

Flexible Component Contracts for Local Resource Awareness*

Andrew Wils, Joris Gorinsek, Stefan Van Baelen, Yolande Berbers, Karel De Vlamincx
Department of Computer Science

K.U.Leuven

Celestijnenlaan 200A

3001 Leuven, Belgium

{ andrew | jorissg | stefanv | yolande | kdvd } @cs.kuleuven.ac.be

April 25, 2003

Abstract

Ubiquitous and pervasive computing paradigms tend to shift towards open service-oriented platforms, in which carelessly written services can undermine the general stability of the whole system. We propose a component-contract based approach to solve these challenges. Resource contracts, combined with an intelligent resource broker and monitoring system, will efficiently deal with the local resource aspects of dynamically reconfigurable component systems.

1 Introduction

Modern software, and especially software that is to be running on future, “ubiquitous computing” systems, is often required to reconfigure, adapt and update (in short, to *evolve*). Multimedia services will run side by side with other, less resource-hungry, yet equally important services. Unfortunately, the limited resource capacities of embedded systems involving CPU power, memory size etc. in future hardware platforms will continue to exist, forcing developers of embedded software to spare resources. As we will further demonstrate in this paper, resource awareness is a necessary property of robust, yet flexible embedded system software.

In sections 2 and 3 we will outline the concept of resource awareness and differentiate between the distributed and local level. The paper continues in 4 with the basic concepts we will use to reason about resource awareness: components and contracts. Using these concepts, we will formulate basic resource declarations in sections 5 and 6. Before concluding the paper (section 9), basic system support and state of the art is discussed in sections 7 and 8, respectively.

*The described work is part of the EUREKA-ITEA projects DESS and EMPRESS, and partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders)

2 Resource Awareness

The upcoming ideas around ubiquitous or pervasive computing systems have started quite some research activities, among them those involving supporting “middleware systems”. A hardware device will no longer be designed to handle a single task, yet serve as a host for a myriad of services. Think of today’s PDA’s that are clearly evolving into multi-functional, multi-medial devices.

As the hardware becomes ubiquitous, users want to transfer their services and tasks easily between devices. This makes services have a very turbulent and often short life-cycle, placing heavy demands regarding flexibility on themselves and the supporting software.

Throughout the paper, we will call one particular setup of services on a hardware device a *service configuration*. Each service typically has need for a certain set of resources, where the deadline of the delivery is often of great importance. Thus a resource allocation must be found to satisfy all the needs of a particular service configuration. Typical requirements involve timing, memory and bandwidth constraints, as well as storage capacity and performance, power capacity and cost of resources (e.g. bandwidth). In spite of the continuous increase in performance of modern embedded hardware¹, many kinds of services tend to demand a considerable (and equally increasing) amount of the available resources on those devices.

As users move from place to place, they also switch from device to device and environment to environment. Thus, the service configurations on these devices change, as well as the resources available to each service. This can lead to service configurations that do not work as expected because of resource shortage. Clearly, the violation of important timing constraints, the complete saturation of the network link, or a premature depletion of battery power is unacceptable to the user. To avoid

¹Unfortunately, industry is struggling with the increased complexity of the design process and the problem of increasing power consumption.

this, the system software on a device needs to be aware of the minimum requirements of all services, and the easiest way to obtain this information is to ask the services. Software needs to be aware - in an abstract way - of the changing limitations of the underlying system.

3 Global and local Resource Awareness

The set of resource constraints formed by the services that a particular user (or perhaps a group of users) needs, represents an advanced distributed resource scheduling problem.

On the *distributed* level, an intelligent load balancing algorithm can decide on a coarse-grained distribution of all required services. The algorithm will have to take into account service dependencies, the preferred location, line of sight for displaying results, free resources of each platform, etc. If it is clear that some services will not be able to meet their resource constraints, *cyber foraging* can offer a solution. Cyber foraging assumes that non-mobile processing power will become ubiquitous as well. The idea is to make use of -untrusted- resources available in the environment.

The reason why this distributed intelligence is not enough is twofold. On the one hand, the distributed level works too coarse-grained. Also, its flexibility is limited: it is often not opportunistic to interrupt a service and move it elsewhere. Because of this, when a distributed allocation has been made, the local nodes will still have to deal with small fluctuations in services and resources.

We propose a *local* resource aware system that takes into account the individual QoS levels each service can function in. This involves the principle of *graceful degradation*[5]. Often, the resource usage of a service is not entirely fixed. Many resource intensive services could function even better, were they to be given more resources. The inverse of this statement also holds, e.g. a video streaming application can reduce its framerate, a maintenance utility can run temporarily in the background, or it could postpone its execution. Depending on the possibilities and relative importance of all services, the system can decide on a resource profile for each service and allocate resources accordingly. The aim is to optimize the resource allocation to provide a maximum user satisfaction.

Let us look at a simple, yet sufficiently realistic example, depicted in figure 1. Imagine a user carrying a portable MP3-player that is equipped with a small 3 inch color display. Its primary use is to download, store, stream and decode music. Suppose the user enters a train station building, while listening to a song he is streaming from his home (Timestamp 1). The information server asks the player (and, of course, its user) if it is allowed to

install a service to show the user around and to order tickets. This service puts some constraints on the CPU to ensure the smooth rendering of pages, maps etc. Meanwhile, the device gains access to the wireless LAN network of the station. The increase in bandwidth causes the music service to switch to a different, less CPU-hungry decoding algorithm. The middleware also decides that the -now minimized- graphic analyzer could cut back on its CPU requirements. This frees enough resources to let our user get around the building and order his tickets comfortably using his own trusted device (Timestamp 2). Should the device not be powerful enough, it could direct the user to a more suitable station terminal.

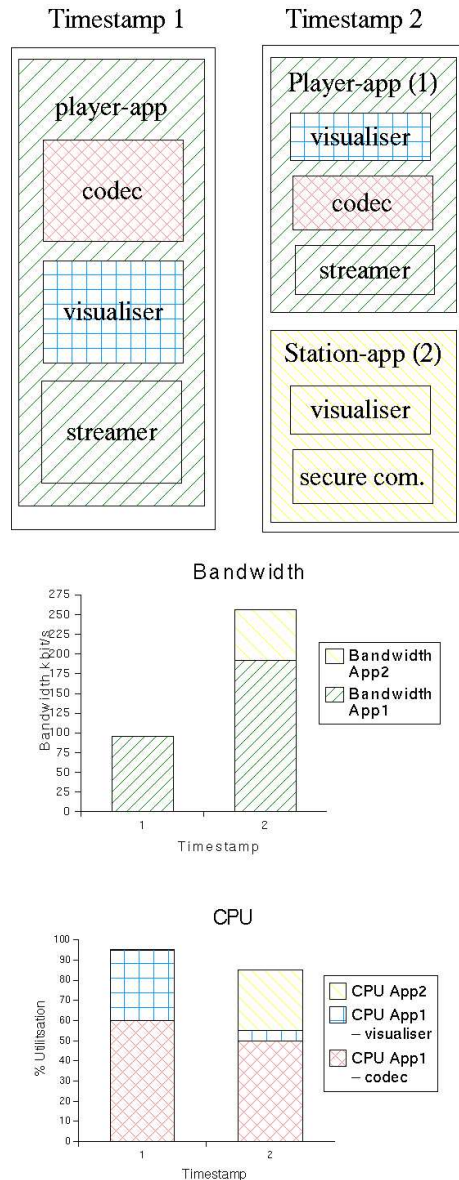


Figure 1: A simple example of changing resource allocations. Two service configurations are shown, along with their respective bandwidth and CPU usage.

4 Components and Contracts

One can try and prove at design-time that for a given service configuration and resource allocation all services behave within expected parameters. When the nature of the device is such that one cannot predict those configurations, or when finding a proof is too time-consuming or difficult, a supporting runtime system becomes necessary. This system complements the underlying OS or middleware and will enforce all services to somehow “declare” their resource usage and constraints *at runtime*. This information will be used to guarantee reconfiguration actions that result in a properly working configuration. The extra information that each service needs to declare could partly be inferred (e.g. through code analysis or benchmarking), and partly added by a developer². Depending on this information, the runtime system can stop or move other -less important- services to perform the reconfiguration, or it can refuse or delay the reconfiguration itself. The remainder of this section outlines 2 important concepts of the runtime system: components and contracts.

4.1 Components

Decomposing a complex ubiquitous computing system into well defined services is a hard task. Coupling between services needs to be very loose, to provide easy reconfiguration of services. Current object-oriented software techniques can help here, yet the strong coupling between objects hinders the efficient modeling of services: considering one object on its own is next to impossible. Component based development solves these limitations. We believe modern component based middleware platforms provide a sound basis to extend and implement the techniques that are necessary to realize ubiquitous computing.

Although the use of components as software building blocks is not new (e.g. the CORBA component model, or Microsoft’s COM), there is no generally agreed on definition of a component. Depending on their use, one can assign different characteristics to components. The DESS component model [2] provides a definition and vocabulary of what a component can and must be within the context of the development of real-time embedded software systems. It defines a common terminology that allows people from different application domains to understand each other about component development. The model has also guidelines on how components and interfaces should be specified, including resource aware behavior. For this paper, it is enough to assume that components are loosely coupled, independent pieces of software that export

²This does not necessarily need to be the component developer: the people behind the Quality Objects framework suggest a special QoS developer[3].

their functionality through a set of interfaces.

The mapping of a service onto a component is not exact. A single component could offer multiple services, but it could also require other components to implement them. Still, the component provides us with the right granularity to implement resource awareness. That is why we will often interchange the words “service” and “component” when talking about resource awareness. The ideas involving resource awareness discussed in this paper are not exclusive to components, but components provide a more natural abstraction mechanism to apply resource awareness on than objects.

4.2 Contracts

In a nutshell, in order to function in a middleware component system and interact with others, a component needs to capture the following:

- *functional* behavior
- *non functional* behavior
- *dynamic aspects* of the above during the component’s lifetime

DESS components capture their behavior based on the contract approach which originated from [1]. In DESS, interfaces are documented at 4 levels:

Syntactic level: Description of the message signatures (name and parameters).

Semantic level: Description of pre- and postconditions.

Synchronization level: Description of the sequence of messages, loops and alternative paths.

QoS level: Includes resource declarations

A contract is then formed if an agreement is reached about these specifications.

5 Resource declarations

The primary use of resource declarations is to enable a component to perform its functions with an adequate quality of service. The QoS level needs to declare the nature, quantity and point in time particular resources are needed. Resources need to be described as abstract as possible (see also section 7 about system support). We propose the use of resource pools, each pool representing the availability of one particular resource. We solve the task of declaring resource usage up-front by differentiating between the “where” and “when” of usage.

First, let us analyze and trace a function call to find out where resources are used. Figure 2 shows how resource use depends on a chain of resource

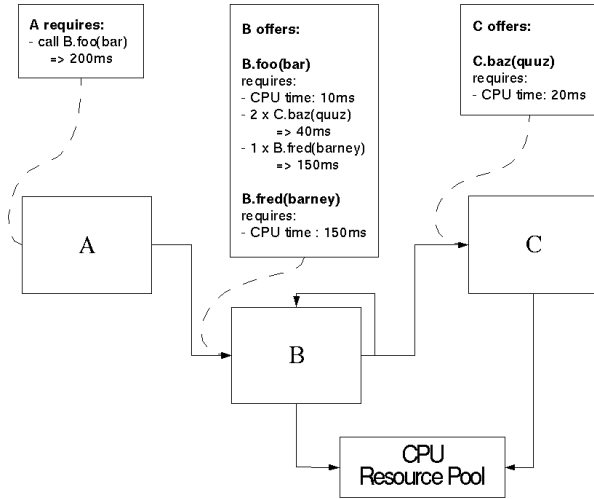


Figure 2: Cascading resource requirements

declarations. As we can see, adding up the declarations from right to left gives us the 200ms of CPU time to execute the call A requires. Similarly, one can apply these declarations to declare the number of bytes to be sent over a network or use of some other service or device.

The second characteristic of resource usage is how often and with which deadline or period the resource will be used. One can split resource costs in:

Deployment, initialization and default

operating costs. Resources are needed to load the component, negotiate interfaces, and start and initialize it, even when it is not used.

Costs for intended (typical) use. Usually, a component is deployed for a reason: if there are not enough resources available for this, the deployment has no purpose.

Costs per additional use. These unanticipated requests can seriously destabilize a carefully tuned component configuration.

Important here is that we need to know, predict or limit up-front how the service is going to be used, if we are to offer guarantees of this service to its users.

6 Resource contracts

As is clear from the above section, a component requiring guarantees for its behavior needs to establish direct or indirect agreements with the resource pools and all components it uses. We will divide these agreements into 2 types of contracts.

The first type is a *resource declaration contract*. A resource declaration contract complements the syntactical interfaces of the component with

the resource use of each function, plus its inter-component calls. Inter-component calls describe for each message of the provided interface which outgoing messages it sends, when and how many. Note that this specification does not violate the black-box principle of a component since it only shows which *component boundary crossing* messages originate from a certain message and therefore not giving away the internal working of the component. Resource declarations can be proposed to the component system; they will enable us to calculate the resource usage of each call. The information in figure 2 captures the information in these contracts in an informal way.

The second type is called *component use contract*. This will describe how the component and its syntactical interfaces will be used throughout its life-cycle. Component use contracts also contain additional constraints to the resource declaration contracts, such as deadlines for timing and bandwidth. When properly negotiated, component use contracts form an agreement between the component itself and the components it uses. They set the QoS levels of the component and provide the required information to determine the schedulability of the proposed component configuration.

Figure 3 shows a simple example of a resource declaration and a component use contract in a component responsible for setting up an audio stream.

QOS SPEC	
RESOURCE DECLARATIONS:	
MESSAGES	//offered messages, resource demands and inter-component connections
MESSAGE (init)	
TIMING USE	= 50 ms;
MESSAGE (Create Stream)	
TIMING USE	= 250 ms;
OUTGOING	= MESSAGE (DataSocket.OpenConnection) : MULTIPLICITY (1);
MESSAGE (FetchStream Packet)	
TIMING USE	= 20 ms;
OUTGOING	= MESSAGE (DataSocket.GetData) : MULTIPLICITY (5);
PERIODICITY	//periodicity of above messages, if any
START	= MESSAGE (Create Stream);
STOP	= MESSAGE (Close Stream);
HOOK	= MESSAGE (FetchStream Packet);
PERIOD	= 40 ms;
COMPONENT USE:	
INIT	//messages executed at deployment
MESSAGE	(Constructor)
TYPICAL	//message executed when fulfilling its function
MESSAGE	(Create Stream)
TIMING SET	= MESSAGE (FetchStream Packet) : TIMING = 30 ms;

Figure 3: Resource declaration and component use contract

7 System Support

Resource contracts in itself do not provide sufficient means to ensure resource awareness. The reservation of resource pools must be coordinated, e.g. by an entity called *resource broker*. As explained, these reservations are dynamic: existing resource allocations will change when service configurations

change. Second, untrusted components must be monitored to keep them from violating their resource contracts.

Enforcing resource awareness poses quite some different problems. Resource needs in itself - especially processing power needs- are hard to describe quantitatively. For instance, the required execution time of different kinds of code can vary greatly depending on the underlying middleware implementation, operating system, hardware, and the data that needs to be processed. One can measure this through benchmarking and off-line analysis and express estimated numbers compared to a fixed reference platform. Likewise, the memory usage of a component may vary depending on the underlying system. Similar methods (involving profiling) exist to monitor memory usage of a component.

8 State of the art

As can be read in [7], the CORBA component model is specially suited for dynamic reconfiguration and is being optimized for embedded and realtime applications. It introduces not only the component concept, but also the component *home* (responsible for the life-cycle of components). RT-CORBA combined with CCM does not support resource aware components. However, the Quality objects framework[3] uses contracts to manage QoS in distributed systems using CORBA. Delegates make the application resource-aware, with the help of reusable system condition objects for resource monitoring. Depending on the current resource situation, the delegates can alter the behavior of tasks needing to be executed. Using special delegates and system condition objects that work together, one could implement local resource awareness in QuO. A plus here is that QuO explicitly supports multiple levels of QoS.

The last few years, the Java platform is working its way into embedded devices. The recent RTJava standard may pave the way for a resource aware Java platform. The RAJE and JAMUS projects both strive for this. RAJE provides facilities to monitor and control resources on a Java platform. RAJE reifies various resources by wrapping them. JAMUS is an experimental platform built around RAJE that tries to solve many of the issues discussed in this paper. Its main goal is to provide a secure runtime environment for resource-aware components. Using resource utilization profiles, a component can specify its resource needs. Unfortunately, for now, these specifications are static. The SEESCOA project [6] has worked on timing and periodicity contracts. Like DESS, they rely on components that are specified on 4 levels.

9 Conclusion

We have shown how resource usage can be declared and agreed upon in component systems using 2 types of contracts. Resource declarations can be made for a component that take into account the inter-component dependencies. For resource aware ubiquitous systems, a supporting runtime system is needed, the two most important subsystems being the resource broker and the monitor.

Currently, we are implementing a prototype resource broker using OSGi[4]. Due to space issues, we have left out a more detailed architectural overview of this runtime system, as well as optimizations and simplifications that can be used when one does not require as much flexibility and/or security.

References

- [1] A. Beugnard, J.M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer IEEE*, 32(7):38–45, 1999.
- [2] The DESS Consortium. Definition of components and notation of components. <http://www.dess-itea.org/deliverables/ITEA-DESS-D144-V02P.pdf>.
- [3] 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)*, 1998.
- [4] OSGi: Open Services Gateway Initiative. <http://www.osgi.org>.
- [5] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, August 2001.
- [6] D. Urting, S. Van Baelen, T. Holvoet, and Y. Berbers. Embedded software development: Components and contracts. In *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, volume 1027-2658 (ISSN), pages 685–690. ACTA Press, 2001.
- [7] N. Wang, D. Schmidt, and D. Levine. Optimizing the corba component model for high-performance and real-time applications, 2000.