Submitted for publication.

Task-based Self-adaptation

David Garlan, Vahe Poladian, Bradley Schmerl, João Pedro Sousa
Carnegie Mellon University
Computer Science Department
Pittsburgh, PA 15213, USA
+1-412-268-5056
[garlan | poladian | schmerl | jpsousa]@cs.cmu.edu

ABSTRACT

Recently there has been increasing interest in developing systems that can adapt dynamically to cope with changing environmental conditions and unexpected system errors. Most efforts for achieving self-adaptation have focused on the mechanisms for detecting opportunities for improvement and then taking appropriate action. However, such mechanisms beg the question: what is the system trying to achieve? In a given situation there may be many possible adaptations, and knowing which one to pick is a difficult question. In this paper we advocate the use of explicit representation of user task as a critical element in addressing this missing link.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**] Patterns, D.2.1 [**Requirements/ Specifications**] Languages, D.2.5 [**Testing and Debugging**] Monitors, Error handling and recovery.

General Terms

Design, Reliability

Keywords

Self-adaptation, self-management, software architecture, task-aware computing, utility-based optimization.

1. INTRODUCTION

Self-adaptive systems are becoming increasingly important. What was once the concern of specialized systems, with high availability requirements, is now recognized as being relevant to almost all of today's complex systems, and particularly those where environmental resources can change radically (e.g., mobile computing) or where systems must continue to run in the presence of failures (e.g., space systems, e-commerce, medical systems).

Currently adaptive systems tend to fall into two broad categories:

1. Fault-tolerant systems: Fault-tolerant systems react to component failure, catching or compensating for errors using a variety of techniques such as redundancy and graceful degradation. Such systems have been prevalent in safety-critical systems or systems for which the cost of off-line repair is prohibitive (e.g., telecom, space systems, power control systems, etc.) Here the primary goal is to prevent or delay large-scale system failure.

2. Resource-aware systems: Resource-aware systems react to resource variation, adapting components so they can function optimally with the current set of resources (bandwidth, memory, CPU, power, etc.) These systems emerged with the advent of mobile computing over wireless networks, where resource variability becomes a critical concern. Adaptation may be local to a given component: for example, one might adjust the fidelity of a video player to accommodate a drop in bandwidth; or one might degrade the accuracy of speech recognition for the sake of response time [6]. Alternatively, adaptation may be global: for example, a system might reconfigure a set of clients and servers to achieve optimal load balancing. Typically, such systems use global system models, such as architectural models, to achieve these results [2][4] [5].

While these systems demonstrate important new capabilities, they tend to beg the important question: how do you choose the appropriate adaptation, given that there may be several possibilities. For example, in the presence of reduced bandwidth a video player might select any of several possible adaptations: reduce the frame rate; reduce the picture size; increase the granularity; eliminate color. Which is the right adaptation?

Of course, the answer depends critically on the use of the system: what the user is trying to achieve with it. Unfortunately, most systems have no knowledge of user goals and intent.

In this paper we describe an emerging complementary aspect of self-managed systems: task-aware adaptation. The key idea is for the system to maintain an explicit representation of user intent, including preferences for quality tradeoffs, and of the nature of the services required, which are contextual pre-conditions for adaptation. In the remainder of the paper we discuss some desirable characteristics of task-aware systems, outline key research questions that arise in developing those systems, and briefly describe the Aura approach to answering those questions.

2. TASK-AWARE SYSTEMS

The central tenet of task awareness is that systems are used to carry out high-level activities of users: planning a trip, buying a car; communicating with others. In today's systems those activities and goals are implicit. Users must map them to computing systems by invoking specific applications (document editors, email programs, spreadsheets, etc.) on specific files.

In a task-aware system user tasks are made explicit. They encode user goals, and provide a placeholder to represent the quality attributes of the services used to perform those tasks. So, for example, for a particular task, in the presence of limited bandwidth, the user may be willing to live with a small video screen size, while in another task reducing the frame rate would be preferable.

Once such information is represented a self-managing system can in principle query the task to determine both when the system is behaving within an acceptable envelop for the task, and also can choose among alternative system reconfigurations when it is not.

However, a number of important research questions arise, and the way we answer them will strongly influence the way we look at and build task-aware systems:

– How do we represent a task? What encoding schemes can best be used to capture the user's requirements for system quality?

– How should we characterize the knowledge for mapping a user task to a system's configuration? As a user moves from task to task, different configurations will be appropriate, even for the same set of applications.

– Should we trigger an adaptation as soon as an opportunity for improvement is detected, or should we factor in how distracting the change will be to the user against how serious the fault is?

– Is the binary notion of fault enough, or do we need to come up with a measure of fault "hardness" – a continuum between "all is well," and "the system is down?"

Over the past five years we have been experimenting with various answers to these questions. Centered on a large ubiquitous computing research project, Project Aura [3], we have evolved a system that, in brief, addresses these questions as follows:

– We represent *a task as a set of services*, together with a set of quality attribute preferences expressed as multi-dimensional utility functions, possibly conditioned by context conditions.

– We define *a vocabulary for expressing requirements,* which delimits the space of requirements that the automatic reconfiguration can cover. The set of requirements for a particular task expresses which services are needed from the system, as well as the fidelity constraints that make the system adequate or inadequate for the task at hand. The required services are dynamically mapped to the available components and the fidelity constraints are mapped into resource-adaptation policies.

– We incorporate the notion of *cost of reconfiguration* into the evaluation of alternative reconfigurations. A high cost of reconfiguration will make the system highly stable, but frequently less optimal; a low cost of configuration will permit the system to change frequently, but may introduce more user distraction from reconfigurations.

– We invert the notion of fault by adopting an econometric-based notion of *system utility*: ranging from 0 (system is not useful at all for the current task) to 1 (system is totally appropriate for the current task). This enables an objective evaluation of configuration alternatives, regardless of the sources of change (either changes on the task/requirements or on the availability of resources and components).

We now describe the architecture of the system that permits such task-based self-adaptation, and elaborate on the above decisions.

3. THE AURA LAYERS

The starting point for understanding Aura is a layered view of its infrastructure together with an explanation of the roles of each layer with respect to task suspend-resume and dynamic adaptation. Table 1 summarizes the relevant terminology.

The infrastructure exploits knowledge about a user's tasks to automatically configure the environment on behalf of the user. First, the infrastructure needs to know *what* to configure for; that is, what the user needs from the environment in order to carry out his tasks. Second, the infrastructure needs to know *how* to best configure the environment: it needs mechanisms to optimally match the user's needs to the capabilities and resources in the environment.

In our architecture, each of these two subproblems is addressed by a distinct software layer: (1) the **Task Management** layer determines *what* the user needs from the environment at a specific time and location; and (2) the **Environment Management** layer determines *how* to best configure the environment to support the user's needs.

Table 1. Terminology.

task	An everyday activity such as preparing a presentation or writing a report. Carrying out a task may require obtaining several *services* from an *environment*, as well as accessing several *materials*.
environment	The set of *suppliers*, *materials* and *resources* accessible to a user at a particular location.
service	Either (a) a service type, such as printing, or (b) the occurrence of a service proper, such as printing a given document. For simplicity, we will let these meanings be inferred from context.
supplier	An application or device offering *services* – e.g. a printer.
material	An information asset such as a file or data stream.
capabilities	The set of *services* offered by a *supplier*, or by a whole *environment*.
resources	Are consumed by *suppliers* while providing *services*. Examples are: CPU cycles, memory, battery, bandwidth, etc.
context	Set of human-perceived attributes such as physical location, physical activity (sitting, walking…), or social activity (alone, giving a talk…).
user-level state of a task	User-observable set of properties in the *environment* that characterize the support for the task. Specifically, the set of *services* supporting the task, the user-level settings (preferences, options) associated with each of those services, the *materials* being worked on, user-interaction parameters (window size, cursors…), and the user's preferences with respect to quality of service tradeoffs.

Table 2 summarizes the roles of the software layers in the infrastructure. The top layer, Task Management (TM), captures knowledge about user tasks and associated intent. Such knowledge is used to coordinate the configuration of the environment upon changes in the user's task or context. For instance, when the user attempts to carry out a task in a new environment, TM coordinates access to all the information related to the user's task, and negotiates task support with Environment Management (EM). Task Management also monitors explicit indications from the user and events in the physical context surrounding the user. Upon getting indication that the user intends to suspend the current task or resume some other task, TM coordinates saving the user-level state of the suspended task and instantiates the resumed task, as appropriate. Task Management may also capture complex representations of user tasks (out of scope of this paper) including task decomposition (e.g., task A is composed of subtasks B and C), plans (e.g., C should be carried out after B), and context dependencies (e.g., the user can do B while sitting or walking, but not while driving).

Table 2. Summary of the software layers in the infrastructure.

layer	*mission*	*roles*
Task Management	***what*** does the user need	• monitor the user's task, context and preferences • map the user's task to needs for services in the environment • complex tasks: decomposition, plans, context dependencies
Environment Management	***how*** to best configure the environment	• monitor environment capabilities and resources • map service needs, and user-level state of tasks to available suppliers • ongoing optimization of the utility of the environment relative to the user's task
Env.	support the user's task	• monitor relevant resources • fine grain management of QoS/resource tradeoffs

The EM layer maintains abstract models of the environment. These models provide a level of indirection between the user's needs, expressed in environment-independent terms, and the concrete capabilities of each environment.

This indirection is used to address both heterogeneity and dynamic change in the environments. With respect to heterogeneity, when the user needs a service, such as speech recognition, EM will find and configure a supplier for that service among the ones available in the environment. With respect to dynamic change, the existence of explicit models of the capabilities in the environment enables automatic reasoning when those capabilities change dynamically. Environment Management adjusts such a mapping automatically in response to changes in the user's needs (adaptation initiated by TM), and changes in the environment's capabilities and resources (adaptation initiated by EM). In both cases adaptation is guided by the maximization of a *utility function* representing the user's preferences.

The Environment layer comprises the applications and devices that can be configured to support a user's task. Configuration issues aside, these suppliers interact with the user exactly as they would without the presence of the infrastructure. The infrastructure steps in only to automatically configure those suppliers on behalf of the user. The specific capabilities of each supplier are manipulated by EM, which acts as a translator for the environment-independent descriptions of user needs issued by TM.

By factoring models of user preferences and context out of individual applications, the infrastructure enables applications to apply the adaptation policies appropriate for each task. That knowledge is very hard to obtain at the application level, but once it is determined at the user level – by Task Management – it can easily be communicated to the applications supporting the user's task.

Each layer reacts to changes in user tasks and in the environment at a different granularity and time-scale. Task Management acts at a human perceived time-scale (minutes), evaluating the adequacy of sets of services to support the user's task. Environment Management acts at a time-scale of seconds, evaluating the adequacy of the mapping between the requested services and specific suppliers. Adaptive applications (fidelity-aware and context-aware) choose appropriate computation tactics at a time-scale of milliseconds. A detailed description of the architecture, including the formal specification of the interactions between the components in the layers defined above, is available in 0.

4. EXAMPLES OF SELF-ADAPTATION

To clarify how this design works, we illustrate how the infrastructure outlined in Section 3 handles a variety of examples of self-adaptation, ranging from traditional repair in reaction to faults, to reactions to positive changes in the environment, to reactions to changes in the user's task.

To set the stage, suppose that Fred is engaged in a conversation that requires real-time speech-to-speech translation. For that task, assume the Aura infrastructure has assembled three services: speech recognition, language translation, and speech synthesis. Initially both speech recognition and synthesis are running on Fred's handheld. To save resources on Fred's handheld, and since language translation is computationally intensive, but has very low demand on data-flow (the text representation of each utterance), the translation service is configured to run on a remote server.

Fault tolerance. Suppose now that there is loss of connectivity to the remote server, or equivalently, that there is a software crash that renders it unavailable. Live monitoring at the EM level detects that the supplier for language translation is lost. The EM looks for an alternative supplier for that service, e.g., translation software on Fred's handheld, activates it, and automatically reconfigures the service assembly.

Resource/fidelity-awareness. Computational resources in Fred's handheld are allocated by the EM among the services supporting Fred's task. For computing optimal resource allocation, the EM uses each supplier's spec sheet (relating fidelity levels with resource consumption), live monitoring of the available resources, and the user's preferences with respect to fidelity levels [7]. Suppose that during the social part of the conversation, Fred is fine with a less accurate translation, but response times should be snappy. The speech recognizer, as the main driver of the overall response time, gets proportionally more resources and uses faster, if less accurate, recognition algorithms. When the translation service is activated on Fred's handheld in response to the fault mentioned above, resources become scarcer for the three services. However, having the knowledge about Fred's preferences passed upon service activation, each supplier can react appropriately by shifting to computation strategies that save response times at the expense of accuracy [1].

Soft fault (negative delta). Each supplier issues periodic reports on the Quality of Service (QoS) actually being provided – in this example, response time and estimated accuracy of recognition/translation. Suppose that at some point during the conversation, Fred brings up his calendar to check his availability for a meeting. The suppliers for the speech-to-speech translation task, already stretched for resources, reduce their QoS below what Fred's preferences state as acceptable. The EM detects this soft fault, and replaces the speech recognizer by a lightweight component, that although unable to provide as high a QoS as the full-fledged version when resources are plentiful, performs better under sub-optimal resource availability.[1]

Soft fault (positive delta). Suppose that at some point, the language translation supplier running on the remote server becomes available again. The EM detects the availability of a new candidate to supply a service required by Fred's task, and compares the estimated utility of the candidate solution against the current one. If there is a clear benefit, the EM automatically reconfigures the service assembly. In calculating the benefit, the EM factors in a cost of change, which is also specified in the user's preferences associated with each service. This mechanism introduces hysteresis in the reconfiguration behavior, thus avoiding oscillation between closely competing solutions. See 0 for the formal details about of this mechanism.

Task QoS requirements change. Suppose that at some point Fred's conversation enters a technical core for which translation accuracy becomes more important than fast response times. The TM provides the mechanisms, if not to recognize the change automatically based on Fred's social context, at least to allow Fred to quickly indicate his new preferences; for instance, by choosing among a set of preference templates. The new preferences are distributed by the TM to the EM and all the suppliers supporting Fred's task. Given a new set of constraints, the EM evaluates the current solution against other candidates, reconfigures, if necessary, and determines the new optimal resource allocation. The suppliers that remain in the configuration, upon receiving the new preferences, change their computation strategies dynamically; e.g., by changing to strategies that offer better accuracy at the expense of response time.

Task suspend/resume. Suppose that after the conversation, Fred wants to resume writing one of his research papers. Again, the TM provides the mechanisms to detect, or for Fred to quickly indicate his change of task. Once the TM is aware that the conversation is over it coordinates with the suppliers for capturing the user-level state of the current task, if any, and with the EM to deactivate (and release the resources for) the current suppliers. The TM then analyses the description it saved the last time Fred worked on writing the paper, recognizes which services Fred was using and requests those from the EM. After the EM identifies the optimal supplier assignment, the TM interacts with those suppliers to automatically recover the user-level state where Fred left off. See 0 for a formal specification of such interactions.

Task service requirements change. Suppose that while writing his paper, Fred recognizes that it would be helpful to refer to a presentation he gave recently to his research group. The TM enables Fred to explicitly aggregate viewing the presentation to the ongoing task. As soon as a new service is recognized as part of the task, the TM requests an incremental update to the EM, which computes the optimal supplier and resource assignment for the new task definition, and automatically performs the required reconfigurations. Similarly, if Fred decides some service is no longer necessary for his task, he can let the TM know, and the corresponding (incremental) deactivations are propagated to the EM and suppliers. By keeping the TM up-to-date with respect to the requirements of his tasks, Fred benefits from both the automatic incremental reconfiguration of the environment, and from the ability to suspend/resume exactly the set of services that he considers relevant for each task.

[1] Additionally, the EM uses these periodic QoS reports to monitor the availability of the suppliers, in a heartbeat fashion.

5. CONCLUSION & FUTURE WORK

We have argued that an explicit representation of user tasks is a critical component for self-managed system, and outlined the way Project Aura has instantiated this concept in its layered architecture. The form of tasks that we capture in this research is relatively simple. Future work is needed to represent more complex user tasks with complex goal structures, ordering dependencies, and the capability of learning.

6. ACKNOWLEDGMENTS

This work was supported by the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298, the National Science Foundation under Grant ITR-0086003, and DARPA under grants N6601-99-02 and F30602-00-2-061.

We thank Mary Shaw and members of CMU's Coda and Aura projects for insights that enabled the architecture presented here to dovetail with research in ubiquitous and resource-aware systems.

7. REFERENCES

[1] Balan, R., Sousa, J.P., Satyanarayanan, M. Meeting the Software Engineering Challenges of Adaptive Mobile Applications. Tech. Report, CMU-CS-03-11, CMU, Pittsburgh, PA, 2003.

[2] Cheng, S.-W. et al. Software Architecture-based Adaptation for Pervasive Systems. *Proc of the International Conf. on Architecture of Computing Systems: Trends in Network and Pervasive Computing,* April 2002. Springer LNCS Vol. 2299, Schmeck, H., Ungerer, T., Wolf, L. (Eds), 2002.

[3] Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P. Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, April-June 2002.

[4] Garlan, D., Cheng, S.-W., Schmerl, B. Increasing System Dependability through Architecture-Based Self-repair. In *Architecting Dependable Systems*, R. de Lemos, C. Gacek, A. Romanovsky (Eds), Springer-Verlag, 2003.

[5] Georgiadis, I., Magee, J., Kramer, J. Self-Organising Software Architectures for Distributed Systems. *Proc. ACM SIGSOFT Wksp on Self-Healing Sys. (WOSS'02)*. Nov. 2002.

[6] Noble, B., et al. Agile Application-Aware Adaptation for Mobility. *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP'97)* October 1997. *Operating Systems Review* 31(5), ACM Press, 276-287.

[7] Poladian, V., Sousa, J.P., Garlan, D., Shaw, M. Dynamic Configuration of Resource-Aware Services. *Proceedings of the 26th International Conf. on Software Engineering* (*ICSE 2004*). May 2004. IEEE Computer Society, 604-613.

Submitted for publication.

[8] Sousa, J.P., Garlan, D. The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. Tech. Report, CMU-CS-03-183, CMU, Pittsburgh, PA, 2003.