AMBIENT COMPUTING USING COMPONENT RESOURCE CONTRACTS

Andrew Wils, Peter Rigole, Yolande Berbers, Karel De Vlaminck
Department of Computer Science
K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
email: { andrew | peterri | yolande | kdv} @cs.kuleuven.ac.be

ABSTRACT
This paper introduces a contract-driven resource management process and supporting middleware framework. We aim at supporting mobile computing tasks as they appear in the areas of Ambient Intelligence (AmI). As available resources in AmI environments fluctuate frequently, the resource usage of AmI tasks must be extremely flexible. The presented component based methodology permits an optimal flexibility in resource distribution and allocation through the use of resource contracts.

User tasks specify their QoS needs in parameterized contract propositions that are used as the basis for a negotiation process. Once a contract has been validated and signed by the middleware, it can be monitored. The supporting decision logic is built around a Java-based expert system. Negotiation and monitoring rules detect misbehaving tasks and environmental changes. Renegotiations result in task adaptations or more drastic measures, such as partial task relocation.

KEY WORDS
ambient intelligence, component based development, embedded systems, resource management, ...

1 Introduction

New computer paradigms such as Ambient Intelligence (AmI) are rapidly extending middleware research. They require user applications to run on a host of devices and strive for the maximum possible Quality of Service (QoS). Hence, dynamic adaptation in all its facets will become a distinguishing feature of AmI applications. This paper presents a mechanism and supporting framework to manage and reserve resources at the local level. The main purpose of this is to maintain a minimum QoS when available resources are scarce. Distributed middleware can also leverage this technology to perform resource-based software relocation decisions.

Resources are allocated to applications after a negotiation process. Section 2 will elaborate on this approach that introduces resource contracts as first class objects. Related work is also discussed in this section. We will further illustrate the approach with a small example in section 3: a slideshow presenter.

The proof-of-concept framework we implemented to support the approach is called CRUMB (Contract-based Resource Monitor and Broker) and is presented in section 4. We will especially focus on the use of an expert system to validate and monitor various resource constraints. Findings of our results and experiences are further explained in section 5. Finally, we address future work in section 6 and conclude our paper in section 7.

2 Resource Contracts

Software envisioned by Ambient Intelligence (*AmI software*) supports a user and his tasks while avoiding user-device interaction as much as possible. To ensure its survival on mobile devices, such software must be extremely robust and flexible in supporting these user tasks because of the frequent occurrence of location changes. As a consequence, two major responsibilities of AmI software are:

task management: continuously supporting the computing functionality expected by the user according to the current task, avoiding user distraction as much as possible

task availability: continuously supporting the user's computing space, hiding variations in the availability of external resources as much as possible

An approach to solve *task management* in pervasive environments has been described in [11]. 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.

Dealing with sudden changes in the availability of resources and ensuring that applications survive these fluctuations implicates that certain knowledge must be at hand:

- resource requirements of different parts of the application
- current resource use of different parts of the application
- resource availability on the host system
- resource availability on other hosts in the environment

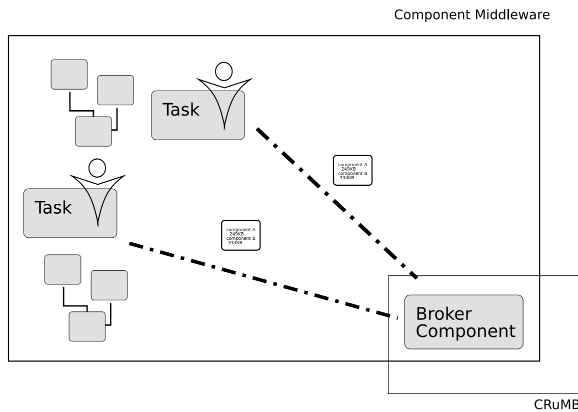

Figure 1. Resource Aware Tasks negotiating resource contracts

Task management will use this information to continuously deal with changes in the availability of resources, through negotiation, renegotiation and application adaptation strategies. The remainder of this section will introduce supporting concepts and processes to support these mechanisms.

2.1 Contractual Component Based Programming

The nature of AmI software suggests the use of software building blocks that are more course-grained than software objects and more easily inter-connectible than web services. Component Based Development (CBD) [8] offers architectural and runtime support for programming in the large.

We use the contract paradigm to map and adapt component specifications to the underlying middleware in order to connect their interfaces. Contractual programming, widely used in object-oriented programming, proved even more useful in CBD, and was extended to cover more levels of interoperation. Beugnard et al. introduced a specific CBD contract notion containing the following levels [2]:

syntactic specification: signatures of messages and parameters

semantic specification: description of component invariants, states, pre- and postconditions

synchronization specification: description of the synchronization protocol by means of a state chart or a sequence chart

QoS specification: non-functional properties related to the interaction, such as message timing, resource usage, result accuracy and reliability.

Seamless component interoperation requires at least a partial matching of all contract levels.

This paper focuses on a concrete solution to match the QoS level of a component specification. Typical AmI scenario's [9] require software components to migrate and adapt to changing environments. In order to maintain a minimum QoS and take use of available resource where appropriate, components and their enabling middleware must be aware of their resource usage.

To facilitate the reuse of a component in a new or changed environment, we adapt its QoS by introducing parameterized contracts. These resource contracts become first class entities that are effectively and continuously subject to negotiation through a resource broker (see figure 1).

2.2 Contract Negotiation

This section explains the notion of multi-party contracts and how they are agreed upon.

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.*

An example set of resource declarations can be found in table 1.

Developing software using the contract paradigm means that designers have to think carefully about several functional and non-functional properties during the development process. The properties of the resources required by some software entities are outlined in a resource declaration. Each resource contract type has a resource declaration 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.

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

At runtime, resource declarations are the basis for creating a resource contract proposal, only the missing parameters are to be filled in. If more than one party is involved in creating a contract proposal based on their resource declarations, a negotiation process may be started to reach an agreement. As soon as the proposal is agreed upon, it may be submitted to the middleware where a validation procedure is initiated. Usually, a set of related contracts is submitted at once. Indeed, a task typically involves more than

one component and/or type of resource. Rejected contract proposals are returned to the task manager of the application 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.

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. The middleware system may use monitoring mechanisms to validate whether all parties involved in signed contracts are behaving accordingly. 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 task 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.

Resource contracts alleviate the programmer of many aspects involving the implementation of adaptive resource-aware components. Moreover, a supporting run-time framework can handle important decision making tasks, such as:

- automatic selection of an appropriate QoS level (based on user expectancy)
- decisions involving global resource usage and allocation (aggressive or not)
- trade-offs between robustness and speed

Removing most of the decision making process from the application software to supporting software improves resource sharing and stability of the overall system. Of course, this functionality comes with a penalty: as the granularity of resource monitoring becomes finer, the resource use of the run-time system itself will increase. The use of trusted software components may remove the need for extensive resource monitoring.

The primary goal of CRUMB is to control user tasks as they have been defined in the field of Ambient Intelligence and context awareness. It does this by striving for the greatest application fidelity while maintaining a strong control on each individual component.

2.3 Related Work

Resource adaptation is a relatively new research topic. Most resource-constrained devices have a well defined and fixed functionality for which resource analysis can be performed at design-time[1]. In fact, real-time constraints and stability issues make industry very reluctant to embrace any form of dynamic adaptation. The advent of multi-functional mobile computing devices and modern computing paradigms resulted in the incorporation of resource awareness in a number of existing middleware solutions. Although resource-aware middleware such as the KaffeOS VM [1] offers basic timing and/or memory accounting support, it generally lacks a general negotiation mechanism that can enforce dynamic adaptation. Our proposed framework offers a higher level of abstraction to programmers of resource-aware software. In this way, our component-based approach is a form of architectural resource adaptation (as described in [3]) with the added advantage of a powerful negotiation mechanism.

A different approach is taken by the Quality Objects framework [4]. QuO explicitly and rightfully separates program functionality from QoS aspects. Adaptability of an application is modeled by the definition of a number of fixed regions for which the programmer defines a number of transitions. Rather than finding an optimal general resource distribution, application specific logic (QoS delegates) tries to come to a suitable QoS level.

Finally, the JAMUS/RAJE framework [7] also offers resource contracts that can be (re)negotiated. JAMUS considers any component a potential threat, but it is unclear how each component is made to request a contract before using a resource. JAMUS does not focus so much on a certain software architecture: it expects general object-oriented applications to use their framework. This makes it difficult to demand software entities to adhere to the contract rules. CRUMB is bound to a component architecture and knows exactly where to expect contracts and how it can postpone or stop a component's execution if necessary. These aspects are part of the architecture and required by design. In addition, CRUMB ameliorates contract expressiveness by allowing multiple quality attributes and multi-party contracts.

3 A Resource-Aware Slideshow

To illustrate our approach, we present an example resource-aware slide presentation task. Figure 2 shows the basic component instance diagram of the task.

It consists of the following component instances:

The Domain Controller manages the actual presentation, and can send out individual slides.

The Slide Renderer processes slide models and produces an annotated graphical representation of the slide. These annotations include transition and animation information.

[1] as opposed to run-time

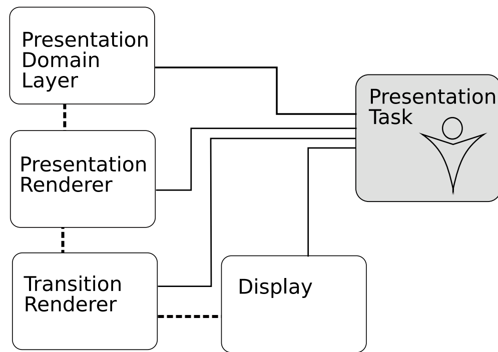

Figure 2. Basic component diagram for the presentation task. Solid lines depict control flow, stroked lines depict data flow.

The Transition Renderer is able to distill animation sequences out of a number of graphical slide representations.

A subset of the involved resource descriptions involving memory are shown in table 1.

The presentation task will first submit a set of contracts that reflects all desired functionality e.g. displaying high-resolution fluidly animated slides. Depending on the used policy and underlying device configuration, this contract may be accepted or not. Which contract gets selected, will depend on the ruling policy of the resource broker. Although it seems logical to favor higher performing configurations, the broker may prefer not to allocate resources aggressively. For example, a heuristic policy could decide to reject a contract when its aggregated memory needs are more than the currently available memory or when the average memory need is more than 20% above the global contract average.

Table 2 lists a number of possible contract sets if only maximum memory needs are taken into account. Note that a change from contract set 1 to set 2 will cause a reconfiguration of the Animation Renderer, whereas contract set 4 does not require the Animation Renderer at all.

4 System support

4.1 The CRUMB Architecture

The CRUMB resource contract management architecture is available as an extension module for the DRACO [10] middleware system (also see section 5. CRUMB's functionality is encapsulated in several interacting submodules, as depicted in figure 3.

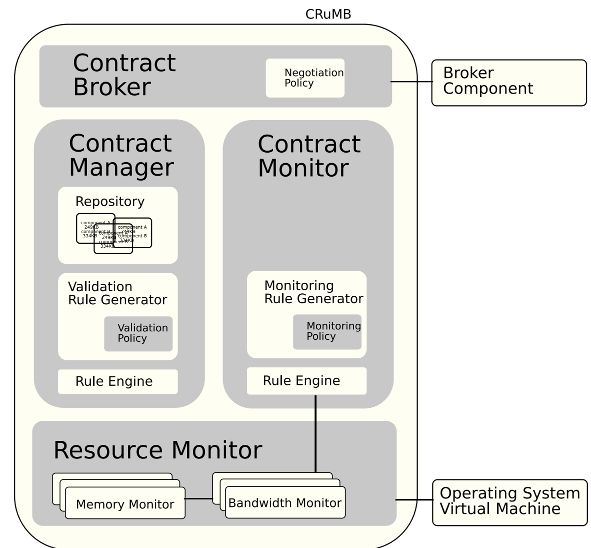

Figure 3. CRUMB General Architecture

Resource Monitor: The resource monitor performs the essential task of monitoring resource availability and resource consumption. By reifying the monitored information, it is made available to and understandable by the other subcomponents.

Contract Manager: The contract manager has the following responsibilities:

1. keeping account of all signed contracts and managing contract removals.
2. checking the feasibility of resource contract proposals and signing them when their feasibility gets positively evaluated.
3. negotiating about contract proposals by interacting with the contract broker.
4. dealing with contract violations reported by the contract monitor.

Contract Broker: As the CRUMB access point, the contract broker contains the negotiation logic for settling mutual agreements between tasks and CRUMB regarding certain contracts.

Contract Monitor: The optional contract monitor validates the use of resources (monitored by the Resource Monitor) against the established contracts. Whenever the use of resources exceeds the usage profile that was agreed upon, a violation is reported to the contract manager.

4.2 Rules for Resources

Figure 3 shows the internals of the Contract Manager and the Contract Monitor. Both use a rule engine of an expert system to perform their vital functions.

Component Instance	Average Memory	Maximum Memory	Max 95th percentile	Min 5th percentile
Domain Controller	274KB	1057KB	459KB	139KB
Slide Renderer	474KB	3247KB	1634KB	639KB
Transition Renderer	5240KB	15021KB	9459KB	7139KB

Table 1. Memory Profiles of the example component instances

Contract Set	Object Animation	Transition Animation	High Resolution
set 1	yes	yes	yes
set 2	yes	no	yes
set 3	yes	no	no
set 4	no	no	yes

Table 2. Example contract sets

Configuration	Total Memory	Used	Contractually Reserved	Free Memory	Possible Contract Sets
Laptop	256MB	68MB	78MB	178MB	1,2,3,4
PDA	32MB	25MB	28MB	4MB	4
Cellphone	2MB	512KB	768KB	1280KB	

Table 3. Example device configurations

The Contract Manager receives Contract Proposals from the Contract Broker and delegates them to the appropriate rule generator. Generated rules use previously submitted contract facts and policy rules to decide whether to sign a set of contract proposals. If a contract is signed, the necessary contract facts are generated for use in subsequent validation procedures. The Contract Monitor is notified of successfully signed contracts so it can produce its own monitoring rules, again according to the ruling policy. As an example, listing 1 triggers a violation if the associated memory bean reports a memory usage that is higher than 512KB.

To deal with renegotiation (requiring changes in existing contracts), contract rules and facts can either be reinserted, or made to reference dynamic contract details objects.

5 First Results

We implemented a proof-of-concept prototype of CRUMB in Java. Though Java has thusfar seldom been considered as an option for embedded software development, we believe it will play a critical role in the development of computing systems for Ambient Intelligence. We chose Jess [6] (which was based on Clips [5]) as expert system, because of its integration with Java.

A non-invasive memory monitoring technique using Java's resource API approximately slowed down the system with 10%. Moreover, the garbage collection mechanism makes profiling data very unreliable if it is not sampled after a garbage collection. That is why we feel memory monitoring would certainly benefit from VM support. Besides our work on memory contracts and memory monitoring mechanisms, we also designed a port-based bandwidth monitoring system and corresponding bandwidth contracts.

6 Future Work

To prove that resource contracts are beneficial to the general performance of a system, we will focus on enhancing existing memory monitoring support. Additionally, we are working on incorporating timing contracts into CRUMB to easily specify and monitor (periodic) deadlines on a group of components.

7 Conclusion

New computer paradigms shift towards the use of various applications on a plethora of embedded devices. We presented an approach to ensure basic QoS and deal with resource management on the local level. This can be used to implement fine-grained as well as larger architectural run-

```
(defrule memoryContractCompA
        (MemoryBean (name beanCompA) (currentMemory ?curr&:(> ?curr 512KB)) )
          =>
        (call (fetch contractMonitor) violationAlert memoryContractCompA)
)
```

Listing 1. Example of a memory monitoring rule

time adaptations.

To apply for resources, tasks need to get their contract propositions signed. Additional contract monitoring is used to adapt wrongly behaving components. The supporting framework uses an expert system to implement monitoring and validation of resource contracts. Although we do not consider an expert system to be generally useful for contract validation, it proved easy and fairly light for monitoring purposes.

References

[1] Godmar Back, Wilson C Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[2] A. Beugnard, J.M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer IEEE*, 32(7):38–45, 1999.

[3] Shang-Wen Cheng, David Garlan, Bradley Schmerl, Joäo Pedro Sousa, Bridget Spitznagel, Peter Steenkiste, and Ningning Hu. Software architecture-based adaptation for pervasive systems. In *International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing*. LNCS, April 2002.

[4] JP Loyall, RE Schantz, JA Zinky, and DE Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, Kyoto, April 1998.

[5] NASA. Clips: C language integrated production system, 2003. http://www.ghgcorp.com/clips/CLIPS.html.

[6] Sandia National Laboratories. Jess: the rule engine for the Java platform, 2004. http://herzberg.ca.sandia.gov/jess/.

[7] Nicolas Le Sommer. A contract-based approach of resource management in informations systems. In *Proceedings of 9th International Conference on Object-Oriented Information Systems (OOIS 2003)*, Genève, September 2003. LNCS.

[8] C. Szyperski. *Component Software: Beyond Objectoriented Programming*. Addison-Wesley, New York, 1998.

[9] the IST Advisory Group. Istag scenario's for ambient intelligence in 2010, 2000.

[10] Yves Vandewoude, Peter Rigole, and David Urting. Draco: an adaptive runtime environment for components. Appendix of the EMPRESS deliverable for Run-time Evolution and Dynamic (Re)configuration of Components, 2003.

[11] Zhenyu Wang and David Garlan. Task-driven computing. Technical Report CMU-CS-00-154, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15217, USA, May 2000.