Mobile Adaptive Tasks Guided by Resource Contracts

Peter Rigole Yolande Berbers
Tom Holvoet
K.U.Leuven, Department of Computer Science
Celestijnenaan 200A
3001 Leuven, Belgium
{peter.rigole, yolande.berbers, tom.holvoet}@cs.kuleuven.ac.be

ABSTRACT

This paper proposes a way to realize the idea of calm computing by adding a dynamic task model into the pervasive computing environment. This task model contains information about the actions to undertake to help a user realize his daily tasks. The task model's mapping onto a deployment plan guides an internal adaptation mechanism, which helps applications to evolve without causing user distraction. In addition, a foraging technique (relocation) is proposed that allows for expanding an application's computing space automatically whenever possible. This technique involves external adaptation mechanisms. Both adaptation mechanisms are driven by resource information and resource contracts that are negotiated between the middleware and the application components. This allows the middleware to do the adaptations automatically, realizing the idea of calm computing.

Categories and Subject Descriptors

D.2.8 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*

General Terms

Design

Keywords

pervasive computing, adaptation, foraging, contracts

1. INTRODUCTION

A pervasive computing system [1] is expected to commit its computing powers to support the daily tasks of its users. The intelligence of these systems hereby ensures that user distraction is reduced to a minimal level. This way, the idea of *The Disappearing Computer* or *Calm Computing* is realized. The reasoning that lies at the basis of this intelligent behavior processes the bulk of available context information to undertake appropriate supportive actions. These actions are performed by applications that are guided by a task model that reflects the user's current task. Since pervasive computing environments are envisioned to be widely supported by mobile devices held by the user or by small devices hidden in the fabric of the user's environment, resources are relatively small on each computing device. Therefore, distributed cooperation must be extensively exploited on these devices. The result should be a rich computing space in which user task support is provided with the resources that are needed to survive on.

In the pervasive computing space, *Cyber Foraging* [2] techniques combined with application adaptation mechanisms can be used to increase the capabilities of resource-limited devices. Such techniques allow applications to forage certain resources from its current environment. Memory, for example, could be foraged by instantiating code on a remote device and thus saving space on the local device. Also, memory could be released - and thus made available for other software entities - by removing application parts that are optional for achieving the current task. A sudden increase in memory on the other hand, e.g. when the user moves to a richer environment, expands the computing space which offers new capabilities to the applications. This expansion is reflected in the context information that is shared among the computing entities.

The only way to achieve flexible adaptive task-based applications and applications that are able to do resource foraging is to make applications inherently resource aware by design. Object- and component-oriented technologies are the state of the art in general purpose software technologies. They lack, however, several assets that are required to live by the pervasive computing paradigm: they are not easily adaptable and not resource aware.

Developing resource-aware applications involves introducing knowledge about the application's resource footprint in the application itself. However, reasoning about an application's resource footprint during the development process seems complex and few design methods support it. We believe that a contract-oriented approach in a component based development methodology can alleviate this burden. Furthermore, a run-time representation, validation and monitoring of resource contracts, combined with resource information spread through context, forms a strong basis of knowledge to guide foraging and adaptation mechanisms.

In the following sections, we first discuss how contracts

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
2nd Workshop on Middleware for Pervasive and Ad-Hoc Computing Toronto, Canada
Copyright 2004 ACM 1-58113-951-9 ...$5.00.

can be combined with component technology, followed by an approach to supports contracts, their validation and monitoring at runtime. Thereafter, we describe how task-driven computing can guide an application's live-cycle. Adaptation mechanisms and their use are discussed in the next section and in conclusion we formulate future work and a vision for future context driven component applications.

2. RESOURCE-AWARE COMPONENT TECHNOLOGY

In a world where people are guided by intelligence spread in ambient computing spaces, software will live its own life and settle there where it can serve its user and where it can harvest the resources it needs to survive. For the sake of self-manageability and for the ease of software developers, such software will not come in large blocks of code, but rather in fine-grained cooperative entities. The current state of the art describes component-based software as a mature extension of object-oriented software development. Besides the advantage of the addition of a level of abstraction, improvement of code reusability and many other, components are also very well suited as a delineated entity for performing code analysis. For example, resource-awareness could be established per component.

In our view, resource-awareness is realized through a description of resource properties by means of a resource declaration for each component. These can be found by analyzing each component's resource behavior:

> **Definition 1 (Resource Declaration).** *A resource declaration is a parameterized description of a certain type of resource a software entity requires when it deploys its functionalities.*

Developing software using the contract paradigm means that designers have to think carefully about several functional and non-functional properties during the development process. It is an essential part of it. The properties of the resources required by some software entities are outlined in a resource declaration. Each resource declaration type has a form that has to be filled in by the designer. This form may be filled in using static numbers (when the property is known precisely in advance) or using a parameterized formula. These parameters are filled in at runtime and are based on initial configuration data of the software entity involved. It may be obvious that defining a resource declaration involves a thorough knowledge of the internals of the software entity and often, testing, measuring and statistical analysis can be required to acquire the data needed in the declaration.

When resource awareness is coupled to the building block of a fine-grained application, the resource-aware nature of the application emerges from its composing components. The unit of composition becomes the unit for resource descriptions. However, design time resource knowledge must be reflected in a runtime equivalent for realizing the real purpose of resource awareness: automated application management. Resource declarations combined with environmental resource availability information, which is part of the context information, form a starting point to do resource foraging. In the next section we propose a mechanisms for making resource commitments effective in a runtime environment.

3. A CONTRACT BASED APPROACH

The purpose of resource declarations is to settle agreements between several software entities about the use of certain resources. These agreements can be seen as a contract between software requesting resources (the application) and the software providing the resources (the middleware). Applications that live by themselves and manage their components (allocating, deallocating and configuring them) at runtime have to compose resource requests based on the combined resource declarations of their composing components. We define these resource request, which we call resource contract proposals, as follows:

> **Definition 2 (Resource Contract Proposal).** *A resource contract proposal contains an agreement on intended resource use by one or several software parties.*

For example, a bandwidth contract proposal may be related to a connector connecting two components which requires an agreement based on the bandwidth descriptions of both components. The information from a set of resource descriptions is combined into a contract proposal. This "combining" of information is the task of the application core (e.g. a central application manager component) and must be done with concrete resource descriptions. So, the parameters from parameterized resource descriptions must have a concrete value (e.g. from the component's configuration).

As soon as a contract proposal is compiled, it may be submitted to the middleware where a validation procedure is initiated. Usually, a set of related contracts is submitted at once because an application typically manages multiple component at the time. Rejected contract proposals are returned to the application core and a cause (e.g. the clause that caused the rejection) for the rejection is given. This cause can then be used to compile a new resource contract proposal and start a renegotiation. The result of a successful validation is a signed resource contract:

> **Definition 3 (Signed Resource Contract).** *A resource contract proposal can become a signed resource contract when the middleware system accepts the contract and signs it. A signed resource contract means that the parties involved must adhere to the agreements in the contract and that the middleware system must provide the resources as described in the contract.*

On acceptance, all contracts of a resource contract set become a signed entity in the runtime system. From that moment on, both the parties involved as the middleware system have a responsibility in respecting the contract. The middleware system may use monitoring mechanisms to validate whether all parties involved in signed contracts are behaving accordingly. Parties violating contracts can be disciplined in several ways. Depending on the middleware configuration, a small violation may be ignored, whereas recurring or more serious violations may cause the rejection of a signed contract. The middleware may allow the managing application to start a contract renegotiation. As long as no contract is signed for the allocation of a certain resource, the software entities involved have no more rights to allocate that type of resource.

In our opinion, the symbiosis of components as building blocks for applications and resource contracts will result in a software architecture that is extremely suited for

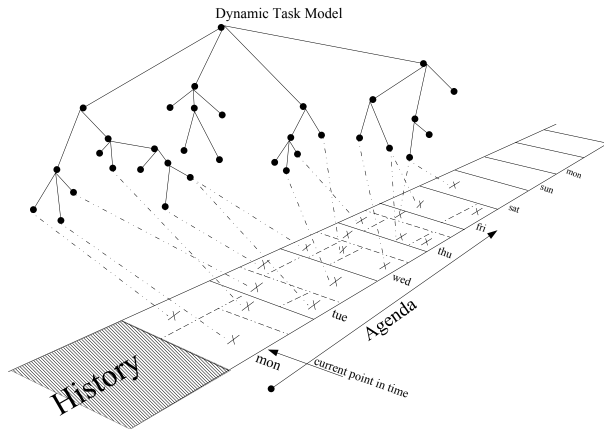

Figure 1: Dynamic task scheduling

enabling flexible pervasive computing solutions. The presence of context information regarding resource availability is paramount, since the automated behavior of pervasive applications will be driven partly by resource contracts.

4. TASK-DRIVEN COMPUTING

Software envisioned by pervasive computing should support the user and his tasks while avoiding user-device interaction as much as possible. This requires the software to be intelligent in the sense that its proactive behavior tries to avoid superfluous distraction of the user. And therefore, it needs detailed knowledge about the goals of the user's task.

Task models [3] can provide useful information about the different tasks and the order in which they need to be executed. They are usually represented in a tree structure in which a task is split up in several subtasks. Traditionally, task models are defined at design-time to specify the steps that have to be executed to realize a goal. Such models, however, do not represent daily real world situations. In reality, there is no seamless course of events that helps a user to realize a goal. Many seams are at the origin of the unpredictable nature of our daily lives. Pervasive systems should know how to deal with this unpredictable factor.

Dynamic task models could take real world events into account by interpreting context information. This would allow for the content of a task to evolve according to the user's context. In addition, the task schedule has to be continuously adjusted to reflect the agenda of the user. Figure 1 illustrates this idea. The dynamic taks model is mapped to the agenda of the user, but this mapping may evolve whenever new evolutions in the course of events require it.

As a task model defines "what should be done" and "when", it is a good starting point for composing a deployment plan. Such plan specifies which software components have to be loaded and how they should be interconnected. This plan should follow the evolutions in the schedule of the task model so that the right software is available at all times. This schedule should aim at achieving continuity from the user's viewpoint, even if task support suddenly fails. Task centered graceful degradation involves smoothly switching to another task if the current task can not be continued (e.g. a required resource drops away, the task lacks certain input, . . .).

The deployment plan specifies components that are required to realize a task and also optional components. These optional components make the task easier, but the task does not fail if they are not available. It is up to the middleware to decide whether to instantiate them or not (e.g. based on resource availability). An example is a spellchecker helping a user to write a tekst. Though it is not required, it may help the user a lot. The spellchecker may be instantiated in a resource-rich environment and left out in a constrained computing environment. Figure 2 illustrates the four views: the task model has a mapping onto deployment plan, the deployment plan leads to resource contracts when the request for instantiation is processed, and on acceptance of all contracts, the composition is instantiated (representing the application that is used to perform the user task).

5. ADAPTATION

In a pervasive computing environment resource management is very important since the computing space consists of a mixture of mobile devices and devices embedded in our daily environment. Optimized resource provision can be realized in a distributed way by using foraging and adaptation mechanisms. Considering the nature of the adaptation, we distinguish two types of adaptations: *external adaptations* (adaptations driven by forces external to the application) and *internal adaptations* (adaptations driven by the application itself).

5.1 External Adaptations

External adaptations are performed by interfering with the runtime instance of an application without having effects on the application's correct[1] execution. In other words, the adaptation should be transparent for the application. Usually only middleware can execute this type of adaptation since only middleware has access deep down to the application's internal structures.

For a component-oriented application, its nature of fine-grained delineated software entities provides what is needed for doing application adaptations. In these applications, there are two commonly used external adaptations: *component reconfiguration* and *component replacement*. Reconfiguration of components is possible when the components provide a special interface that allows for certain changes in the components configuration (such as changing a buffer size, changing a frame rate, . . .). Examples in literature are [4, 5, 6]. Component replacement is a more general used technique (e.g. live update mechanisms). One component is swapped by another in an atomic action. It transfers the component's internal state and execution to the new component without stopping the application. The goal in both cases of the adaptation is to change the behavior of the application with respect to its resource use. This means that contracts will be renegotiated resulting in either an increase or decrease in the use of certain resources. As a consequence, the functional behavior of the application changes as well, but without crippling its ability to support the user's task.

A third type of external adaptation is *component relocation*. This is a foraging technique that forages external resources such as memory and CPU cycles by moving a component from one host to another. By instantiating the component on the other host and removing it on the local host,

[1]According to its contracts

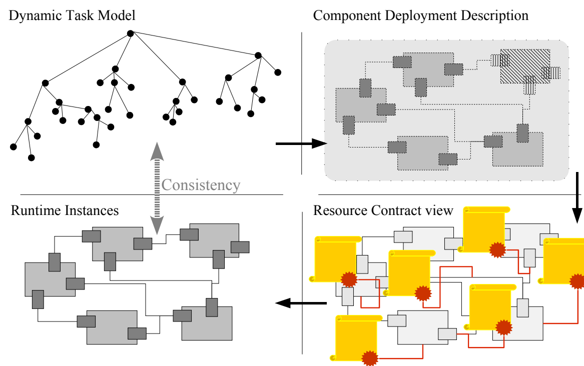

Figure 2: Task model is consistent with runtime component instances

resources are freed locally and consumed remotely instead. An disadvantage of this technique is that it usually increases bandwidth use between the hosts involved and the tradeoff must be considered.

Component relocation involves a resource allocation problem, for which several constraint-solving solutions have already been worked out. An appropriate solution would be one that works in a pervasive context. This means that the algorithms working out a new component configuration should find a solution in a reasonable time. One approach could be using an incremental constraint solver [7], because relocation means working out a new (and better) solution starting from a given solution (configuration). The information the relocation algorithm needs can be acquired two sources. On one hand from context information (remote resource availability) and on the other hand from the resource contract view[2] of a component composition (see figure 2).

5.2 Internal Adaptations

Internal adaptations are adaptations performed by the application itself. In a pervasive context, this type of adaptation is mainly driven by the user's personal context, and especially the context that is related to the progress of the user's current task. The goal of the adaptations remains the same: handling resources economically while still supporting the user as good as possible. The key idea here is that user support changes through time (because of the evolving task of the user) and thus the application does not need to offer all of its functionality at the time. The evolution of the application through time is guided by context and knowledge about the user's task. This means that parts of an application that are not needed at the moment can be removed, which saves resources. Optional components, for example, can be instantiated only when a user is in a resource-rich environment.

One approach to solve task management in pervasive environments has been described in [8]. This paper focuses on guaranteeing task availability and proposes a contract-driven solution involving a design methodology supported by middleware that allows for adaptable task behavior at runtime.

[2]which is a runtime instance of a resource meta model

5.3 Pervasive Adaptations

Coordination between internal and external adaptation in pervasive systems is an interesting challenge that could result in extremely flexible computing spaces where robust user support is the ultimate goal. Several schemes for cooperation of external and internal adaptations are possible. External forces may request an internal adaptation (e.g. middleware wants to instantiate a additional application and needs more free resources), internal adaptations may be realized in combination with an external adaptation (e.g. more functionality is added by the application but the middleware can only instantiate the component remotely).

6. IN CONCLUSION

In pervasive computing systems the tradeoff between functionality and resource use must be considered whenever resource availability changes. This states the importance of *resource information* being available as *context*. Therefore, middleware must be able to discover, find, process and interpret resource information to perform the described mechanisms. *Resource contracts* are the means for managing resources in a predictable way in a distributed environment. *Component-oriented* approaches deliver applications with a fine-grained composition structure which ensures the flexible nature required for the presented *adaptation mechanisms*. A *task-centered* approach ensures that applications move along with the user's activities.

7. REFERENCES

[1] Mark Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–10, Sept 1991.

[2] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive computing*, 1(2):22–31, April-June 2002.

[3] Jan Van den Bergh and Karin Coninx. Contextual concurtasktrees: Integrating dynamic contexts in task based design. In *Second IEEE Conference on Pervasive Computing and Communications Workshops*, pages 13–17, Orlando, USA, 14 Mar 2004.

[4] Vanegas R, Zinky JA, and Loyall JP. Quo's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, England, Sept 1998.

[5] Loyall JP, Schantz RE, Zinky JA, and Bakken DE. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing*, Kyoto, Japan, 20-22 Apr 1998.

[6] David Pierre-Charles and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In *Proceedings DAIS'04*, volume 2893 of *LNCS*, pages 1–14, Paris, France, Nov 19-21 2003.

[7] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer Human Interaction*, 8(4):267–306, December 2001.

[8] Zhenyu Wang and David Garlan. Task-driven computing. Technical Report CMU-CS-00-154, School of Computer Science, CMU, Pittsburgh, USA, May 2000.