\}$, $\{Supp_1, Supp_2\}$, would be a throw-away).

For two resources, let *MaxQoS(i, p1, p2)* denote the value of maximum possible QoS utility possible when considering only the first *i* suppliers and the available level of the first resources is respectively *p1* and *p2*. Then the following recursive relationship holds:

$$MaxQoS(i, rAvl1, rAvl2) = max \{ BestOf(Supp_i(p1', p2') + MaxQoS(i-1, p1-p1', p2- p2')\}$$

where the maximum is taken over all allocation units *p1', p2'* such that $p1 \leq p1'$ and $p2 \leq p2'$.

In simple terms, the above recursive formula states that in order to determine the maximum QoS utility possible using the first *i* suppliers only ($Supp_1, Supp_2, \ldots, Supp_i$), we need consider allocating partial amounts (*p1', p2'*) of the available resources to $Supp_i$, and the remainder among the first *i-1* suppliers. Using dynamic programming, all smaller instances of the problem have been solved and cached, providing for the fast lookup of the expression:

$$MaxQoS(i-1, p1-p1', p2- p2')$$

In this thesis, we will refer to the dynamic programming solution as `DMaxQoS_ResSieve`, where "D" stands for dynamic programming.

Let's re-cap the algorithms that we have:

`AMaxQoS`:

- **input** Task *T*, **input** SupplierAssignment *SA1*, **input** ResourceAvailabilityVector *res*,
- **output** QoSetPoints *setPoints*, **output** real *maxQoSUtility*,

`DMaxQoS_ResSieve`:

- **input** Task *T*, **input** SupplierAssignment *SA1*, **input** ResourceAvailabilityVector[1..nResSieve] *resourceSieve*,
- **output** QoSetPoints[1..nResSieve] *arrSetPoints*, **output** real[1..nResSieve] *maxQoSUtilityArray.*

The output variables are arrays indexed by items from the resource sieve. For each vector in the resource sieve, the algorithm produces a solution: the maximum possible QoS utility and the QoS set-points for each supplier in the assignment.

To analyze the runtime performance of these algorithms, let's introduce the following variables:

- ***nSvc***, the number of services in the task (this was already defined earlier),
- ***nQoS***, the size of the profile of a typical supplier, i.e. the number of QoS-resource tuples that the supplier profile contains,
- ***nResources*****,** the number of resources in the environment,
- ***nResPoints*****,** the number of points in the sieve of a single resource, on average,
- ***nResSieve*****,** the size of the resource sieve. This number is O(***nResPoints*^ *nResources***).

`AMaxQoS` has a running time of O(***nSvc*** * log(***nQoS***) ***** ***nQoS***).

`DMaxQoS_ResSieve` has a running time of O(***nSvc*** * ***nQoS*** * ***nResSieve***).

9.2.2 Calculating maximum overall utility by Iterating over Supplier Assignments

In the beginning of this section, we decomposed the expression of overall utility into three terms. We saw that the first two terms: change penalty and supplier affinity utility, are dependent only on the choice of a candidate supplier assignment (change penalty also depends on the active configuration chosen in the previous time window, but that configuration is already fixed.) In order to solve the Stage 1 problem completely, we can calculate the maximum overall utility of each supplier assignment and choose the best one.

- Step 1: perform the pre-calculations: GenAllSA, CalcSuppUtil, and CalcChPen,
- Step 2 (loop): for each supplier assignment:

- find he maximum possible value of the QoS Utility given the current resource conditions (MaxQoSU),
- calculate and record overall utility (OU) as follows: CP + SU + MaxQoSU,
- keep track of the supplier assignment with the best OU so far,

- Step 3: declare the supplier assignment with OU as the winner.

The above algorithm can be further improved using a simple heuristic. In step 1, for each supplier assignment, we can add the two terms, CP and SU, and call this number the *supplier-level contribution* to overall utility. Then we can sort the supplier assignments in the decreasing order of their supplier-level contributions. The perfect QoS utility attainable (not just among the suppliers available) is bounded. That bound is simply the sum of the weights of the QoS Utility functions and can be calculated up-front, or can be normalized to 1. That bound can be used together with the supplier-level contribution from the higher scoring supplier assignments in order to eliminate certain lower-scoring supplier assignments entirely. We show the pseudocode of the algorithm in the exhibit below. In the pseudocode, calls to routines that we have described earlier are shown in bold typeface. Angle brackets, <>, are used to make tuples on the fly.

```
OptInstUtility(   input Task T,
                  input List<Supplier> availableSuppliers,
                  input SupplierAssignment currentSA,
                  input ResourceAvailabilityVector res)
    output SupplierAssignment bestSA,
    output QoSSetPoints qosSetPoints,
    output real maxOptUtil
{
    SupplierAssignment allSAs[] = GenAllSA(T, availableSuppliers);
    real suppUtilPerSA[] = CalcSuppUtil(T, allSAs);
    real chPenaltyPerSAPair[][] = CalcChPen(T, allSAs);
    real perfectQoSUtil = 1;
    real maxOptUtilSoFar = - INF;
    for each iSA in allSAs
    {
        <QoSSetpoints setpoints, real maxQoS> = AMaxQoS(T, iSA, res);
        real penalty = chPenaltyPerSAPair[currentSA, iSA];
        real ou = maxQoS + penalty + suppUtilPerSA[iSA];
        if ( maxOptUtilSoFar < ou )
        {
            maxOptUtilSoFar = ou;
            bestSA = iSA;
            qosSetPoints = setpoints;
        }
        if ( (maxOptUtilSoFar - perfectQoSUtil ) > maxQoSForThisSA )
            break;
    }
    maxOptUtil = maxOptUtilSoFar;
    return <bestSA, qosSetPoints, maxOptUtil>;
}
```

The worst case running time of `OptInstUtil` is O(***nSA***) * O(`AMaxQoS`) = O(***nSA*** * ***nSvc*** * log(***nQoS***) * ***nQoS***). `AMaxQoS` algorithm scales well with the increase in the size of the QoS profile of the suppliers, with a running time that is proportional to log(***nQoS***) * ***nQoS***. The other factor, O(***nSA***), is equal to O(***nAltSupp*** ^ ***nSvc***) and get potentially large with the increase in the number of available suppliers as well as the number of the active services in all active tasks. In the worst case, when all the suppliers are equally preferred by the user, the algorithm has to loop through each supplier assignment in order to find the best one.

On the other hand, if supplier preference scores are distributed uniformly, we have demonstrated that the early termination condition in the algorithm reduces the number of supplier assignments several orders of magnitude [48]. For example, if ***nAltSupp*** = 10, ***nSvc*** = 8, and maximum possible value of overall utility is at least .66 (out of the perfect value of 1), then the number of supplier assignments that the algorithm has to search is only about 330. By comparing that number with the value of ***nSA*** = 10^8 = 10,000,000, we see that the termination condition provides a significant speedup when there number of the supplier assignments is very large.

9.3 Stage 2: Anticipatory Strategy with Perfect Predictions

The problem in this stage is the simplest instance of a configuration problem that uses predictions. The decision-making horizon is greater than 1, and predictions are required to solve this problem. We assume that all predictions are perfect.

For simplicity, throughout this section we make some simplifying assumptions about the task and task requests:

- there is only one active task,
- the task is activated at time 0,
- the duration of the task is TD, expressed in prediction scale units.

The algorithm we are about to describe is general enough so that all of the above assumptions can be dropped. At the end of the section we will briefly describe how the algorithm can generalize to multiple tasks, and multiple activations / deactivations of the same task, as long as predictions of activation / deactivation times of all tasks are known in advance.

Because predictions are perfect, we can assume that predictions of resource availability are known at least till the duration of the task. This ensures that the decision-making horizon covers the active period of the task, from time 0 to TD.

Recall, from section 7.4, that the objective we need to maximize is the accrued utility:

$$\arg\max_{SEQ \in Set(Seq)} E\left[AU\Big|_{cT}^{cT+DH}(SEQ) \mid cT \right]$$

By definition, accrued utility under the expected value sign is equal to the following sum:

$$AU\Big|_{cT}^{cT+DH}(SEQ) = \sum_{i=0}^{DH-1} IU(Conf(cT+i), Conf(cT+i-1))$$

First, let's enumerate the available inputs:

- Task *T*, its activation time, 0, and deactivation time, *TD*. This corresponds to the first input in the anticipatory model of configuration,

- List <Supplier> *AvailableSuppliers*. As in the reactive model, this is the list of the available suppliers and their corresponding QoS-resource profiles,

- ResourceAvailabilityVector[0..TD-1] *resPath*. This is an array of resource availability snapshots, one per time window. For example, for some future time *s*, *resPath*[*s*] is the predicted level of the resources expressed as a ResourceAvailabilityVector object. Because the predictions are perfect, the value of the prediction does not change over time and has not uncertainty, i.e. all values in that array are non-random vectors,

Using the routines and algorithms presented in the previous section, we pre-compute the following variables:

- SupplierAssignment [1..nSA] *allSAs*. This is the list of all alternative supplier assignments that satisfy Task *T*, and can be computed using `GenAllSA`,

- Real [1..nSA] *supplierUtil*. This is the supplier utility portion of overall utility, calculated for each supplier assignment according to the preferences in Task *T*. This can be computed using `CalcSuppUtil`,

- Real [1..nSA][1..nSA] *changePenalty*. This two dimensional array is the matrix of change penalties. In entry *iSA, jSA* we record the change penalty that is incurred in the hypothetical scenario of switching from supplier assignment *iSA* to *jSA*. This is computed using `CalcChPen` routine,

- ResourceAvailabilityVector [1..nResSieve] *resourceSieve*. This a sieve of all possible resource vectors,

- Real [1..nSA][1..nResSieve] *maxQoSUtil*. This two dimensional array is the matrix of maximum QoS utility possible per supplier assignment for each possible resource state. In entry *iSA*, *jResSieve* we store the value of maximum possible QoS utility in the hypothetical scenario when the level of available resources is equal to *resourceSieve* [*jResSieve*] and we are considering supplier assignment *iSA*. This array can be efficiently computed using the algorithm, `DMaxQoS_ResSieve`.

We solve the problem at hand using dynamic programming. We algorithm tabulates over time going backwards and over the list of available supplier assignments. The algorithm works by solving partially instances of the problem by computing optimal sequence of configurations in the hypothetical scenario when the task starts at future time *s* and lasts until time *TD* (where *s* iterates

from *TD* down to 0). In this manner, the algorithm solves all partial problem instances, including the original instance of the problem.

Let *PartialMaxAU(jSA, s)* denote the maximum accrued utility possible in the hypothetical scenario when the task starts at future time *s* and lasts until TD, AND at time *s* we select supplier assignment *jSA* to run:

- Real [1..nSA][1..TD-1] *PartialMaxAU*,

The following equations describe the termination condition and the induction in dynamic programming:

- *PartialMaxAU [jSA][TD-1] = maxQoSUtil[jSA][resPath[TD-1]] + supplierUtil[jSA]*. This equation holds because TD-1 is the last time during which the task is active, and we have a 1-period optimization problem. This equation provides the termination condition for dynamic programming,

- *PartialMaxAU [jSA][s] =* $\max_{kSA}$ *{ maxQoSUtil[jSA][resPath[TD-1]] + supplierUtil[jSA] + changePenalty[jSA][kSA] + PartialMaxAU[kSA][s+1]} = maxQoSUtil[jSA][resPath[TD-1]] + supplierUtil[jSA] +* $\max_{kSA}$ *{ changePenalty[jSA][kSA] + PartialMaxAU[kSA][s+1]}*. This is the recursive equation that we use to move the dynamic program one step backwards. Intuitively, this equation describes how to solve an instance of the problem with longer remaining length of the task, if instances of the problem with shorter remaining length have already been solved.

We also need to record the value of *kSA* that maximizes the sum:

- SupplierAssignment [jSA][s] *nextSA* denotes the value of *kSA* for which the maximum is achieved in the recursive equation above.

Here is the pseudocode of the algorithm:

```
MaxAUPerfect( input Task T,
                        input real TD,
                        input List<Supplier> availableSuppliers,
                        input ResourceAvailabilityVector[1..TD-1] resPath)
    output real maxOptAU,
    output SupplierAssignment [] saSeq,
    output QoSSetpoint [] qosSetPointPerSASeq
{
    SupplierAssignment allSAs[] = GenAllSA(T, availableSuppliers);
    real suppUtilPerSA[] = CalcSuppUtil(T, allSAs);
    real chPenaltyPerSAPair[][] = CalcChPen(T, allSAs);
    ResourceAvailabilityVector resSieve[] = GeneResSieve();

    int nSA = allSAs.length;  // size of the array
    int nResSieve = resSieve.length;
    real perfectQoSUtil = 1;
    real maxOptUtilSoFar = - INF;
    real maxQoSUtil[1..nSA][1..nResSieve];
```

```
    // calculate maxQoSUtil for all suppliers / all poits in res sieve
    for jSA = 1 to nSA
        maxQoSUtil[jSA] = DMaxQoS_ResSieve(T, allSAs[jSA], resSieve);

    real partialMaxAU[1..nSA] [0..TD-1];
    int nextSA[1..nSA] [0..TD-1];
    // calculate the values of partialMaxAU for the LAST time period
    for jSA = 1 to nSA
    {
        partialMaxAU[jSA][TD-1] = maxQoSUtil[jSA][ LookupSieveIndex(resPath[TD-1]) ] +
                    suppUtilPerSA[jSA];
        nextSA[jSA][TD-1] = NULL;
    }
    for s_time = TD - 2 downto 0
    {
        for jSA = 1 to nSA          // outer loop is for each SA
        {
            int localNextSA = -1;
            real runningMaxAU = -INF;
            real temp = 0;
            real thisSAsMaxQoS = maxQoSUtil[jSA][LookupSieveIndex(resPath[s_time])];

            for kSA = 1 to nSA              // inner loop is to find the best "next"
            {
                temp =  chPenaltyPerSAPair[jSA][kSA] + partialMaxAU[kSA][s_time+1];
                if (temp > runningMaxAU )
                {
                    runningMaxAU = temp;        localNextSA = kSA;
                }
            }
            nextSA[jSA][s] = localNextSA;
            partialMaxAU[jSA][s] = runningMaxAU;
        }
    }
    maxOptAU = - INF;
    int bestIndex = -1;
    for jSA = 1 to nSA
    {
        If ( partialMaxAU[jSA][0] > maxOptAU )
        {
            maxOptAU = partialMaxAU[jSA][0];
            bestIndex = jSA;
        }
    }
    // populate saSeq[] and qosSetPointPerSASeq[] and return;
}
```

The algorithm is executed once when the task is activated. It calculates a sequence of configurations that are temporally optimal, i.e. will give the highest accrued utility over the duration of the task. Because predictions are perfect, this sequence is *ex post* optimal, i.e. we don't need to modify the sequence until the task completes.

This algorithm can be generalized to multiple tasks, with varying activation and deactivation times – so long as the task request operation predictions are perfect. The dynamic programming algorithm will simply have to account for the set of active tasks as it steps backwards in time. Changes in the active tasks that are in response to user requests are not penalized per the analytical models.

To analyze the running time of the algorithm is dominated by two phases: (1) calculating the optimal instantaneous QoS for each SA and each point in the resource sieve, and (2) calculating the optimal sequence of configurations using dynamic programming.

As noted earlier, `DMaxQoS_ResSieve` has a running time of O(***nSvc*** * ***nQoS*** * ***nResSieve***) per supplier assignment. That routine is invoked once per supplier assignment, resulting in aggregate running time of O(***nSvc*** * ***nQoS*** * ***nResSieve*** * ***nSA***). The dynamic programming phase takes O(***nSA*** * ***nSA*** * ***TD***). The total running time is on the order of:

$$O(nSvc * nQoS * nResSieve * nSA) + O(nSA * nSA * TD).$$

Instead of using `DMaxQoS_ResSieve` algorithm for calculating the optimal instantaneous QoS, we have the option of using the `AMaxQoS` algorithm. We will need to invoke the latter a total of ***nSA*** * ***TD*** times, as we need the optimal instantaneous QoS for each supplier assignment and resource state pair (because of perfect predictions, there are only ***TD*** possible resource states). Thus, the overall performance of the algorithm would be:

$$O(nSvc * \log(nQoS) * nQoS * nSA * TD) + O(nSA * nSA * TD).$$

9.4 Stage 3.1: Resource Predictions with Uncertainty

In this section, we demonstrate how the dynamic programming algorithm can be successfully generalized to the case of imperfect resource predictions. Per the problem parameters for stage 3.1, we know that task predictions are perfect, and there are no non-perishable resources.

For simplicity, throughout this section we make the same simplifying assumptions about the task and task requests made in the previous section:

- there is only one active task,
- the task is activated at time 0,
- the duration of the task is TD, expressed in prediction scale units.

Furthermore, we will assume that only predictions for one resource are imperfect. Generalizing to the case with imperfect predictions for multiple resources is a simple exercise and will only increase the computational complexity of the algorithm.

Resource predictions are represented using a recursive data structure. Each node contains references to its, their probabilities conditional on the current node, the resource value, the depth relative to the root node, and a number of housekeeping items.

In addition to the data structure for resource predictions, we need a data structure for storing the results of the calculation necessary to find the optimal configuration. There is a natural correspondence between the calculations and the prediction nodes. In order to save space and simplify the design we can store the results of calculations inside the prediction node.

Just like the case with perfect resource predictions, results must be stored for each supplier assignment to be able to run the dynamic program. Based on these observations, here are the definitions of the relevant data structures:

```
typedef struct _resourcePredictionNode
{
    struct _ resourcePredictionNode         * parent;
    struct _ resourcePredictionNode         * children;
    real                                    * probabilities;
    int                                     countChildren;
    resNodeTypeEnum                         nodeType;
    ResourceVector                          resourceValue;
    struct _resourceSimulationAndResult     * sims;
    int                                     countSA;    // number of sims, equal to nSA
} ResourcePredictionNode;

typedef struct _resourceSimulationAndResult
{
    struct _resourcePredictionNode          * owningNode;
    real                                    instQoSUtil;
    real                                    suppUtil;
    real                                    penalty;
    real                                    PartMaxAU;
    int                                     * nextSA; // the number of these is countChildren
} ResourceSimulationAndResult
```

The problem inputs are as follows:

- Task *T*, its activation time, 0, and deactivation time, *TD*,
- List <Supplier> *AvailableSuppliers*,
- ResourcePredictionNode * *prediction*. This is the root prediction node.

As in the case with perfect predictions, we use dynamic programming. We modify the DP algorithm from the previous section. Let *PartialMaxAU(rPredNode, s, jSA)* denote the expected value of maximum accrued utility possible in the hypothetical scenario when:

- the task starts at future time *s* and lasts until TD,

- resources are in state *rPredNode* in the prediction tree,

- AND in the beginning of the task we select supplier assignment *jSA* to run.

Notice that unlike in the case with perfect predictions, *PartialMaxAU* is not a guaranteed value of future accrued utility. Because of predictive uncertainty, the actual value is a random variable, and *PartialMaxAU* is the expected value of that random variable.

The terminal equation and the recursive rule equations are as follows:

- *PartialMaxAU (rPredNode, TD-1, jSA) = maxQoSUtil[jSA][rPredNode.resourceValue] + supplierUtil[jSA]*,

- *PartialMaxAU(rPredNode, s, jSA) = maxQoSUtil[jSA][rPredNode.resourceValue] + supplierUtil[jSA] +* $E_{childrenOf(rPredNode)}[\ max_{kSA}\ \{$ *changePenalty[jSA][kSA] + Partial-MaxAU(rPredNode, s+1, jSA)}]*. This recursive equation calculates the expected value of future accrued utility over all children nodes of the current node.

Next, we show the pseudocode for the calculation of *PartialMaxAU*. For brevity, we omit calls to helper functions, initialization of variables, and stopping conditions for the recursion.

```
RecursiveCalcPartMaxAU(ResourcePredictionNode * curNode)
{
    …
    for rChild = 1 to curNode -> countChildren
        RecursiveCalcPartMaxAU(& curNode->children[rChild]);
    for jSA = 1 to nSA
    {
        real expPartMaxAU = 0;      // the value of PartMaxAU(curNode, curNode->d, jSA)
        for rChild = 1 to curNode->countChildren;
        {
            real bestOfPartMaxAU = -INF;
            int indexOfBest = -1;
            real temp = 0;
            for kSA = 1 to nSA
            {
                temp = curNode->children[rChild].sims[kSA].PartMaxAU + ChPen[jSA][kSA];
                if (temp > bestOfPartMaxAU)
                {
                    bestOfPartMaxAU = temp;
                    indexOfBest = kSA;
                }
            }
            expPartMaxAU += curNode->children[rChild]. probabilities * bestOfPartMaxAU;
            curNode ->sims[jSA].nextSA[rChild] = indexOfBest;
        }
        curNode-> sims[jSA].PartMaxAU = expPartMaxAU;
    }
}
```

Based only on the recursive call in the first loop, it is obvious that the algorithm requires exponential time to run. If the number of children in each node of the prediction tree is ***nBF*** (BF here stands for branching factor), and the duration of task is ***TD***, then the running time of the algorithm is O(***nBF^nTD***). The triple-nested "for" loops that calculate PartMaxAU contribute a constant factor that is equal to roughly O(***nSA*** * ***nSA*** * ***nBF***).

To reduce the running time of the algorithm, we considered a number of options. All of these options require sampling fewer nodes in the resource prediction tree.

The first approach is to use Monte Carlo simulations. Each sample in the simulation is a possible resource path, and the "Perfect" algorithm from section 9.3 can be used to determine the optimal sequence and the sample maximum accrued utility, MaxAU. A Monte Carlo simulation samples a large number of paths and calculates the best configuration, together with the expected accrued utility. The primary disadvantage of this approach is that resource outcomes that are in the near future are sampled as many times as outcomes that are farther in the future. Because uncertainty is smaller in the near future, this perhaps results in oversampling in the near future outcomes. Also, in order to use Monte Carlo simulations, we would have to modify the current representation of predictions. Using the current representation will result in significant inefficiencies, because sample paths would be generated only from the prediction tree, producing many samples that share the same path.

Another approach is to sample more intensively in regions where resources are scarce, and less intensively in regions where resources are plentiful. This approach to sampling has the potential benefit of providing a better estimation of PartMaxAU on the basis of sample paths when it matters most. Indeed, when the available level of some resource is sufficiently large, then the resource does not create a capacity constraint. Consequently, instantaneous QoS utility at that resource level would tend to be close to the maximum possible, and sampling more would not necessarily help obtain a better estimate.

The third approach traverses the complete prediction tree up to depth *d*, where *d* is an integer less than or equal to 10. Starting with nodes at depth *d*, it only samples the middle node. To make this approach more efficient, the prediction tree can be built according to the sampling needs. The resource prediction API described in section 8.4 is built around this idea. We believe this approach provides the best approximation with the model and the representation of resource predictions we have chosen. Ideas from the second approach can be combined into this one, providing more opportunities for reducing running time and improving the effectiveness of sampling.

We have chosen to implement the third sampling approach. With this approach, the running time of the dynamic program is reduced to O(***nBF^d***) * O(***nSA*** * ***nSA*** * ***nBF***) + O(***nSA*** * ***nSA*** *(***TD-d***)). We also need to account for the running time of the following `DMaxQoS_ResSieve` routine, which is equal to O(***nSvc*** * ***nQoS*** * ***nResSieve*** * ***nSA***). All other helper routines have running times that are dominated by at least one of the three terms. So the running time of the entire algorithm is equal to:

O(***nBF^d*** * ***nSA*** * ***nSA*** * ***nBF***) + O(***nSA*** * ***nSA*** *(***TD-d***)) + O(***nSvc*** * ***nQoS*** * ***nResSieve*** * ***nSA***)

9.5 Stage 3.2: Task Predictions with Uncertainty

In this section we solve the anticipatory dynamic configuration problem with imperfect task predictions. The only additional complexity in this stage is uncertainty in the task predictions. We described the task prediction model in section 6.5.3. According to that model, the time of activation and deactivation of known tasks as well as task durations are modeled using a probability mass function. Only two of the three variables are modeled for each activation instance of each task. The model differentiates between *tight* and *non-tight* predictions of these variables. When predictions are tight, they are expressed using a one-tail or two-tail Gaussian probability functions. When predictions are not tight, they are expressed using a probability mass function.

We modify the algorithm used in section 9.3 designed for the stage 2 problem. We express the available task predictions using a recursive data structure and generalize the "perfect" algorithm slightly for our purpose.

For simplicity of describing the algorithm, we will make the following simplifying assumptions:

- there is only one known task,
- the task is activated at time 0,
- the task is deactivated at some future time which is a random variable with the following probability mass function, $\{(p_1, TD_1), (p_2, TD_2), \ldots, (p_k, TD_k)\}$. The TD_i are integers in increasing order, i.e. $TD_i < TD_j$ for all $1 < i < j < k$, and the p_i are positive numbers that add up to 1. Notice that the deactivation time of the task is also the duration of the task, because we have assumed that the task starts at time 0. This probability mass function states that the task will be deactivated at the end of time period TD_i with probability p_i.

We can use the above probability mass function to obtain the following conditional probabilities:

- Pr(task ends at the end of period TD_i | task has not ended at the end of period TD_{i-1}), and
- Pr(task does not end at the end time period TD_i | task has not ended at the end of period TD_{i-1}).

The above events are complementary, so their probabilities must add to 1. Using the definition of conditional probability, we have: Pr(ends at TD_i | hasn't ended at TD_{i-1}) = Pr(ends at TD_i AND hasn't ended at TD_{i-1}) / Pr(hasn't ended at TD_{i-1}). Pr(hasn't ended at TD_{i-1}) = $p_i + p_{i+1} + \ldots + p_k$, and Pr(ends at TD_i AND hasn't ended at TD_{i-1}) = Pr(ends at TD_i). Thus, we have:

- Pr(ends at TD_i | hasn't ended at TD_{i-1}) = $p_i / (p_i + p_{i+1} + \ldots + p_k)$.

Similarly, we compute the probability of the complement event:

- Pr(doesn't end at TD_i | has not ended at TD_{i-1}) = $(p_{i+1} + \ldots + p_k) / (p_i + p_{i+1} + \ldots + p_k) = (1 - p_i) / (p_i + p_{i+1} + \ldots + p_k)$.

We can represent these probabilities using a binary prediction tree. The leaf nodes record task-ending outcomes, while the non-leaf nodes record the fact that the task will go on. Each non-leaf node can have one or more children.

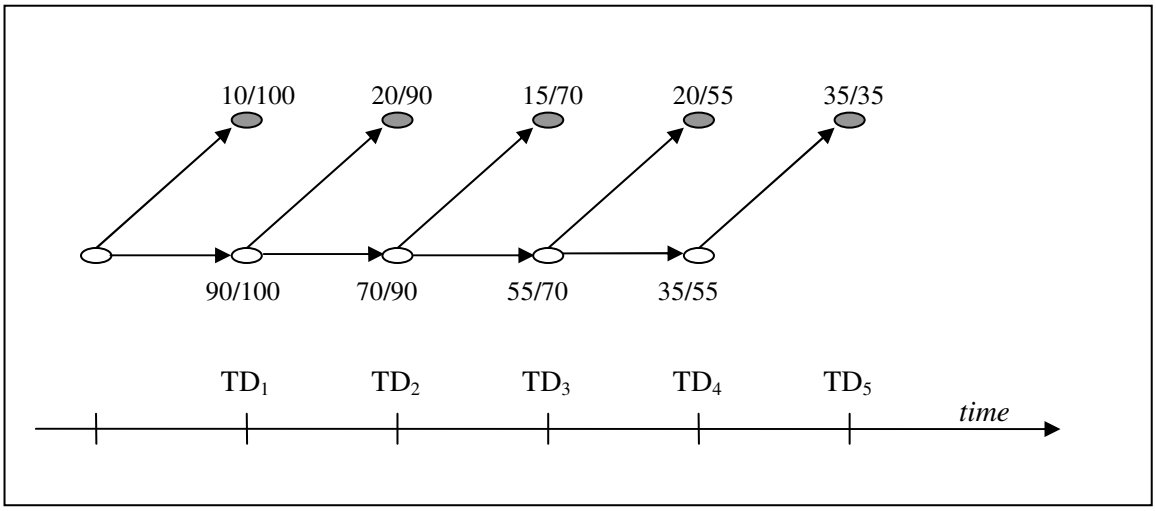


Figure 14: Prediction tree showing the probability distribution of the task duration.

To obtain a better intuition for this representation, let's consider an example. We have the following probability mass function for the duration of the task:

- $\{(10/100, TD_1), (20/100, TD_2), (15/100, TD_3), (20/100, TD_4), (35/100, TD_5)\}$.

The binary prediction tree is shown in Figure 14. The figure shows time on the horizontal axis. The leaf nodes are filled with gray shading. Non-lead nodes are unfilled. The quotients immediately next to each node show the probability of that node conditional on the parent node.

For example, the probability that the task will end at the end of period TD_2 conditional on the event that the task lasts beyond period TD_1 is 20 / 90. Notice that here 20/100 is equal to p_2 (the probability that task lasts until the end of period TD_2 in the original prediction function), while 90/100 is equal to $p_2 + p_3 + p_4 + p_5$, and 20 / 90 is the ratio of the two probabilities. This is consistent with our calculations earlier.

We can use this representation to extend the dynamic program from section 9.3. The dynamic program tabulates backwards in time, using expected value over the children of the current node to calculate optimal partial accrued utility.

```
typedef struct _taskEventPredictionNode
{
    struct _taskEventPredictionNode          * parent;
    struct _taskEventPredictionNode          * leftNode;  // leaf node
    struct _ taskEventPredictionNode         * rightNode; // non-leaf node
    enum                                     nodeType;  LEAF, NON_LEAF
    struct _taskEventSimulationAndResult     * sims;
    int                                      countSA;    // number of sims, equal to nSA
    int                                      time;
} TaskEventPredictionNode;


typedef struct _taskEventSimulationAndResult
{
```

```
    struct _taskEventPredictionNode     * owningNode;
    real                                instQoSUtil;
    real                                suppUtil;
    real                                penalty;
    real                                PartMaxAU;
    int                                 * nextSA; // the number of these is at most 2
} TaskEventSimulationAndResult;
```

The inputs to the problem are as follows:

- Task *T*, its activation time, 0, and the prediction for its deactivation time expressed as probability mass function: $\{(p_1, TD_1), (p_1, TD_1), \ldots, (p_k, TD_k)\}$. We use this function to generate a recursive prediction tree, TaskEventPredictionNode * *taskPrediction*,

- List <Supplier> *AvailableSuppliers*,

- ResourceAvailabilityVector[0.. TD_k-1] *resPath*. Notice that resource predictions are available as far as the task is likely to last.

Next, we show the two equations necessary to implement dynamic programming:

- *PartialMaxAU(tPredNode, jSA) = maxQoSUtil[jSA][resPath[tPredNode.time]] + supplierUtil[jSA]*. This is the equation for terminal condition that applies when *tPredNode* is a leaf (terminal) task prediction node,

- *PartialMaxAU(tPredNode, jSA) = maxQoSUtil[jSA][resPath[tPredNode.time]] + supplierUtil[jSA] +* $E_{childrenOf(tPredNode)}[\ \max_{kSA}\ \{\ changePenalty[jSA][kSA] + (tPredNode, s+1, jSA)\}]$. This equation applies when *tPredNode* is a non-leaf node. The expected value is taken over its children. As mentioned earlier, a non-leaf task prediction node can have at most two children. This has important consequences on the running time of the algorithm. Notice that the recursive equation is identical to the one in section 9.4, with the exception of the parameters in the expression for

The recursive equation in this problem stage is identical to the one in stage 3.1. Consequently, the dynamic programming algorithm is also identical to the one in section 9.4. Therefore, we omit the pseudocode of the algorithm for brevity. The algorithm has a branching factor of one, resulting in a running time that is linear in the decision-making horizon. The performance of the algorithm is equal to:

$$O(\boldsymbol{nSA} * \boldsymbol{nSA} * \boldsymbol{TD}) + O(\boldsymbol{nSvc} * \boldsymbol{nQoS} * \boldsymbol{nResSieve} * \boldsymbol{nSA})$$

9.6 Stage 3.3: Non-perishable, Non-renewable Resources

Non-perishable, non-renewable resources such as battery introduce a unique challenge to the configuration problem. This challenge arises as a result of the intertemporal fungibility of battery. The current usage of battery by user's own tasks depletes the charge and reduces the future available level of battery. On the other hand, any unused battery energy can be transferred for use in

the future. Also, the battery can be charged up-to its maximum charge capacity, providing for future use of the hardware device without access to wall electricity.

First, we make a number of realistic assumptions about the behavior of the battery:

- Turning a hardware device off by shut-down or hibernation over short periods of time does not deplete the battery. In stand-by mode, battery energy is used to refresh the RAM, and this assumption fails to hold. Over longer periods of times such as days or weeks, the battery can deplete without being used,

- There is no battery leakage in the system. Some faulty batteries might show significant drainage of the battery over very short periods of time, indicating that battery is leaking. This might be due to deteriorating battery chemistry, a short circuit in the battery or in the device. This assumption allows us to exclude faulty batteries from consideration and only focus on healthy batteries,

- The system provides accurate accounting of battery. The system can report the following parameters on demand: (1) the maximum charge capacity in watt-hours and (2) the remaining battery capacity in watt-hours. Although many systems can report design capacity as well, this is not required for our purpose. The design capacity is the maximum charge capacity for a battery that has just been put into use. The maximum charge capacity is the maximum amount of energy that the battery can store when fully charged. The maximum charge capacity is less than the design capacity. For brand new batteries, the maximum charge capacity is very close to the design capacity. The remaining capacity shows how much energy is left in the battery and is always less than the maximum charge capacity.

The intertemporal substitution possibilities of battery create a unique challenge, because the battery can be drained at different rates per unit time. When battery is drained at a higher rate, applications or devices can potentially provide a higher quality of service. Either the user or a software component acting on behalf of the user, can control the drainage rate of the battery by manipulating various knobs. Settings for controlling the drainage rate of the battery are widely available on off-the-shelf devices such as laptops, handheld devices, and phones. For example:

- A software or hardware knob for dimming the display brightness. Typically, the brighter the display, the faster the battery drains. On a typical laptop or handheld device, there are anywhere from half a dozen to a dozen brightness levels,

- A software knob for scaling the voltage of the CPU. This feature is found in the power-saving controls of the battery and is made available by either the device manufacturer or the OS vendor. By scaling the voltage of the CPU down, the available CPU cycles are reduced. On a typical device, there are 3-4 voltage scaled modes,

- Various knobs for shutting off parts of the device when not used. For example, the hard drive can be shut off, the display can be put into power-save mode, etc.

9.6.1 Background Research on Battery Drain

In his thesis [17], Flinn demonstrated several important results about battery energy usage and accounting. First, he measured the battery drain due to various hardware parts, e.g., the CPU,

display, memory, hard disk, network card. For those devices that can operate at different settings that correspond to different power consumption levels, he measured the battery drain in each setting. For example, the battery drain of the display depends on the brightness of the display. Similarly, the battery drain of the CPU depends on the voltage mode of the CPU. Next, he profiled adaptive applications in a way that measured the battery drain due to a single fidelity operation such as decoding a stream.

The following are the findings based on Flinn's work that apply to our work:

- We can measure the base level of energy drain per unit time that a particular hardware device consumes at the base power level,

- We can measure the incremental drain of battery energy drain per unit time that is due to an incremental change in the mode of operation of a particular device. For example, we can calculate the battery drain for each brightness level of the display. Based on those measurements, we can calculate incremental drain due to incremental improvements in display brightness,

- We can measure the incremental drain of battery energy per unit time that is due to an particular application while it executes at each capability level. The incremental level of battery drain is in addition to the base power drain of the entire computer,

9.6.2 Modeling Battery Drain for Configuration Purposes

Our work on modeling the battery drain is based on Flinn's work and is fully consistent with his work. Our work goes further by using the user's preferences to make decisions on allocations of battery over time.

The key modeling principle in our work, and the main differentiator of our work, is to *evaluate* the effect of incremental improvements in quality of service of applications and devices relative to their incremental battery drain and to *decide* the incremental level of battery drain that benefits the user most in the form of increased the accrued utility.

To demonstrate how we can do that, we consider three cases of modeling the battery drain and making configuration decisions according to user's preferences. The first case concerns the brightness of the display. The second case concerns the voltage scaling of the CPU. And the third case concerns the quality of service of a typical fidelity-aware application. These cases are not exhaustive, but they demonstrate key modeling principles.

9.6.2.1 Display Brightness

We differentiate two mutually exclusive cases of user's preferences with respect to display brightness: (1) the user cares about display brightness and let's the automatic configuration software manage the display brightness together with all tasks of the user, or (2) the user either does not care about display brightness, or manages the display brightness manually, overriding the system.

In the first case, we require the user to define a task that has one service, "*show Dsplay*". This service has one quality attribute: "*brightness*". The user provides preferences for this task, just like any other task, using the Task Manager.

The Environment Manager includes this task as part of its configuration decisions. First, there is a single supplier of a "*show Display*" service available in the environment. This supplier represents the display of the device. The profile of this supplier enumerates the available brightness levels and the corresponding battery drain per unit time. The drain rate recorded by the supplier's profile is the incremental rate above the baseline.

The "*show Display*" task is an active task, and the Environment Manager controls the brightness level according to the preferences of the user.

On the other hand, if the user does not care about display brightness or prefers to manually control the brightness, then the "*show Display*" task is not made active. In this case, the Environment Manager does not control the brightness of the display. Instead, the control of the brightness is left to the default mechanisms in the operating system or manual control of the user.

9.6.2.2 Voltage Scaling of the CPU

Modern CPUs for mobile platforms have power-saving features that utilize voltage scaling. Reducing the voltage of the CPU preserves energy, but also reduces the clock speed of the CPU.

Maximum Performance Mode			Battery Optimized Mode		
Frequency	Voltage	Power Consumption	Frequency	Voltage	Power Consumption
1.20 GHz	1.40 V	22.0	800 MHz	1.15 V	9.8 W
1.13 GHz	1.40 V	21.8	733 MHz	1.15 V	9.3 W
1.06 GHz	1.40 V	21.0	733 MHz	1.15 V	9.3 W
1.00 GHz	1.40 V	20.5	733 MHz	1.15 V	9.3 W
933 MHz	1.40 V	21.1	733 MHz	1.15 V	9.3 W
866 MHz	1.40 V	19.5 W	667 MHz	1.15 V	8.9 W
800 MHz	1.15 V	9.8 W	533 MHz	1.05 V	6.1 W
800A MHz	1.15 V	9.8 W	500 MHz	1.05 V	5.9 W
750 MHz	1.15 V	9.4 W	450 MHz	1.05 V	5.7 W
733 MHz	1.15 V	9.3 W	466 MHz	1.05 V	5.8 W
700 MHz	1.10 V	7.0 W	300 MHz	0.95 V	3.0 W

Figure 15: CPU and power consumption tradeoffs of Mobile Intel Pentium III Processor.

In Figure 15, we reproduce a table provided by Intel Corporation. The table shows the available performance modes of the CPU, listing the clock speed and power consumption in each mode.

In order to take advantage of the clock-speed and power consumption trade-off offered by the CPU, we capture the information shown in that table using a supplier profile. This supplier is called "CPU Power Settings" and serves a special purpose that is different than application suppliers. This supplier's profile has a small number of QoS-resource pairs, corresponding to the number of voltage scaled modes. Each point captures the battery drain due to the CPU and the *complement* of CPU availability. The complement is the difference between the maximum clock speed of the CPU and the available clock speed at the voltage scaled mode.

For the CPU shown in the table, the maximum clock speed is 1,200 MHz without voltage scaling. The first row in the battery optimized mode in Figure 15 shows a clock speed of 800 MHz and power consumption of 9.8 Watts. The corresponding resource vector in the supplier's profile is (400 MHz, 9.8 Watts). It appears that the supplier consumes CPU but reduces the battery drain. While this does not accurately reflect the physics of the situation, the mathematics is correct.

The choice of a power-saving mode for the CPU does not affect user's utility directly. Therefore, unlike in the case of display brightness, there is no need to elicit additional preferences from the user or create a new service. In fact, the user is not aware of this service. Instead, the Environment Manager uses an internally defined task with one service that is designed for this purpose only. The service is provided by the "CPU Power Settings" supplier, and the inclusion of that supplier in the configuration decision expresses the feasible trade-off space between battery and CPU.

9.6.3 Expressing the Intertemporal Substitution Possibilities of Battery Energy

In our model, we express the intertemporal substitution possibilities of the battery. In each time period, the rate of battery consumption is a decision variable bounded by some maximum and minimum power consumption rates. We calculate the maximum and minimum consumption rates by profiling the device. We also calculated incremental consumption rates that are divide the interval between the minimum and maximum rates into equal intervals.

For example, a laptop might have a maximum power consumption rate of 26 Watts and a minimum rate of 20 Watts. We can choose an incremental consumption rate of 0.5 Watt to obtain 13 different power consumption rates. On the other hand, we can choose an incremental consumption rate of 1 Watt and obtain 7 different possibilities.

In this manner, the power consumption rate in each time period is a decision variable. If the remaining charge of the battery is 48 Watt-hours, then the battery can last anywhere between 1.85 hours to 2.4 hours. But the intertemporal substitution possibilities control more than just the remaining battery life. The battery consumption rate can change over time, e.g. starting low but then increasing, or vice versa.

We use a tree to express all possible power consumption rates over time. Each node in the tree expresses the remaining charge. The root node shows the remaining battery charge right now. The children of each node show possible consumption rates for the next immediate time period, and the amount of charge that would be left after transition to that node. The tree has a recombining property. For example, by consuming power at 20 Watts in the first period and then at 26 Watts in the second period, we will transition to the same node as when consuming at 26 Watts first, and then 20 Watts next.

Unlike the nodes in the prediction trees for bandwidth, the nodes in this tree are choices for configuration. With this distinction, we are now ready to describe the configuration algorithm that makes battery consumption rate decisions over time.

First, we define a data structure to record the tree that expresses the intertemporal expenditure possibilities of battery. Each node has the remaining available battery in that node, the rate of power consumption, and pointers to all the children. Also, we reserve memory to store the results of computations, as usual.

```
typedef struct _depletableResourceNode
{
    struct _depletableResourceNode      * parent;
    real                                resourceValue;
    real                                remainingCapacity;
    struct _depletableResourceNode      * children;
    int                                 countChildren;  // number of children
    struct _depletableResSimAndResult   * sims;
    int                                 countSA;    // number of sims, equal to nSA
    int                                 time;
} DepletableResourceNode;
```

Let *PartialMaxAU(deplResNode, s, jSA)* be the maximum accrued utility possible when starting from node *deplResNode* with supplier assignment *jSA*.

The terminal equation and the recursive rule equations are as follows:

- *PartialMaxAU (deplResNode, TD-1, jSA) = maxQoSUtil[jSA][deplResNode.resourceValue] + supplierUtil[jSA]*,

- *PartialMaxAU(deplResNode, s, jSA) = maxQoSUtil[jSA][deplResNode.resourceValue] + supplierUtil[jSA] + $max_{childrenOf(deplResNode)}$\{ max_{kSA} { changePenalty[jSA][kSA] + PartialMaxAU(deplPredNode, s+1, jSA)}}*. The outer maximum is over all children of *deplResNode* that are feasible. To be feasible, a node must have a positive amount of resource, i.e. *resourceValue* must be greater than zero.

Contrast the recursive equation for *PartMaxAU* in this case with the same equation in the case with resource uncertainty. In the latter, the recursive formula calculated the expected value of the future *PartMaxAU* across all child nodes. In this case, the formula calculates the maximum value of *PartMaxAU* across all child nodes. The feasibility constraint ensures that we don't allocate more of the resource than the available capacity.

We can extend this model and algorithm with two additional improvements. First, we can allow user to value the remaining energy after the task. For this purpose, the user needs to provide a preference function for possible remaining charge levels.

Next, we analyze the runtime performance of algorithm. We need to calculate the number of nodes in the recombinant tree. If the tree has a branching factor of ***nBF***, and the horizon decision is ***TD***, then we have ***s * (nBF-1) + 1*** nodes at time *s*. By summing that expression for the values

of *s* from 0 to ***TD-1***, we get the total number of nodes as follows: ***(nBF-1) * TD(TD-1)/ 2 + TD***. In each node, we calculate the maximum using two nested for loops. The first loop iterates over possible supplier assignments, and the second, over the feasible child nodes. So the running time of calculations in each node is O(***nSA * nBF***). In aggregate, the running time of the algorithm is as follows:

$$O(nBF * TD^2) * O(nSA * nBF) + O(nSvc * nQoS * nResSieve * nSA)$$

9.7 Stage 4: Combining Predictions with Uncertainty

In this section, we demonstrate how multiple predictions for different variables are combined when solving the configuration problem. We also describe how the existing dynamic programming algorithm can be used to solve the problem with multiple predictors.

Our strategy of combining predictions from multiple sources is to combine the individual prediction trees into a single tree with more branches. The nodes in the combined tree express joint outcomes and their probabilities. The probabilities of the nodes in the combined tree are obtained using the probabilities of the nodes in the original trees and the correlation between the variables being predicted. We don't address the problem of determining the correlation between the predicted variables in this thesis. Instead, we address two simple cases: (a) when the predicted variables are statistically independent or (b) when those variables are perfectly correlated.

In the following three sections, we describe three cases of combining predictions: (1) two resource predictions, (2) two task request predictions, and (3) a resource prediction and a task request prediction. In each case, we use examples to provide the reader with the basic intuition.

9.7.1 Combining Two Resource Predictions

We use an example to demonstrate how resource predictions are combined. In this example we have two resource instances being predicted. Each of these resource instances has an aggregated predictor that is expressed using a prediction tree with a branching factor of 3. The combined prediction tree has a branching factor equal to 9, the square of 3.

Let the root node of the first prediction be RA, and its three has children be RA1, RA2, and RA3. These child nodes correspond to high, medium, and low outcomes of the 1 step-ahead prediction of the resource. Similarly, let the root node of the second prediction be RB, and its three children be RB1, RB2, and RB3. Again, these child nodes correspond to the high, medium, and low outcomes of the 1 step-ahead prediction of the second resource.

In the combined tree, we label the parent node with a tuple, (RA,RB). It has 9 children, corresponding to the pair-wise combinations of each of the three children from the "A" tree with each of the three children from the "B" tree: (RA1,RB1), (RA1,RB2), (RA1,RB3), (RA2,RB1), (RA2,RB2), (RA2,RB3), (RA3,RB1), (RA3,RB2), (RA3,RB3). The (RA1,RB1) node corresponds to the outcome of the world when both the first and second resources are in high state. The meaning of the other 8 nodes is similar, in each case combining the individual resource outcomes to obtain the joint outcome.

To determine the probability of the child nodes in the combined prediction tree, we need the probabilities of the nodes in the original tree as well as the correlation between the resources.

When the resources are independent, the probability of each combined node is the product of the probabilities of the nodes in the original tree that make up the joint node. For example, the probability of the node (RA1,RB1) is the product of the probabilities of RA1 and RB1. Recall, that the probabilities of the nodes in both original as well as the combine trees are conditional on the parent.

This strategy of combining predictions increases the branching factor of the resulting prediction tree, and causes the runtime performance of the algorithm to increase. For example, if we have 3 independent resources, and each resource has a prediction tree with a branching factor of 3, then the combined tree will have a branching factor of 27. Despite the large branching factor, the running time of the optimization algorithm can still be kept feasible by limiting the depth of branching. However, to maintain the same absolute running time, we need to limit the depth of branching even further.

9.7.2 Combining two Task Predictions

Let's consider an example with two task request event predictions. The first prediction tree has 6 nodes: A0, A1, A1', A2, A2', and A3' (here, nodes with an emphasis are leaf nodes, e.g. A1'). The second prediction tree has the same shape, and we use letter B to prefix its nodes: B0, B1, B1', B2, B2', and B3'. The combined prediction tree has 18 nodes and is shown in figure xxx. We note that 18 = 2 * 3^2. Let's prove that the number of nodes in the combined tree is proportional to the square of the height. Let's denote by G(n) the number of nodes in the original prediction tree. It is easy to see that G(n) = 2 * n. Denote by F(n) the number of nodes in the combined prediction tree. Let's first observe that:

$$F(n) = 1 + 1 + 4 * (n-1) + F(n-1) = 2 + 4(n-1) + F(n-1)$$

Let's see where each of the terms in the above formula comes from. When combining the two threes, we get a root node, (A0,B0), and 4 branches. The first branch is a single leaf node, (A0',B0'). Then there are two branches that are identical in shape to the original trees. The root nodes of these two branches are the following: (A1,B1') and (A1',B1). The fourth branch starts with node (A1,B1) and is a combined tree with a smaller height.

Next, let's calculate the value of F(n) as a function of n. For n = 2, we have F(2) = 8 = 2 * 2^2. By induction, let's assume that F(n) = 2n^2. According to our recursive formula, F(n+1) = 2 + 4n + 2 n^2 = 2 (n+1)^2.

There is another, more intuitive approach to visualizing the combined prediction tree. We represent a prediction using as a set of possible path outcomes. Using our earlier example, per Figure 16, the first tree has the following possible paths:

- {A0, A1'}, {A0, A1, A2'}, {A0, A1, A2, A3'}.

Similarly, the "B" prediction tree has 3 possible paths, as follows:

- {B0, B1'}, {B0, B1, B2'}, {B0, B1, B2, B3'}.

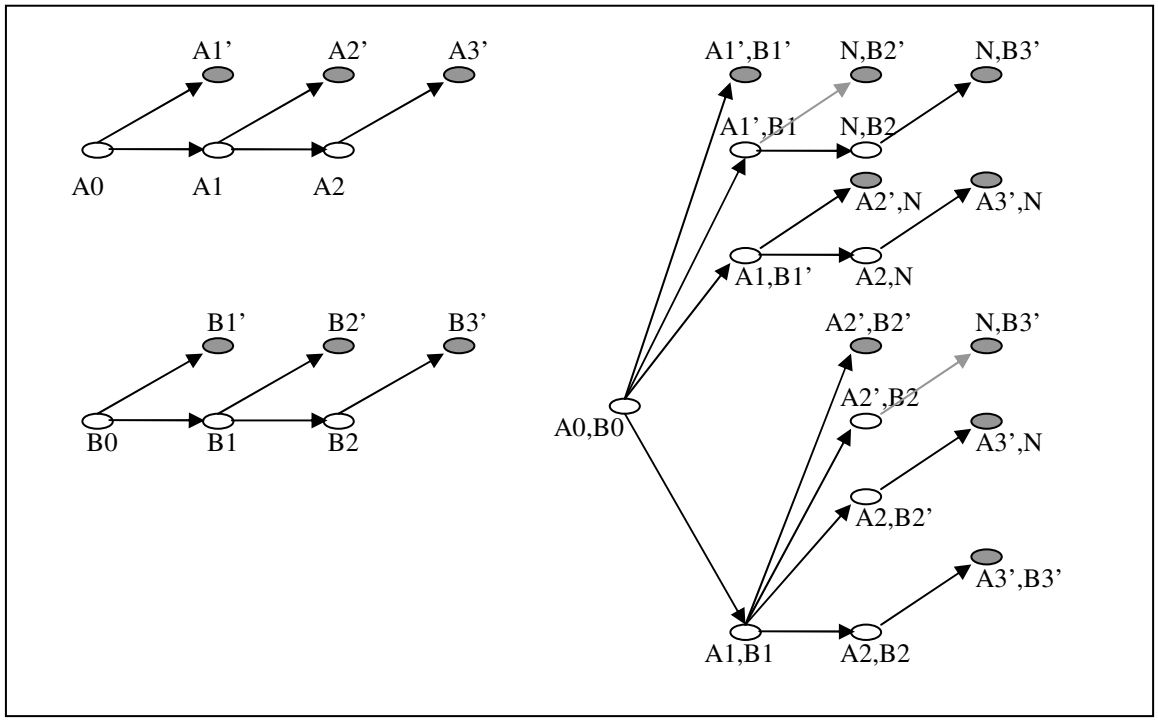


Figure 16: Combing the predictions of two independent task request events.

The combined tree has 9 possible paths. The following 3 paths are obtained by combining {A0,A1'} path with all "B" paths:

- {(A0,B0), (A1',B1')},
- {(A0,B0), (A1',B1), (NULL,B2')},
- {(A0,B0), (A1',B1), (NULL,B2), (NULL,B3')}, etc,

Because {A0,A1'} does not continue past period 1, we use the NULL outcome as a placeholder.

We obtain the probabilities of the nodes in the combined tree using the probabilities of the nodes in the original tree. When the events predicted by the two trees are independent, then the probability of a node in the combined tree is the product of the original nodes. For example, the probability of the node (A1,B1) is the product of the probabilities of nodes A1 in tree 1 and A2 in tree 2. Recall that all probabilities are conditional on the parent node.

Another noteworthy case is that of completely correlated events. For example, suppose that tree 1 is predicting the time when one task will be deactivated by the user and tree 2 is predicting the same event for another task. If these events are completely correlated, then the user will terminate both tasks at the same time. This information can be used to prune the combined tree quite a bit. In fact, if we prune out all the nodes that have probability zero, we will be left with a tree that has the same shape as either of the original trees.

9.7.3 Combining Resource Predictions with Task Predictions

Solving the configuration prediction with resource predictions and task predictions requires combining the trees of both predictions into a single tree. The tree expresses the probabilities of joint outcomes: future resource levels and future times of task requests. The dynamic programming method continues to apply as before. The algorithm required to solve this case is exactly the same.

9.8 Stage 5: Predictions of Resources, Tasks, and Battery

In this stage of the problem, we address the most general formulation of the anticipatory problem of configuration in which resource predictions and task request predictions have uncertainty, and battery is part of the configuration decisions.

The main challenge in this problem that has not been addressed so far in any of the previous algorithms is how to combine prediction of future events with the intertemporal substitution possibilities of battery.

Prediction trees of resource availability and prediction trees of task requests are similar because both express future outcomes and associated probabilities. The future outcomes are decided exogenously, and the system must be prepared to plan for any future outcomes along the prediction trees. To reflect that, the recursive equation for dynamic programming uses expected value calculation over the children of each tree node to determine the expected future accrued utility possible in that tree node.

Unlike those prediction trees, the intertemporal substitution possibilities of battery are expressed as a tree in which each node is an alternative that the system can choose. The recursive equation for dynamic programming has a choice and uses the maximum operator to determine which of the child nodes to select.

Therefore, combining the two types of tree mingles two orthogonal concerns. The recursive equation for dynamic programming requires combining expected value calculation with the maximum operator. In all other respects, the algorithm is the same.

9.9 Chapter Summary

In this chapter we have described several algorithms for near-optimal configuration of the environment. In order to tackle the complexity of the problem incrementally, we first defined multiple stages of the problem. Each stage is built upon a pervious stage by adding a new dimension to the problem, or by relaxing an assumption. The initial stage problem is in the context of the reactive model. The following stages are all in the context of the anticipatory model.

To solve the problem in the reactive model of configuration, we described two different algorithms. These algorithms tackle the problem of resource allocation among multiple applications in a one-period setup. One of the algorithms uses a greedy selection tactic to gradually find a near-optimal allocation of resources among the running applications. The other algorithm uses a dynamic programming approach to build a solution of the problem by solving its sub-problems. The second algorithm is exact, but solves a slightly different problem.

To solve the problems in the anticipatory model of configuration, we use a dynamic programming approach that tabulates backwards in time. In each case, the dynamic programming algorithm uses resource predictions expressed as a tree. In case of perfect resource predictions, the tree has a single branch, and the problem is relatively simple. The optimal sequence of configuration can be calculated *ex ante*, and this solution remains optimal *ex post*, thanks to the assumption that resource predictions have no uncertainty *ex ante*. In case of imperfect resource predictions, the tree is branching, and each branch describes possible resource paths and their probabilities. We modify the original algorithm to maximize the expected value of accrued utility. With some clever heuristics, we are able to reduce the exponential running time of the algorithm.

We also describe algorithms for imperfect task predictions and for non-perishable resources. In each case, the original dynamic programming algorithm is carefully modified to account for possible future outcomes and their probabilities.

10 Thesis Validation

This dissertation has set out to demonstrate the following thesis statement:

> ***The configuration of the computing environment can be substantially automated by tailoring it to the needs and preferences of an individual user for each task, while simultaneously satisfying the requirements of optimizing utility and practicality while coping with uncertainty.***

To demonstrate this thesis, we have described an approach that has both analytical and runtime parts. The two analytical models of configuration, the reactive and anticipatory, formalize the notion of utility for user's tasks and express automatic configuration as a mathematical problem of maximizing the *expected utility* of the user from the running state of the environment under the constraints of the computing environment.

Using the analytical model as a formal basis, we have designed and implemented a software infrastructure for automatic configuration. Our contributions to the infrastructure are threefold: (1) the Environment Manager, which is responsible for configuration decisions, (2) a framework that can calculate aggregate resource predictions, and (3) a programmatic interface between the prediction framework and its consumers.

In section 1.4 of Introduction, we decomposed the thesis statement into the following claims: (1) the control of the configuration can be substantially automated, (2) configuration can be tailored to individual user preferences so that the requirements of utility and practicality are simultaneously satisfied, and (3) our approach works under uncertainty. In sections 10.1, 10.2, and 10.3, we provide evidence to demonstrate each claim. In section 10.4, we summarize our findings and argue that by the virtue of demonstrating the 3 claims, we have successfully demonstrated the thesis.

10.1 Automating the Control of the Configuration

In this section we address the claim of substantial automation of the control of configuration. Our strategy for demonstrating substantial automation is to compare the configuration of the environment with and without our system, and to provide quantitative and qualitative estimates of the overhead to the user in both cases.

Today, mundane configuration activities such as finding and starting applications, and manipulating their runtime settings require significant manual effort. As explained in the Introduction, manually configuring applications, resources, and devices has a number of drawbacks: it imposes significant cognitive burden on the user and requires expertise that lay users lack or don't want to learn. Furthermore, manually configuring applications or resources can be difficult to do well for even experienced users.

Automating the configuration of the environment rids the user of burdensome administrative chores, while providing an important step towards distraction free, seamless computing in which

users can spend their precious cognitive resources on their tasks and not have to worry about configuration burden.

To argue that our approach substantially automates the control of the configuration, we state and demonstrate two hypotheses with respect to two broad classes of configuration activities: (1) finding, starting and stopping applications, and (2) manipulating application settings to change QoS output or control resource consumption. We use two dimensions of user overhead: (a) the *number of input events*, i.e., the number keyboard strokes and mouse clicks that a user must execute to make configuration happen and (b) the *knowledge* that a user must acquire in order to be able to perform configuration. We estimate knowledge qualitatively, referring to the number of menus, windows, control panels, pop-ups, and help screens that the user must find, access, and read in order to be able to effect certain configuration actions. We don't measure if the user can do manual configuration well or even adequately. The two hypotheses are:

> **Hypothesis 1.1:** The infrastructure for automatic configuration reduces the number of input events that a user must execute to perform configuration activities.
>
> **Hypothesis 1.2:** The infrastructure for automatic configuration reduces the knowledge required to perform configuration activities.

We take a top-down approach to estimating the input events and the knowledge required to per-

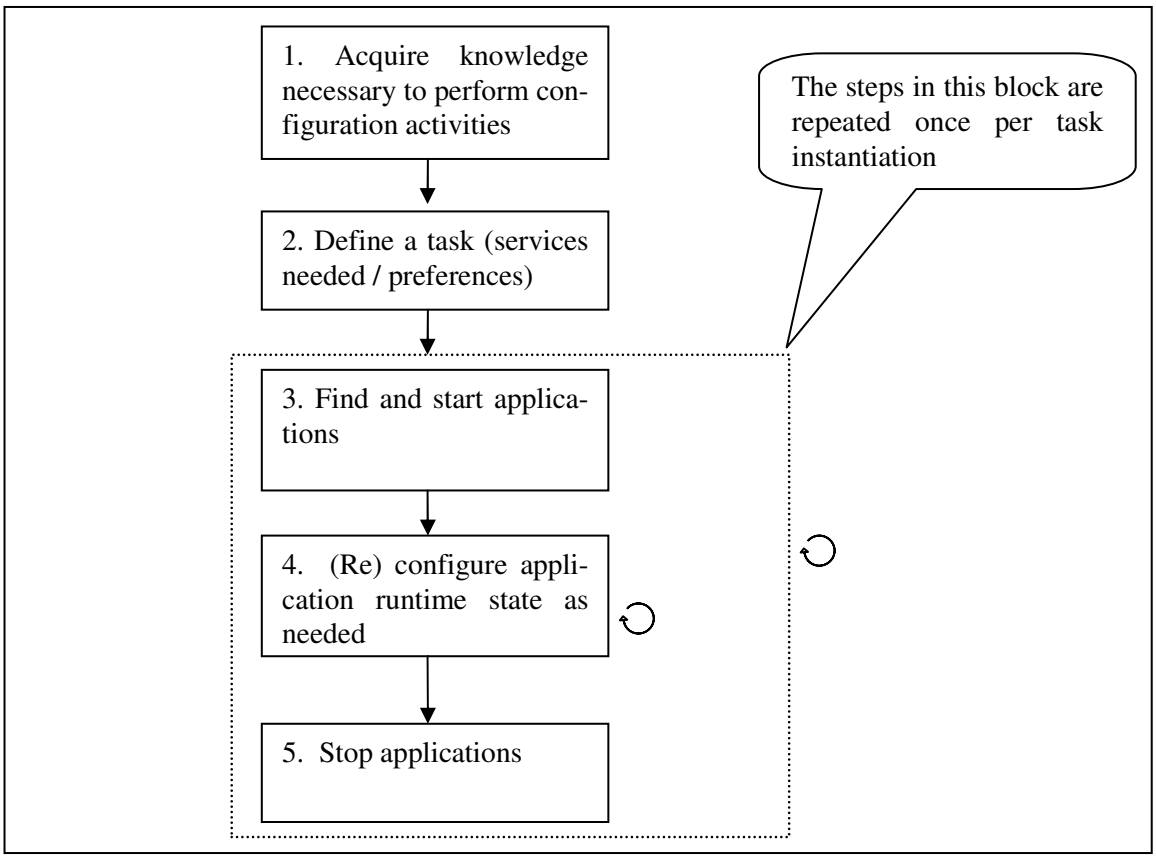


Figure 17: A workflow of a user's configuration activities for one task. Each box shows a configuration phase. Looping arrows next to a configuration phase indicate it is repeatable.

form configuration activities. To that end, we first consider a simplified workflow of a user's computing interactions (shown in Figure 17), with a focus on configuration phases.

All the overhead (either in terms of input events or in terms of knowledge acquisition) that a user might incur is in the following 5 phases:

- Phase 1: acquire knowledge needed to perform configuration,
- Phase 2: define a task in terms of services needed and preferences,
- Phase 3: find and start applications for the task, set their initial runtime state,
- Phase 4: Reconfigure the runtime state of the applications in the task, as needed,
- Phase 5: Stop applications in the task.

Next, we discuss each phase of configuration separately, focusing on the differences in both the frequency as well as the intensity of overhead required under two scenarios: when the user does not have the benefit of the new infrastructure, and when the user does have the benefit of the new infrastructure.

10.1.1 Phase 1: Acquire Knowledge Needed to Perform Configuration

Whether equipped with the new infrastructure or not, the user needs to acquire the knowledge required for configuration activities once per his lifetime (although, the user might need to re-enforce his knowledge from time to time). However, the primary difference in the amount of knowledge required to perform configuration with and without the system is what the user needs to know to do configuration in both cases.

With the new system, the user needs to learn how to define services and specify preferences. We calculated the number of windows, widgets, and help screens that the user needs to know about in order to be able to use the Prism Task Manager to define tasks and input preferences. We estimate that the amount of knowledge is comparable to the amount of knowledge required to manipulate the settings of just one application, RealOne Media Player (see Appendix B). The advantage offered by the infrastructure for automatic configuration is the reduction in the incremental acquisition of knowledge required even with the addition of new service types, new tasks, new applications or new versions of existing applications.

Without the system, knowledge required to make manual configuration is proportional to the number of available applications, application versions, and platforms. Furthermore, without the system, the knowledge required to perform certain activities, e.g., limiting the resource usage of an application is difficult to find even for expert users. Similarly, controlling the QoS settings of applications that do not provide easy-to-use knobs is also very high. Thus, the system provides the user with the ability to "learn-once and apply-everywhere".

We also conducted a usability study [53] to investigate if users can use Prism to input preferences. The participants received a 15-minute tutorial session, which covered not only the knowledge required to use the system, but also background material about resources and applications. Based on the collected evidence, the participants were able to define preferences using plain English instructions. Furthermore, the behavior of the system satisfied the objectives provided in the

instructions. Thus, the results of the study provide further evidence that users can use the new system successfully.

Based on the results our surveys a representative set of applications and the system, we conclude that the knowledge required to perform configuration with the new system is significantly lower.

10.1.2 Phase 2: Define a Task in Terms of Services and Preferences

Configuration activities in Phase 2 are a consequence of the new infrastructure. Without the new infrastructure, activities in phase 2 are not needed. With the infrastructure, the user has to bear the burden of defining the services each task, and for each service, providing the required inputs to define the preference functions and the weights. The overhead of defining a task is only in terms of the input actions, because the knowledge required to define the task has already been fully accounted in phase 1.

The overhead in terms of the input actions required depends on the size and richness of the task, i.e., the number of services and the average number of QoS attributes in each service.

Let:

- ***nSvc*** be the number of services in the task,

- ***nQoS*** be the number of QoS attributes per service.

The number of input events required to define the task completely is ***O(nSvc*(nQoS+1))*** with a constant factor that is in the low single digits. If the user chooses to define custom preferences for each service in the task, then the constant factor is about 5. The user can take advantage of the ability to define and save preferences, or use pre-defined preference templates. Should the user decide to do so, the number of input events is reduced to ***O(nSvc)***.

We used the infrastructure to define a task with two services: "*play video*" and "*browse web*", and custom preferences for 7 dimensions (5 QoS and 2 supplier). We counted 30 mouse events.

10.1.3 Phases 3 and 5: Find and Start Applications, Stop Applications

Without the new infrastructure, the number of input events required for finding and starting applications depends on the number of applications in the task. To find and start each application in the task, the user typically has to execute a small number of input events, anywhere from 1 to 5. If the number of services in the task is ***nSvc*** and the average number of input events required to find and start each application is ***d***, then the number of input actions required to resume the entire task is ***nSvc*d***. For stopping the applications in the task, the number of events required is ***nSvc***. The total number of input events required for resuming and suspending the task once is ***nSvc*(d+1)***.

With the new infrastructure, resuming or suspending the task requires two mouse clicks. This number does not depend on the number of services in the task.

During the lifetime of a task, the user is likely to instantiate the task several times. As the number of task instantiations grows, the user reaps additional benefit from the new infrastructure, because the incremental cost of resuming and suspending a task is much lower with the infrastructure.

10.1.4 Phase 4: Configure / Reconfiguration Application Runtime Settings

In this section, we will refer to make use of the data collected while surveying the applications. We learned about the available knobs to control the quality settings of a small set of applications, and counted the input events required to access the menu items, windows, pop-ups, control panels that need to be navigated in order to change the runtime setting of the applications.

Without the infrastructure for automatic configuration, the number of actions required to change one quality setting requires between 2 and 5 actions. We need to account for multiple services in the task and multiple QoS dimensions per service. For example, in a task with two services: "play video" and "browse web", to perform manual reconfiguration just once, we calculated over 20 input events.

- Real Player:
    - Access "Preferences": 2,
    - Test bandwidth: 4,
    - Select Bandwidth level: 4,
    - Modify CPU settings: 2.
- Internet Explorer:
    - Access "Advanced Options": 3,
    - Modify richness settings: 2 + 1 per checkbox selected,
    - Manage add-ons: 2 + 1 per add-on added removed,
    - Limit bandwidth: 2.

With the new infrastructure, once the services in the task are defined and the preferences are elicited, the user does not need to execute any additional input events for manipulating the runtime state of the applications. Recall, from the discussion about Phase 2, that to define a task with two services ("*play video*" and "*browse web*"), we calculated about 30 input events.

During each instantiation of the task, as the task is active, changing resource conditions might require reconfigurations. Without the new system, the user must execute a large number of input events to perform configuration activities. With the system, there are not input event required by the user, as the system automatically configures the application runtime settings. As the number of required reconfigurations increases, the manual cost of configuration increases linearly, while the infrastructure keeps this cost to virtually none.

10.1.5 Summary of the Results

The benefit of using the infrastructure is obvious. The infrastructure requires a once-per-task cost of 30 input events to define the services and input the preferences. Re-configuration is seamlessly handled by the infrastructure, requiring no intervention by the user. Without infrastructure, while there is no up-front cost, each re-configuration requires 20 input events. So after 2 reconfigurations, the user will start to reap benefits from the infrastructure.

In this section we have demonstrated the thesis claim that our approach substantially automates the control of the configuration. We have demonstrated that our approach reduces the number of input events required by the user to perform two classes of configuration activities: (1) starting and stopping of applications, and (2) re-configuring the runtime state of applications in response

to changes in resources or changes in user's task. We have also demonstrated that the amount of knowledge that the user must acquire in order to perform configuration activities is reduced when using the infrastructure. After the user learns how to use the infrastructure to define services and preferences, she does not need to acquire any more knowledge even if the user must use new applications, new application versions, or applications on different operating platforms.

10.2 Satisfying Utility and Practicality (Reactive Model and Strategy)

In this section, we address the thesis claims about the requirements of automatic configuration: practicality and utility. In this section, we limit our evaluation to the reactive strategy of configuration, leaving evaluation of the anticipatory strategy for the next section.

With respect to the requirement of practicality, we set out to demonstrate the following two hypotheses:

> **Hypothesis 2.1.1 (Resource Footprint):** The resource footprint of the software infrastructure for automatic configuration is comparable to background processes such as virus and spyware checkers and does not exceed 2% of the available resources on a commodity laptop computer that was manufactured 3-6 years ago.
>
> **Hypothesis 2.1.2**: **(Latency)** The latency of configuration actions, measured from the time a user requests a task operation until the infrastructure implements the request, is comparable to the latency of similar interactive applications. In particular, latency that is less than or equal to 1 second is an excellent result, while latency under 7 seconds is acceptable.

To address hypothesis 2.1.1, we measure the utilization of the critical resources (CPU, RAM, network, and battery) by the infrastructure. For CPU, we first measure the system variable called CPU Time. To obtain CPU utilization, we divide the CPU Time of each process by the amount of clock time. For memory, we measure the working set (physical memory) system variable. We don't profile the energy drain of the infrastructure. However, the incremental usage of the energy due to the infrastructure should be proportionate to the CPU usage. Thus, the CPU utilization of the infrastructure should provide a good approximation and a rather generous upper bound for the energy utilization.

The size and the frequency of the messages between the infrastructure components provide a good proxy for the network utilization by the infrastructure.

To address hypothesis 2.1.2, we measure the latency of resume and suspend configuration requests: the total time elapsed between the time when the user requests a task operation from the Task Manager and the time when that request is completed. As mentioned earlier, the latency can be broken down into the following portions:

- Communication overhead between the TM and EM, and EM and Suppliers,
- Data structure creation and manipulation overhead (object creation, parsing of messages, marshalling / un-marshalling, etc),

- Overhead of configuration algorithms (time it takes for a configuration algorithm to complete calculations from the time it was invoked),

- Instantiation overhead. This is time spent by Suppliers to effect changes in the applications.

We only measure the total latency of a configuration request, i.e., we don't provide a breakdown of time spent in the four phases above.

With respect to utility, we have set out to demonstrate the following hypothesis:

> **Hypothesis 2.2:** Using preferences tailored for each task, the infrastructure is able to nearly-optimize user's overall utility of the task.

To demonstrate this hypothesis, we provide both experimental and analytical evidence. First, we use the same experimental setup used to validate hypotheses 2.1.1 and 2.1.2 to we measure the overall utility of the configuration selected by the software and then separately, using offline calculations, we check whether that configuration has the highest utility of all possible configurations. Next, we provide theoretical justification why the software finds configurations that either optimize or nearly-optimize the overall utility.

10.2.1 Experimental Setup

Our experiments were conducted on an IBM ThinkPad T30 laptop manufactured in 2002. The laptop has a 1.8 GHz Pentium 4 processor and 512 MB of RAM. We specifically selected this laptop, because it is significantly resource-constrained when compared to recent laptop models and serves as a good proxy for a modern handheld device like a smart phone or a palm-held computer. The laptop runs Windows XP operating system with service pack 2, and Java Virtual Machine, version 1.4.2. All infrastructure components implemented in Java use this virtual machine version. The laptop supports multiple voltage-scaled levels of the CPU.

In our experiments, we launched the following infrastructure components:

- The Prism Task Manager, implemented using Java, and launched in its own Java Virtual Machine,

- Environment Manager, implemented a combination of Java and ANSI C programming languages, and launched in its own Java Virtual Machine,

- Suppliers for the following applications: Microsoft Word providing "edit Text" service, Microsoft Excel providing "spreadsheet" service, Windows Media Player providing "play Video" service, JMF (Java Media Framework) Player. The JMF Media Player supplier is implemented in Java, while the suppliers of the Microsoft applications are implemented in C/C++,

- Half a dozen supplier stubs providing richness to the environment. These script-controlled programs can receive and transmit Aura protocol messages. From the point of view of Prism and the Environment Manager, these stubs appear like real suppliers. The purpose of running this stubs is to increase the size of the capability space that the Environment Manager must search,

- Proxies for resource managers. These are configuration files providing resource availability information to the Environment Manager.

Our experimental task has three services: "*play Video*", "*browse Web*", and "*edit Slides*". Based on both actual as well as stub suppliers, there were 80 alternative supplier assignments for the Environment Manager to consider.

We conduct two experiments using the same setup. The purpose of our first experiment is to collect measurements of resource utilization when the infrastructure resumes and suspends tasks. In this experiment, we measure the latency of resume and suspend requests as well as the CPU and RAM usage required for resume and suspend. In each trial, we resume the task, let it run for about 20 seconds, and then suspend it. We repeat that sequence ten times. We measure the latency using code injected into the source of Prism. In the background, we measure the CPU and memory utilization of the infrastructure using standard operating system APIs and tools (Process Explorer from Microsoft SystInternals battery of utilities).

In our second experiment, we are interested in the resource utilization of the infrastructure as it continually performs monitoring and re-configuration. We turn on task monitoring in the Environment Manager, setting the monitoring interval at 30 seconds (monitoring interval is the length of the discrete time window). For as long as a task is active, the Environment Manager recalculates the optimal configuration using the most recent values of the input variables every 30 seconds. Every time the configuration is re-calculated, the results are communicated to the Task Manager, which must process that information. In this experiment, we resume the task once, and let it run for 5 minutes. 5 minutes allows sufficient time over which to average the resource usage of the Environment Manager.

As a basis for comparison, we measure the resource footprint of a background process, the Symantec Antivirus Real Time Virus Scan. We believe that this program makes a good comparison, because it executes in the background continually, using the CPU in short bursts and at a low utilization rate.

10.2.2 Addressing Practicality

In this section, we report the results of the experiments in support of hypotheses 2.1.1 and 2.1.2. We report the average, minimum, maximum, and standard deviation of the latency of resume and suspend configuration requests over the experimental run. In addition, we also report the average by excluding the first experiment run (the first trial takes longer, as the caches need to be warmed up).

For resume, we measure the time elapsed between the time when the request is made from the Task Manager and the time when the response to that request is received from the Environment Manager. For resume, the latency excludes any file input / output, because that's not part of the overhead of the infrastructure. For suspend, we measure latency with and without the file input / output.

As the results in Table 9 demonstrate, the average latency of resume is just under 1.5 seconds (1,500 milliseconds) and the average latency of suspend excluding file input / output is under 150 milliseconds.

Table 9: The latency of configuration actions in milliseconds.

The statistics are taken over 10 trials, except for "Avg-x-First", which excludes the first trial.

	Resume Excluding File I/O	**Suspend Including File I/O**	**Suspend Excluding File I/O**
Avg	1,433	2,049	135
Avg-x-First	1,284	2,206	139
Min	811	731	90
Max	2,623	3,775	190
StDev	507	1,037	33

We note that only a small portion of the aggregate latency is due to the configuration algorithm. We isolated the algorithm as a callable library and experiment with the algorithm separately. Our tests showed that the latency of configuration decisions is on the order of dozens or hundreds of nanoseconds. In section 10.3, we will report additional results on the latency of the algorithm.

Next, we report on the resource utilization of the infrastructure. We are interested in the average rate of CPU utilization by the processes of the infrastructure. The CPU utilization of the infrastructure is due to two things: (1) resume and suspend of tasks, and (2) monitoring and reconfiguration. In the first experiment, we measured the resource utilization from resume and suspend operations. In the second experiment, we measured resource utilization due to monitoring and reconfiguration. To report the average CPU utilization by the infrastructure, we estimate: (1) how many times the infrastructure would need to perform resume and suspend, and (2) how many times the infrastructure would need to perform monitoring / re-configuration. Over a 60 minute time interval, we estimate the infrastructure to do no more than 5 resume / suspend, and exactly 120 monitoring / re-configuration calculations.

Table 10: The resource utilization of the infrastructure, the task, and other processes.

	Working Set (KB)	CPU Time Res/Susp (sec)	Number of Res/Susp in 60 mins	CPU Time Mon + Re-conf. (secs)	CPU Util (%)
Environment Mgr	11,656	1.11	5	0.21	0.84%
Task Mgr	17,672	0.60	5	0.14	0.56%
Supplier Stubs	8,088	0.28	5	0.05	0.21%
Internet Explorer Supp	4,476	0.04	5	0.01	0.04%
PowerPoint Supp	4,654	0.04	5	0.01	0.04%
Total, infrastructure	46,546	2.07	5	0.42	1.69%
Java JMF Player	30,980	N/A	N/A	N/A	39.2%
Internet Explorer	28,184	N/A	N/A	N/A	7.72%
PowerPoint	11,680	N/A	N/A	N/A	0.11%
Symantec Antivirus	54,540	N/A	N/A	N/A	1.08%
All Other Processes	300,070	N/A	N/A	N/A	5.50%
Total System Usage	472,000	N/A	N/A	N/A	58.58%

The results are shown in Table 10. The first 6 rows show results for processes that belong to the infrastructure. The 7th row is the total for the infrastructure. Rows 8-10 show the resource utilization of the applications that were selected for the user's task. Row 11 is the RtvScan process of the Symantec Antivirus. Row 12 shows the approximate resource utilization of all other processes running, and row 13 shows the total system usage. The first and last columns are respectively the physical memory and CPU percentage utilized by each process. The second and fourth columns are the CPU times measured during experiments one and two (these are only relevant for the infrastructure). For the processes belonging to the infrastructure, i.e., for rows 1-6, to calculate average CPU utilization (the variable in the last column), we have used the following formula:

(CPU_Time (Susp/Resume) * 5 + CPU_Time (Mon/Reconf) * 120) / 60

This formula allows us to convert CPU usage in terms of seconds into average percentage CPU utilized. We estimate about 5 resume and suspend operations during a one-hour continuous execution of the system, and exactly 120 monitoring / reconfiguration calculations.

As we can see from the table, the CPU utilization of the entire infrastructure is under 1.7% of the total CPU available, and in the same range as the CPU utilization of the Symantec Antivirus Real Time Virus Scan (1.69% vs. 1.08%). The total working memory set of the processes of the infrastructure is under 47 MB, which is about 9% of the total RAM of the 512MB available on that system. This number is a little high. However, we note that three of the processes belonging to the infrastructure execute in their own Java Virtual Machines (JVM). A "Hello, World!" Java program executing in its own JVM has a working set of 8 MB. A Java Virtual Machine implementation on a more resource constrained device is likely to have a much smaller memory footprint. Furthermore, there are opportunities for reducing the memory usage of the infrastructure by combining multiple components into a single VM. We have successfully integrated infrastructure components into an OSGi framework, which allows sand-boxed execution of multiple components in a common virtual machine. Deployment in this manner reduces the memory utilization of the infrastructure by about 6 MB per infrastructure component that is executed in the common virtual machine.

In conclusion, we believe that the overhead of the infrastructure, both latency as well as resource usage, is well within an acceptable range. Based on these results, we claim that the implementation satisfies the thesis claims set forth in the Introduction.

10.2.3 Addressing Utility

Our implementation of the reactive strategy of configuration uses the `OptInstUtility` algorithm described in section 9.2. `OptInstUtility` iterates through the list of candidate supplier assignments in descending order of supplier utility. For each supplier assignment, we invoke the `AMaxQoS` algorithm in order to calculate the maximum possible QoS for that supplier assignment under the current available level of the resources. We use a condition to check if we have found the best overall utility and break from the loop early, without having to iterate through the list of all supplier assignments.

We used the experimental setup described in section 10.2.1. The experimental task had 3 services: "*play video*", "*browse web*", and "*edit slides*". We injected the environment using a combination of actual suppliers and stubs, creating 80 possible supplier assignments. The `OptInstUtility` algorithm iterated through 36 supplier assignments before breaking from the loop.

In order to check whether the configuration found by the algorithm had the best possible overall utility, we used another algorithm to verify the results. The verifying algorithm was similar to `OptInstUtility`, but did not use a break condition to terminate the loop early, and instead iterated through each of the 80 supplier assignments. The verifying algorithm used `DMax-QoS_ResSieve` algorithm in order to calculate the best possible QoS utility for each supplier assignment over a fine-grained sieve of possible resource levels. In our experiments, the configuration selected by `OptInstUtility` was also selected by the verifying algorithm, indicating that `OptInstUtility` was optimal. However, in general we can not claim that `OptInstUtility` can find the best configuration. In order to guarantee the optimality of `OptInstUtility`, we must be assured that `AMaxQoS` finds the optimal QoS utility for each supplier assignment. Unfortunately, that's not possible to guarantee.

In the following section, we present comparisons between several algorithms for configuration based on accrued utility. In particular, we compare the reactive and anticipatory algorithms against each other and against some naïve schemes.

10.3 Claims about Uncertainty

In this section, we address claims about uncertainty. Recall, from the Introduction, that we set out to demonstrate two claims about uncertainty. First, we claim that uncertainty in the resource predictions can be encoded and transmitted efficiently at runtime. Second, we claim that the requirements of utility and practicality can be satisfied in the face of uncertainty in the predictions.

We use analytical reasoning to validate the first claim. To validate the second claim, we use experimental results from simulations.

10.3.1 Predictions with Uncertainty can be Encoded and Transmitted Efficiently

In section 8.4, we described the aggregate resource prediction protocol which is designed to allow the consumer of predictions to request predictions while controlling the prediction parameters that affect accuracy and cost of predictions: (1) the prediction scale, (2) the prediction horizon, and (3) the prediction detail. In the protocol, aggregate predictions are encoded using a tree data structures. The tree approximates possible future paths of the available level of the resource and the associated probability of each path. The tree has a higher branching factor closer to the root. Deeper nodes of the tree have only single branches. This allows to dramatically decrease the size of the tree, reducing the generation and transmission cost of predictions. The shape of the tree is defined by the following parameters:

- $\boldsymbol{S}$, the scale of prediction in seconds,
- $\boldsymbol{H}$, the horizon of prediction in seconds (here, $\boldsymbol{H}$ must be divisible by $\boldsymbol{S}$),
- $\boldsymbol{BF}$, the branching factor,
- $\boldsymbol{D}$, the maximum depth at which branching factor is $\boldsymbol{BF}$ ($\boldsymbol{D} \leq \boldsymbol{H/S}$).

Let $\boldsymbol{nWin = H / S}$. This is depth of the tree. The number of points required to describe the prediction tree is the sum of the following terms:

- $1+BF^2+BF^3+ \ldots +BF^D + BF^D * (nWin - D) = (BF^{D+1} - 1) / (BF - 1) + BF^D * (nWin - D)$.

We argue that the transmission cost of the tree can be kept low while assuring that enough predictive information is communicated. Furthermore, the consumer of the predictions can control the cost of generating and transmitting. This is useful when the resources are scarce. For example, when the network link between the aggregate predictor and the consumer has limited capacity and is also used by the applications in the user's tasks, then the Environment Manager can request to reduce the size of the predictions at the expense of prediction detail and accuracy.

Table 11 and Table 12 show the number of nodes in the prediction tree based on different values of the tree parameters. The size of each prediction report message is proportionate to the number of nodes in the tree. Prediction report messages are sent at a frequency equal to the prediction scale. At a prediction scale of 30 seconds, the transmission cost of the predictions is reasonable for a range of the parameter values. For example, for ***BF*** = 3, ***D*** = 4, and ***nWin*** = 30, the prediction tree has 2,227 nodes. Assuming each node requires 4 bytes, prediction reporting should require 15 KB of network every 30 seconds. We believe this is a reasonable overhead.

Table 11: The number of nodes in a prediction tree with branching factor 3.

Branching Factor = 3		***D*, Branching Depth**				
		2	**3**	**4**	**5**	**6**
***nWin*, Decision Horizon**	**15**	130	364	1,012	2,794	7,654
	20	175	499	1,417	4,009	11,299
	25	220	634	1,822	5,224	14,944
	30	265	769	2,227	6,439	18,589
	40	355	1,039	3,037	8,869	25,879
	50	445	1,309	3,847	11,299	33,169

Table 12: The number of nodes in a prediction tree with branching factor 4.

Branching Factor = 4		***D*, Branching Depth**				
		2	**3**	**4**	**5**	**6**
***nWin*, Decision Horizon**	**15**	229	853	3,157	11,605	42,325
	20	309	1,173	4,437	16,725	62,805
	25	389	1,493	5,717	21,845	83,285
	30	469	1,813	6,997	26,965	103,765
	40	629	2,453	9,557	37,205	144,725
	50	789	3,093	12,117	47,445	185,685

The prediction reporting protocol is not only efficient in terms of transmission cost, but also exposes a cost-accuracy trade-off by providing the consumer with the ability to select prediction parameters that control prediction accuracy and detail. We believe this is a novel feature in interface design.

10.3.2 Utility and practicality In the face of Uncertainty

Our evaluation of the utility and practicality of the anticipatory strategy is based on simulation. Due to the lack of supporting infrastructure for providing task predictions and online resource predictions, we do not have a functioning infrastructure that can make configuration decisions based on the anticipatory strategy of configuration.

However, we believe that we can sufficiently evaluate claims about practicality and utility of the anticipatory strategy of configuration. We have designed a standalone test bed that implements the anticipatory algorithms. The test bed uses input data that is consistent with the models of configuration, and invokes the anticipatory configuration algorithms much the same way that the Environment Manager would invoke. When requisite functionality to provide predictions becomes available in the Task Management and Resource Management layers of the infrastructure, it should be a matter of modest amount of engineering effort to integrate our algorithms into the Environment Manager and demonstrate the feasibility of the anticipatory strategy of configuration.

In order to ensure that the anticipatory algorithm will be usable in practice, we have imposed much tighter evaluation criteria on the overhead of the algorithm. We require that both in terms of the latency and in terms of the resource footprint, the implementation of the anticipatory algorithm add no more than 10% of the overhead of the entire infrastructure. Based on the results reported in section 10.2.2, the latency of each configuration decision should not exceed 150 milliseconds, average CPU utilization of the algorithm should not exceed 0.16% of the available CPU, and the memory utilization of the algorithm should not exceed 4 MB. Based on those criteria, we have formulated the following hypotheses:

> **Hypothesis 3.2.1 (Resource footprint):** The resource footprint of the anticipatory algorithm for automatic configuration expressed as a percentage of the resource footprint of the entire infrastructure does not exceed 10%.
>
> **Hypothesis 3.2.2 (Latency)**: The latency of one configuration decision by the anticipatory algorithm does not exceed 150 milliseconds.

To evaluate the utility of configuration decisions, we need to measure the *ex post* accrued utility of a sequence of configuration decisions made by the anticipatory algorithm using *ex ante* imperfect predictions. We can compare that utility with one of two other measures of accrued utility:

- the *ex post* accrued utility of a sequence of decisions made by the anticipatory algorithm that has the benefit of *ex ante* perfect predictions.
- the *ex post* accrued utility of the reactive algorithm.

Comparison with the first measure will give us an indication of how close is the anticipatory algorithm to the perfect accrued utility that is only possible to attain with an "oracle" prediction. Comparison with the second measure will give us an indication of improvement of the anticipa-

tory algorithm over its reactive counterpart. Therefore, we have formulated the following hypotheses:

> **Hypothesis 3.3**: The anticipatory algorithm that uses *ex ante* imperfect predictions nearly-optimizes the *ex post* accrued utility when compared to the perfect accrued utility. Furthermore, the anticipatory algorithm that uses *ex ante* imperfect predictions achieves better *ex post* utility than the reactive algorithm.

For evaluation, we have implemented the following algorithms:

- **Reactive**, the reactive algorithm that solves the Stage 1 problem (section 9.2),
- **Oracle**, the anticipatory algorithm that solves the Stage 2 problem (section 9.3),
- **Anticipatory**, the anticipatory algorithm that solves the Stage 3.1 problem (section 9.4),[6]
- **Random**, an algorithm that first selects and commits to a random suite of suppliers, randomly modifies selects capability points from the profiles of those suppliers so that resource availability constraint is not violated.

Our experimental task had two services: "*browse web*" and "*play video*". We profiled 3 video players and 3 web browsers. The author of this dissertation provided preferences for the task. Our experimental setup had three resource instances: CPU, physical memory, and downstream network bandwidth. For CPU and memory availability, we used a number of known pattern and bounding predictors. For memory availability, we used an ARMA(5,4) linear recent history predictor as well as a number of known pattern and bounding predictions.

10.3.2.1 Addressing Practicality

First we conducted an experiment to measure the latency of the anticipatory algorithm. We set the prediction parameters as follows: ***BF*** = 3, ***nWin*** = 24, and varied ***D*** from 2 to 6. To accurately measure the overhead cost of one configuration decision, we averaged over 500 invocations of the algorithm. We conducted experiments using 2 modes of the CPU: maximum performance and slowest possible performance. The results for latency are show in Table 13.

Table 13: The latency of one configuration decision, in milliseconds (***nWin*** = 25).

	Branching Depth				
	2	3	4	5	6
Max	0.5	1.95	7.65	24.2	72.5
Slow	1.71	6.15	23.3	72	201.5

[6] Due to the lack of experimental data that measures the battery drain of applications, we did not experiment with the algorithm that solves the Stage 3.3 problem involving non-perishable, non-renewable resources.

During the experiments, the CPU was utilized at 100%, and about 95% of the utilization was due to the process executing the algorithm. Using a prediction scale of 30 seconds, we calculate the CPU utilization of the algorithm as follows: Latency / 30,000. The results for average CPU utilization are shown in Table 14.

Table 14: Average CPU utilization due to the **Anticipatory** algorithm (***nWin*** = 25).

	Branching Depth				
	2	3	4	5	6
Max	0.002%	0.007%	0.026%	0.081%	0.242%
Slow	0.006%	0.021%	0.078%	0.240%	0.672%

As we can see from the results show in Table 13 and Table 14, both hypotheses, 3.2.1 and 3.2.2, are satisfied when ***D*** is less than 5. The additional resource overhead added by the anticipatory algorithm is under 10% of the total infrastructure resource overhead when executing the reactive strategy. Similarly, the additional latency due configuration decisions made by the anticipatory algorithm is less than 10% of the infrastructure's total latency when making configuration decisions using the reactive strategy.

10.3.2.2 Addressing Utility

In this section, we present experiments to demonstrate the claim about utility. Based on the results in the previous sub-section, in our experiments to evaluate utility we have set the parameters of the anticipatory algorithm as follows: ***BF*** = 3, ***D*** = 3, and ***nWin*** = 24. Using these values will ensure that the latency and resource overhead are acceptable, and the simultaneous satisfaction of utility and practically under uncertainty.

Recall that we are interested in *ex post* accrued utility of the following 4 algorithms[7]:

- **Reactive**, the reactive algorithm that solves the Stage 1 problem (section 9.2),
- **Oracle**, the anticipatory algorithm that solves the Stage 2 problem (section 9.3),
- **Anticipatory**, the anticipatory algorithm that solves the Stage 3.1 problem (section 9.4),
- **Random**, an algorithm that first selects and commits to a random suite of suppliers, randomly modifies selects capability points from the profiles of those suppliers so that resource availability constraint is not violated.

[7] Due to the lack of experimental data that measures the battery drain of applications, we did not experiment with algorithms that solve problems in Stages 3.3, 4, and 5.

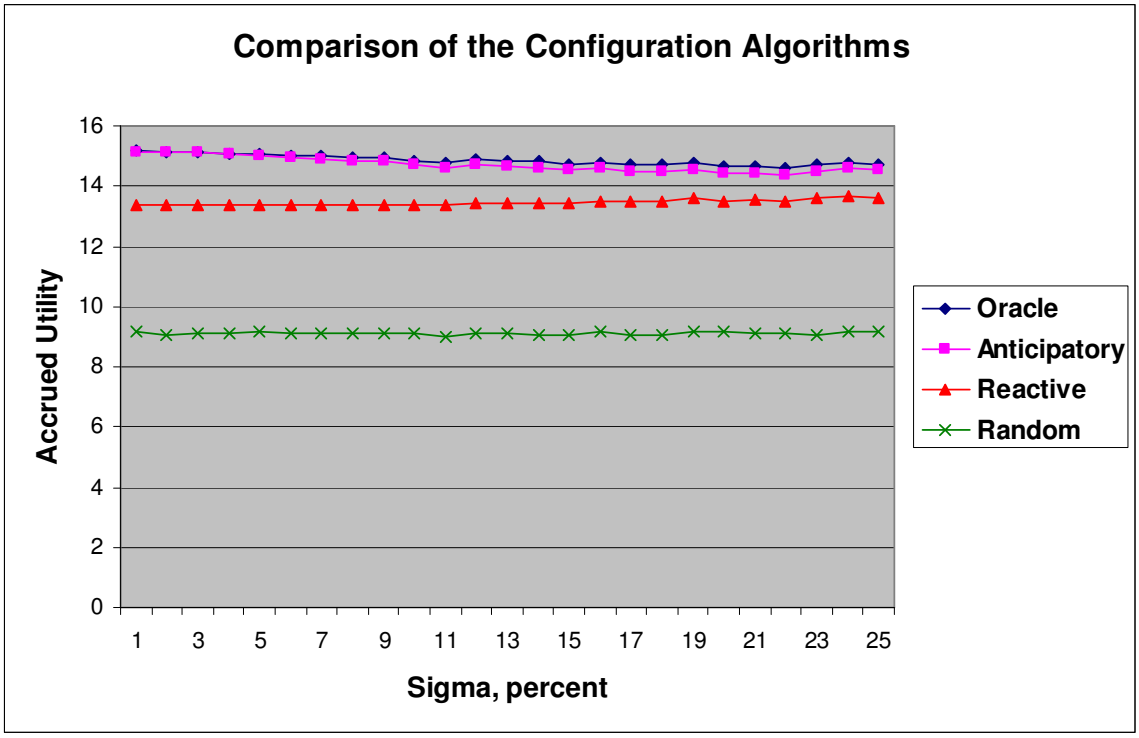


Figure 18: Comparison of configuration algorithms (average over 500 simulation trials).

We use Monte Carlo simulation to estimate the mean value of *ex post* accrued utility across a number of realizations of available resource paths. Each simulation has 500 trials. In each trial, we use the prediction model of resource availability to generate an actual resource path, one value at a time, and invoke each algorithm to make a sequence of configurations. The **Reactive**, **Anticipatory**, and **Random** algorithms make one configuration decision after observing one realized value of the resource. The **Oracle** algorithm is invoked in retrospect, after all realized values of the resource are known. Thus, the **Oracle** algorithm has the benefit of perfect predictions.

We conducted experiments by varying the accuracy of the predictions. Recall that the accuracy is concisely captured by the *sigma* parameter, the error of the predictions. We expressed *sigma* as a percentage of the mean value of the prediction. When *sigma* is small, predictions are more accurate. As the value of *sigma* gets larger, the accuracy of predictions decreases. We varied *sigma* from 1% to 25%. The results are shown in Figure 18. Based on the experiment duration of 24 time units, maximum possible accrued utility is 24. **Oracle** algorithm is the best, followed by the **Anticipatory**, then **Reactive**, then **Random** algorithms. Notice that the advantage of the **Anticipatory** algorithm slowly disappears as *sigma* increases. To emphasize the gain of each algorithm over the next best algorithm, let's define the following measures:

- **Gain_Oracle** is the difference between the utility of Oracle and Anticipatory,
- **Gain_Anticipatory** is the difference between the utility of Anticipatory and Reactive,
- **Gain_Reactive** is the difference between the utility of Reactive and Random,
- **Total_Gain** is the difference between the utility of Oracle and Random.

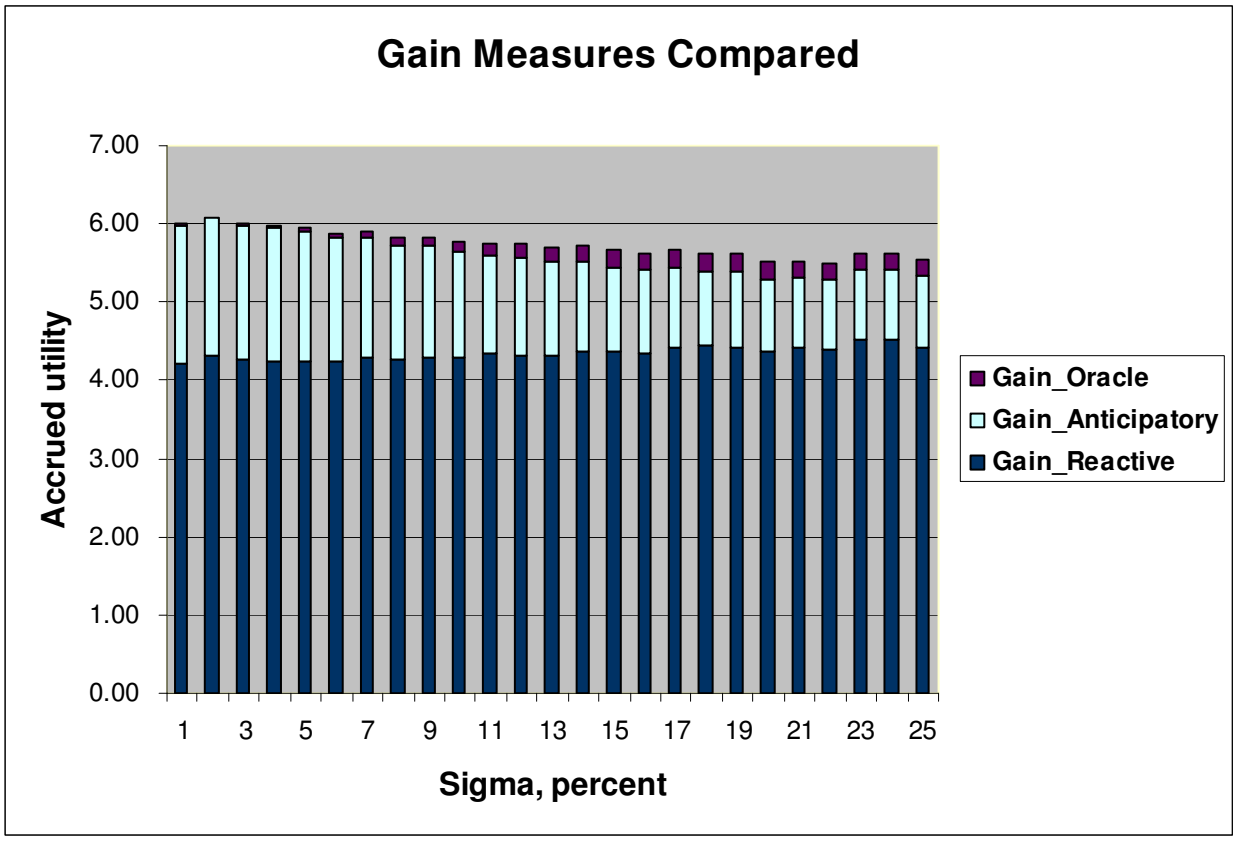
Furthermore, let's define **Percent_Gain_xxx** to be the percentage of **Gain_xxx** in **Total_Gain** for each gain measure "xxx".

Let's consider a different interpretation of the results by comparing the size of the different gain measures as a percentage of the total. As Figure 19 shows, **Gain_Anticipatory** is quite significant when compared to the **Gain_Reactive**, ranging from 42% (for *sigma* = 0.01) to 20% (for *sigma* = 0.25). Furthermore, **Gain_Oracle** (i.e., the gap between the **Oracle** and **Anticipatory**) ranges from 0.17% to 4.51% as a percentage of **Total_Gain**. In the face of already optimizing behavior of the **Reactive** algorithm, we claim that the gain of Anticipatory is significant, and its handicap relative to Oracle is rather insignificant.

Throughout this dissertation, we have repeatedly mentioned that the advantage of the **Anticipatory** algorithm over the **Reactive** algorithm is due to the penalties from switching suppliers. We investigated the effect of the magnitude of the penalties relative to the non-penalty portion of instantaneous utility.

In the experiments shown above (Figure 18 and Figure 19), the magnitude of the penalty of changing each supplier is 0.17. Thus, switching one supplier costs -0.17 in accrued utility, and switching both suppliers costs -0.34. Compared to the maximum possible instantaneous utility (1.00), the penalties are 17% and 34%, respectively. Compared to the average instantaneous utility of the **Anticipatory** algorithm during the experiment (see Figure 18), the penalties are 26% and 52%, respectively.

Figure 19: Comparing the gain measures (average over 500 simulations trials).

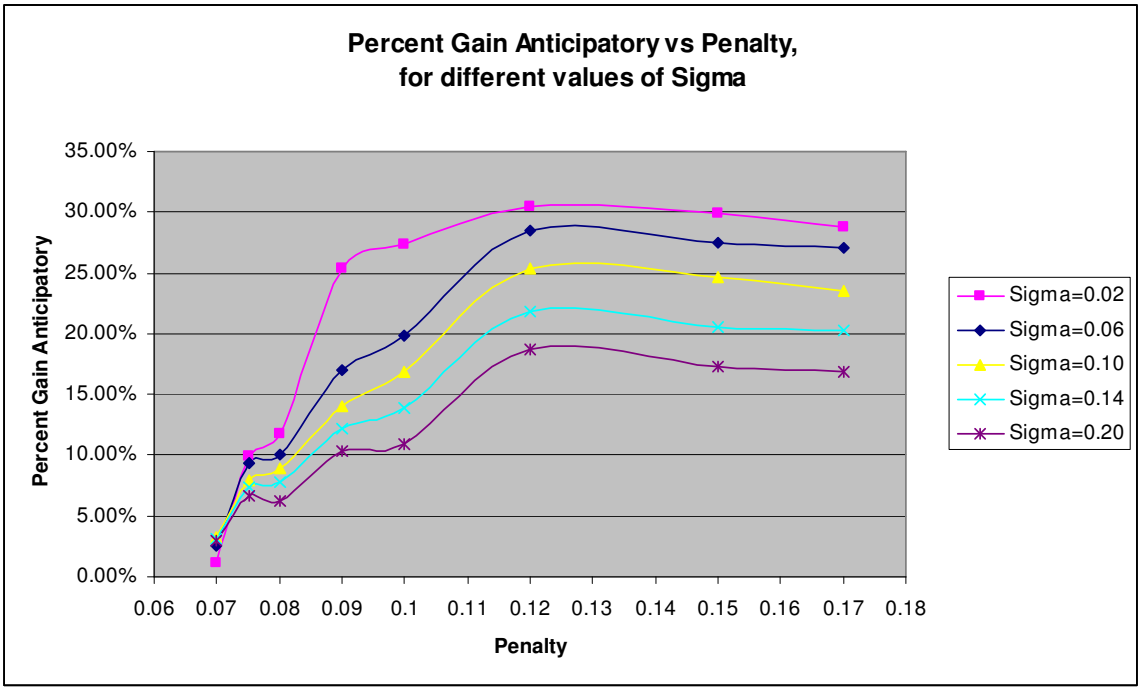


Let's review the expected effect of penalties on the difference in the accrued utility of **Anticipatory** and **Reactive**

- when penalties are zero, there should be no difference in the accrued utility of the **Anticipatory** and **Reactive** algorithms because **Reactive** is provably optimal over time,
- when penalties are very small, the difference should be negligible,
- as the magnitude of the penalties increases, we should be able to observe an increase in the advantage of **Anticipatory** over **Reactive**.

By varying the magnitude of the penalties from 0.01 (negligible) to 0.50 (very high), we observed a ramp-up in the **Percent_Gain_Anticipatory** as penalty increased from 0.07 to 0.12 (Figure 20).

Figure 20: The effect of varying the magnitude of the penalties on **Percent_Gain_Anticipatory**.



We investigated why there is such a significant ramp-up in **Percent_Gain_Anticipatory** as penalties increase from 0.07 to 0.12. The advantage of the **Anticipatory** algorithm is due to the magnitude of the penalties. As the penalties are negligible, the **Reactive** algorithm is able to switch to a supplier assignment with a higher overall instantaneous utility without significant loss to accrued utility. As the magnitude of the penalties gets larger, the cost of change starts to chip away from the accrued utility of the **Reactive** algorithm. When the magnitude of the penalties gets very large, the **Reactive** algorithm will not find it rewarding to switch at all, instead opting to find a better resource allocation and configuration among the currently running suppliers.

We calculated the difference in the instantaneously utility (without penalty) of the best and worst supplier assignments for a large set of possible levels of resource availability. The values of those differences fell in the range, [0.108,0.17]. The upper value of that range, 0.17, is the critical level for the sum of both penalties that makes switching prohibitively expensive for **Reactive** (dividing by two gives us the critical level of the magnitude of one penalty, 0.95). Furthermore, we calculated the difference between the instantaneous utility (without penalty) of the best and worst supplier assignments while allowing only one supplier to change. The values of those differences fell in the range of [0.082,0.098]. The upper value of that range, 0.098, is the critical level for the magnitude of the penalty that makes switching expensive for **Reactive**. Not surprisingly, the ramp-up range observed earlier, (0.07,0.12), includes the critical thresholds.

The significance of the observed values depends on the preferences of the user and the profiles of the available suppliers. Under different preferences, we might observe different thresholds. However, we have discovered an important insight. Because the thresholds for the critical values of the penalties can be calculated up-front with relatively little cost, the infrastructure for automatic configuration can calculate the expected gain of using the **Anticipatory** algorithm and make a determination whether to use it or the less expensive **Reactive** algorithm.

But what is a reasonable magnitude for the supplier change penalty? By calculating the amount of time that switching a supplier takes, we can obtain a lower bound for the value of the penalty relative to the maximum value of instantaneous utility. We argue that the penalty should be higher than that, and perhaps significantly higher, in order to account for disrupting user's mental state and concentration. Let's reasonably assume that switching a supplier interrupts user's task for 5 seconds (the actual switching time might be a little faster). Relative to a prediction scale of 30 seconds, a 5 second interruption results in a 17% penalty. So a reasonable lower bound for the penalty of switching one supplier should be 0.17.[8] And much larger penalties might be necessary to properly account for disrupting the user.

Based on the results of the experiments, we claim the following conclusions:

- The **Reactive** strategy of configuration provides significant improvement (~46%) in accrued utility, when compared to a random strategy that merely finds feasible configurations,

- The **Anticipatory** strategy of configuration improves modestly (7%-13%) over the **Reactive** strategy. The improvement in the **Anticipatory** strategy is more pronounced when predictions are more accurate. However, the **Anticipatory** strategy shows reasonably improvement even for values of *sigma* as high as 25%,

- Considering the optimizing strategy of the **Reactive** algorithm, we believe it is worthwhile to compare the improvement of **Anticipatory** over **Reactive** (**Gain_Anticipatory**) with improvement of Reactive over Random (**Gain_Reactive**). **Gain_Anticipatory** expressed as a percentage of **Gain_Reactive** ranges from 42% to 21%,

- The Anticipatory strategy of configuration nearly optimizes accrued utility, as compared to the accrued utility of the Oracle strategy.

---

[8] Our choice of setting penalty to 0.17 in our first set of experiments was in part justified by this argument.

Based on these results, we claim that we have demonstrated hypothesis 3.3. Furthermore, we claim that our approach satisfies the requirement of utility under uncertainty.

10.4 Summary: Demonstrating Thesis

In this section, we first verify that each claim made in the Introduction has been addressed. Next, we argue that by the virtue of addressing each claim, we have demonstrated the thesis of this dissertation.

Table 1 in the introduction set forth the claims required to demonstrate the thesis of this dissertation.

In section 10.1, we addressed claim 1. We argued that the control of the configuration can be substantially automated by demonstrating that user's overhead is reduced when using the new infrastructure for configuration. Our evidence was based on counting the number of input events that the user must execute to effect configuration with and without the infrastructure. Furthermore, our evidence was based on qualitative comparison of the knowledge that the user must acquire to perform configuration both with and without the new infrastructure.

In section 10.2, we addressed claims 2.1 and 2.2. First, in section 10.2.2, we provided experimental evidence that the infrastructure has low runtime overhead in two dimensions: latency of configuration actions and resource footprint. Next, in section 10.2.3, we demonstrated that in the experimental setup used, the configuration decisions of the infrastructure maximize user's utility. We further argued that the algorithms implemented in the infrastructure can not guarantee to maximize the utility, but are guaranteed to be nearly-optimal in all cases.

In section 10.3, we addressed claims about uncertainty, 3.1, 3.2, and 3.3. In section 10.3.1, we demonstrated that the interface for aggregate resource prediction encodes predictive uncertainty efficiently, and also exposes a cost-accuracy trade-off to the prediction consumer. This supports claim 3.1. In section 10.3.2.1, we provided experimental evidence in support of 3.2, i.e., the anticipatory algorithm of configuration has low overhead. And lastly, in section 10.3.2.2, we demonstrated that the anticipatory algorithm nearly-optimizes the accrued utility of the user. This supported claim 3.3.

By the virtue of satisfying all the claims set forth in the introduction, we argue that we have demonstrated the thesis of this dissertation.

11 Discussion, Future Work, and Conclusion

In this chapter we critically reflect on the entire thesis. In section 11.1, we discuss the design decisions we made while developing our approach. We describe alternatives that we considered, compare our decisions with the alternatives, and justify our decisions. In section 11.2, we present ideas for future work. Some of the future work requires only a modest amount incremental effort to implement in the existing infrastructure. Others go well beyond the scope of our current work and require significant amount of research and implementation effort. In section 11.3, we enumerate the contributions of this thesis to the state of computer science and software engineering research. In section 11.4, we summarize the work and conclude the dissertation.

11.1 Design Decisions Made and Alternatives Considered

11.1.1 Model of Preferences

In our work, we use a linear-additive model of preferences (see section 5.2.1). The linear-additive model of preferences is rooted in microeconomic theory and has been used in decision theory to help individuals make decisions or to help understand the decisions made by individuals.

There are other ways to express preferences. In this section we discuss whether other preference models might be more appropriate, more expressive, or result in other configuration choices.

One alternative is a model that Sousa proposed in his thesis. Sousa's model is a multiplicative model of preferences in which weights are used as exponents and dimension-wise utilities are multiplied to obtain overall utility. The equation for overall utility in Sousa's model is as follows:

$$U_{overall}(SvcSet) = \prod_{i=1}^{n} \left(h_{S_i}^{x_{S_i}} F_{S_i}^{W_{S_i}} * \prod_{\forall d \in QoS\,\dim(S_i)} f_{S_i,d}^{w_{S_i,d}} \right)$$

We first compare our model to Sousa's for expressiveness. We argue that in terms of expressiveness, the two models are equivalent, because both models allow expressing preferences for each dimension of concern separately. Furthermore, an additive model can be transformed into a multiplicative one and vice-versa. In particular: (1) any linear-additive model such as ours can be transformed into an equivalent multiplicative model by using exponentiation and (2) any multiplicative model such as Sousa's can be transformed into an equivalent linear-additive model by using logarithm.

In order to compare an additive and a multiplicative model of preferences in terms of configuration choices made, we need to understand how each model orders the same set of configuration alternatives. For that, let's consider a simple configuration problem in which there are two dimensions of concern, e.g., frame rate and frame size. Let:

- frame rate range from 6 to 24 frames per second,

- frame size range from 10,000 to 1,000,000 pixels.

We use sigmoid functions to evaluate the utility of frame rate and frame size as follows:

- utility of 6 frames per second is 0, utility of 24 frames per second is 1, and utility of intermediate values is a linear combination of the utilities at those endpoints (e.g., utility of 12 frames per second is 0.33, and utility of 18 frames per second is 0.5),

- utility of 10,000 pixels be 0, utility of 1,000,000 pixels be 1, and utility of intermediate values is a linear combination of the utilities at those endpoints (e.g., utility of 400,000 pixels is about .39).

Also, we assign both dimensions equal weights, i.e., 0.5 each. Both our and Sousa's model use these weights and preference functions, but our model uses addition to obtain overall utility from dimension-wise utilities and Sousa's uses multiplication. Let's consider the iso-util curves, i.e., the curves that describe equally-valued configurations under our and Sousa's preference models.

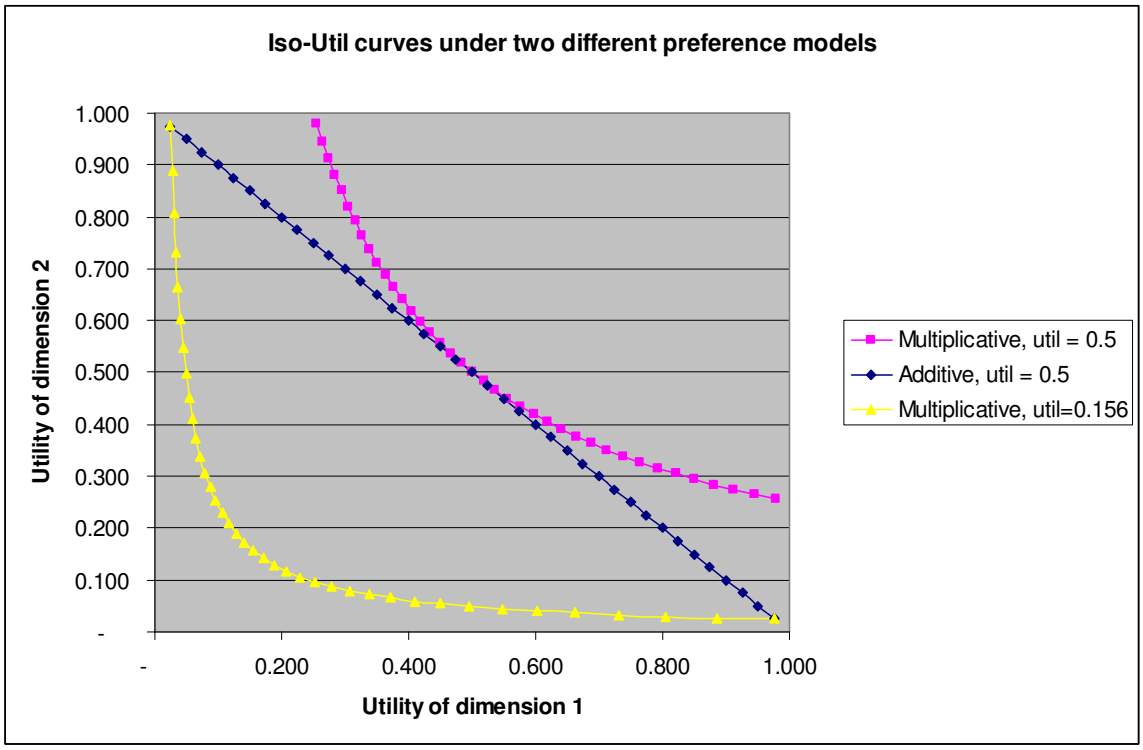


Figure 21: The iso-util curves under two different preference models.

As Figure 21 shows, the iso-util curves under our model are straight lines with slope -1. The straight line shown in that figure is the set of all configurations with overall utility 0.5. In Sousa's model, the iso-util curves are hyperbolas. The higher positioned hyperbola in the figure is the set of all configurations with overall utility 0.5. The lower positioned hyperbola is the set of all configurations with overall utility 0.156. Let's evaluate two different configurations under both models.

- 18 frames per second and 500,000 pixels has utility of 0.5,

- 8 frames per second and 900,000 pixels has utility of 0.31.

Under Sousa's model of preferences, the overall utilities of the two configurations are 0.5 and 0.31, respectively, i.e., the first configuration has much higher utility. Under our model, the two configurations have the same utility.

In general, Sousa's model encourages configuration choices in which dimension-wise utilities are closer together. In that respect, Sousa's model implicitly assumes that there is a dependency among the dimensions of the task. Our model treats dimension-wise utilities as entirely interchangeable, i.e., 0.1 utility improvement in one dimension is enough to off-set a 0.1 utility decrease in another dimension. This is consistent with the four independence assumptions of generally accepted preference models (see section 2.1 and [62]).

In practice, usability studies are necessary to determine which model gives a total order for the set of all possible configurations that is closest to user's own. Furthermore, in our infrastructure, we have implemented the ability to work with both preference models.

11.1.2 Do Preferences Make a Difference in Practice?

Our model of utility takes into account two types of preferences: (1) supplier preferences and (2) QoS preferences. But in practice, do QoS preferences make a difference in configuration decisions? In other words, is it possible that despite including QoS preferences in decision-making, configuration decisions are still dominated by supplier preferences?

We argue that QoS preferences can make a difference, i.e., less preferred suppliers are selected because they are more resource-efficient in providing the same level of QoS. We provide experimental evidence to support our argument.

We have analyzed the profiles of two suppliers for "*play video*" service: the Windows Media Player and the JMF Java Media Player. We played 8 different video streams in each application, and measured the resource usage of each application for each stream. For each stream, the bandwidth usage of the two applications is identical. However, the two applications have different CPU and memory requirements. In particular:

- The JMF Java Player has higher memory requirement than the Windows Media Player, because Java-based programs are generally more memory intensive,

- The CPU requirements of the Java Media Player, on the other hand, are lower than the CPU requirements of the Windows Media Player. Also, the Java Media Player can not provide play certain high definition formats, while the Windows Media Player requires significantly more CPU for very low quality videos.

We defined a task with only one service, and evaluated the two applications using different preferences[9]. The task has 4 preference dimensions: one supplier dimension and 3 QoS dimensions.

[9] An appropriate way to reason about different preferences is in the context of different use cases.

We varied the weight assigned to the supplier dimension and the QoS dimensions, and we varied which supplier is favored.

When the weight assigned to the supplier preferences was very high in relation to the weights for QoS dimensions, and one of the suppliers was favored heavily, then the favored supplier had the best configuration most of the time. However, when the weights were allocated among the 4 dimensions equally, and neither supplier was heavily favored, then the available level of resources determined which supplier had the best configuration. When the CPU is plentiful, then the Windows Media Player has the best configuration. On the other hand, when the CPU is limited (e.g., on a lower-end or older model device, or when CPU is ran at a slow speed to help battery last longer) then the Java Media Player has the better configuration.

This experiment clearly demonstrated that QoS preferences matter and including those in configuration decision-making makes a difference in the configuration chosen.

11.1.3 Comparing Anticipatory and Reactive Configuration Strategies

In this dissertation, we have designed and implemented two configuration strategies, namely, the reactive and anticipatory. But was it necessary to design two strategies? Furthermore, are both strategies useful?

We answer those two questions by demonstrating that each strategy has a comparative advantage and describe the circumstances under which such a comparative advantage would hold.

First let's discuss the differences among the two configuration strategies. The strategies differ in two respects: (1) how much input information is used to make configuration decisions and (2) how long the decision-making horizon is. The anticipatory strategy requires predictive inputs (predictions of task request operations and predictions of resource availability). Such predictive inputs must be available at least as far ahead into the future as the decision-making horizon. Conversely, predictions beyond the decision-making horizon are not needed, because they are inconsequential for the configuration decisions. So the strategies can be distinguished from each other using only a single parameter, the length of the decision-making horizon.

We measure the length of the decision-making horizon (DH) in units of prediction scale. The reactive strategy has a decision horizon of one unit of time, while the anticipatory strategy has a horizon of multiple units. The anticipatory configuration must look-ahead multiple units of time in order to make a single decision, while the reactive only must look-ahead one unit.

Recall that predictions of future events contain uncertainty. The two strategies differ in how they treat uncertainty in the predictions of future events. The reactive strategy ignores such uncertainty entirely, by waiting until all future uncertainty has resolved, and then reacting. The anticipatory strategy leverages quantified uncertainty of future events and plans its decisions based on predictions. In this manner, the anticipatory strategy handles uncertainty instead of ignoring it.

We have summarized the differences between the two strategies in Table 15.

Table 15: Summary of the differences between the reactive and anticipatory strategies of configuration.

	Decision Horizon (DH)	Requires Predictions?	Uncertainty	Computational Cost as a Function of DH
Reactive	1	No	Ignores	Const
Anticipatory	N	Yes	Handles	O(N^2)

In terms of the cost of computation, the anticipatory configuration is more expensive, because of the cost of the anticipatory planning. The computational complexity of the anticipatory configuration is potentially exponential in the length of the decision horizon. In Chapter 9, we described a heuristic for reducing the computational cost of the anticipatory strategy significantly; however, even with the reduction in the theoretical running time of the anticipatory algorithm, the runtime cost of the anticipatory algorithm is significantly greater than that of the reactive.

Because of the increased computational cost of the anticipatory algorithm, it is useful to identify the conditions under which the anticipatory strategy is likely to be better than the reactive strategy. There are many ways to quantify how one strategy is better than the other one. Does one strategy apply in situations when the other does not? If both strategies are applicable, does one strategy make better configuration decisions as measured by the accrued utility? How about the computational cost of each strategy?

We have identified the following four factors: (1) managing the resource allocation of non-perishable resources such as battery, (2) the magnitude of change penalties, (3) the relative order of supplier assignments under different resource conditions, and (4) the accuracy of resource predictions.

First, the anticipatory strategy can reason about non-perishable resources such as battery, while the reactive strategy can not. If automatic management of battery or other perishable resources is required, the reactive strategy is not usable. However, in situations where the user prefers to control the battery drain manually, the reactive strategy is still usable.

Second, when there are non-trivial change penalties, the anticipatory algorithm has the potential to be better (as demonstrated in section 10.3). Conversely, when there are no change penalties, then the reactive algorithm is provably optimal over time. Thus, in such situations, the anticipatory strategy does not have any advantage over the reactive one. In section 11.1.5, we argue that the distractions to the user are sufficiently large so that penalties can not be ignored. Furthermore, it is possible to imagine an extension of our model in which the cost of disruption changes over time. In this case, the advantage of the anticipatory strategy might be even more pronounced, because it can plan better when to switch penalties.

The third factor that can affect the added benefit of the anticipatory configuration is the ordering among the different supplier assignments under all possible resource conditions. The ordering is a function of the preferences, supplier profiles, and resource availability. If the order of supplier assignments in terms of overall instantaneous utility does not change as the resource conditions change, then the configuration selected initially will always be the best one. In fact, it is sufficient that one supplier assignment have higher maximum overall utility under all resource conditions, ensuring that it is the supplier assignment selected by both strategies. As the available level

of the resources changes, that supplier assignment is still the best one, and no supplier changes will be required to maximize overall utility. While changes to the runtime state of the suppliers in that supplier assignment might be required, our models do not penalize such changes. Consequently, both the reactive and the anticipatory strategies will always select that particular supplier assignment to run.

We can check whether the conditions described in the above paragraph hold. The requisite calculations are already performed as part of the initial computation in the anticipatory algorithm. Recall, from section 9.3, that we calculate the maximum possible QoS utility for each supplier assignment under each hypothetical resource condition. Based on such calculations, we can check if the total order of the supplier assignments changes depending on the available level of the resources. In practice, we have seen this to be the case, because different suppliers have different resource profiles, making some suppliers better when the resources are scarce and others to be better when the resources are plentiful.

The fourth factor is the accuracy of the resource predictions. The experiments in section 10.3 demonstrated a relationship between the accuracy of suppliers and the difference in utility between the anticipatory and reactive strategies. We measure the accuracy of resource predictions by the sigma parameter, the standard deviation of the prediction error. If prediction accuracy is good (i.e., the sigma parameter is small relative to the predicted values), then the anticipatory strategy is likely to outperform the reactive strategy. On the other hand, if the accuracy is bad (i.e., the sigma parameter is large), then the advantage of the anticipatory strategy diminishes. In practice, the accuracy of the predictions depends on a number of factors, e.g., how well utilized the resource is (see, for example, [49]).

In summary, we defend our decision to design and implement both strategies. The two strategies provide a choice in the space of cost-benefit alternatives. The anticipatory strategy is more costly in terms of resource requirements but it has two advantages over the reactive strategy: (1) can model and manage non-perishable resources like battery, and (2) has the potential to outperform the reactive strategy. The reactive strategy can be used as good solution when the computational power in the environment is limited. On the other hand, when the CPU is not the limiting resource, the anticipatory strategy can be used in the hopes of getting better accrued utility.

11.1.4 Accuracy of Supplier Profiles

Our approach to automating the configuration depends on the availability of supplier profiles. In this section we address the issue of accuracy of the suppliers, i.e., does our model capture the resource requirements of applications sufficiently and accurately for the purpose of configuration?

Recall from section 5.2.2, that a supplier profile is a relationship between the capability (QoS) space of the service and the resource space. A profile is defined as a set of <QoS, Resource> pairs. A supplier's resource requirements for a particular level of QoS are captured by the corresponding resource vector. We have considerable evidence that the resource requirements can be accurately estimated for suppliers that have QoS attributes (e.g., for suppliers of services such as "*play video*", "*browse web*", "*translate speech*").

However, for those suppliers that do not have user-perceived QoS attributes, the resource requirements in the supplier's profile might be less accurate. For suppliers that provide services such as "*edit text*", "*edit slides*", "*spreadsheet*", the requirements of the resources depend on the number and the size of the user files. Because the size of the file is not an explicit QoS dimen-

sion, there is no way in our current model to capture the resource requirement based size of the file. There are two issues at hand: (1) recording the number and the size of the files for those services that do not have QoS dimensions, and (2) deciding how many application instances to start when multiple files are needed by the user's task. We describe both issues in detail and argue that addressing those issues will require a modest amount of design and implementation effort in our current analytical framework.

First, let's consider the impact of file size and number of files on resource requirements. The size and the number of files that an application must open will affect the physical memory required for the application. Because the current supplier profile model does not record the size of the information assets, the prediction for the resource requirement might not be accurate. This deficiency in the model can be overcome by adding two special parameters to the service description: (1) the number of files in the service and (2) the size(s) of the files in the service. These parameters are recorded and maintained by Prism in the service description, and communicated to the Environment Manager when requesting services. Furthermore, the supplier profile is modified to reflect these parameters. We replace the <QoS, Resource> vector pairs in the supplier profile with <QoS, FileSize, Resource> triplets, where FileSize is vector that records the sizes of the files. And lastly, at configuration time, the Environment Manager uses the number and the sizes of the files to lookup the resource requirement in the profile of the supplier.

Second, when the user needs to work on multiple files of the same type, e.g., multiple text documents, there are two alternative ways to define the user's task that affect how many application instances need to be started. In the first alternative, multiple files can be defined as part of one service. In this alternative, the Environment Manager will need start one instance of an application that provides the requisite service. In the second alternative, for each file, a separate service can be defined in the task. In this case, whenever possible, the Environment Manager can request a different instance of the application for each file. The user might prefer the second alternative in cases when the user is concerned that a crash in one of the application instances might affect his other files. A number of off-the-shelf applications, e.g., Microsoft Word and Excel, Internet Explorer, will allow multiple instances of the executable to be launched as separate operating system processes. Consequently, each instance of the executable can be used to open one or more files or URLs. Launching multiple instances of the application costs more in terms of the resources, especially RAM. We are interested in accurately profiling the resource requirements of the suppliers in both alternatives.

The resource requirements in the two alternatives will generally be different. For example, opening two 2MB documents in the same instance of Microsoft Word will generally require less RAM than opening each file in a separate Word instance. The difference in resource requirements is due to the fixed resource overhead required to launch the application, and the incremental resource overhead required for each file. We can express these alternative possibilities in the supplier's profile by structuring the profile into two sections: the initial resource requirements and the incremental resource requirements. The Environment Manager will use information from both sections to calculate the resource requirement of requested services. When multiple files are attached to one service, then the initial resource requirement is counted once. When multiple files are attached to different services, then the initial resource requirements are counted multiple times, because multiple instances of the application will be launched.

For scoping reasons, our model of the supplier profiles currently does not accurately account for multiple files or the sizes of the files. However, we believe that the incremental effort required to design and implement such a change will be modest, and will not affect the results of this thesis.

11.1.5 Cost of Disrupting the User

Recall that our model of utility imposes penalties to discourage switching suppliers in order to avoid disrupting the user (see section 5.2.1 for the definition of the penalties). But is that a reasonable model to capture all disruption and switching costs? Furthermore, what would be the impact to the infrastructure and configuration algorithms if we adopt a more elaborate model of user attention and disruption?

As described in section 5.2.1, we have a simple model for capturing inconvenience to the user as a result of disruptions. Our model allows assigning penalties to certain kind of system changes that are not explicitly requested by the user. Specifically, our model allows the user to assess a supplier change penalties for each service in each task. For example, if a "*play video*" service in some task has a penalty then the penalty must be assessed, if the system decides to switch a running supplier, S1, with an alternative, S2. The penalty does not depend on S1 and S2. The penalty is not assessed, if the running supplier is not switched, but a change is made to its runtime state, or the switch is a result of user's explicit request.

Recent research in user task prioritization has demonstrated that when the user is interrupted, his performance on the task suffers. Furthermore, cost of interrupting the user is more severe in the middle of the task, and less severe in the beginning or at the end of the task. The cost of interrupting the user is significant and can not be ignored. Based on those results, we believe that using supplier change penalties is an appropriate way to account for the inconvenience to the user the user.

One can imagine a more general approach to modeling the attention of the user. Perhaps user's attention can be modeled as a multidimensional resource, accounting for the number of times that the user can be interrupted, the specific cost of each interruption, and the duration of each interruption. Perhaps we can allow the number of interruptions to accrue, and penalize more heavily as the total number of interruptions per unit time exceed some thresholds. Furthermore, we could even treat adjustments in the QoS attributes of suppliers as interruption to the user, and assign smaller penalties to such adjustments.

The enhancements described in the above paragraph will complicate the model of utility significantly and potentially increase the computational cost of the configuration algorithms. Under this new model of the resources, setting aside issues for evaluating / eliciting the cost of the various kinds of interruptions, our two models of configuration will require different amount of input information.

The reactive model of configuration will require periodic updates of the time-varying penalties of user interruption for different types of changes. These time-varying penalties will depend on the state of the user, progress on the task, and the type of change. The numerical values of the penalties will be obtained by the Task Management layer, e.g., using a process of elicitation. The Task Manager will periodically communicate the time-varying penalties to the Environment Manager, which will use the penalties to evaluate alternative configurations. The changes to the algorithm to accommodate the penalties will be minimal.

The anticipatory model of configuration will require predictions of the future costs of the different types of change penalties. These predictions will be time-varying, and will depend on the user's future progress with the task and other factors. The predictions will be obtained by the Task Management level and provided to the Environment Manager. The anticipatory strategy

and the algorithm will need minor changes to handle time-varying penalties. The computational cost of the anticipatory algorithm will increase somewhat; however, we don't expect the increased cost to affect the feasibility of running the algorithm in near-real time.

The enhanced penalty / cost model would require a re-examination of the runtime costs and the advantages of the two configuration strategies. In particular, an interesting open question is: would the anticipatory configuration provide an additional advantage due to the predictions of time-varying penalties, and how would the accuracy of predictions affect that advantage.

11.1.6 Applying the Prediction Framework Outside of the Current Context

The technical material and the results in this thesis have been presented in the context of Aura, a software infrastructure for pervasive computing. We would like to explore whether parts of our contribution can be applicable outside of that context. In particular, can the resource prediction framework be useful in other domains?

We collaborated with the designers of Rainbow [9], a self-healing system, to find if our prediction framework can be helpful in the context of Rainbow. Rainbow uses architectural style as a basis for self-repair. Rainbow models the system's behavior to ensure that the system is in a desired operating range. When the system deviates from this operating range, Rainbow chooses among a set of strategies to attempt to restore the normal behavior of the system. It uses real-time monitoring of both the target system and the environment to develop a model of the current system state and the available resources in the system. Currently, the monitoring provides only the current state of the system. Therefore, Rainbow can only react to changes to the current state of the system.

Resource predictions can help Rainbow when evaluating and choosing between alternate strategies of adaptation. For example, by knowing the probability that the available level of a critical resource such as bandwidth 5 minutes from now will be below a certain threshold, Rainbow can choose a strategy that quiesces lower priority client sessions so that the remaining client requests will continue to be satisfied within a tolerable latency. If, on the other hand, the probability is high that the bandwidth will be restored to levels that will naturally bring the system back within its desired state, Rainbow can choose reduce the fidelity of some or all of the client sessions.

Currently, Rainbow utilizes the DASADA gauge and probe infrastructure for resource and system monitoring. Probes monitor and collect information from the target system and from the environment. Probes publish the monitored information on the probe bus. Gauges subscribe to information from probes by listening on the probe bus. Gauges perform computation using input information from the probes and publish the output on the gauge bus. Rainbow uses information from both gauges and probes to make decisions. The left side of Figure 22 shows a runtime view of the probe and gauge architecture.

We can integrate our prediction framework into the architecture of Rainbow as follows. A runtime instance of the entire prediction framework can be represented as a sophisticated Rainbow gauge (see the right side of Figure 22). Monitoring information can be provided into the framework using the probe bus; such information can be provided by one or more probes on the probe bus. The consumer component needs to be programmed according to the specific predictive information needs of Rainbow. The consumer initiates aggregate prediction reports using the Aggregate Prediction Protocol when requested to do so by a gauge client.

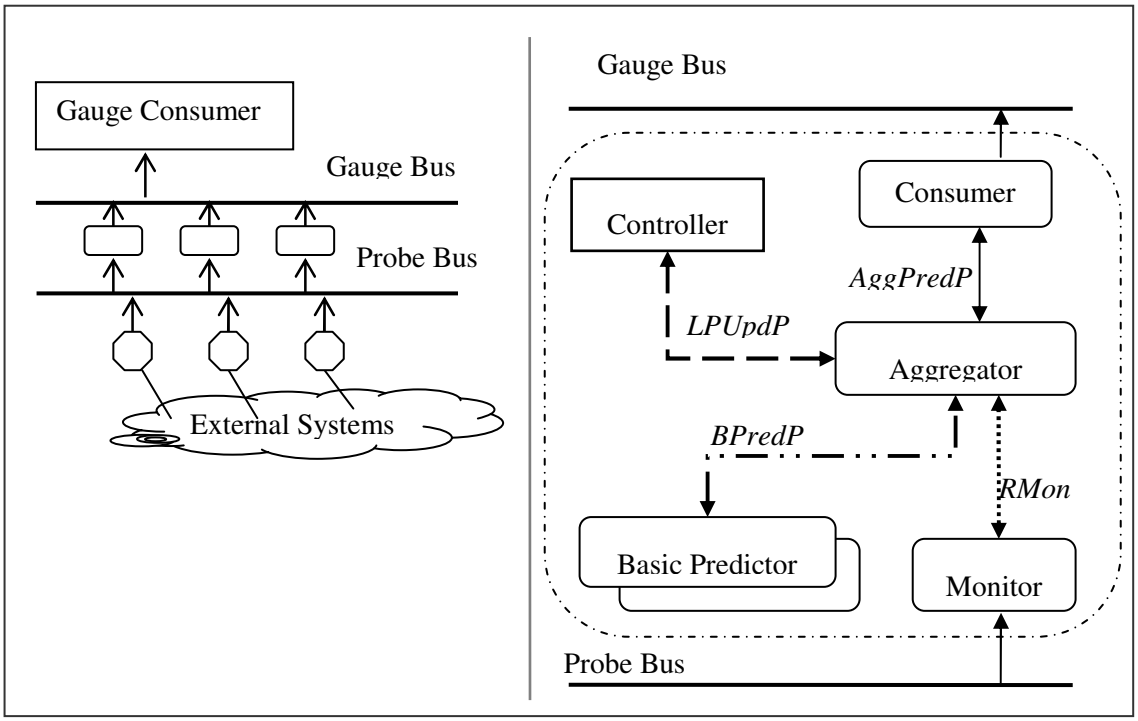


Figure 22: The left side of the figure shows the probe and gauge infrastructure of Rainbow. The right side of the figure shows the resource prediction framework wrapped as a Rainbow gauge.

The consumer uses the prediction tree as input to perform computations, and publishes the result on the gauge bus. The design of the prediction framework allows some flexibility in runtime deployment options. For example, a single runtime instance of the consumer can provide multiple types of information. Or, as an alternative, multiple instances of a consumer can be deployed, each responsible for computing specific type of predictive information.

The integration is consistent with the probe-gauge architecture. The framework conforms to the architectural requirements of a gauge, because it receives inputs from the probe bus and provides outputs to the gauge bus. Furthermore, as a gauge, the framework can be configured using the standard gauge configuration interface. The configuration parameters of the prediction framework, e.g., the parameters of the linear recent history prediction model can be set using the gauge control interface.

11.2 Future Work

11.2.1 Ensuring the Privacy of User's Tasks

Automatic configuration deals with private data of the user, including user's files and task descriptions. While automatic configuration provides desired features to the user, it also creates security and privacy concerns. For scoping reasons, we have not addressed such concerns in this thesis. However, jointly with Joao Sousa and Bradley Schmerl, we have done a modest amount of design and implementation towards ensuring the security and privacy of user's data in a seamless computing world.

In terms of the requirements for security, a user's data must be adequately secured using encryption and authorization. Second, there are bound to be trust issues between the user and the environment, and a trust model is needed to address such trust issues. For example, should the user trust the suppliers in the environment? Conversely, should the environment provide services to any user?

We argue that some of the security features can be added to the architecture of our infrastructure with only modest changes to the existing components. In particular, we have early evidence that authentication and encryption can be integrated into the existing infrastructure with modest amount of changes. Furthermore, our security design ensures that automatic configuration continues to incur low overhead. We use a combination of public key and symmetric keys encryption to authenticate both users and suppliers, and secure user's data. Because public-key based handshakes are expensive computationally, we use those sparingly. The bulk of encryption is done using much cheaper symmetric key cryptography. Our encryption design uses a separate encryption key for each task instance, ensuring that only the Task Manager, the Environment Manager, and the suppliers activated for that task have access to user's private data.

We discovered that tackling the issue of trust is a much harder problem. Our objective is to design a two-way trust model between the user and the environment. We have been faced with two important design decisions: (1) how to capture the extent to which the user trusts the suppliers in the environment and the extent to which the environment trusts the user, and (2) should the selection of the suppliers include preferences for user's level of trust for those suppliers, or should trust be treated as an entirely orthogonal concern.

It is tempting to apply the existing preference model to trust levels. In fact, enhancing the infrastructure based on such a design choice will require small amount of changes, because both the Task Management layer and the Environment Management layer are equipped to work with additional preference dimensions. On the other hand, we don't have any evidence whether applying preferences to trust levels is justified and secure.

Ensuring the security and privacy of a user's task remains the subject of ongoing work.

11.2.2 Defining Environment Scope and Environment Boundaries

In our work, we have not addressed the problem of defining the environment scope and boundaries. We assumed that the available suppliers know which Environment Manager to register with, and that the list of available suppliers does not change. We further assumed that the administrative boundaries of the Environment Manager is fixed and does not change during configuration.

Deciding the administrative domains of the Environment Managers, their relationships (e.g., containment, neighboring, etc), and which Environment Manager should the suppliers register with are difficult problems.

Identifying environment boundaries is a difficult problem, because it requires combining two scoping strategies. The first scoping strategy is dictated by the existing boundaries of computing networks. Currently, large IT deployments use network and administrative domains to define groups of computers, users, devices, and resources. For example, Windows Server domains are widely used for this purpose. Network and subnet divisions are standardized and widely used.

Unfortunately, network and domain-based division of resources and devices does not allow describe the physical location of the hardware and applications. Typically, only applications and devices that are in the physical proximity of the user can be used to provide services for the user's tasks. For example, a remote computer with a great speech entry program is of little value to the user, if the computer is located two hundred feet away and the user is not willing to travel there or does not know the location of the computer.

A better way to scope environments for the purpose of configuration is to use geographical proximity and containment of spaces. Jiang and Steenkiste [26] proposed a hybrid location model to identify the location of users and devices. The Aura Location Identifier (ALI) system is a hybrid between standard network naming and geographical naming systems. An ALI is a standard Uniform Resource Identifier and has three parts: (1) the protocol (ali://), (2) the network address, which is a standard IP address or network name resolvable using DNS, and (3) the resource, which identifies a geographical space. The network address of an ALI identifies a network domain in the existing internet network hierarchy, while the resource portion identifies a geographical space. Here is an example ALI, ali://cmu.edu/wean_hall/8th_floor/8100_corridor/. This ALI identifies the 8100 Corridor on the 8th Floor of the Wean building that falls in the cmu.edu domain.

The ALI system allows expressing the containment relationship of the geographical spaces in the real world. For example, ali://cmu.edu/wean_hall/8th_floor/8100/ space is contained in ali://cmu.edu/wean_hall/8th_floor, which in turn is contained in ali://cmu.edu/wean_hall.

We believe that environment boundaries should be decided using the ALI location system because of several potential advantages: (1) environments can be defined and identified very narrowly, e.g., indicating a room in which the user is working, (2) the hierarchical containment of the spaces allows the possibility of requesting services from any space that contains the location of the user.[10]

In order to make use of the ALI location system for the purposes of environment scoping, two problems need to be solved:

- finding one or more Environment Mangers that provide configuration services in a location identified by an ALI,

- deciding the boundaries and containment of a mobile environment such as a laptop.

Solving the first problem is necessary to enable mobile computing. When a mobile user enters a new environment, she will need to find the local Environment Manager to request services,

This problem can be solved using an approach already used to resolve the network server names of machine IP addresses. A network of naming servers can be utilized which are responsible for resolving the ALI of an environment to the network coordinates (the IP address and the network port) of the corresponding Environment Managers. The resolution service can be provided as

[10] Defining a location hierarchy is an important and difficult problem. For example, in Pittsburgh, PA there are a number of overlapping and intersecting location hierarchies, e.g., zip codes, neighborhoods, voting precincts.

part of the local DHCP service, because the DHCP already solves a similar problem. When the user's computer obtains a DHCP IP address, the DHCP server can be used to provide the location of the local Environment Manager. By finding the network coordinates of one of the local environment manager, the bootstrap problem can be solved.

The second problem arises when a mobile user carries his laptop into a new environment. In that scenario, how should the suppliers on his laptop be treated? Should they be made available as part of the room's local environment, or should they be kept available only on the laptop's own environment? This is a difficult question to answer because of privacy and security concerns. On the one hand, allowing the local suppliers to join the local environment makes them available to other users that might be authorized in the local environment. On the other hand, keeping them hidden from the local environment requires treating the laptop's environment and the room's environment separate for configuration purposes. How should we allow the the mobile user wants to use suppliers available on his laptop as well as suppliers available in the fixed environment on the room?

For the above mentioned reasons, the problems of deciding the environment scope and boundaries must be addressed in order to facilitate the deployment of configuration infrastructure.

11.2.3 Configuration There and Then: Combining Motion Profiles

Our two analytical solve the problem of configuration "here and now". It is possible to imagine configuration "there and now" or "there and then". Addressing such configuration problems requires two additional research problems which are rather broad in scope:

- requesting services environments other than the local environment that the user is in,
- calculating user's motion profiles.

We discussed the first problem in the previous section, 11.2.2. As for the second problem, let's consider two scenarios different scenarios to motivate it.

In the first scenario, imagine a user that needs to work on his task in approximately 30 minutes. However, the user has a choice where he can work on his task. This choice is encoded as a small subset of possible locations, e.g., alternative coffee shops or alternative airport gates. The user would like to work in the location that provides the best possible support in terms of resources and suppliers for his task.

This problem of configuration can be tackled by enhancing our existing infrastructure. User's Task Manager can make "what-if" configuration request in each candidate location and then help guide the user towards the location which can provide the best expected utility for the user.

In the second scenario, imagine a user that is continuously mobile, e.g., a field biologist or a travelling salesman or deliveryman. The user has tasks in progress, e.g., download and process data, upload data, etc. The problem of configuration is local to the user's current location; however, his location is continuously changing. This problem is much more difficult, because it requires prediction user's motion profile, and then superimposing the prediction of the resources over the motion profile of the user. If the motion path of the user can be predicted in advance with accuracy (e.g., in the case of a deliveryman, the path might be fixed, unless there are road accidents), then the resources predictions can be calculated as a function of both time and space. However,

we believe that this problem is still considerably more difficult then the problem we solved in this thesis. A number of factors, e.g., schedule delays, path deviations, can affect the accuracy of resource predictions, and significant amount of research and investigation is required to address those issues.

11.3 Contributions

This dissertation offers contributions at three levels: an approach to automating the configuration of the computing environment, an analytical framework that structures the problem of configuration, and an infrastructure that supports and validates the approach. The following subsections elaborate the three contributions in detail.

11.3.1 Contributions of the Approach

We have demonstrated an approach that substantially automates the control of the configuration. Our approach eliminates some of the routine chores that lay users must routinely perform in order to configure applications needed for their daily tasks. Our contributions are threefold:

- **Automating the Configuration.** We have demonstrated that the control of the configuration can be substantially automated. Our approach relieves the user of routine administrative chores of finding and starting applications, and setting application runtime state. Our approach continually monitors the state applications and resources, and makes changes to the applications as needed. Our approach requires the user to define his tasks in terms of abstract services and preferences, and uses the definition of the task automate the control of the configuration. We have demonstrated that our approach significantly reduces the amount of effort that the user must spend to control the applications and resources needed for his tasks.

- **Satisfying two Competing Requirements: Utility and Practicality.** We have demonstrated that our approach satisfies to two important competing requirements: utility and practicality. Utility measures how well our approach satisfies the preferences of the user into by mapping abstract user needs into concrete capabilities in the environment. In terms of practicality, our approach has low overhead: both the latency of configuration requests and the resource footprint of our approach are acceptably low.

- **Quantifying and Addressing Predictive Uncertainty.** We have proposed two strategies for automating the configuration: the reactive and the anticipatory. These strategies differ in the way they treat the uncertainty in the predictions of future events. The reactive strategy ignores any uncertainty, and only makes configuration decisions in response to changes in the environment, e.g., changes in resource changes, or changes in the user's task. The anticipatory strategy relies on quantified predictions of future events. We have described a model of predictive uncertainty for the resources, and demonstrated the simultaneous requirements of utility and practicality can be satisfied by the anticipatory approach

11.3.2 Analytical Contributions

Our approach defines analytical models that capture the dimensions, inputs, and constraints of the problem of automatic configuration. The problem of configuration is to find a sequence of near-

optimal configurations over a period of time such that the expected utility of that sequence to the user is nearly optimal according to the preferences of the user. Our contributions are two-fold:

- **Structuring the Problem for Feasibility.** We have shown how to separate the essential dimensions of the problem into three spaces: utility, capability, and resource. We have defined a model of preferences, which are mathematical functions that map from the capability space into the utility space. We have also defined a model of application profiles, which are summaries based on the historical performance of applications. Formally, application profiles define a relation between the resource space and the capability space. By combining user preferences, application profiles, and resource availability, we have formulated the problem of configuration as an optimization problem, in which the objective is to select a suite of applications, decide their runtime settings, and resource allocation among them so that user's utility is optimized.

- **Designing Efficient Algorithms for Configuration.** We have designed and implemented efficient configuration algorithms that exploit the analytical structure of the configuration problem. We have demonstrated an incremental staging of the problem instances, so that the complexity in the problem is handled gradually. For the problem in each stage, we have provided one ore more algorithms that solve it.

11.3.3 Architectural and Implementation Artifacts

We have contributed to the design and implementation of project Aura, a software infrastructure for pervasive computing. Our contributions include the design and the implementation of Aura components and connectors, as follows:

- **Environment Manager.** We have designed and implemented the Environment Manager, the component responsible for configuration decisions. The Environment Manager uses inputs from three sources: the Task Manager, the Suppliers, and the Resource Managers, in order to make configuration decisions. The Environment Manager monitors the state of the environment, and makes adjustments as needed, ensuring that the utility to the user is maximized or nearly maximized according to user's preferences.

- **Resource Prediction Framework.** We have designed a resource prediction framework, which allows combining multiple types of predictive information about a resource's availability into an aggregate prediction. While the design of the resource prediction framework is an essential part of the architecture, the framework itself is a self-contained entity that can be used in other contexts and domains.

- **Aggregate Resource Prediction API.** We have designed the interface to the Resource Prediction Framework, ensuring a potentially broader application of the framework. The interface has a number of desired features: it provides considerable flexibility to consumer of predictions control for decided the essential parameters of prediction: prediction scale, prediction horizon, and prediction detail. The interface also exposes information about the cost and accuracy of predictions, allowing the consumer to make determination of the prediction parameters based on its prediction needs, desired accuracy, and cost. The prediction interface is not tied to a particular statistical model and prediction.

- **Supplier SDK.** We have designed a reusable library for rapid development of suppliers in both Java and C++ programming languages. The library allows third-party developers

to develop suppliers that can interoperate with the infrastructure. We have provided API documentation for the library, and working implementations for about a half a dozen suppliers.

11.4 Conclusion

In this dissertation, we investigated the feasibility of automating the control of the configuration of applications and resources, while ensuring that two critical requirements of utility and practicality are simultaneously satisfied. We demonstrated a novel use of predictions to make configuration decisions. We demonstrated two strategies of configuration: reactive and anticipatory. The reactive strategy does not use predictions. As part of the anticipatory strategy, we defined a model of predictive uncertainty, and demonstrated that configuration can be automated while ensuring practicality and utility under uncertainty.

Prior to our work, research in task-oriented computing demonstrated how to elicit task requirements from the user. Research in fidelity-aware computing demonstrated how to guide the adaptation of individual applications towards a single goal. Our work bridged an existing gap between the two research areas, showing how to use the task requirements of the user to select appropriate applications and then guide their adaptation according to those requirements.

Our hope is that the work described in this dissertation has contributed to research in software engineering and pervasive computing. In particular, our thesis provides an important step towards value- and utility-driven software engineering. Traditionally, cost considerations have been the primary driver in software design and implementation. Our approach advocates maximizing the value to the end user, while demonstrating that the runtime overhead can be kept acceptably low.

Our thesis provides evidence that the design of engineering solutions can successfully mediate between multiple dimensions of concerns. When there is a single dimension of concern, designing an optimal solution is often easy. Multiple dimensions of concern make the design and engineering of practical solutions challenging. This thesis serves as one data point in demonstrating the feasibility of engineering solutions that address practicality, tractability, and utility in the presence of multiple objectives.

12 References

1. G. Abowd, E. Mynatt. Charting Past, Present and Future Research in Ubiquitous Computing. *ACM Transactions on Computer-Human Interaction*, 7(1), pp 29-58, March 2000.
2. R. Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics 6, 1957*.
3. R. Bent and P. Van Hentenryck. Regrets Only! Online Stochastic Optimization under Time Constraints. *Proceedings of the 19th AAAI*, 2004.
4. S. Butler. Security Attribute Evaluation Method. A Cost-Benefit Approach. *Proc Int'l Conf in Software Engineering (ICSE)*, 2002.
5. R.K. Balan. Simplifying Cyber Foraging. *Ph.D. Thesis, Carnegie Mellon University Technical Report CMU-CS-06-120, 2006.*
6. R.K. Balan, J.P. Sousa, M. Satyanarayanan. Meeting the Software Engineering Challenges of Adaptive Mobile Applications. Carnegie Mellon University Technical Report, CMU-CS-03-11, February 2003.
7. L. Capra, W. Emmerich and C. Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Soft Eng, Volume 29, Num. 10, pp. 929- 945 (2003).*
8. S.W. Cheng et al. Software Architecture-based Adaptation for Pervasive Systems. International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing. Karlsruhe, Germany. *LNCS* Vol. 2299, Schmeck, Ungerer, Wolf, (Eds.) April 2002.
9. S.W. Cheng. Rainbow: Cost-effective, software architecture-based self-adaptation. *PhD thesis, Carnegie Mellon University Technical Report CMU-ISR-08-xxx*, 2008.
10. T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*. MIT Press. 1994.
11. The DAML Services Coalition (multiple authors), "DAML-S: Web Service Description for the Semantic Web", *Int'l Semantic Web Conference (ISWC)*, 2002.
12. E. Dashofy, D. Garlan, A. Koek, B. Schmerl. xArch: an XML Standard for Representing Software Architectures. http://www.isr.uci.edu/architecture/xarch/
13. E. DeLara. Component-Based Adaptation for Mobile Computing. *PhD Thesis, Rice University Computer Science Technical Report, TR03-414, 2003.*
14. P. Dinda, D. O'Hallaron. Host Load Prediction Using Linear Models. *Cluster Computing*, 3:4, 2000.
15. P. Dinda, and D. O'Hollaron, An extensible toolkit for resource prediction in distributed systems, Technical Report CMU-CS-99-138, Carnegie Mellon University, July 1999.
16. The Proceedings of the 1-7 International Workshop in Economics Driven Software Engineering Research (EDSER), affiliated with the International Conference on Software Engineering (ICSE), 1999-2005.
17. J. Flinn, Extending Mobile Computer Battery Life through Energy-Aware Adaptation, Jason Flinn, *PhD thesis, Carnegie Mellon University Technical Report CMU-CS-01-171, 2001.*
18. J. Flinn, M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. *Proc Symp Operating Syst Principles (SOSP)*, 1999.
19. K. Gajos. Rascal - a Resource Manager for Multi-Agent Systems In Smart Spaces. *Proceedings of CEEMAS'01*, 2001.
20. V. Galtier, et al. Predicting resource demand in heterogeneous active networks. *Proceedings of MILCOM 2001*.
21. P. Garbacki, D.H.J. Epema, M. van Steen. An Amortized Tit-For-Tat Protocol for Exchanging Bandwidth instead of Content in P2P Networks. *Proc. First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, Boston, MA 2007.
22. D. Garlan, R. Monroe, D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, Leavens and Sitaraman (Eds), Cambridge University Press, pp. 47-68, 2000.
23. D.Garlan, D.Siewiorek, A.Smailagic, P.Steenkiste. Project Aura: Toward Distraction-Free Pervasive Computing. IEEE Pervasive Computing, April-June 2002.
24. S. Gurun, C. Krintz, and R. Wolski. NWSLite: A Light-Weight Prediction Utility for Mobile Devices. *Proc. International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2004.
25. J. Flinn, M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. *Proc Symp Operating Syst Principles (SOSP)*, 1999.
26. C. Jiang and P. Steenkiste. A hybrid location model with a computable location identifier for ubiquitous computing. *In Proceedings of Ubicomp 2002. Lecture Notes in Computer Sience Vol. 2498*. Gothenburg, Sweden, September 2002.

27. H. Hamann and H. Worn. A Space- and Time-Continuous Model of Self-Organizing Robot Swarms for Design Support. *Proc. First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, Boston, MA, 2007.
28. I. Han; H.-S. Park; Y.-K. Jeong; K.-R. Park. An integrated home server for communication, broadcast reception, and home automation, Consumer Electronics, IEEE Transactions on, Vol 52 (1), 2006.
29. P. Hentenryck, et al. Online Stochastic Optimization Under Time Constraints. Working paper, last accessed April 2008 at http://www.cs.brown.edu/people/pvh/aor5.pdf.
30. M. Jones, D. Rosu, M. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *Proc Symp Operating Systems Principles (SOSP)*, 1997.
31. W. Koch W., E.M. Schulz, R. Wright, et al. What is a Ratio Scale? Rasch Measurement Transactions 9:4, p. 457, 1996.
32. F. Kon, et al. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. *Proc. USENIX Conference on OO Technologies and Systems (COOTS)*, 2001.
33. E. de Lara, D. S. Wallach, W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. *Proc. USENIX Symp on Internet Technologies and Systems (USITS)*, 2001.
34. C. Lee, et al. A Scalable Solution to the Multi-Resource QoS Problem. *Proc IEEE Real-Time Systems Symposium (RTSS)*, 1999.
35. J. Magee, J. Kramer. Concurrency, State Models & Java Programs. John Wiley & Sons, 1999.
36. Moore, M.; Kazman, R.; Klein, M.; & Asundi, J. "Quantifying the Value of Architecture Design Decisions: Lessons from the Field", Proc. 25th Int'l Conf on Software Engineering (ICSE), 2003.
37. D. Narayanan. Operating System Support for Mobile Interactive Applications. *PhD Thesis, Carnegie Mellon University Technical Report CMU-CS-02-168, 2002*.
38. D. Narayanan, J. Flinn, M. Satyanarayanan. Using History to Improve Mobile Application Adaptation. *Proc. 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 2000.
39. D. Narayanan, M. Satyanarayanan. Predictive Resource Management for Wearable Computing. *Proceedings of the the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, May 2003 San Francisco, CA.
40. The Network Weather Service on the internet. http://nws.cs.ucsb.edu/ewiki/. Last accessed July 2007.
41. R. Neugebauer and D. McAuley. Congestion Prices as Feedback Signals: An Approach to QoS Management. *Proc. ACM SIGOPS European Workshop*, 2000.
42. B. Noble, et al. Agile Application-Aware Adaptation for Mobility. *Proc ACM Symp Operating Systems Principles (SOSP)*, 1997.
43. B. Noble. Mobile Data Access. *PhD Thesis, Carnegie Mellon University Technical Report CMU-CS-98-118, 1998.*
44. U. Norbisrath, I. Armac, D. Retkowitz, P. Salumaa: Modeling eHome Systems. S. Terzis (ed.): *4th Intl Workshop on Middleware for Pervasive and Ad-Hoc Computing*, Melbourne, Australia. ACM Press, 2006.
45. U. Norbisrath, C. Mosler, I. Armac: The eHome Configurator Tool Suite. *Firstl Internaitonal Workshop on Pervasive Systems (PerSys 2006)*, Montpellier, France, 30-31 2006, LNCS 4278, p. 1315-1324, Springer, 2006.
46. D. Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114, (1999).
47. V. Poladian, S. Butler, M. Shaw, D. Garlan. Time is Not Money: the case for multi-dimensional accounting in value-based software engineering. *In Proc. 5th Workshop on Economics Driven Software Engineering Research (EDSER)*, 2003.
48. V. Poladian, João Pedro Sousa, David Garlan, Mary Shaw. Dynamic Configuration of Resource-Aware Services. *In Proc. 26th Intl Conf. On Software Engineering (ICSE 2004)*. Edinburgh, May 2004.
49. Y. Qiao, J. Skicewicz, P. Dinda. An Empirical Study of the Multiscale Predictability of Network Traffic. *Proc Intl. Symposium on High Performance Distributed Computing (HPDC)*, 2004.
50. A. Ruszczyński and A. Shapiro. 2003. Stochastic Programming: Handbook in Operations Research and Management Science 10. Elsevier.
51. M. Satyanarayanan. Pervasive Computing: Vision and Challenges. In IEEE Personal Communications, August 2001.
52. J.P. Sousa. Scaling Task Management in Space and Time: Reducing User Overhead in Ubiquitous Computing Environments. *PhD thesis, Carnegie Mellon University Technical Report CMU-CS-05-123, 2005.*
53. J.P.Sousa, R.K.Balan, V.Poladian, D.Garlan, M.Satyanarayanan. Giving Users the Steering Wheel for Guiding Resource-Adaptive Systems. *Carnegie Mellon Technical Report, CMU-CS-05-198.*
54. J.P. Sousa, D. Garlan. The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. *Carnegie Mellon Technical Report, CMU-CS-03-183.*

55. J.P. Sousa, B. Schmerl, V. Poladian, and A. Brodsky. UDesign: End-User Design Applied to Monitoring and Control Applications for Smart Spaces. *Proc. 2008 Working IFIP/IEEE Conference on Software Architecture*, Vancouver, BC, Canada, 18-22 February 2008.
56. S. Stumpf, et al. The TaskTracer system. *20th National Conference on Artificial Intelligence (AAAI-05)*, Intelligent Systems Demonstrations, Pittsburgh, PA, July 9-13, 2005.
57. The Task Tracer Project Home Page at Oregon State University. Last accessed in April, 2008 at http://eecs.oregonstate.edu/TaskTracer/.
58. M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, pp 94-100, September 1991.
59. R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. *In Proceedings of High Performance Distributed (HPDC)*, 1997.
60. R. Wolski, et al, The network weather service: A distributed resource performance forecasting system. *Journal of Future Generation Computing Systems*, 1999.
61. R. Wolski, et al. Predicting the CPU availability of time-shared Unix systems. *Proc Intl Symp High Perf Dist Computing (HPDC)*, 1999.
62. Yoon, K. Paul and Hwang, Ching-Lai. *Multiple Attribute Decision Making: An Introduction*, Sage Publications, 1995.
63. W. Yuan and K. Nahrsted, Energy-Efficient CPU Scheduling for Multimedia Applications, To appear in *ACM Transactions on Computer Systems*.

Appendix A: Time Series Analysis Background

In this section, we present a light introduction to the time series theory and the classical decomposition approach. This will serve as a motivation and starting point for defining resource predictor types and operations among them.

In time series analysis, we consider a discrete time model. Let t denote an integer time. A time series is an infinite vector of values indexed by time. We denote time series like this: $\{Y_t\}$, $\{X_t\}$, etc. The classical decomposition breaks up time series $\{Y_t\}$ into three components as follows:

$$Y_t = m_t + s_t + X_t.$$

Here,

$\{Y_t\}$ is the original time series,

$\{m_t\}$ is the trend component,

$\{s_t\}$ is the known pattern component (called a seasonal component in time series literature),

$\{X_t\}$ is the remainder, a stationary series.

The trend component captures a steady change, typically an increase, in the series. The trend can be linear, quadratic, exponential, etc. The known pattern component captures periodic patterns in the series, e.g. increases and decreases that repeat reliably over a defined period of time. There may be multiple known pattern components, e.g. hourly, daily, weekly, or monthly. When multiple season components are present, the decomposition formula above will have additional terms.

For the purposes of research in this thesis, the use of a trend component does not seem to be appropriate. Such a component would be appropriate if, for example, modeling the growth of the average processing power of commodity desktop computers over a very long period of time while assuming that Moore's law holds.

In the time series analysis using the classical decomposition approach, once the trend and known pattern components are removed, the remaining time series will be *stationary*, a technical term that guarantees the series has certain properties. A stationary time series can be analyzed using a model that captures serial correlation in the series. Serial correlation, or autocorrelation, is relationship between he adjacent values of a time series. The objective of this analysis would be to determine a model (model selection) that does a good job of predicting future values based on past observations and solving for the best-fit parameters (parameter inference) of the model.

The *analysis* process, shown in diagram in Figure 23, is performed first. A historical trace of the series $\{Y_t\}$ is first analyzed to determine trend and known pattern components. Then these are removed to obtain a stationary series, $\{X_t\}$. Model selection and parameter estimation is done using $\{X_t\}$.

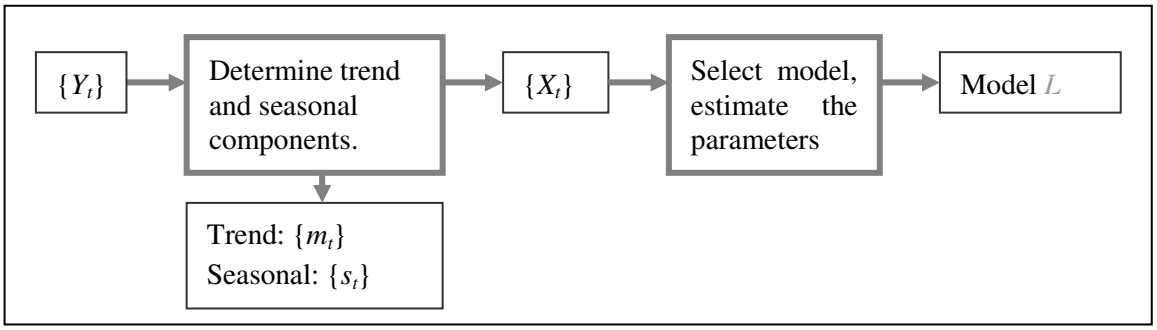


Figure 23: Diagram showing the time series *analysis* process. Starting with the original time series {Y_t}, we use classical decomposition to determine and remove trend, {m_t}, and seasonal components, {s_t}. We then perform model selection and parameter estimation over the stationary series, {X_t}, to generate a

During prediction (see Figure 24), we start out with the first s observations of the series, {Y_t}: Y_1, Y_2, …, Y_s. We remove trend and known pattern components to obtain the first s observations of the series, {X_t}: X_1, X_2, …, X_s. Then we use the model, L, to predict the value of X_{s+1}. To obtain a prediction Y_{s+1}, we work backwards, i.e. add the known pattern component and re-trend the value of X_{s+1} to obtain a prediction for the Y_{s+1}. As new values of {Y_t} become available, we repeat the same steps. We can save on computation by caching the values of intermediate calculations, i.e. the values of the series {X_t}.

A time series model is a mathematical formula that expresses relationships found in the adjacent values of the series. When earlier values of the series can help partially predict future values, it is said that there is serial correlation in the time series. Some of the simplest forms of serial correlation can be found using autoregressive (AR) and moving-average (MA) models. Formally, an

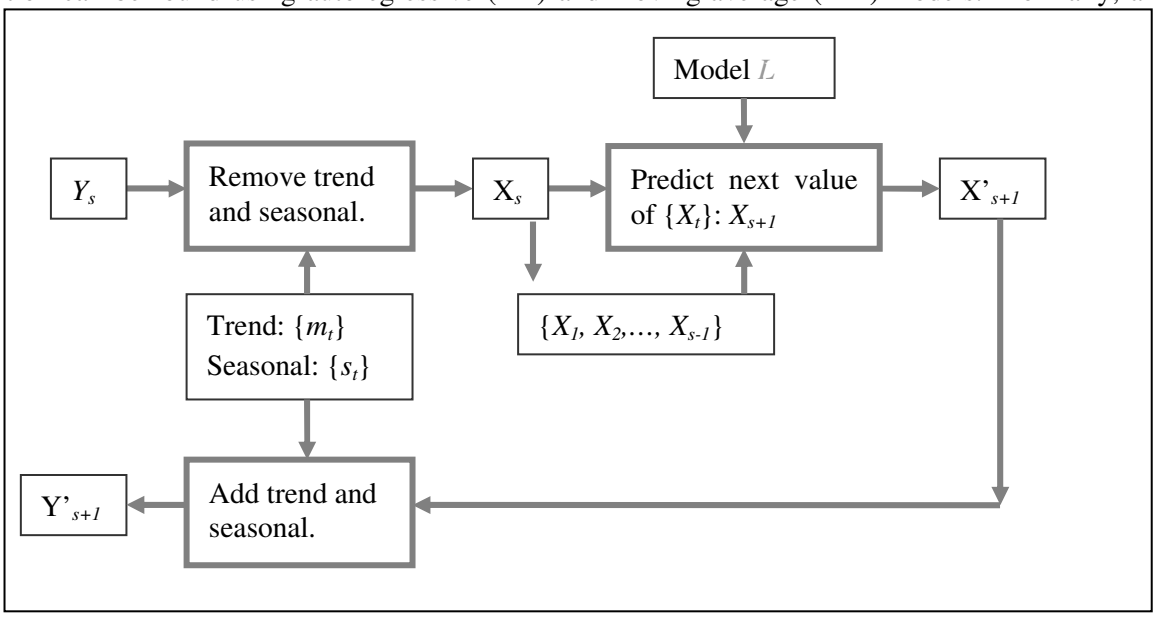


Figure 24: This diagram shows prediction of future values of time series, {Y_t}. As new a value of series {Y_t} is observed, e.g., Y_s, we calculate the corresponding value of X_s. Then we make a prediction for X_{s+1} based on a model L. The predicted valued of X_{s+1} and pre-computed seasonal and trend components are used to make a prediction for Y_{s+1}.

autoregressive, moving average model, ARMA(p,q), is an equation of the following form:

$$X_{t+1} = \varphi_1 X_t + \varphi_2 X_{t-1} + \ldots + \varphi_p X_{t-p+1} + Z_{t+1} + \theta_1 Z_t + \theta_2 Z_{t-1} + \ldots + \theta_q Z_{t-q+1}$$

Here, p is the autoregressive order, q is the moving average order, $\{X_t\}$ is the series being predicted and $\{Z_t\}$ is the series of prediction errors. $\varphi_1, \varphi_2, \ldots, \varphi_p$ are the autoregressive parameters, and $\theta_1, \theta_2, \ldots, \theta_q$ are the moving average parameters. The values of the series $\{Z_t\}$ are jointly normally distributed with a standard deviation σ, i.e. each $Z_t \sim N(\mu_{t+1}, \sigma)$. The size of the variable σ relative to the mean of the series $\{X_t\}$ shows the size of the uncertainty. The degenerate case of a linear predictor with σ parameter value equal to zero shows no uncertainty. For further background on linear models and predicting resource availability, see [14].

Appendix B: Analysis of Manual Configuration

In this appendix, we summarize our notes from surveying representative applications for the purpose of quantifying manual overhead of configuration. We surveyed the following off-the-shelf media players and web browsers: RealOne Player, Windows Media Player, Internet Explorer Browser, and FireFox Mozilla Browser. We present detailed notes about the RealOne Player and Microsoft Internet Browser applications.

Real One Player. RealOne player is a general purpose, multimedia player from Real Networks. We surveyed version 10.2 of the player for Win32 (32 bit Microsoft Windows) platforms. The application provides a number of advanced settings for controlling the quality of the video playback and resource consumption.

The controls for these settings are found from menu item "Tools -> Preferences". First, under the "Hardware tab", there is a sliding scale for controlling the CPU usage of the player. Surrounding text instructs the user "*to move the slide if the computer seems sluggish while playing*". Based on that text, we assume that the application can reduce its own CPU usage if requested to do so, but at the expense of some quality attribute. We believe that either the resolution or the frame update rate of the video playback will be affected, but no information was found in help files shipped with the application to support our beliefs.

Second, under the Connection tab, we found two drop-down lists for selecting the level of available bandwidth. The drop-down lists are labeled: "Normal" and "Maximum". The surrounding text states that if the same presentation is available in multiple encodings, the player will select one based on the settings. The value selected in the drop-down controls how much bandwidth the application is allowed to consume, if multiple streams are available. We were unable to infer the meaning of the two labels and how the selections in each drop down exactly affects the bandwidth consumption from the surrounding text. We used the online help provided with the application to further study the effect of the two bandwidth-controlling knobs. We discovered that the drop-down labeled "Normal" should be set slightly below the available bandwidth, and the drop-down labeled "Maximum" is applicable only when the Real Player is accessing streams authored using "Sure Stream", presumably a proprietary technology. "Sure Stream" allows the Player to modify the quality of the playback based on connection conditions.

Third, under the Connection tab, there is a button labeled: "Test Connection". Hitting that button initiates a battery of tests for the available bandwidth. The test is based on active probing, i.e., the application downloads some content from known internet addresses. The test takes approximately 10 seconds. We tested this feature several times, spacing the tests a few minutes apart. Using a residential, cable-based broadband internet, we discovered that the level of available bandwidth fluctuates over time.

While these knobs provide the means for controlling the quality vs. resource tradeoff of the application, successfully using the knobs for optimal experience hinges on a number of assumptions:

- the application is the only process that consumes significant amount of the available resources such as CPU and bandwidth,

- provided knobs can be used to guide the application towards a desired trade-off between different dimensions of application quality,

- the user can find the knobs, understand their intended use, and successfully manipulate the knobs based on the surrounding text or help.

Multiple dimensions of application quality (e.g., frame rate and resolution), multiple applications, and changing resource conditions greatly complicate the usage of those features. When environment conditions change, the user might need to manipulate multiple settings in multiple applications.

Internet Explorer. We surveyed version 6.0 of the Internet Explorer browser from Microsoft Corporation for the Windows platforms. The browser provides a number of features to control the quality of the browsing experience. Most of these features manipulate the richness of the web pages, but not latency of loading pages.

The majority of the controls are located on the “Advanced” tab in the “Options” menu. The options menu can be accessed by following menu items “Tools -> Internet Options". The controls are toggles for downloading pictures, sound, videos, and animations. Each toggle can be turned on and off independently, providing a large number of possibilities. When turned off, these controls instruct the browser not to download rich content. As a result, the bandwidth consumption of the browser is reduced.

In addition to features that are native to the browser application, various plug-ins (also known as add-ons) can be disabled, further reducing bandwidth usage. The Adobe Flash Player plug-in and the Java Virtual Machine plug-in are two commonly used add-ons that can be bandwidth intensive. Add-ons are accessible via menu option "Tools -> Manage add-ons".

The Internet Explorer application does not provide explicit controls for trading latency with bandwidth consumption. We used a third-party bandwidth limiting proxy to accomplish this. Browser’s traffic is routed through a proxy server installed on the same hardware. The proxy provides controls for allowing maximum bandwidth utilization per process. Configuring this setup requires two steps. First, the user must access the Internet Explorer Proxy settings from the Connections, by following menu item "Tools -> Internet Options". Second, the user needs to find, install, and configure a third-party bandwidth limiting proxy. To adequately perform that, the user must known about proxy programs, find one from a vendor that can be trusted, install it, configure it, and have the browser correctly interoperate with the proxy.