Dynamic Configuration of Resource-Aware Services

Vahe Poladian, João Pedro Sousa, David Garlan, Mary Shaw
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, USA
{vahe.poladian, jpsousa, garlan, mary.shaw}@ cs.cmu.edu

Abstract

An important emerging requirement for computing systems is the ability to adapt at run time, taking advantage of local computing devices, and coping with dynamically changing resources. Three specific technical challenges in satisfying this requirement are to (1) select an appropriate set of applications or services to carry out a user's task, (2) allocate (possibly scarce) resources among those applications, and (3) reconfigure the applications or resource assignments if the situation changes. In this paper we show how to provide a shared infrastructure that automates configuration decisions given a specification of the user's task. The heart of the approach is an analytical model and an efficient algorithm that can be used at run time to make near-optimal (re)configuration decisions. We validate this approach both analytically and by applying it to a representative scenario.

Keywords: Ubiquitous computing, resource-aware, multi-fidelity applications, service composition, resource allocation.

1. INTRODUCTION

Despite steady increases in computing resources, such as processing power, network bandwidth, and battery capacitance, applications' demand for these resources continues to grow and exceed supply. The problem is exacerbated in ubiquitous environments such as automobiles, coffee shops, and airport kiosks, where availability of resources is both limited and changing.

In response, we are beginning to see a new class of applications that are engineered to be "resource-aware" and "multi-fidelity" [11][16]. Such applications typically provide varying qualities of service depending on the resources available to them. For example, some video players can be configured to play at reduced frame rates in situations where bandwidth is limited.

While the creation of such applications is a necessary first step, it does not completely solve the problem. The main issue is that while resource-aware applications can work well in isolation, user tasks often encompass more than one activity, requiring resources to be allocated among several applications. For example, the task of writing a review for a documentary clip may require a video player, a text editor, and possibly a browser for searching the web for related information. Decisions on resource allocation made independently by each application may not yield an optimal composite solution.

This problem is further complicated by the availability of alternative ways to realize the same task in a given environment. On a typical system, video playback is possible using applications such as Windows Media, RealOne, and QuickTime. Combined with several possible text editors and web browsers, the task of writing a clip review can be supported by a large number of different application bundles. Thus the problem of *configuration* involves both the selection of applications as well as allocation of resources.

One obvious, but deficient, way to solve the configuration problem is to have the user manually configure the system. Unfortunately, while a user may know *what* he wants, he may not know *how* to realize it in a particular environment. Requiring a user to understand the low-level details of a potentially unfamiliar computing environment, including available resource levels and available applications, configuring the applications, and adapting to changes in the environment is clearly unreasonable.

In this paper we show how to partially automate system configuration for resource-aware applications. The key idea is to provide a system infrastructure independent of, and external to, applications [20]. Such infrastructure makes dynamic configuration decisions based on inputs describing users' quality of service requirements, resource and service availability, and application fidelity as a function of assigned resources. The heart of this infrastructure is an analytical model, and associated algorithm, that provides efficient, near-optimal configuration decisions.

To achieve the goal of partially automated configuration, the model separates concerns into three spaces: user utility, application capability, and computing resources; and two mappings. A mapping from capability space to utility space expresses the user's needs and preferences. A mapping from capability space to resource space expresses the fidelity profiles of available applications. Available resource levels are provided by the system, and constrain the configuration space to a feasible region. Configuring the system formally reduces to finding a point in the capability space that (1) maximizes user utility, and (2) satisfies the resource constraints. As we will show, this point identifies which applications to run, and its projection into the capability subspace of each application identifies the quality level for that application. Furthermore, it identifies when it is feasible, and desirable, to perform reconfiguration decisions, based on explicit modeling of the cost to make a change.

The rest of the paper is structured as follows. Section 2 surveys related work and highlights the novelty of this work. Section 3 defines the main requirements for automatic configuration, enumerates the expected Software Engineering benefits, and presents our approach. The mathematical formulation of the model and the expression of the underlying optimization problem are presented in Section 4. Section 5

presents an algorithm for the search problem, and analyzes its running time and scalability. Section 6 illustrates the application of our approach to a representative scenario. Section 7 presents an evaluation of our approach, highlights the contributions, discusses software engineering benefits, and enumerates related research that is out of scope of this paper.

2. RELATED WORK

Our work leverages multi-fidelity and resource-aware application research such as Odyssey and Puppeteer [5][6][11][16], but tackles the new problem of multi-component integration, configuration, and reconfiguration. Although somewhat related, this kind of *automatic configuration* is distinct from the automatic configuration being investigated in other research [13]. There, configuration is taken in the sense of *building and installing* new applications into an environment, whereas here, it is taken in the sense of *selecting and controlling* applications so that the user can go about his tasks with minimal disruption. Further, our work builds on service location [18] and discovery protocols and systems [9].

Resource scheduling [10], resource allocation [12][15] and admission control have been extensively addressed in research. Odyssey [15] addresses simultaneous adaptation of fidelity-aware applications, but it lacks a notion of task-wide user preferences. From analytical point of view, closest to our work are Q-RAM [12], a resource reservation and admission control system maximizing the utility of a multimedia server based on preferences of simultaneously connected clients; Knapsack algorithms [17]; and winner determination in combinatorial auctions. In our work, we handle the additional problems of selecting applications among alternatives, and accounting for cost of change.

Dynamic resolution of resource allocation policy conflicts involving multiple mobile users is addressed in [2] using sealed bid auctions. While our work shares utility-theoretic concepts with [2], 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. In our work, the objective is to choose among possibly thousands of configurations so as to maximize the objective function of one user. As such, our work has no game-theoretic aspects, but faces a harder computational problem. Furthermore, our work takes into account tasks that users wish to perform.

Rascal [8] defines resources as physical devices in a smart room, and configures resources by managing connections, interdependence of devices, wiring, and allocates resources among competing applications using coarse-grained utility. In our work, resources are CPU cycles, network bandwidth, memory, and our notion of utility is much more fine-grained.

Reducing user disruption in everyday computing is the broad goal of Project Aura [7]. The work herein focuses on the formal underpinnings of the mechanisms for the automatic configuration of the computing environment around a user. Such mechanisms are used in the middleware infrastructure supporting task-oriented, distraction-free computation in Project Aura.

3. APPROACH

In this section we define the configuration problem, pose specific technical challenges that need to be solved to provide automatic configuration, and describe our approach.

3.1 Terminology

We adopt the following, reasonably well-established terminology in approaching a solution to the problem [19]. The set of devices, applications, and resources that are accessible to a user at a particular location constitutes the computing *environment* for the user. Applications (and devices) in an environment provide *services,* such as video playing, text editing, web browsing. In a given environment there may be many applications, or *suppliers*, that can provide the same service. The richer the environment, the more suppliers are likely available for a particular service.

Users carry out *tasks,* such as reviewing a video, planning a vacation, or selling a house. Each task typically requires the use of multiple services. Today's systems provide only weak support for such tasks, and users typically have to manually configure the environments, by finding, directly invoking, closing, and switching between specific suppliers for the desired services.

Applications consume *resources* (such as computing cycles, network bandwidth, and battery power) in providing their services. In many environments (for instance, when the user only has a portable device available) resources are scarce. Furthermore, resource levels may change while a user is carrying out a given task. For example, bandwidth may vary over time in shared network environments, such as a shared DSL line at a coffee shop, or a wireless LAN on a university campus.

To provide an acceptable level of service despite the scarcity of resources, applications (often termed "multi-fidelity" or "resource-aware") are designed to reduce the quality of operation (e.g., exclude or reduce rich media from web-pages, reduce accuracy of speech recognition), and as a result consume less resources [6][11][16]. Such applications offer quality and resource tradeoffs to users. The following is a small subset of fidelity-aware applications: (1) media players such as Windows Media Player, RealOne Player, QuickTime, (2) browsers such as Internet Explorer, Netscape , Opera, (3) speech recognition software such as Janus, PanGlossLite, or the Microsoft Speech API.

3.2 The Challenge

The principal goal of our approach is to reduce user distractions with the problem of configuration and reconfiguration of computing environments. With that in mind, we would like to allow a user to specify *what* he wants, on a per-task basis, in terms of the services and their observable qualities, and then to automatically determine *how* to realize that in a changing environment. To meet this goal, a solution must address several key requirements.

R1. *Provide a representation of user needs and preferences that is expressive, efficient, and automatically processable.*

To automate configuration, the infrastructure needs to have sufficient input about the user's needs. What services are needed for the task? Given alternative suppliers of a service, which one should be chosen? What are the preferred tradeoffs between the quality dimensions, e.g. if the bandwidth drops, which dimension of quality should be reduced? The system should be able to determine answers to all these questions based on a specification of the user's task. Such specification should be efficiently represented. For example, enumerating a user's evaluation of each possible environment configuration state is infeasible, as that state may be rather large.

In the present work, we focus on the formalism to represent user needs and preferences, and on the mechanisms to exploit such representation for automatically configuring environments, rather than on the mechanisms to *elicit* those needs and preferences. The later is an important research topic in Human-Computer Interaction, and will be better addressed after the form and substance of the information required for supporting the automatic configuration of environments is well established.

R2. *Provide a representation of the capabilities of an application, including the services it provides and the relationship between the quality levels it supports and the resource levels it demands.*

While a specification of the user's task is needed to compute how good a given configuration is, in order to determine possible configurations of the environment, a well-specified description of the environment is necessary. What are the available suppliers? What quality and resource tradeoffs do the suppliers offer?

R3. *Provide an efficient algorithm for computing a near-optimal configuration of applications and quality levels, given a specification of the user's task, and a specification of the current environment.*

To determine the best possible match between user's needs and environment capabilities at run time, an efficient algorithm is needed to find that match. Ideally, the algorithm will employ provably correct strategies to find an optimal, or near-optimal, solution from the user's point of view, avoiding an exhaustive search of the space, which may be very large.

R4. *Provide a mechanism to minimize disruption to the user resulting from changes in the environment, or in user intent.*

As resources and supplier availability change over time, an instantiated configuration may become either infeasible or sub-optimal. Thus, the infrastructure should have mechanisms to adapt the configuration. In making such adaptations, it is desirable to reduce unnecessary disruption to the user.

Additionally, should the user's task require an additional service, disbanding a running service, or adjusting the user preferences for a running service, the infrastructure should be able to handle changes gracefully, in an incremental fashion.

3.3 Software Engineering Benefits

As argued above, an automatic mechanism for the dynamic configuration of computing environments is needed to (a) find optimal solutions in a potentially large space from the point of view of the user, that is, in a utility-theoretic sense; and (b) reduce the user's disruption in configuring those environments for supporting his tasks.

Additionally, we would expect a good solution to provide engineering benefits for application developers, specifically:

- making it easier to develop resource-adaptive applications by factoring a number of mechanisms out of the applications and into a common infrastructure;
- making it feasible to improve the overall quality of service to the user by using theoretically sound strategies for optimizing the configuration and resource allocation among all applications involved in a task, rather than having to rely on application-centric heuristics;

3.4 Approach

Our approach is based on optimizing the match between the needs of a user task and the environment capabilities. In practice, finding such a match corresponds to maximizing user's utility for a specific task. We express user's utility by means of *preference functions* that map from a multidimensional *capability* space to a uni-dimensional *utility space*. Further, we express concrete capabilities of the suppliers in an environment by means of *application profiles* that specify tradeoffs between the *capability space* and *resource space*.

In the following subsections, we elaborate on these concepts and extend on an example, introduced earlier, in which the user's task is to prepare a review of a documentary movie. Recall that to support this task, the user requests three services: *video playing*, *text editing*, and *browsing*.

3.4.1 Utility Space

Utility is a measure of user's happiness with respect to possible outcomes. For the purposes of our paper, the utility space provides a formal representation of how useful the possible configurations of services in the environment are relative to a specific task. We encode utility in the interval [0,1] of the real numbers, where 0 utility corresponds to the environment being unacceptable for the task; and 1 corresponds to user satiation, in the sense that increasing the capabilities of the environment will not improve the user's perception of usefulness for the specific task.

3.4.2 Capability Space

The capability space is the Cartesian product of the QoS dimensions for each service, and of an additional dimension that captures supplier-specific features. Specifically, the latter indicates the *supplier* type – for example, for *text editing*, possible values would be MS Word, Notepad, Textpad, Wordpad, UltraEdit, Emacs, etc. Supplier type is a compact representation for the availability of user-desired features, such as automatic spell checking, or sophisticated text formatting.

Examples of quality of service dimensions are, for video playing, frame rate, frame quality, and audio quality; for browsing, latency of page loading, and richness of the pages. The unit and range of possible values are dimension-specific. For example, frame rate is measured in integer frames per second and lies in the range from 0 to 30. The richness of a web page is described by the discrete enumeration: {Images, No Images}.

To determine whether a particular aspect of a service is a quality dimension, we follow the following criteria:

- does not depend on choosing a particular supplier,
- the level can be varied (adapted) at run time, and
- the resource demand varies with the level of quality.

Video frame rate satisfies all three. First, all video players support the notion of frame rate; second, it can be changed dynamically at run time; and third, higher frame rate typically demands more CPU and bandwidth. On the other hand, spell-checking support for text editors is a feature that is not a quality dimension. First, although common, not all text editors support spell checking. Second, it would be awkward, and probably ineffective, to enable or disable spell checking at run time in response to resource variations.

3.4.3 Resource Space

The resource space describes the computational resources of the environment. For the purposes of this paper, we consider four types of resources: CPU, memory, battery, and network bandwidth. The resource space is the Cartesian product of individual resource dimensions. Although we account for three types of resources, in a particular environment, the number of resource dimensions can be more than three. For example, if the environment includes two computers, then the CPU of each is accounted for as a separate resource dimension. Further, upstream and downstream bandwidth of a particular computer can be accounted as a separate resource.

3.4.4 User Preferences

User preferences are a collection of functions that evaluate how useful the environment is from the point of view of his task. Formally, they map from the capability space to the utility space. For each point in the capability space, user preferences help compute a real-numbered utility value. The capability space is potentially large but structured, and user preferences are designed to take advantage of such structure. For the purposes of this paper, user preferences capture two concepts. First, *QoS preferences* express user's utility for each possible level of service in each individual dimension of quality of service (QoS), and tradeoffs among these dimensions. And second, *supplier preferences* capture which specific applications are preferred to supply those services.

To illustrate supplier preferences, we refer again to the example of reviewing a clip. For taking notes (*text editing* service), the user may prefer MSWord over Notepad or Emacs and be unwilling to use the *vi* editor at all. Note that representing supplier preferences by discriminating the *supplier type*, e.g. preference of MSWord over Notepad, is a compact representation for the preferences with respect to the availability of desired features, such as spell checking, richness of editing capabilities, and to the user's familiarity with the way those features are offered.

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 both process and elicit, we make two simplifying assumptions with respect to their form. 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. 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 encoded by 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. image quality) 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 *audio fidelity*, whose values are *high*, *medium* and *low*, we simply use a discrete mapping (table) to the utility space. In the case studies we evaluated so far, we found the expressiveness of the forms above to be sufficient.

3.4.5 Application Profiles

Typically, an application 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. 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. Despite discrete approximation, our approach does allow the handling of a rich capability space. For example, the capability

space of a specific video player application can have 90 points, which is made possible by combining 5 frame rates, 6 frame sizes, and 3 audio qualities. Such a capability 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.

Application profile specifies a discrete enumeration of the capability points supported by an application and corresponding resource demand for each point.

Note that specific *mechanisms* for obtaining and expressing application profiles exist. As demonstrated in [14], resource demand prediction based on *historical* data from experimental profiling is both feasible and accurate. Further, *metadata* can be used to express application profiles [3].

4. MATHEMATICAL FORMULATION

In this section we present a mathematical formulation of the model. We define the utility space, capability space, and resource space, and define allowed operations in each space. We formulate the configuration problem as an optimization over a search space. In later sections we describe how our implementation realizes this mathematical model through efficient algorithms and shared infrastructure.

4.1 Utility Space

The utility space is represented by the real number interval [0, 1]. The user's happiness with an outcome is represented by a utility value. The user is happy with utility values close to 1, and unhappy with utility values close to zero. Given two outcomes, to determine the preferred one, we compare their utilities. Higher utility corresponds to the preferred outcome.

4.2 Capability Space

The capability space $\boldsymbol{C_s}$ corresponding to service s is the Cartesian product of the individual quality dimensions d of the service:

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

For example, for *video playing* service the quality dimensions are the *frame update rate*, the *frame size*, and *audio quality*. Thus, the capability space of video playing is three-dimensional.

Cartesian product is used to combine the capability space of two services. For distinct services s and t, their combined capability space is formally expressed as:

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

For example, a *web browsing* service has two quality dimensions: *latency* and *page richness*, and *video playing* has 3 dimensions of quality. Thus joint capability space of *video playing* and *web browsing* has 5 quality dimensions.

4.3 Resource Space

The resource space $\boldsymbol{R}$ is the Cartesian product of the individual resource dimensions r of the entire environment $\boldsymbol{E}$:

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

Examples of resource dimensions are: CPU cycles, network bandwidth, memory, and battery. The actual number of resource dimensions is dependent on the environment.

4.4 User needs and preferences

The user expresses the requirements for a task by specifying services needed and the associated preferences. A shared vocabulary of services and service-specific quality dimensions must exist between the user and the system. Developing such a vocabulary is a subject of related research and out of the scope of this paper, but we give insights to the essential characteristics of such a vocabulary [4].

The user specifies a requested service using its type, which is part of the common vocabulary. For example, to watch a video, user requests a *video playing* service. The user specifies an operation for the service: *add*, *replace*, and *disband*. The add operation requests a new service, disband requests that the current supplier for the service be shut down, and replace requests that the current supplier of a previously requested service be disbanded, and another supplier be added. Note that supplier changes resulting from a replace command should be accounted as user-initiated, and should not carry disruption costs.

4.4.1 QoS Preferences

QoS preferences specify the utility function associated with each QoS dimension. The names of the QoS dimensions are also part of the shared vocabulary. The utility of service s as a function of the quality of service is given by:

$$U_{QoS}(s) \triangleq \prod_{d \in QoS\ \dim(s)} F_d^{c_d}$$

where for each QoS dimension d of service s, $F_d : dom\ (d) \rightarrow (0,1]$ is a function that takes a value in the domain of d, and the weight $c_d \in [0,1]$ reflects how much the user cares about QoS dimension d. As an example, *video playing* has a QoS dimension of *frame update rate*. The function $F_{frameRate}$ gives utility for various frame rates, and $c_{framerate}$ specifies the weight of frame rate.

Weighted product specifies an "AND" semantics when combining QoS dimensions. A utility value of zero in one dimension indicates that the user is not interested in the configuration even if the quality of other dimensions is high.

4.4.2 Supplier Preferences

To evaluate the assignment of specific suppliers, we employ a supplier preference function, which is a discreet function that assigns a score to a supplier, based on its type. Also, we account for the *cost of switching* from one supplier to another at run time.

Precisely, the utility of the supplier assignment for a set a of requested services is:

$$U_{Supp}(a) \triangleq \prod_{s \in a} h_s^{x_s} \cdot F_s^{c_s}$$

where for each service s in the set a, $F_s : Supp\ (s) \rightarrow (0,1]$ is a function that appraises the choice for the supplier for s; and the weight $c_s \in [0,1]$ reflects how

```
HashMap SuppPrefs;    // supplier preferences
HashMap QoSPrefs;     // qos preferences
HashMap SuppReg;      // registered suppliers
Config BestConfig(Set reqstdSvcs){
  // 1. QUERY
  Map  suppListsBySvc;
  for each svc in reqstdSvcs do{
    List  suppList = null;
    Pref  suppPref = SuppPrefs.get(svc);
    // query for supp based on svc type
    suppList = query(SuppReg, svc, suppPref);
    suppListsBySvc.put(svc.type, suppList);
  }
  // 2. GENERATE configs, compute supp pref
  List configs = GenConfigs(suppListsBySvc);
  configs = sort(configs);
  // 3. EXPLORE the QoS space
  int indexBestConfig;
  float overallUtilBest = 0.0;
  for each i from configs.size-1 to 0 do {
    Config cCur = configs.get(i);
    if (overallUtilBest > cCur.suppPrefUtil)
        break;
    cCur = searchQoS(cCur, QoSPrefs);
    if (cCur.overallUtil > overallUtilBest){
        indexBestConfig = i;
        overallUtilBest = cCur.overallUtil;
    }
  }
  return configs.get(indBestConfig);
}
GenConfigs(Map suppListsBySvc){
  List configs = new List(MAX_INT);
  int depth = 0;
  Config partialConfig = null;
  GenConfigsRecur(depth, configs,
       suppListsBySvc, partialConfig);
}
GenConfigsRecur(int d, List configs,
   Map suppListsBySvc, Config partialConfig){
  if ( d == suppListsBySvc.size()){
    configs.add(new Config(partialConfig));
    return;
  }
  List suppList = suppListsBySvc.getByInd(d);
  for each supp in suppList do{
    partialConfig.add(d, suppList);
    GenConfigsRec(d+1, configs,
            suppListsBySvc, partialConfig);
    partialConfig.remove(d);
  }
}
```

Figure 1. Pseudocode of the algorithm

much the user cares about the supplier assignment for that service.

4.4.3 Accounting for Switching of Suppliers

Among the technical challenges to automatic configuration, is requirement **R4** in 3.2.

The term $h_s^{x_s}$ above (4.4.2) expresses a change penalty as follows: h_s indicates the user's tolerance for a change in supplier assignment: a value close to 1 means that the user is fine with a change, the closer the value is to zero, the less happy the user will be. The exponent x_s indicates whether the change penalty should be considered (x_s=1 if the supplier for s is being exchanged by virtue of dynamic change in the environment) or not (x_s=0 if the supplier is being newly added or replaced at the user's request).

4.4.4 Overall Utility

Overall utility is the product of the QoS preference and supplier preference. The overall utility over a set a of suppliers is:

$$U_{overall}(a) = \prod_{s \in a} h_s^{x_s} \cdot F_s^{c_s} \left(\prod_{d \in QoS\,\dim(s)} F_d^{c_d} \right)$$

4.4.5 Application Profiles

Application profiles describe the relationship between the capability points supported by applications, and the corresponding resource requirements. Formally, the quality resource mapping of supplier p is a partial function from the capability space of service s to the resource space: $QoSprof_p : C_s \mapsto R$. The range of the function is the subset of the capability space that is supported by the supplier.

4.5 The Optimization Problem

The optimization problem is to find a supplier assignment a, and for each supplier p in this assignment, a capability point such that the utility is maximized:

$$\underset{\substack{p_s \in Supp(s) \\ f_d \in dom(d)}}{\arg\max} \prod_{s \in a} h_s^{x_s} \cdot F_s^{c_s}(p_s) \cdot \left(\prod_{d \in QoS\,\dim(s)} F_d^{c_d}(f_{p,d}) \right)$$

The maximization is over a set of constraints, which we express below. The capability constraint stating that the chosen point $f_{p,d}$ is in the capability space for supplier p is as follows:

$$\forall_{p \in Supp(s)} f_p = \otimes_{d \in QoS\,\dim(s)} f_{p,d} \in C_p$$

And to ensure that the resource constraints are met:

$$\sum_{p \in Supp(s)} QoSprof_p(f_p) \le |R|$$

where summation is in the vector space $\boldsymbol{R}$ of resources, and the inequality is observed in each dimension of that space. In non-mathematical terms, this constraint expresses the fact that the aggregate resource demand by all the suppliers can not exceed the resource supply.

5. ALGORITHM AND ANALYSIS

In this section we solve the optimization problem. As identified by requirement **R3** in 3.2, the algorithm must be efficient to be usable at runtime. Two metrics we are interested in are the latency of computing an answer to a given instance of the problem, and in the computational overhead of the algorithm.

5.1 The Algorithm

The algorithm works in three phases: (1) query, (2) generate, and (3) explore. In the first phase, it queries for relevant suppliers for each service in the task. In the second phase, it combines suppliers into configurations and ranks them according to the supplier preference only. In the third phase, it explores the quality space of the configurations. The pseudocode of the algorithm is shown in Figure 1.

The double product term of the utility formula in 4.5 allows for a clever exploration strategy. The outer product is the supplier preference score. It can be computed at the time the supplier assignment is known (in phase 2), and can be used as an upper bound for overall utility during the explore phase. Since overall utility is the product of supplier preference and QoS preference, and the latter is bounded by one, then maximum overall utility is bounded by supplier preference. The break in the loop in BestConfig takes advantage of that fact.

Consider a simple example. Assume that two services are requested. For each service, there are two possible suppliers: a_1 and a_2 for the first service, b_1 and b_2 for the second, yielding 4 possible configurations as shown in Table 1. The search space can be divided into 4 quadrants, each representing the capability space of a specific configuration. We are searching for a point with the highest utility.

As noted, the maximum utility that can be achieved within each quadrant is bounded by the supplier preference portion of utility. These observations help provide a stop condition for the search: once a point is found that has overall utility of Λ, there is no need to explore configurations with supplier preference portion of utility of less than Λ.

Table 1. The structure of the search space

a_1,b_1	a_1,b_2
a_2,b_1	a_2,b_2

In Table 1, the shading of each quadrant reflects the hypothetical values of supplier preference portion of utility for each configuration: the darker the shade, the higher the value. Assume these values are: .8, .6, .4, and .2. Each of these values is an upper bound for maximum *overall utility* possible from the respective quadrant. We explore inside the quadrants, starting from the darkest. If the maximum utility for the quadrant a_1, b_1 is higher than 0.6, then at this time we know the best point in the entire space is found, and can stop the search. If not, we continue the search in quadrant a_2, b_1, and so on.

Exploring the quality space of a configuration is a variant of a 0-1 Knapsack problem, called multiple dimensional, multiple choice 0-1 Knapsack. Multiple dimensions refer to the multiple constraints that are present in the problem. Multiple choice refers to choosing one among a set of similar items. In our problem, resources map to knapsack dimensions and the capability space of one service maps to one set of similar items. This is a well-studied problem in the optimizations research, and is at the core of such optimization problems as winner determination in combinatorial algorithms. [12][17] show the problem to be NP-complete, and give approximation algorithms. [17] gives an exact solution that is demonstrably fast on inputs drawn from certain probability distributions.

One of the approximating algorithms to the problem uses utility to resource ratio as a metric for ranking the capability points, then applies greedy selection and LP-relaxation to find a near-optimal answer. In the multiple resource case, quadratic weighted-average is used to compute a single resource currency from multiple resources, and the solution to the single resource case is reused iteratively [12].

In our solution, SearchQoS invokes a third-party library called Q-RAM, the package described in [12].

5.2 Analysis

To analyze the running time of the algorithm, let:

- **n** be the number of requested services
- **P** be the total number of available suppliers
- **p** be the number of suppliers for a given service type
- **q** be the size of the capability space of a supplier.

P and **p** describe the *richness* of the environment, and can potentially increase as more applications, hardware, and devices are made available. **q** describes the capability richness of a supplier. It is reasonable to assume that the size of the user task is limited to a small number of applications. Thus **n** is bounded.

Next we analyze the running time of the three phases.

The query phase retrieves items from a hashtable. Retrieving one item can be done in constant time. **n** retrievals from a hashtable take O(**n**) time.

The generate phase is a recursion of depth **n**, with a loop of size **p** at each level. Thus, it takes O($\mathbf{p}^{\mathbf{n}}$).

The explore phase in the worse case takes O($\mathbf{p}^{\mathbf{n}}$) * O(searchQoS). The size of the QoS space of a configuration of **n** suppliers each of which has a capability space of size **q** is O($\mathbf{q}^{\mathbf{n}}$). Approximation algorithm we use can search that space in time O(**n*****q***log**q**) [12][17]. Thus the explore phase takes O($\mathbf{p}^{\mathbf{n}}$) * O(**n*****q***log**q**) in the worst case, and dominates all other terms. The first term, O($\mathbf{p}^{\mathbf{n}}$), presents a possible scalability bottleneck.

Let us demonstrate how the exploration strategy described earlier helps tackle that bottleneck. Recall the break condition in the explore phase, illustrated in the example introduced in 5.1. The number of configurations that are explored will depend on the distribution of the supplier preference values, and Λ, the highest achievable utility value. Let's assume an average number of suppliers per service **p** = 10, and a specific distribution of supplier preference values that is uniform, i.e. the most preferred supplier scores 0.9^0, the next one scores: 0.9^1, etc. In Table 2, we show the number of configurations generated, and the number of configurations that are actually explored depending on the value of maximum achievable utility, Λ, and number of services in the task, **n**.

Table 2. Number of configurations generated and explored for various values of *n*, and Λ, maximum utility achieved

	n=1	2	3	4	…	8
Generated	10	10^2	10^3	10^4	…	10^8
Λ = .9	2	3	4	5	…	8
Λ = .81	3	6	10	15	…	36
Λ = .73	4	10	20	35	…	120
Λ = .66	5	15	35	70	…	330

The first row shows the number of services. The second row shows the number of configurations generated, which is $\mathbf{p^n}$, in this case, $10^{\mathbf{n}}$. In each subsequent row, we show the number of configurations that are sufficient to explore, if the maximum utility shown in the first column in that row is actually achieved by some configuration. For instance, for a task with 4 requested services, even if the maxim utility achievable is as modest as $\Lambda = 0.6$, then the number of supplier configurations explored is 126, which is two orders of magnitude smaller than the 10^4, the total number of configurations.

5.3 Reconfiguration

The algorithm also handles reconfiguration. When there is a running configuration, the utility from the best computed configuration is compared with the *observed* utility of the running configuration, and a switch is made if the latter is lower than the former. Because of the cost of change term in formula 4.4.2, some of the suppliers in the running configuration may get an advantage.

6. CASE STUDY

In this section we report on a case study of configuring an environment for the task of reviewing a documentary video clip. The task requires 3 services: *video playing*, *text editing*, and *browsing*. The user watches the clip, takes notes, while browsing the net for information. The quality dimensions are as follows: for video playing: frame rate, frame size, and audio quality; for browsing: latency of loading pages, and richness of the pages (pages have graphics that are not required for the task can be helpful); for text editing: none.

We performed the case study in two steps. In the first step we collected application profile data, specified preferences, and identified resource limits. In the second step, we ran a prototype implementation of the algorithm.

6.1 Input Data Collection

As an experimental platform, we chose an IBM Thinkpad 30 laptop, equipped with 256 MB of memory, 1.6 Ghz CPU, WaveLAN card, and Windows XP Professional. In power saving mode, the CPU can run at a percentage of the maximum speed, effectively creating a tight CPU constraint.

The model requires three inputs: (1) user preferences, (2) application profiles, and (3) resource availability. For the purposes of this experiment, we used synthetic preferences intended to be representative of the tasks. We identified several applications that supported various facets of the task. Those applications were installed on the laptop. To obtain application profiles, we measured resource usage corresponding to a small set of capability points. We performed this profiling *offline*, with each supplier running separately. Resource availability is as follows: 400 MHz of processing power, when the CPU is running at ¼ of the baseline speed; 64 MB of free memory after excluding the memory taken by the operating system and other essential critical systems; and 512 Kbps of bandwidth, provided by an 802.11 wireless access point backed by a DSL line.

The applications used in the experiment were:

- Video players: RealOne and Windows Media,
- Text editors: TextPad, WordPad, Notepad, Microsoft Word, and GNU Emacs,
- Browsers: Internet Explorer, Netscape, and Opera,

These suppliers allow a total of 30 = 2*5*3 configurations.

We measured CPU and physical memory load using Windows Performance Monitor. We used *percent processor time*, *working set* counters of the *Process* performance object to measure CPU and memory utilization respectively. We took the sampling *average* over a period of time. The performance monitoring API does not provide per process network statistics, so the mechanism for measuring bandwidth demand was different in each case, as explained below.

For a representative clip to watch, we obtained a two minute trailer of a movie in Windows native .wmv and Real Networks native .rpm in several different bit-rates. Where cross-player compatibility is supported, we obtained additional capability points. For example, RealOne plays .wmv format. Also, players provide quality knobs, allowing improved quality in exchange for higher CPU utilization. For example, Windows Media player supports video smoothing that provides higher frame rate than the rate encoded in the stream. For each player, 32 points quality points were sampled. To measure bandwidth demand, we consider the bit-rate of the stream, and cross-check with the application-reported value. We observed that the CPU consumption of different players can be widely different for the same quality point.

We measured CPU and memory used while typing and formatting text for 2 minutes with each text editor. We observed that the memory consumption of different text editors can be widely different.

All browsers surveyed support a text-only mode, providing two points in the page richness dimension. To obtain different levels of latency, we used a bandwidth-limiting http proxy, and pointed the browser to the proxy. We measured latency by allowing the following bandwidth limits: 28, 33, 56, 128, 256, 512 Kbps. Our script included a sequence of approximately 15 pages with a mix of both text graphics on the internet. Each test started with a clean browser cache. 16 quality points were sampled. We observed that the browsers have very similar resource consumption patterns.

Although we realize that the methods for obtaining the resource consumption measures are not precise, we believe that they yield good enough approximations for this feasibility analysis.

Note that the capability space of a configuration of suppliers has approximately 500 points (32*16*1), based on the samples taken. 30 configurations together provide a capability space of approximately 15,000 points.

6.2 Prototype Evaluation

The algorithm is guaranteed to find an optimal assignment of suppliers. Furthermore, it will obtain the optimal set of quality points for the suppliers, as long as Q-RAM finds the optimal point inside each quadrant. Whenever Q-RAM

returns a near-optimal answer, our algorithm will return a near-optimal set of quality points.

Additionally, we evaluated a prototype implementation of the algorithm according to two metrics: (1) latency, and (2) system overhead. Latency measures the time it takes to compute an answer, from the time that a client program requests it. Overhead measures percent CPU and memory utilization of the algorithm. To adapt the configuration in response to environment changes, it might be necessary to run the algorithm periodically. Thus, the overhead of the periodic invocation provides a useful metric.

The latency of computing the best configuration averaged over 10 trials was 531 ms. In the query and generate phases, the algorithm spends less then 10 ms each. In the explore phase, it spends just under 500 ms (approximately 10 ms was due to parsing the request, and formatting the answer). The bulk of the time in the explore phase was due to external process invocation and file input-output (Q-RAM package is an external executable). Thus, the latency can be significantly reduced by linking into Q-RAM in-process.

We invoke the algorithm a total of 50 times in 5 second intervals over a period of 250 seconds, and measure *average CPU utilization*. Average CPU utilization is 3.8%. This overhead is fairly low, and can be further lowered by running the algorithm less frequently, e.g. once per 10 or 25 seconds. Alternatively, the algorithm can be run in response to events that may lead to changing the configuration, e.g. in response to changes in suppliers, resource levels, or user preferences.

Memory usage of the Java Virtual Machine process running the algorithm is approximately 8.8 MB.

The case study presented here only addresses the performance of the optimization algorithm, using data collected via profiling. Ongoing work on the Aura platform [19] addresses the runtime instantation and monitoring of applications utilizing the analytical model described in this paper.

7. EVALUATION

In this section we discuss how our approach meets the requirements of Section 3.2, highlight software engineering benefits of the approach, discuss the limitations of the model and scope of the work, and indicate future work.

7.1 Addressing the Requirements

We set forth four requirements in Section 3.2 (to conserve space, we omit the text of the requirements).

R1: *Representation of user task.* Our approach allows a user to specify services needed, and preferences for observable qualities of the services. Through preferences, the user can express fine grain control over resource allocation between applications, and guide adaptation policies of applications. Although the space of possible configurations is large, the encoding of preferences is efficient because it exploits the independence of quality dimensions. Sigmoid functions further improve the efficiency of the encoding, requiring three numbers per quality dimension: two for the knees of the sigmoid curve, and one for the weight of that dimension.

To enable automatic processing of a user request, our approach employs a shared vocabulary of service names, quality dimensions and preference encodings. Representation of user's task is only one aspect of a broader problem. Another important issue is the *elicitation* of preferences, which is not addressed in this paper, but is a subject of ongoing research.

R2: *Representation of application capabilities*. Our approach meets this requirement by specifying the capabilities of an application using a shared vocabulary of service names and quality dimensions, and using discrete enumeration of vector pairs to specify the relationship between quality and resources. This is a general solution, as it does not rely on the existence of functional forms between application quality and resource usage. Our approach uses historical profile data to estimate demand at runtime.

R3: *Optimal and efficient algorithm*. We have presented an algorithm for efficient searching of the configuration space. We have demonstrated analytically that the algorithm finds a near-optimal answer in polynomial time by taking advantage of the special structure of the problem, which is NP-hard. Using a case study, we have demonstrated that the algorithm has a latency of less one second on an input of moderate size.

R4: *Reconfiguration*. Our approach explicitly models the cost of disruption to the user from possible reconfigurations. Specifically, our approach considers on-the-fly switching of suppliers as a possible source of disruption, allows the user to express his tolerance for such a change as a preference function, and takes the latter into account when doing reconfiguration.

Our approach also allows user-initiated incremental changes to a running configuration.

7.2 Software Engineering Benefits

Although the primary focus of this paper has been on addressing issues of efficiency and optimality, the approach also embodies a number of significant engineering benefits.

First, our approach provides an integration architecture (via a shared infrastructure) that helps maximize the benefits of resource-aware applications. Today's default integration architecture requires each such application to make resource-awareness decisions independently. In our approach applications are composed through intermediaries and resource allocation decisions are made through a shared infrastructure that can provide better overall utility to the user by considering the capabilities of the entire system.

Second, our approach addresses usability issues of today's applications. By providing mechanisms that automatically translate from user preferences of the qualities they desire to the configuration of specific services, we reduce the cognitive burden of users in taking full advantage of resource-configurable applications.

Third, our approach adds new functionality to existing applications that makes them better suited for resource variable environments. Specifically we incorporate in our configuration infrastructure the ability to make *reconfiguration* decisions, factoring in issues of user distraction.

Fourth, our approach can contribute to reductions in engineering costs for resource-aware applications themselves

by shifting the burden of decision-making into the infrastructure.

7.3 Limitations of the Model

Our approach relies on a small number of assumptions, which arguably are reasonable for the problem at hand. For completeness, we discuss these assumptions:

- Our multiplicative model of preferences for QoS requires that these dimensions satisfy certain *independence* assumptions. Proving that independence conditions hold in each case is not possible. However, there is evidence, that even though in practice independence does not hold completely, additive (and consequently multiplicative) preferences provide good enough approximations [1][21].
- The model depends on accurate prediction of applications' resource demand. Related research suggests [14], that historical profiling can be used to accurately predict demand. Although we realize that the methods for obtained the resource consumption measures for the case study described in Section 6 are not precise, we believe that they yield good enough approximations for this feasibility analysis.

7.4 Other Research

To address the configuration problem completely, several related problems need to be solved. In this work, we address only a subset of them. The following related problems are being addressed elsewhere:

- Elicitation of user tasks [4] and preferences. For the approach to work in practice, user-friendly and accurate elicitation of preferences is required.
- Development of resource- and fidelity-aware applications [5][11][16].
- Development of an infrastructure for runtime monitoring of resource supply levels [16].

7.5 Conclusion

An important emerging requirement for computing systems is the ability to adapt at run time, taking advantage of local computing devices, and coping with dynamically changing resources. In this paper, we have presented an analytical model and an efficient algorithm for the dynamic configuration of resource-aware services in ubiquitous computing environments. We demonstrated that it is feasible to automatically configure such environments, given a specification of user preferences. Our analysis shows that the presented algorithm is scalable to rich environments. Additionally, the preliminary evaluation of the prototype implementation yielded promising performance. Furthermore, our approach embodies a number of software engineering benefits. Future work includes runtime instantiation of applications, monitoring of the environment, and scaling the approach to more than one device under a user's control.

8. ACKNOWLEDGMENTS

This research is supported by the National Science Foundation under Grants CCR- 0205266, ITR-0086003, by the Sloan Software Industry Center at Carnegie Mellon, and the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298. Authors would like to thank Bradley Schmerl for help in designing the system, and Sourav Ghosh for help in integrating QRAM.

9. References

[1] S. Butler. Security Attribute Evaluation Method. A Cost-Benefit Approach. *Proc Int'l Conf in Software Engineering (ICSE)*, 2002.

[2] 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).*

[3] L. Capra, W. Emmerich and C. Mascolo. Reflective Middleware Solutions for Context-Aware Applications. *Int'l Conf on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION),* 2001.

[4] The DAML Services Coalition (multiple authors), "DAML-S: Web Service Description for the Semantic Web", *Int'l Semantic Web Conference (ISWC)*, 2002.

[5] J. Flinn, M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. *Proc Symp Operating Syst Principles (SOSP)*, 1999.

[6] J. Flinn, E. de Lara, et al. Reducing the Energy Usage of Office Applications. *IFIP/ACM Int'l Conf on Distributed Syst Platforms* (Middleware), 2001.

[7] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: Towards Distraction-Free Pervasive Computing. *IEEE Pervasive Computing,* Volume 21, Number 2, April-June, 2002.

[8] K. Gajos. Rascal - a Resource Manager for Multi-Agent Systems In Smart Spaces. *Proceedings of CEEMAS'01*, 2001.

[9] Jini. www.jini.org. Accessed: Sep. 2003.

[10] M. Jones, D. Rosu, M. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *Proc Symp Operating Systems Principles (SOSP)*, 1997.

[11] 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.

[12] C. Lee, et al. A Scalable Solution to the Multi-Resource QoS Problem. *Proc IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[13] F. Kon, et al. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. *Proc. USENIX Conference on OO Technologies and Systems (COOTS)*, 2001.

[14] 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.

[15] R. Neugebauer and D. McAuley. Congestion Prices as Feedback Signals: An Approach to QoS Management. *Proc. ACM SIGOPS European Workshop*, 2000.

[16] B. Noble, et al. Agile Application-Aware Adaptation for Mobility. *Proc ACM Symp Operating Systems Principles (SOSP)*, 1997.

[17] D. Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114, (1999).

[18] Service Location Protocol. www.srvloc.org. Accessed: Sep. 2003.

[19] J.P. Sousa, D. Garlan. The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. *Carnegie Mellon Technical Report, CMU-CS-03-183*, 2003.

[20] J.P. Sousa, D. Garlan. Improving User-Awareness by Factoring it Out of Applications. *Proc System Support for Ubiquitous Computing Workshop (UbiSys)*, 2003.

[21] Yoon, K. Paul and Hwang, Ching-Lai. *Multiple Attribute Decision Making: An Introduction*, Sage Publications, 1995.