Evolution Management and Process for Real-Time Embedded Software Systems

Run-time Evolution and Dynamic (Re)configuration of Components: Model, Notation, Process and System Support

Deliverable 2.4-2.5
Edited by Jens Gerlach (FIRST) & Stefan Van Baelen (K.U.Leuven)

12.12.2003
Version 1.0
Status final
Public Version
Run-time Evolution and Dynamic (Re)configuration of Components:
Model, Notation, Process
and System Support

Authors/Partners:

<table>
<thead>
<tr>
<th>Partner</th>
<th>Author</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraunhofer FIRST</td>
<td>Jens Gerlach</td>
<td><a href="mailto:jens@first.fhg.de">jens@first.fhg.de</a></td>
</tr>
<tr>
<td>Fraunhofer FIRST</td>
<td>Hans Werner Pohl</td>
<td><a href="mailto:hans@first.fhg.de">hans@first.fhg.de</a></td>
</tr>
<tr>
<td>K.U.Leuven</td>
<td>Joris Gorinsek</td>
<td><a href="mailto:Joris.Gorinsek@cs.kuleuven.ac.be">Joris.Gorinsek@cs.kuleuven.ac.be</a></td>
</tr>
<tr>
<td>K.U.Leuven</td>
<td>Stefan Van Baelen</td>
<td><a href="mailto:Stefan.VanBaelen@cs.kuleuven.ac.be">Stefan.VanBaelen@cs.kuleuven.ac.be</a></td>
</tr>
<tr>
<td>K.U.Leuven</td>
<td>Andrew Wils</td>
<td><a href="mailto:Andrew.Wils@cs.kuleuven.ac.be">Andrew.Wils@cs.kuleuven.ac.be</a></td>
</tr>
<tr>
<td>University of Magdeburg</td>
<td>Danilo Beuche</td>
<td><a href="mailto:danilo@ivs.cs.uni-magdeburg.de">danilo@ivs.cs.uni-magdeburg.de</a></td>
</tr>
</tbody>
</table>

Document History:

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Editor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Dec 03</td>
<td>1.0</td>
<td>Stefan Van Baelen</td>
<td>Public version based on internal version 1.1</td>
</tr>
</tbody>
</table>

Filename: D2.4_D2.5_v1.0_Public_Version.doc

This document is part of the work of the EUREKA Σ! 2023 – ITEA 00103 project EMPRESS.
Copyright © 2002-2003 EMPRESS consortium.

ITEA Title: EMPRESS
# Run-time Evolution and Dynamic (Re)configuration of Components:
## Model, Notation, Process and System Support

## 1 Introduction

1.1 Configuration versus Reconfiguration

1.2 Problems of Runtime Evolution in Composed Systems

## 2 Design Techniques and Patterns for Runtime Evolution

2.1 Feature Modeling and Aspect-Orientation

2.2 Components, Homes and Containers

2.3 Proxies

2.4 The Bridge Pattern

2.5 State Transfer

2.5.1 Abstract State Representation

2.5.2 Rudimentary Extraction

2.5.3 Hybrid Approach

2.6 Introspection and Meta Information

2.6.1 Introspection

2.6.2 Meta Information

2.7 Update Control

2.7.1 Quiescence

2.7.2 Time of Update

## 3 Resource Aware Components

3.1 Resource Awareness and Runtime Evolution

3.2 Limited Memory Capacity as a Reason for Resource Awareness

3.3 Design of Resource Aware Components

3.4 Local versus External Control of Resource Usage

3.5 Draco and CRuMB: resource-aware component middleware for robust runtime evolution

3.5.1 Goal of CRuMB

3.5.2 The DRACO component runtime environment

3.5.3 Target of CRuMB

3.6 Scenarios of Use for CRuMB

3.6.1 Practical Applications

3.7 Resource Declarations and Contracts

---

ITEA Title: EMPRESS
Run-time Evolution and Dynamic (Re)configuration of Components:
Model, Notation, Process
and System Support

3.7.1 Declaring Resource Usage ................................................................. 22
3.7.2 Describing Reservations .................................................................. 23
3.7.3 Example Declarations ........................................................................ 24
3.7.4 Negotiating Contracts ....................................................................... 24
3.7.5 Dealing with Non-Contracted Components ......................................... 24
3.8 Architecture of DRACO and CRuMB ..................................................... 25

4 Java™ Service Update supported by the Bridge Pattern .............................. 25
4.1 Eager Update of Registered Services in OSGi ..................................... 25
4.1.1 The Update ......................................................................................... 27
4.1.2 Synchronization Issues of Service Update ......................................... 28
4.1.3 Generation of Bridge Class ................................................................. 29
4.2 Lazy Update of non registered Services ................................................. 30

5 Literature ..................................................................................................... 34
1 Introduction

In this report we discuss several issues of the run-time evolution and dynamic reconfiguration of components in embedded systems. It starts with a presentation of simple scenarios for exchange and discusses several patterns and related techniques that can be used to provide system support for dynamic exchange of components (see Section 2). Care has been taken not to rely on services of a particular software platform. This is important because embedded devices often provide only limited support for services that have been already standardized for desktop and business environments. Another important issue discussed is the distinction between dynamic reconfiguration and dynamic configuration. This is crucial because deeply embedded systems often do not require dynamic changes of a running system.

Section 3 discusses resource awareness of components as a means to determine the time of an update. Section 4 is more platform dependent than the other sections. It uses Java reflection to implement a tool which generates bridges (i.e. the Bridge design pattern) for Java classes.

There is a great variety of component system architectures. They differ in required computational power of the hardware, communication- and synchronization techniques, whether components may have states or not, and so on. First, we discuss dynamic configuration versus dynamic reconfiguration, this will be interesting for autonomous systems, too. Second, we will have a short look at problems rising in case of updating dynamically composed systems.

1.1 Configuration versus Reconfiguration

The dynamic aspect of a software system begins already at startup time and not foremost at runtime. Therefore, dynamic changes of software systems can be divided into two classes:

- Dynamic Configuration which takes place at startup of a software system. During the configuration process the system loads all necessary components to provide the desired functionality, but does not provide any service yet. All loaded components are in their initial state determined either by built-in defaults or startup configuration information provided by external means like command line parameters etc.

- Dynamic Reconfiguration happens when a service needs to be changed in a system that is already running. This in general requires migrating the state of the system in its original configuration to the equivalent state in the changed configuration.

Embedded systems have limited memory and/or processing power. Dynamic reconfiguration is needed only by a small number of embedded systems. As a result only these systems should have to pay the costs for the additional functionality like the dynamically exchange of components. These costs stand for more memory and processing power consumption at runtime.

However, most embedded systems work in a very static environment and the task of the system does not change over the lifetime. These systems need at the most dynamic configuration at startup-time. While this dynamic configuration implies more time at the startup
of system in contrast to statically configured systems, the resource usage during runtime is less than for dynamic re-configurable systems.

1.2 Problems of Runtime Evolution in Composed Systems

Let us consider a consumer – producer system to illustrate the problems that arises in composed systems in case of the dynamic update of components. In this classic example (see Figure 1) there are at least three components: producer, consumer, and the channel. Two of them, producer and consumer, may have their own control flow.

![Producer Consumer Example](image)

Figure 1: Producer Consumer Example

We consider the following issues that are caused by evolution.

- Synchronization of exchanges must be organized.
- After an exchange of a “service” component (the channel) the “client” components (the producer and the consumer) must find it, for example by a lookup mechanism.
- State transfer is an important issue. In particular, the transformation of temporary state that arises from message processing.
- Another important issue are changes in the interfaces.

Even in case the interfaces are stable, some problems can occur. For example, let the message format be generic by using strings. New variants of producers and consumers can differ in the interpretations of these messages.

Of particular interest is when the consumer is replaced by a new version that consumes different messages. In this case, message transformations might be necessary even the producer is updated at the same time (e.g. to deal with messages that are in the channel, or those buffered while performing an update).

2 Design Techniques and Patterns for Runtime Evolution

A basic design principal to support dynamic change for component systems is separation of concerns. There are two important aspects, namely, low coupling and high cohesion.

"Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements" (cf. [1]).

"In terms of object design, cohesion is a measure of how strongly related and focused the Responsibilities of an element are” (cf. [1]).
Obviously, Low Coupling eases the task of component exchange and makes it less error-prone. High Cohesion, on the other hand, reduces the amount of things to exchange.

2.1 Feature Modeling and Aspect-Orientation

Providing a statically as well as dynamically (re)configurable component system implies the existence of more than one variant for a single component. The management of these variants adds another level of complexity to the component developers and users. Component developers have to cope with the differences in implementations in some way, so that most of these different variants are shared. Component users have to choose between the different variants, which share many properties, but important properties like memory usage may be quite different.

Feature modeling (cf. [10]) allows managing and providing configurable components. It describes the common and variable features of a domain like a component or a set of components. A special component is specified by choosing certain features (feature set). That is the providing of the configuration process.

![Feature model diagram](image)

Figure 2: The managing of the configuration of a component

Figure 2 shows a simple feature model of a component with concept node ‘component’. Feature node ‘configuration’ is a mandatory feature and forces choosing between static configuration, dynamic configuration and dynamic reconfiguration. Feature node ‘A’ until feature node ‘B’ are another possibilities to specify a certain component.

Aspect oriented programming could be used to achieve separation of concerns. This means that the dynamic (re)configuration is treated as an aspect orthogonal to the functional aspects of the component. Therewith, implementations which only allow static configuration can be adapted to implementations which provide dynamic (re)configuration by an aspect weaver (cf. [12]).
2.2 Components, Homes and Containers

One important point in the design of a component architecture is the conceptual separation of the system in components, and possibly, component homes and container. The functional aspects of services are concentrated in the components. Components must clearly state their provided and required interfaces. Component ‘homes’ are ‘Factories’. They manage the life cycles of individual components and encapsulate the complex operations to perform.

The component ‘container’ is responsible for general activities of the system, e.g. naming, binding, communication support, and XML-parsing. These activities are often referred to as ‘technical concerns’ in contrast to ‘functional concerns’ that perform the business logic of an application (cf. [2]). In case there are conceptually no component homes, the container also has to manage the life cycle of components.

2.3 Proxies

An important example is the separation of the component’s functionalities from its bindings to suppliers. This especially is useful for hiding the locality of the suppliers. Proxies are widely used in parallel and distributed programming. For remote access proxies are (generally) generated by so called stub generators (see Figure 3).

![Figure 3: A supplier proxy](image)

As the Proxy pattern the following Bridge pattern means an additional indirection. However, there is a difference. A supplier can be related to many proxies that ‘belong’ to its clients. On the other hand a supplier has only one Bridge. In general, both clients and proxies reside in the same address space. The same holds for suppliers and their bridges.

2.4 The Bridge Pattern

A problem is the clients’ awareness of components exchange. Client's component proxies (local or remote) can still point to the old component that may exist no more.
The bridge design pattern provides a natural solution through an additional indirection. The aim of the Bridge design pattern [15] is to “decouple an abstraction from its implementation so that the two can vary independently”. The Bridge pattern is useful when “you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time”. This approach is also known as ‘Handle/Implementation’, ‘Cheshire Cat’, or ‘Protected Variation’.

Protected Variation can be expensive because of its time and space overhead. Appropriate tools generating the code to perform the indirection task however can minimize the additional programming effort. Occasionally, especially for small devices, the overhead of the Bridge pattern may be too high to be acceptable.

In case no Bridge is used an expensive (‘eager’) method is to halt the (generally distributed) system and to update all the proxies of the component’s clients, too. A second (‘lazy’) way is to force a client to perform a new “lookup” by means of exception when it calls an updated or exchanged component.
2.5 State Transfer

A component may have state that must be preserved over the update. In a simple case it may be enough to equip the new component with a constructor having an old component object as single argument. The designer of the new version has to know the old implementation. However, this tight binding of the new component to the old one may hinder reusability. The introduction of `getState` and `setState` functionality appears to be more appropriate. This resembles the Memento design pattern[3]. In contrast to the Memento pattern this functionality belongs to the component implementing classes, not to the memento classes (`getState` of the Memento pattern corresponds to `setState` here). Moreover, in contrast to the Memento pattern it seems to be a good idea to define a generic memento class. This generic implementation can rely on Java Properties or XML descriptions as possible candidates.

We consider two possible approaches to establish state transfer: using an abstract representation and rudimentary extraction. We compare both and discuss their advantages and drawbacks. Finally, we propose another hybrid approach.

2.5.1 Abstract State Representation

It should be possible to create an abstract state representation on which the concrete state of a component is mapped. A major advantage of this approach is that it is only necessary to interpret the abstract state representation in order to initialize a new version of the component.

Another advantage is that the processing of the abstract state representation and the initialization of the new component do have to happen on the embedded device itself. XML parsing requires relatively resource-consuming operations which could interfere with normal operation of the running program.

Also, when several old versions have to be updated to the latest new version, only one transformation function has to be used (i.e. the transformation form the abstract representation to the implementation of the new version). Also, using this abstract representation one doesn’t have to worry about the concrete implementation of the component that has to be updated since the abstract representation makes use of a common terminology.

However, this common terminology should be used consistently for each abstract state representation, so a well-defined ontology is needed. In the DiPS/CuPS (cf. [9]) framework, such an ontology is used. Since this is a component framework for network protocol stacks building such ontology is possible due to the restricted application area of the system. Unfortunately, the EMPRESS component framework will be much more general than this and we believe that building an ontology covering such a broad application area is too complicated for practical use.
2.5.2 Rudimentary Extraction

In this approach no intermediate state representation is used. The components state is extracted, interpreted and transformed by a custom transfer function. This transfer function is version dependent and will have to be customized for each new version of a component that has to be updated.

An advantage of this approach is that the designer of a certain component doesn’t have to build an abstract representation of the components state, which can save quite a lot of work. Moreover, since most components will never be updated, building an abstract state representation for each component at design-time could be considered a waste of effort.

One could argue that building an abstract state representation can be done at the time of a pending update. This is true, but this approach loses much of the advantages of “standard” abstract state representation since it depends on the implementation of the old component again.

2.5.3 Hybrid Approach

Both of the above approaches have their advantages and disadvantages. Therefore it might be a good idea to use a combination of both. We can use an abstract state representation for parts of a component state such as collections (e.g. representations for hash tables, vectors, linked lists...). For such a confined application area, building a suitable ontology should be
possible. For other parts of the state we could use rudimentary extraction.

2.6 Introspection and Meta Information

Components should provide functionality to inform their environments about their provided and required interfaces. This information can be used for searching components with desired functionalities in case they are not known in advance.

2.6.1 Introspection

Languages like Java™ and C# offer a great variety of reflection function calls which conveniently can be exploited. The Java Micro-edition provides only restricted reflection capabilities.

C++ so far only has Run-Time Type Identification (RTTI), which provides limited support for introspection. However, in the context of the Corba Component Model (CCM), a more useful introspection mechanism is provided. This does not necessarily mean that Java or C# are better languages than C++. It rather shows that both C# and Java™ have integrated middleware support into their respective platforms.

2.6.2 Meta Information

In case no language supported reflection can be applied, the system designer can use the "Reflection Pattern". So called Meta objects provide information about objects, which can be retrieved by clients of them. In this context XML descriptions of components and configuration are useful. They can be used for deployment, reconfiguration and for the introspection functionality.

Moreover, in this section we describe how we can add meta-information to components that can be used to determine the amount of work needed to perform a certain update. A key factor to estimating the effort an update will require is to know what the impact of that update is on the running application. Therefore we need to know in detail how that component interacts with its neighbors and what constraints apply to those transactions.

We can describe the communication between two components in an XML based format and attach a contract to that description to specify the constraints on the transactions described in there. We refer to the XML specification of communication as meta-protocols and to the contracts associated with them as inter-component contracts.

Using the meta-protocols and inter-component contracts associated to the component to be updated we can determine before the update what components will have to be updated as a consequence of this action. (This propagation of updates through the application is the so-called ripple effect).

When we know which components have to be updated we can start estimating the cost of that update. To do this we will compare the component contracts, the meta-protocols and the inter component contracts of the old with the new versions.
2.7 Update Control

Updating a component is possible when that component is:

- Quiescent and self-contained
- The update will not violate any constraints in the running system

Quiescent and self-contained basically means that the component isn’t active and that it’s in a consistent state.

To ensure that an update doesn’t violate any constraints in the running system we will have to determine a time at which a given update can be performed safely. In the next 2 sections we discuss the two requirements stated above in more detail.

2.7.1 Quiescence

Kramer’s definition (cf. [7]) of quiescence (in the context of communicating nodes) is:

A node is quiescent if:

- It is not currently engaged in a transaction that it initiated,
- It will not initiate new transactions,
- It is currently not engaged in servicing a transaction, and
- No transactions have been or will be initiated by other nodes, which require service from this node.

In such a quiescent state, a component state is both consistent and frozen.

If a component is not in a quiescent state we cannot guarantee that no information will be lost when an update is performed. For instance: when a component is updated while its in the middle of an algorithm, the preliminary results of the calculations so far will probably be lost. Note that it is our belief that using components and an underlying system to deliver component messages partly eliminates the fourth requirement of the definition. If we can delay messages to the component being updated then it’s not necessary to prohibit starting new transactions with this component. These can be delayed until the update is performed and be serviced by the new version.

Determining if a certain component is in a quiescent state is relatively easy for components since its container handles all communication. The container can perform some bookkeeping of all incoming and outgoing transactions and can publish this information by the life cycle interface.

2.7.2 Time of Update

Determining when an update can be performed safely is a bit trickier. First of all: determining the ideal time for an update is a problem that is undecidable as proven by Gupta (cf. [6]).

However, it is possible to determine a safe moment to perform a given update using simple heuristics based on the currently available resources and the estimated resource consumption of the update.

To establish this we will need the following:

- A resource-aware component system with dynamic scheduling and reconfiguration abilities
A means to estimate the resource consumption of a component-update

Resource-aware components with detailed information on their minimal, current and maximal resource usage

Therefore we need resource awareness, which is discussed in the next section.

Scheduling an update requires some knowledge about the resource consumption of an update. Some of the factors that contribute to this are easily measured (such as the byte size of a component instance) but others are not (such as the amount of work needed to transfer the state of the old component to the new one). State transfer is according to us the most resource consuming part of an update. To estimate the resource used for the transfer, we need extra information about how different two component versions are. For this we will rely on meta-information.

3 Resource Aware Components

3.1 Resource Awareness and Runtime Evolution

In embedded systems, run-time evolution will more and more be associated with resource awareness: the ability of a component to adapt itself or to be adapted to the restrictions of the runtime environment. Indeed, many visionary computing scenarios involve embedded hardware devices that are becoming multi-modal and multi-functional. MP3 players and mobile phones are turning into PDA’s and vice versa. However, the resources of these devices remain limited, and for flexible software platforms to operate safely and robustly, they must be resource-aware. Moreover, dynamic reconfiguration of resource-critical services relies even more on the resource knowledge of the underlying platform. That is why we regard resource awareness as an important aspect of this work package, although it is not on itself part of the methods described in Chapter 2.

A simple example for self-adaptation provides the stable_sort algorithm from the C++ standard library. This algorithm attempts to allocate a temporary memory buffer, and its run-time complexity depends on how much memory is available. In general, however, the problem of resource awareness is not restricted to a single component and must consider the cooperative usage of restricted resources.

Here is a short list of typical constraints of embedded devices.

- Processor performance,
- Time,
- Memory capacity (RAM, flash, mass storage),
- Input/output (sensors, signals, networks).

3.2 Limited Memory Capacity as a Reason for Resource Awareness

As an example for different approaches to handle resource awareness a component in a device with limited memory capacity is considered.
In the first scenario, the component itself is responsible for the adherence to the restrictions. In this case, the component receives at startup a resource constraint structure (RCS), which contains a description of the constraints. Alternatively, these structures can also be passed to the component on its request. The structure could be formatted in XML in which case the definition of the constraints would be done in an XML schema. If the component must allocate a certain amount of memory it checks whether it exceeds the limit and, if necessary, tries to release other memory it had requested beforehand. A benefit of this approach is that the resource constraint is handled inside the component. A problem is how to be sure that the component really adheres to the constraint.

In the second scenario, resource awareness is enforced by the runtime system. The component is not directly aware of how much memory is available. If the limit is exceeded the runtime systems signals a resource constraint violation. The runtime system can use exceptions or other mechanisms for signaling. This approach relies on a steady control through the runtime system.

### 3.3 Design of Resource Aware Components

Like many other component related properties, resource awareness should be treated as an aspect, which should be realized with the least possible interaction with other parts of components, to allow reuse of these parts in scenarios where resource awareness is not necessary and would consume resources without gain. Aspect oriented programming and scalable and adaptable components are very helpful in this context (see Section 2.1).

The resource awareness of components is not only a problem for dynamic (re)configurations but also for static configurations. So far, it is a global problem of embedded systems and therefore it has to be solved generally.

To be able to reconfigure components we need resource aware components. We can achieve this by using contracts (cf. [8]) that specify the minimal and maximal resource-usage of the component.

Components can implement their resource-awareness in different ways. First of all, as mentioned before, it is desirable to separate functional from non-functional behavior. Following this rule, a component implementation should by definition not be aware of the resources it uses. Aside from using Aspect-Oriented Programming (AOP) for this, one could put some sort of delegate or wrapper (such as a component container) in charge of most of these responsibilities. This way, there are at least two options to adapt the behavior of a component:

1. Provide an adaptable implementation. The wrapper or delegate can then use this implementation to change the functional behavior in function of the resource-usage of the component. An example of this is a wrapper that sets the frame rate of a component providing a video stream.

2. Change the entire implementation when other behavior is needed. This could be useful when an entirely different algorithm has to be selected to change the resource usage of a component. The video component mentioned in the previous paragraph could be replaced by one that uses a different compression scheme.

The second approach has a number of interesting advantages over the other:

- The implementation and behavior could be changed radically to adapt to the changing Quality-of-Service (QoS) requirements.
• The developer of the component does not have to care about changing the behavior due to QoS requirements during the implementation.
• The running code would be optimal and efficient, as it would only have to deal with one set of resource constraints.

Of course, all the problems inherent to live-updates would apply here. Furthermore there would have to be multiple implementations present in the component, possibly increasing the footprint of the component and the amount of code that needs to be installed on the system. Finally, there are certain cases where it would be much more efficient to change the behavior of a component by passing it some parameters. An interesting example of this is a component that encodes a video stream. When it is forced to reduce the bit rate of the stream, it can first drop some frames, or perhaps decrease the resolution of the video stream. When the bit rate would have to drop below a certain threshold, the component could be forced to change its implementation and switch to an algorithm that is more efficient at low bit rates.

Now remains the question: who will hand out the RCS to the component containers and control each component’s resource consumption? The notion of a resource broker seems fit to do such a task. A resource broker could keep an eye on the general resource status of the system, and take appropriate actions to ensure the correct QoS behavior of all the components. The responsibilities of such a service would depend on the level of control one would want. We will present 3 levels and discuss them briefly.

1. passive: when all components can be fully trusted, the broker could be reduced to a simple bookkeeping service. It would only have to be given some initial data on the resources of the system. Based on this and the RCSs that have been given out it can calculate the available resources for itself and negotiate further RCSs.
2. monitoring: in this phase the broker will have the ability to monitor the system resources each component uses, and control the components accordingly. When the RCS is violated, the broker could signal this to the component (using the abovementioned exception). When the violation persists, the component can be given another RCS or, in the long end, be stopped.
3. active: the broker can make sure components do not make excessive use of their resources by pre-empting their use of it. For example, the broker could change the priorities of components running in a RTCORBA ORB [20].

The broker is an instance that could be extended to support a dynamic QoS-reconfigurable system. This way, based on the sets of RCSs components support and the importance of each component, a satisfying component configuration setup could be created on a device that has limited resources. The addition of new components could trigger a reconfiguration where a number of existing components would have to sacrifice some resources to be able to accommodate the new ones. An example of this in the network domain is REMOS [21]. It offers an architecture that collects information about the local network, and allows applications to adapt themselves accordingly.

### 3.4 Local versus External Control of Resource Usage

There are two kinds of resource aware systems both with a significant higher resource usage compared to non resource aware component systems.

Resources awareness can be realized locally on the target system or it can be monitored and be checked on a remote system. Depending on the available resources resource awareness can be realized also partially local and partially remote.
The live monitoring of resource usage of components needs to be at least partially, in most cases however completely realized on the target system. Checks of static information about components with regard to their resource usage, they can be done either on the target system itself during runtime for (re)configurable systems and on a external system for all kind of systems.

Especially for deeply embedded systems the resource awareness needs to be on external systems due to insufficient resources on the target.

3.5 Draco and CRuMB: resource-aware component middleware for robust runtime evolution

This section describes actual system support for building resource-aware components in the form of 2 complementing frameworks: DRACO and CRuMB.

3.5.1 Goal of CRuMB

CRuMB is a framework that was developed in EMPRESS to facilitate the development of resource aware components. It offers local control of resource usage and many of the techniques discussed in section 3.3.

The goal of CRuMB is threefold:

- **to enable robust run-time evolution of software components**: CRuMB supports dynamic life-cycle management of components. With CRuMB, one can deploy, remove and update components at runtime using the techniques presented in this deliverable without interfering with global system behaviour. These structural reconfiguration actions allow changes to the following:
  - Location of components: services can be moved from device to device to follow its user
  - Implementation of components: bug-fixes and enhancements can be installed at runtime
  - How components interact, messages or RPCs

  This allows a long lived service to maintain its availability despite changes in its execution environment

- **to add robust resource-aware behaviour to component based systems**: Dynamic component systems cannot predict up front what component configurations will be formed and if the system is able to accommodate enough resources. CRuMB ensures that every configuration is a safe one. Each component’s QoS (Quality of Service) constraints are made explicit and guaranteed by the underlying system

- **enable run-time evolution of QoS behaviour**: a component’s QoS constraints should be flexible in order to maximize the number of services the hardware can run, while keeping an adequate QoS for each one

Current state of the art component based systems either offer no safe and fine-grained resource management or no safe dynamic evolution. Indeed, the embedded systems industry is still sceptical about anything that is dynamic, and research is only slowly moving towards more dynamic real-time systems.
3.5.2 The DRACO component runtime environment

Rather than implementing CRuMB on its own, we have realized CRuMB as a module in a component framework called DRACO.

DRACO is a runtime component environment developed within task 2.4/2.5, designed with deployment on current and future embedded systems in mind. Therefore, it aims to be lightweight, robust, as well as extensible. Functionality is deployed in the form of components that follow the guidelines and specifications of the previous paragraphs.

The DRACO runtime serves as part of the basis for the work of K.U.Leuven in EMPRESS and future research projects. Features that will be investigated and implemented in DRACO include distribution, live-updates and a resource aware monitor and broker (CRuMB). DRACO accomplishes its extensibility through its flexible and modular architecture, which is explicitly designed to accommodate new functionality.

Figure 8: DRACO architecture

Figure 8 describes the general architecture of the component runtime. It should be noted that modules, the abstraction used to extend DRACO, can easily obtain and manipulate message flows through extensive use of observer-based communication and fixed API's.

Figure 9 gives an example of a DRACO component blueprint. The portgroup keyword indicates that the port can have a number of instances. Also, notice that the updateSpeed message is automatically commented out. Indeed, as the message is outgoing it is not a necessary part of the component blueprint’s port interface. Nevertheless, the message is specified for documentation purposes, and to be able to test for composability with other components.
The generated interface can further be implemented with special component mechanisms and plain Java code. Automated translation is provided to transform the component into a Java-compliant set of classes.

Component composition is done in DRACO through a scriptable shell interface. Figure 10 shows a script that can directly be used in DRACO to create several component instances and connect them.

```java
package components.speedometer;

component speedometer {
    portgroup clock_settings 1 {
        // Message Parameters:
        //    parameter: java.lang.Integer
        message setClock {
        }
    }
    portgroup speed_update UNLIMITED {
        /* Incoming message
        // Message Parameters:
        //    parameter java.lang.Integer
        message updateSpeed {
        }
        */
    }
    portgroup tick 1 {
        // Message Parameters:
        message tick {
        }
    }
}
```

Figure 9: example of a generated DRACO component blueprint

Load Component wheel_sensor_1 from wheel_sensor.jar
Load Component speedometer_1 from speedometer.jar
Connect wheel_sensor_1/tick with wheel_sensor_1/tick

Figure 10: example of a generated component instance group creation script

### 3.5.3 Target of CRuMB

CRuMB targets embedded and soft-realtime systems that have adequate resources to serve as a component platform. These systems act as a server for services that are packaged in
loosely coupled components. Components are built to be easily migrated, interchanged and combined. Contemporary PDA’s, residential gateways, desktop PC’s as well as future ubiquitous computing devices are potential candidate component servers.

Amongst the resource requirements services (and thus components) pose are:

- **Timing**: certain code fragments need to be executed within a certain time frame. Typically, this involves a single, often recurring task for a service, e.g. an engine control loop or video stream decoding.
- **Memory**: memory is needed to accommodate code, data and runtime variables.
- **Bandwidth**: a recurring task can need to reserve a portion of the aggregate bandwidth of the system.
- **Storage**: the ability to write data to persistent media at a certain rate.
- **Power**: a side-effect of most resources is the battery drain they cause.
- **Cost**: another side-effect, e.g. electricity or bandwidth costs.

The 3 resources CRuMB is aiming at most are CPU power, memory and bandwidth. We find that these constraints are the most difficult to tackle, yet few efforts exist to create general broker for them. On the other hand, CRuMB also endeavours to have its support for resource constraints to be easily plugged in. This way, we want to gradually extend CRuMB to deal with more types of resources, and offer an API for others to do the same.

### 3.6 Scenarios of Use for CRuMB

We are convinced that the use of resource contracts greatly contributes to the flexibility and robustness of many types of software. To illustrate this, we present a number of example applications and scenarios in which CRuMB would increase the overall user satisfaction.

#### 3.6.1 Practical Applications

##### 3.6.1.1 A Graphic Intensive Application

Multimedia services are becoming more and more important in portable devices. User interaction is increasingly done using CPU-intensive, yet bandwidth-friendly vector graphics. An example of such a protocol is Macromedia Flash. This pseudo-standard can bring down even high-performant desktop PC’s, if configured to use all anti-aliasing and other techniques. A service on an embedded device wishing to display such graphics, should instead negotiate a resource contract in which a quality level is fixed. E.g. the animation could be fullscreen or not, the lines could be anti-aliased, and so on. Depending on the policy CRuMB uses, the system can automatically decide which services should be prioritised. This way, the user is always enjoying the highest quality level on his system.

##### 3.6.1.2 A Resource Aware Spelling Checker
This example shows how non-realtime applications could also benefit from resource contracts. Editing documents is in itself not a very resource-consuming task. Adding intelligent behaviour to a word processor however, is. An example of such a resource-consuming task is the spelling.

or grammar checker. This activity often runs as some sort of background process in modern word processors. Desktop-sized systems usually have the computing power required to do real-time spelling checking. Even then however, the user could load a large document, upon which the checker starts analysing the entire document, reserving resources and reducing the responsiveness of other applications and tasks.

The bottom line here, is that the importance of the spelling checker needs to be established, and its priority and behaviour should be adjusted according to the importance and available resources. E.g. the spelling checker could be configured to do run-time checking at startup. Addition of new services could force the checker to run at less-than-real-time speed, or even to suspend its operations entirely. Naturally, when the user requires a spelling check, or as more resources get freed, the system is configured to allocate more resources to the checker.

3.6.1.3 Automatic Security Updates

Users are increasingly confronted with leaks and security bugs in their software. Every week, at least one component of a desktop system needs to be replaced to fix these, often requiring a reboot of the system. For a user of a ubiquitous device, often operating in “hostile” environments, these security fixes are at least as important desktop user. Of course, that user would expect no or minimal change in the device’s behaviour while installing security fixes. Using CRuMB, this update can be negotiated and scheduled at a convenient moment. Some services may have to decrease their quality level when the update is installing, yet the user suffers no real inconvenience.

3.7 Resource Declarations and Contracts

The primary use of resource declarations and contracts is to quantitatively measure and guarantee a component’s quality of service. CRuMB requires components to be resource-aware, which means they know about the resources they require. More specifically, CRuMB requires components to declare the nature, quantity and point in time particular resources are needed.

To effectively describe resources in an abstract and platform-dependent manner, we propose the use of resource pools, each pool representing the availability of one particular resource. When declaring resource usage up-front, we will differentiate between the declarations and reservations of resources. Declarations describe the resource usage of the component’s interface methods, whereas reservations are negotiated agreements about the component’s requirements.
3.7.1 Declaring Resource Usage

In order to guarantee that each component gets the resources it needs, CRuMB needs the following information from them:

- what resources are needed if a component is used e.g. the method x takes 50 ms of CPU time to complete (declarations)
- when components will be used e.g. method x will or can be executed every 100 ms (constraints)
- how is a component used e.g. method x needs to be executed within 70 ms (reservations)

Resource declarations describe the resource requirements of all interfaces the component exports. This information is necessary when other components want to make use of the interfaces. To encourage low-coupling among components, a component needs not provide resource declarations for the components it uses. However, it will need to include which and how much their interface methods will use those components. To illustrate this, consider Figure 11. It shows how resource use depends on a chain of resource 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.

![Figure 11: Cascading resource requirements](image-url)
The described resource declarations are formalised in 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 are a form of information towards the component system; they will enable us to calculate the resource usage of each call. The information in Figure 11 captures the information in these contracts in an informal way.

3.7.2 Describing Reservations

Next, we can establish concrete agreements on how the component is to be used in the future. If a system is to guarantee correct QoS behaviour for all its components, it needs to know up-front what to expect. Until a contract is agreed on, no guarantees can be made E.g. a video stream component can be asked to download a video at a rate of 300Kb/s. If a contract is present for such a stream, the download can proceed. If not, a new contract can be negotiated; this negotiation may fail. A contract, however, can be made as early as deployment time. That way the system can reserve the appropriate resources beforehand.

One can split up component use resource costs in:

- **Deployment, initialisation and default operating costs.** Resources are needed to load the component, negotiate interfaces, and start and initialise 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. Examples of typical use could be:
  - a one shot action
  - guarantees for an incoming call (reserve-ahead)
  - a periodical call

- **Costs per additional use.** These unanticipated requests can seriously destabilise a carefully tuned component configuration; hence, they are scheduled and admitted depending on available resources on a per-call basis.

A (component use) contract describes 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.
3.7.3 Example Declarations

Figure 12 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
RESOURCES 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(FetchStreamPacket)
  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(FetchStreamPacket);
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(FetchStreamPacket): TIMING = 30 ms;
```

Figure 12: Resource declaration and component use contract

3.7.4 Negotiating Contracts

As mentioned above, it is the component use contract that is subject to negotiation. To offer an adequate level of flexibility, the following mechanisms are used during the negotiation process:

- a component can offer a propositional contract to the system. In case the contract is refused, a new contract can be offered. This enables a component to consider other algorithms (e.g. consuming less resources or favouring other ones).
- to minimise the number of negotiation steps, a component may also specify its constraints in ranges: e.g. the deadline of a message is preferably 20 ms, but the information is still useful after 50 ms.

3.7.5 Dealing with Non-Contracted Components

In a resource-aware system, it is essential to “sand-box” non-resource-aware components, to
keep them from disturbing the carefully negotiated resource constraints. Normally, they would only be allocated resources when others do not need them. It could be wise to split the resource pools, meaning both parties could benefit from a fixed set of e.g. half the resources, where the resource-aware party enjoys all earlier mentioned benefits. Typically, the resource-aware components will be too important to give them only a part of the available resources.

3.8 Architecture of DRACO and CRuMB

A more elaborate description of DRACO, CRuMB and their architecture can be found in Appendix A and B.

4 Java™ Service Update supported by the Bridge Pattern

In this section we give two patterns how do use Bridges (see 2.4) to enable runtime update of Java components. In both cases the exchange of components at run-time needs some preparation at design-time.

In Subsection 4.1 we consider service components which are registered by the container respectively the component framework as it is typically the case in OSGi. In Subsection 4.2 we consider the update of unregistered service objects (including temporary objects on local scopes).

4.1 Eager Update of Registered Services in OSGi

The Open Service Gateway Initiative (OSGi) is a consortium that has specified a Java™ component framework [14] for delivering, activating, and replacing services over wide-area networks to local-area networks and devices. Since OSGi server implementations can be quite small, it is a good candidate when looking for supporting runtime evolution in embedded systems.

OSGi deployment components – which are called bundles -- interact by sharing so-called service objects. Services are registered by one component and referenced by others. A major problem of OSGi component exchange at runtime is the replacement of service objects. Shutting down all dependent components and restarting the re-placed sub-system, as recommended by OSGi, seems overly expensive. Here we describe the use of the well-known Bridge design pattern to decouple service replacement from bundle updates. Instead of registering services only references to instances of automatically generated

A service is nothing more than an object of a class that implements one or more service interfaces. Figure 13 gives a simple example of a class FooBar that implements two service interfaces IFoo and IBar.

A bundle offers a service by registering it at the OSGi server. An example of registering a service is shown in .Figure 14 Note that in order to register a service object the names of service interfaces must be supplied.
Bundles that wish to use services do not directly request service objects. In order to obtain access to a registered service, a bundle queries the OSGi server by providing names of service interfaces.

```java
String[] names = {"IFoo","IBar"};
ServiceRegistration reg =
    context.registerService(names, new FooBar(),...);
```

**Figure 14: Register a service**

If a service object has been registered under these interface names, the OSGi server returns an object of type ServiceReference. The requesting bundle obtains a “real” reference to the registered service object only through such a ServiceReference.

```java
ServiceReference ref = context.getServiceReference("IFoo");
IFoo service = (IFoo)context.getService(ref);
Service.foo(); // call the service method
```

**Figure 15: Get a registered service**

A bundle that has registered a service can unregister it as long as it holds the corresponding ServiceRegistration object (see Figure 16). All what the registering bundle has to do is calling the unregister method and registering a new service object as shown here.

```java
reg=context.registerService(names,new NewFooBar(),...);
reg.unregister();
```

**Figure 16: “Unregister” a service**

However, this only works if the registering bundles had anticipated that it might be necessary to replace a service object. Even more severe is that this kind of update may lead to dangling reference in
bundles that have obtained a ServiceReference. One way to avoid dangling service references in client bundles is to use service listeners of OSGi. However, solely relying on listeners shifts the burden on the clients of a service. OSGi recommends that instead of updating individual service objects their registering bundle is updated which implicitly includes that the bundle is stopped and new version of the bundle is started. OSGi ensures that all registered services of a bundle are unregistered when the bundle is stopped. It is the task of the new bundle start method to register corresponding service objects.

In the following we show that this expensive solution can be avoided and that service objects can be individually updated without invalidation of references.

### 4.1.1 The Update

The basics of our Bridge pattern transformation is best demonstrated by its application to the example. The service implementing class and the interfaces remain unchanged. Note that the generated bridge only refers to the services interfaces and not to a particular service class.

```java
0  public class FooBar_Bridge implements IFoo, IBar {
1    private Object impl;
2    public Object getImpl() { return impl; }
3    public void setImpl(Object object) { impl = object; }
4    public void foo() { ((IFoo)impl).foo(); }
5    public int bar(int i) { return ((IBar)impl).bar(i); }
6  }
```

**Figure 17: The Bridge**

Using the generated bridge class is quite simple. Instead of registering a FooBar service object, a bundle has to register a FooBar Bridge. Since client bundles refer to a service only through its interface(s) they notice no difference.

The registering bundle can now easily replace a service object at run-time as it is shown.

```java
0  ServiceReference ref =
1    context.getServiceReference("IFoo");
2  FooBar_Bridge bridge =
3    (FooBar_Bridge)context.getService(ref);
4  bridge.setImpl(new NewFooBar());
```

**Figure 18: The dynamic update by an exchange bundle**

The update can even be performed by another bundle.

With services following the Bridge pattern a service can be explicitly invalidated through bridge.setImpl(null). This is not possible if (non-bridge) service objects are directly
registered since the Java garbage collector can free a service object only when no client bundle references it any longer.

4.1.2 Synchronization Issues of Service Update

In this section mainly we discuss issues of synchronization when applying the Bridge pattern to represent services. Other important issues are state transfer (see Subsection 2.5), security and the treatment of service factories.

Regarding the relationship of service update and the OSGi security concepts [14], we only mention that for changing the service implementation a bundle must have the same service permissions as for registering the service. Updating service factories requires both updating the factory and up-dating the customized service implementation objects. One problem that cannot be solved easily is that with the current OSGi specification service registering bundle has no access to the individual service objects of other bundles.

Even in the case that there are no synchronization issues for the original system we have to ensure the thread safety of the system extended by bridge classes. First we consider the simple case that all methods of the service object are declared as synchronized. If there is no state transfer, we define the setImpl() method as follows:

```java
public void setImpl(Object object) {
    synchronized(impl) { impl = object;}
}
```

If state transfer is an issue it will not be sufficient to only synchronize inside of the getImpl and setImpl methods because another thread may have changed the state of the service between these calls. One possibility to solve this problem is to synchronize “outside” of the bridge.

```java
synchronized(bridge.getImpl()) {
    Object obj = bridge.getImpl();
    // compute newImpl depending on obj
    bridge.setImpl(newImpl);
}
```

Figure 19: Synchronizing with state transfer

Now we consider the case that the service methods have not been declared synchronized. The aim is to allow (at least in principle) that service invocations work as they do without bridge. However, an update must not take place while a service is in use and vice versa (mutual exclusion). Figure 20 presents one solution. The bridge class gets an additional private integer member useCount that holds the actual number of users. In case it is not zero the thread invoking the setImpl method has to wait. Later on the thread can be waked up by the last thread that invoked foo. On the other hand, while performing an update (which may include a state transfer) no foo method can enter because the incrementation of useCount is also synchronized with the bridge object.
4.1.3 Generation of Bridge Class

Since we want to deal with unanticipated changes it is of paramount importance that users themselves need not to program the bridge classes. Fortunately, the generation of the bridge classes is straightforward and so (as in Hicks’ system [18]) it can easily be automated. We present two approaches. The first one generates the bridge classes at design time. The second approach uses a generic and very flexible bridge implementation. Both rely on Java™ reflection.

4.1.3.1 Automatic Generation At Design Time

A convenient way to perform the code generation is to use the reflection mechanisms of the Java™ platform. No external tools are necessary. It is no problem if the target platform does not fully support reflection—as in the case of the Java™2 Micro Edition [19]—since reflection is used at design time only. Using reflection, it is not hard to automatically generate constructors of the bridge class which initialize the private impl member through the related constructors of the implementation class. Registration of bridge-service then looks almost exactly as without bridge.

4.1.3.2 Using Dynamic Proxies

By defining a single universal Bridge class with Java™ proxies there is no need to generate bridge classes at design-time (see Figure 21). This approach requires comprehensive support for run time reflection which is unrealistic for embedded systems. In particular J2ME [19] does not support Java™ proxies. Moreover, there are higher run time costs caused by additional indirections. The client calls the Java™ proxy, an object of a class with no (in the program visible) name. This object calls the invoke method of the Bridge object, which finally calls the method’s invoke method.

Figure 20: Synchronization in the general case

```java
0 void foo() {
1    synchronized(this) {useCount++;}
2    ((IFoo)ipml).foo();
3    synchronized(this) {
4        useCount--;
5        if (useCount == 0) notifyAll();
6    }
7 }
8 public synchronized void setImpl(Object object) {
9    while(useCount != 0) wait();
10   impl = object;
11 }
```

Figure 21: The bridge invocation handler
For convenience we define static methods of the Bridge class (see Figure 22). The newInstance method generates the proxy with invocation handler which are related to the final serving object. The proxy object is registered by the server bundle. The method setImpl is used to perform the dynamic update, getImpl is defined analogously. As is in the case of generated bridge classes there are no changes for clients of a service.

```java
public class Bridge implements InvocationHandler {
    private Object impl;
    private Bridge(Object obj) { impl = obj; }
    public Object invoke(Object proxy, Method method, Object[] args) {
        Object result = null;
        try {
            result = method.invoke(impl, args);
        } catch (Exception e) { ... }
        return result;
    }
    // static methods ...
}
```

We also want to mention another way to define the invocation handler. It is possible to enrich the interfaces by the setImpl and getImpl methods inside of the newInstance method. This requires that the invoked methods are checked within the Bridge.invoke method.

```java
public static Object newInstance(Object obj) {
    return Proxy.newProxyInstance(
        obj.getClass().getClassLoader(),
        obj.getClass().getInterfaces(),
        new Bridge(obj));
}
public static void setImpl(Object proxy, Object obj) {
    Bridge bridge =
        (Bridge)Proxy.getInvocationHandler((Proxy)proxy);
    bridge.impl = obj;
}
```

**Figure 22: Static methods of Bridge**

4.2 Lazy Update of non registered Services

In this subsection we will show a way of how to prepare components that are not registered at a framework by means of a simple example. Therefore we consider the evolution of the (slightly modified) Foo class of the last subsection.

We will meet here an additional requirement. Program code, that uses Foo before
Run-time Evolution and Dynamic (Re)configuration of Components:
Model, Notation, Process
and System Support

preparation, can remain unchanged.

```java
0  Class Foo {
1    // ...
2    public Foo() { /* constructor code */ }  
3    public int foo() { /* code */ }  
4    // ...
5  }
```

There is a concept to meet whose requirements (mainly) are consequences of the Bridge pattern.

- No public data members are allowed, only public member functions.
- The class’ first constructor has no arguments (here for simplicity).
- Private access only operates to other members of the same object. Private members of other objects of the same type cannot be accessed.
- Restricted use of “this”, it must not be exported to other objects.
- Inheritance can be used for being callable like the superclass and for inheritance of code but not for polymorphism.
- In case the actualization of the component states should be done member functions “getState” and “setState” must be implemented. (Abstract State pattern 2.5.1).

Finally, the next requirement is very important for safety.
- There is only one thread that calls the component’s public functions. (The container’s exchange functionality can be called by other threads).

```java
0  class Foo_Implementation implements Foo_Interface {
1    // ...
2    public Foo_Implementation() { /* same code */}
3    public int foo() { /* same foo code */}
4    // ...
5  }
```

The result of the preparation of the original class consists of two classes and one interface.
The first class is the implementation class, in our example “Foo_Implementation”. It only differs in the class name, (of course) in the names of constructors and additionally it implements the generated interface. The second class is the Bridge with the same name as the original class, in our case “Foo”. This makes sense because no changes are necessary in the surrounding code.
In our example we implemented a “lazy” exchange. A component is not to exchange in the moment the exchange administration takes place. It is to exchange in the first moment when it is called after. The static member “type” is the actual administrated type and the non-static member “myType” is the actual type the component really has. The container’s administration functionality only exchanges the static member “type”.

An advantage of this lazy mechanism is that components can be updated where ever they are located, the stack included. On the other hand, a drawback is that active components never being called can never be updated. In this case one has to design the system in the more usual way: Components have to register in a “container” and the update has to be performed eagerly.

A tool that is implemented with the reflection facilities of Java can easily support the generation of the bridge and the interface; no advanced compiling techniques have to be used. Even the very simple generation of the implementation class is a bit more sophisticated. However, for instance, it can be done by “lex”.

While the tool mentioned above needs reflection at design-time the container who is the runtime environment of the components needs reflection at run-time. The container only expects that the components meet the concept. However, the names of classes, the number and names of member functions are especially in the evolving context completely unknown at design time of the system.
Finaly, at run-time, evolution can take place. According to the evolved requirements a new

```
0 class Foo {
1    private static Class type =
2        Container.forName("Foo_Implementation");
3    private Class myType = type;
4    private Foo_Interface impl;
5    public Foo() {
6        impl=(Foo_Interface)Container.getInstance(type);
7    }
8    public int foo() {
9        if (type != myType) {
10           // State state = impl.getState();
11           myType = type;
12           impl =
13              (Foo_Interface)Container.getInstance(type);
14           // impl.setState(state);
15        }
16        return impl.foo()
17    }
18    // ...
19 }
```

implementation is written (and compiled and tested).

Evolution is performed respectively initiated by a container call:

```
Container.exchange("Foo","New_Foo_Implement");
```
5 Literature

2. Small Components: A project to define and implement a component architecture for small, embedded or mobile systems http://www.voelter.de/projects/smallComponents.html
4. J. Cheesman, J. Daniels, UML Components, Addison-Wesley 2001
7. J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change

import java.lang.reflect.*;

class Container {
  // ...
  public static void exchange(String Bridge, String Impl) {
    try {
      Field f = forName(Bridge).getDeclaredField("type");
      f.setAccessible(true);
      f.set(null,forName(Impl));
    } catch(Exception e) { /* ... */}
  }
  public static Class forName(String s) {
    try {
      return Class.forName(s);
    } catch(Exception e) {
      return null;
    }
  }
  public static Object getInstance(Class c) {
    try {
      Constructor[] cs = c.getConstructors();
      return cs[0].newInstance(new Object[0]);
    } catch(Exception e) {
      return null;
    }
  }
}


15. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


DRACO: An Adaptive Run-Time Environment for Components

Deliverable D.2.4-D.2.5 Appendix A
Written by Yves Vandewoude & Peter Rigole, K.U.Leuven,
Edited by Stefan Van Baelen, K.U.Leuven

12 December 2003
Version 1.0
Status final

Public Version
This document is part of the work of the EUREKA Σ! 2023 – ITEA 00103 project EMPRESS. Copyright © 2002-2003 EMPRESS consortium.

Authors/Partners:

<table>
<thead>
<tr>
<th>Partner</th>
<th>Author</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>K.U.Leuven</td>
<td>Yves Vandewoude, Peter Rigole</td>
<td><a href="mailto:Yves.Vandewoude@cs.kuleuven.ac.be">Yves.Vandewoude@cs.kuleuven.ac.be</a></td>
</tr>
</tbody>
</table>

Document History:

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Editor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Dec 2003</td>
<td>1.0</td>
<td>S. Van Baelen, K.U.Leuven</td>
<td>Public version based on internal version 1.4</td>
</tr>
</tbody>
</table>

Filename:  D2.4_D2.5_v1.0_Appendix_A_Public_Version.doc
DRACO: An Adaptive Runtime Environment for Components

Abstract
This paper introduces the architecture of DRACO (Distributed Reliable and Adaptive Components), a modular and extensible component runtime system for embedded devices. It adheres to the SEESCOA component methodology\(^1\) which explicitly models component interaction using port and connector concepts and allows for dynamic adaptations of component oriented applications by rewiring components at runtime. The SEESCOA component language and the constructs it introduces are presented using two example components. In addition, a mapping from the SEESCOA component language to Java is worked out. In order to preserve its small footprint, the DRACO runtime environment can be extended with extension modules to implement a variety of features (e.g., distribution, runtime contract monitoring or dynamic updating). These extension modules can influence the message delivery process using message handlers. To illustrate the power of this technique, two extension modules are worked out in detail.

Keywords
Middleware, Dynamic Component Configuration, Runtime Adaptable Systems, Embedded Systems

1 Introduction
For a long time, increasing reuse and modularity has been the focus of enormous efforts in both academic and industrial research. The reasons for this are clear: reuse decreases development time and thus reduces time to market, increasing modularity isolates changes to one location and therefore reduces the complexity of software maintenance and modification. The result is that many modern object-oriented languages as Java or C# successfully stimulate reuse through extensive open APIs. Since its introduction in the mid 1990’s, component-oriented development takes this paradigm one step further by exploiting extreme low coupling between different components and integrating other features as concurrency, persistence and distribution. As such, it can be argued that the component-oriented methodology is the first that truly realizes the idea of mass-producible software as described by McIlroy in 1969 [1].

Adoption for the development of embedded systems is significantly slower. Mainly due to their scarce resources and real-time behaviour, they have a history of low level design in which reuse and maintainability are often sacrificed for speed and efficiency. Recently however, development for embedded systems is shifting towards high level languages and systems like Java. One important reason is

\(^1\) Funded by a scholarship from the Belgian IWT
\(^1\) Funded by the Flemish Institute for the Promotion of Scientific-Technological Research in Industry
that many problems that arise when using Java with embedded systems have been resolved or at least alleviated in the last couple of years (e.g. improved garbage collection algorithms, virtual machines with low memory footprint, ...). Perhaps the intention of Nokia to release 100 million Java-enabled units by the end of 2003 ([2]) illustrates best that in the near future embedded applications do not necessarily have to be low-level applications.

Components also, are a hot topic, since the majority of their advantages are also applicable to embedded systems: next to their reduced time-to-market and development costs, components fit extremely well in the context of large-scale distributed systems and are excellently suited for mass deployment.

In the context of SEESCOA, a cooperation project between the KULeuven and three other Belgian universities, a component methodology was developed for embedded systems. One of the merits of SEESCOA is establishing a formal definition of a component and introducing an approach in which different components are dynamically connected to form the application ([3]). In addition, a tool to support component-oriented design (the CCOM Composer tool: [4]) and a component runtime were developed.

The current SEESCOA executing environment ([5]) is able to execute an application constructed using SEESCOA components and it has been deployed successfully on an embedded system in several case studies (e.g. see [6] for a distributed camera surveillance system).

However, from our experience implementing larger cases using the SEESCOA methodology, the current runtime environment proved to be too hard to extend. In this paper, we introduce DRACO, a much improved version of the current SEESCOA runtime. The DRACO component runtime is a lightweight, robust and extendible component system that adheres to the SEESCOA component philosophy. Components written for SEESCOA and that follow the SEESCOA guidelines and specifications are able to run on DRACO without large modifications.

The remainder of this paper is organized as follows. In section 2, we first introduce some relevant concepts of the SEESCOA methodology. Section 3 presents the DRACO architecture. This architecture has its impact on component development. In section 4 we discuss how a component can be written for DRACO. In section 5, we illustrate how the DRACO runtime can be extended using extension modules. Two examples are worked out in detail:

- A module for dynamically updating a running application.
- A module that can be used to monitor timing contracts.

Related work on components, dynamic updating and timing is discussed in section 6. We discuss future work in section 7 and conclude in section 8.

2 SEESCOA stands for Software Engineering for Embedded Systems using a Component-Oriented Approach. The project is funded by the Belgian IWT.
therefore we will limit ourselves to a short description of the most important concepts of SEESCOA that are relevant for the DRACO runtime environment.

2.1 Component blueprint and component

Definition 1 (Component Blueprint) A component blueprint is a reusable entity and contains the type description and implementation of a component. It is a static construct that has no run-time meaning. Component blueprints have an identifier, a version and can be stored in a catalogue.

Definition 2 (Component) A component is a reusable documented entity that is used as a building block in applications. It is an instantiation of a component blueprint and as such it has a run-time existence and state. It performs a specific function, and to carry out this task it operates and interacts with other components within the component system. Components are composed by means of their interfaces: they can provide an interface to and require an interface from other components.

To perform their services, components often need to interact. A component has three main activities: receiving messages, performing computations on behalf of these messages and sending out messages.

2.2 Ports

A port is used to allow for inter-component communication.

Definition 3 (Port) A port is a communication gateway for components: a component uses ports to communicate with other components. Every component has zero, one or more ports associated with it.

There is a strong dependency between a port and an interface: a port is a representation of an interface of a component. Therefore ports can only be connected if their associated interfaces\(^3\) match. In SEESCOA, the port concept specifies which interfaces a component implements and how many connections are allowed simultaneously. A major advantage of this restriction is that with this additional knowledge about the usage of the component, the developer can make more accurate QoS statements about the services the component delivers. Hence, it is required that DRACO enforces both of these restrictions. In SEESCOA, three types of ports exist:

Single Port: A single port allows for one-on-one communications. In the CCOM Composer tool, this port is represented by a rectangle (figure 1(a)).

Multiport: One multiport of dimension \(n\) is conceptually identical to \(n\) single ports as it allows for \(n\) connectors to be attached simultaneously. Although messages can be sent to the entire multiport as such (in this case it behaves as a multicast port), the intended behaviour of a multiport is to send messages to a specific index. Conceptually, a multiport is analogous to a call center: a connection is granted to a multiport unless it is already involved in its specified maximum of connections. Once connected, conversation is one-to-one. As depicted in figure 1(b), the symbol of a multiport depends on its dimension.

\(^3\)The interface of ports in SEESCOA is specified on 4 levels: syntactic, semantic, synchronization and quality of service. For more information see [3].

3
Multicast Port: A multicast port of dimension \( n \) is a single port that can have \( n \) connectors attached to it. Messages sent to a multicast port are always sent to all connectors attached to it. It is therefore not possible to differentiate between different receivers. Also, a multicast port can never receive messages. The graphical notation of a multicast port is a trapezium (figure 1(c)).

The dimension for both multiports and multicast ports may be \( \infty \).

2.3 Connectors and Contracts

Definition 4 (Connector) A connector represent a functional connection between components. This means that components will send messages to each other using this connector as a kind of tunnel.

When connecting two ports by means of a connector, the CCOM Composer tool checks the compatibility of both ports at design time. However, since foolproof checks are not always possible, it remains important that port compatibility can be checked at runtime.

Definition 5 (Contract) A contract imposes a non-functional constraint (for instance a timing or memory constraint) on a component or on a group of interacting components. Contracts can be associated with components, ports and connectors.

We have decided not to model contracts explicitly in the DRACO runtime. However, if contract checking is desired, an optional monitor can be added (see section 5.2).

Composing components

Two components can only be composed with each other, if their interfaces match. This principle is illustrated in figure 2. A basic static check of interface (port) compatibility is performed by the CCOM Composer tool at system design time.

3 The DRACO Runtime

During the design and implementation of DRACO, following requirements had to be met:
Robust:  DRACO is intended to be used in systems with high reliability demands.

Small:  Since DRACO will be deployed on systems with limited resources, a small footprint is required. The current DRACO implementation is contained in a jar with a size < 65 kB.

Extendible:  To keep the base system very small and lightweight, DRACO must be extendible in order to offer more advanced features when necessary. Extensions that will be implemented for DRACO include Distribution, Live Updates, Contract Monitoring and Resource Management.

Performant:  The implementation will be as efficient as possible and will try to eliminate all unnecessary overhead. DRACO should be at least as performant as the current SEESCOA runtime.

3.1 Architecture overview

The DRACO architecture is shown in figure 3. Its core system consists of 6 modules. Each of these modules implements a strict interface by inheriting from fixed base classes. At startup, the core is dynamically constructed using the builder pattern ([7]). Since the builder reads an XML file describing which implementation to use for each of the core modules, modifying or replacing one core module has no impact whatsoever on the rest of the system. The ability to easily customize its core makes DRACO an excellent platform for various types of research (e.g. replacing the scheduler would allow us to investigate the influence of the scheduling algorithm on the execution of a component based application, . . .). Once instantiated, however, the core is considered to be fixed. In order to keep the complexity (and size) of DRACO sufficiently low, no attempt was made to allow for unanticipated modifications of the DRACO core at runtime.

Component Manager:  The component manager is responsible for creating components and ports. It maintains references to all component instances in the component system⁴ and will allow for queries to retrieve components that implement a specific component blueprint.

Scheduler:  The scheduler is responsible for scheduling the messages for delivery. The scheduler must make sure that the order of messages are preserved for each port (i.e. if a message a is sent before

⁴ In a distributed context, each node will run the entire DRACO system consisting of the six core modules. Therefore, the component manager is only aware of components in his own node.
message \( b \), \( a \) will be scheduled for execution before \( b \). In a distributed context, there is always exactly one scheduler available for each component system. The scheduler is also the only entity in the component architecture that deals with threads. It is the responsibility of the scheduler to make sure that no more than one execution thread accesses a component at any moment in time.

Message Manager: The message manager is the module responsible for the delivery of the messages. It will interact with the connector manager to retrieve the connector associated with the port from which the message originates.

Connector Manager: The connector manager is responsible for creating and maintaining the connectors between component ports. It also maintains the message handlers that are associated with these connectors and that guide the message delivery.

**DRACO control interface (DCI):** The DCI is responsible for interfacing the DRACO system with the outside world (e.g. to load new components, to start or stop applications, …). It provides a clean and complete API that can be used to interact with the component system itself. This API is then used by a shell that takes care of the actual interaction with the user. By separating the user interaction from DRACO, it is possible to use different interaction shells depending on the situation. A graphical shell can be used on a high performance desktop, while a thin layer with minimal functionality can be used when resources consumption is an issue. The shell is a separate jar which is provided to DRACO at startup.

Module Manager: This core module is responsible for extending the DRACO component runtime with extra functionality that may not always be required: extension modules. These extension modules can be loaded and unloaded at runtime. A few examples of extension modules that can be inserted in DRACO are a Distribution module, a Dynamic Updating module, a Contract Monitoring module and a Visualization module. Since the exact tasks and thus requirements of extension modules are unknown in advance, they can influence the message flow at various points (see section 3.2). Also, extension modules can register themselves with the DRACO core modules to receive notification of a number of events.

To allow for application components to interact with the DRACO runtime environment, each of the core modules provides a (limited) component interface. These interfaces are multiports to which component ports can be connected and information can be retrieved. To separate the internal workings of DRACO from application development, this interface is deliberately kept limited. It is used to provide a variety of services to components (e.g. timer functionality).

### 3.2 Journey of a message

In DRACO, messages are sent asynchronously between components. In order to send a message from one component's port to another, several steps are required: the destination port must be determined, the message needs to be scheduled for delivery and finally it must be executed. In some cases (e.g. in a distributed context when messages pass system boundaries) additional steps are required. This path which is followed by a message is called the message chain.

It is clear that this chain should be kept short and that the steps constituting the chain should be implemented in an efficient way. However, extension modules should be able to subscribe themselves to
Figure 3: Overview of the DRACO architecture

various events of this message chain (e.g. the delivery of a message). In addition, they should also be able to control this message chain and interact with the delivery of messages in order to implement a variety of features as live updates, distribution, resource management, contract monitoring and others. For example, in order to replace a component at runtime, the dynamic update module will need to freeze the incoming flow of messages to a component in order to shut it down for replacement (see section 5.1). In DRACO, this is realized with message handlers.

The path followed by a message traveling from component A to component B consists of 3 major parts: the sending message chain, the scheduler and the receiving message chain. A schematic overview is shown in figure 4.

Sending message chain

The first part of the journey begins when a component sends a message through one of its ports and ends when the message is passed on to the scheduler for delivery. As can be seen on figure 4, the component passes on the message to its port $P_A$. The message is then handed over to the message handler associated with the connector attached to the port. In the most simple case, this message handler immediately passes on the message to the scheduler for delivery. However, in more complex scenarios, additional message handlers can be introduced by core or extension modules in order to intercept the sending of messages. For instance, there could be a sending message handler for the Contract Management module to time stamp the messages exchanged between components (see section 5.2). Message handlers are linked to form a queue, such that messages will be passed on from the first to the last handler. The
last handler is responsible for delivering the message to the scheduler.

Using the six core modules, this message chain is implemented in 6 steps which are shown in figure

**Component → Port:** When a component needs to send a message, it does so by contacting the port through which the message will be sent. Since ports are implemented as inner classes (see section 4) this is achieved using a local call (see arrow 1).

**Port → Message Manager (core):** The port delivers the message to the message manager (arrow 2), which will retrieve the connector that is associated with the given port from the connector manager (arrow 3). For every connector, the connector manager manages 2 message handlers for each direction (from component A to component B and vice versa): a sending message handler and a receiving message handler. The receiving message handler will be used for the delivery of the message after it has been scheduled for execution by the scheduler. The sending message handler that is being retrieved is the first handler of the sending chain shown in figure 4. As such, the message manager will then pass on the message (with the receiving message handler) to the sending message handler (arrow 4).

**Message Manager → [Send Message Handler]**↑: All the message handlers can inspect or modify the messages, and will then pass it on to the next handler in the chain (arrow 5). The last message handler will eventually hand over the message to the scheduler (arrow 6).
Receiving message chain

As mentioned in the previous section, the scheduler receives two objects: the message to be scheduled and the receiving message handler associated with the current connector. The scheduler will queue the message until it is ready for execution. The number of queues used by the scheduler depends on the number of threads that were specified during the startup of DRACO. In a multi-queued system, it is the responsibility of the scheduler to ensure that the order of messages over a specific connector is preserved. Therefore, messages that are being send over the same connector will always be placed in the same queue.

When the scheduler has selected a message for delivery, it allocates a thread for the execution of this message, and passes on the message to the receiving message handler that is associated with the message. The principle behind the receiving sequence is identical to the sending sequence: there is a chain of receiving message handlers that process the message (e.g. the timing monitor can read out the time stamp added to the message by his peer in the sending message chain) and subsequently pass it on to the next handler in the chain. The last handler delivers the message to the port at the end of the connector. This port will then dispatch the message to the actual method associated to the message.

When the message execution has been completed, control is returned to the scheduler. From this moment the thread that performed the message processing can be reallocated to another message or be suspended, depending on the actual implementation of the scheduler.

4 Implementing Components

As shown in the previous sections, the internal structure of DRACO is built for extendibility, and the concept of message handlers may not be the most intuitive to work with. Fortunately, when developing components, no knowledge of this structure is required whatsoever. In this section we illustrate the development of a component. The example we develop here is intentionally kept very simple to focus on specific issues concerning component development for DRACO.

We will work out a small application consisting of two components: a NumberGenerator and a NumberDisplay. The first component will generate a random number on request and will broadcast it to all interested parties. It has two ports: a MulticastPort Out through which it will output its
generated number and a PortGroup Control through which he will receive all requests. The second component prints out all values it receives through its Input port.

4.1 SEESCOA syntax for components

The implementation of these two components is shown in figure 6. The syntax consists of standard Java code, enhanced with a few extra keywords. The component keyword indicates the start of a component implementation. It consists of a number of variables (e.g. $rnd in 6(a)), a number of methods (not shown for this trivial component) and the description of its ports. The declaration of a multicast port is straightforward since it can not accept messages. It suffices to specify its existence using the multicastport keyword. Two parameters are required: the name of the port and the maximum number of simultaneous connections that are allowed. A port group has a similar declaration, but since it can accept messages, these messages must be declared as well using the message keyword. Inside the definition of a message, the code to be executed when this message is received on the port must be implemented.

New messages can be created using the statement message varName = messageName. After its creation, this message can be sent out through any connected port. If the component on the other side of the connector does not accept messageName, the system responds at runtime with a CannotDeliverMessage.

In addition, two operators have been added: (:: and ..). The operator :: is used to access fields of a particular message. messageName::fieldName = "Hello" will assign the string Hello to a field called fieldName in the message messageName. Retrieval is similar: String someString = (String) messageName::fieldName will assign the content of fieldName to a newly created String variable. The .. operator is used on a port to send out a given message. Inside the implementation body for a message, the implicit parameter $inMessage refers to the received message.
4.2 Mapping to Java

As shown in figure 6, the SEESCOA syntax offers a short and powerful notation to express component interactions. Using a preprocessor, this syntax is converted into standard Java. The result after conversion is shown in figure 7. As can be seen in these code snippets, the translation is straightforward. In Java, each port is implemented as an inner class. This has a number of important advantages:

1. Messages are defined for each port. Therefore, DRACO can enforce that messages can only be sent to ports that accept these messages.

2. The use of inner classes allows for the implementation of messages to use all the variables or internal structures of the component.

3. All ports of a component and all their messages are defined in the same place, giving a good overview of the functionality of the component.

The preprocessor is also responsible for adding the constructor of the component and registering the different ports with DRACO (see figure 7).

5 Extending DRACO

To illustrate the power of message handlers, two extensions will be worked out in more detail. In section 5.1, we will describe a dynamic update module that will make use of message handlers to freeze a component before it is replaced. In section 5.2, a timing monitor is presented that uses the handlers to insert time probes.
5.1 Dynamic Updating of Applications

To safely replace a component at runtime, following steps are typically required:

1. The component is put in an inactive\(^5\) state.
2. The new component blueprint version is instantiated.
3. The internal state of the old version is transferred to the new version.
4. The connectors are rewired to the new component version.
5. The new component is activated.
6. The old component is removed.

Suppose we have the initial situation depicted in figure 4 and we wish to replace component \(B\) with a new version. The user will first instruct the system to load the DU (Dynamic Update) module. This module will then interact with the receiving message chains for all the ports of component \(B\) (we assume only a single port on component \(B\) for brevity). In order to freeze component \(B\), it will set itself as the receiving delivery message handler for component \(B\) (the DU Module will store the current delivery handler to restore the situation when the update has been completed) and it will send a message Freeze to the component. The situation after loading the Dynamic Updating module is shown on figure 8. When the update is initiated, a number of messages may be present in the scheduler. In figure 8, the green messages are intended for one of the ports of \(B\). Blue messages are intended for other components and will not be seen by the DU Module as they have different message handlers. This reduces unnecessary overhead. The red message is the Freeze message sent by the DU Module itself.

The green (and red) messages are intercepted by the DU delivery handler as they pass through the message handler chain. As they are intercepted, the following actions are taken:

1. If the message is not the Freeze message that was sent by the DU Module itself, it is just passed on to the component for execution. Afterwards, the control is returned to the scheduler. In this case, the dynamic updating module performs exactly the same functionality as the original default delivery handler.

2. If the message is the Freeze message that was sent by the dynamic updating module at load time, it will pass on this message to the component for execution. Afterwards, however, it will not return control to the scheduler. Instead, it will first perform an update as described in the following section.

The update process itself

The steps discussed earlier are then executed:

\(^{5}\)More accurately, it must achieve quiescence (see [8]). Since the focus of this paper is the use of message handlers in Draco, an in depth explanation of dynamic update techniques is beyond the scope of this paper.
1. The component is already in inactive state after the Freeze message has been executed. Note that if the component does not support the Freeze message, a CanNotDeliverMessage response is returned to the DU Module. Regardless of whether the component supports the message however, it is indeed in inactive⁶ state, since the DU Delivery Handler is blocking all future requests to this component.

2. The DU Module will then create an instance of the new component version, by contacting the Component Manager. At the same time, the ports of this component are created as well. A reference to the Component Manager is received through the Module Manager. This creation is done through a direct synchronous call. A reference to the new component version is given to the DU module.

3. The old version of component B is passed on to the newly created component. At runtime, it is assumed that the new component version contains the necessary code to import the internal state of the previous component version (for information on how this can be achieved we refer interested readers to [10]).

4. The DU Module contacts the Connector Manager and updates the connectors for each of these ports of Component B.

5. Finally, after a possible but not required transition period (which may prove useful if rollback

⁶If the component is involved in transactions, achieving a quiescent state is more complex. In this case, techniques as those described in [9] can be used.
functionality would be required), in which the DU Module can continue to inspect messages that are sent to component B, the DU Module requests that the component manager removes the previous component version and its ports from the system to free resources. It then concludes by removing itself from the system by reinstating the DeliveryHandler it replaced.

5.2 Monitoring Timing Constraints

In this section, a second example that illustrates the power of message handlers is worked out: an extension module to monitor timing contracts.

The contract concept enables the specification and verification of non-functional properties. Contracts can be attached to components, their ports or the connectors between them. Key idea is that a designer can select from a set of predefined contract templates and attach these to a component model in order to impose a particular non-functional constraint. Once a contract has been added to a model, it can be verified by means of a contract verification algorithm that is part of an underlying monitoring system. This monitoring system will be implemented in DRACO as an optional module, namely the Contract Monitor module.

One particular contract type is the timing contract; it lets the designer specify timing constraints imposed on the interactions among components. Timing contracts are attached to connectors since the connector is the construct that represents such an interaction.

Timing Contract Concepts

Timing contracts are based on two important concepts, namely hooks and hook occurrences. Both refer to a particular point of interest on the extended MSC\(^7\) associated to a port or connector. A hook refers to an action occurring in an MSC and exists in three types: send hooks, receive hooks and eoA (end-of-activation) hooks. A send hook corresponds to the sending of a message, a receive hook to the receiving of a message and an end-of-activation hook refers to the end of processing of a message. Since an extended MSC can contain loop blocks, alternative blocks and optional blocks, one also needs a way to specify the occurrence number of a particular hook. In this paper we will introduce a simplified notation for both concepts: \([\text{ComponentName}] . [\text{PortName}] . [\text{MessageName}] . [\text{HookType}]\) represents a hook, whereas \([\text{ComponentName}] . [\text{PortName}] . [\text{MessageName}] . [\text{HookType}] . [\text{OccurrenceNumber}]\) represents the \(\text{OccurrenceNumber}^{th}\) occurrence of the hook. \(\text{OccurrenceNumber}\) is also allowed to be ALL, which refers to all occurrences of that particular hook. More information on this notation and its semantics can be found in [11].

Timing Contracts: DeadlineTC en PeriodicityTC

Two subtypes of a timing contract have been worked out: a deadline timing contract and a periodicity timing contract. A deadline contract imposes a deadline on the occurrence of two hooks, while a periodicity contract states that a particular hook has to occur periodically (each occurrence in its associated period). Both timing contract types have their corresponding runtime verification algorithms.

\(^7\) An MSC (Message Sequence Chart) represents the interactions among communicating entities. In SEESCOA extended MSC's are used to describe the communication protocol of ports and connectors. An extended MSC also provides support for modeling loops and alternative/optional branches.
Event Gathering and Contract Monitoring

In order to monitor contracts, and more in particular timing contracts, a distinction has been made between the gathering of events and the monitoring of contracts pertaining to them. In the case of timing contracts, one is particularly interested in hook occurrence events. Since hook occurrences correspond to message interactions, the monitoring system needs to intercept the messages exchanged among the components. Once intercepted, every event needs to be timestamped and then this information is sent to the contract monitoring subsystem.

**Event gathering:** Intercepting hook occurrence events is done through the insertion of specific message handlers (time probes) in the send and receive message chains associated to connected ports. Each time a particular message is delivered into the chain, the time probe will intercept the message. It then extracts information that uniquely identifies the event: the component and port that sent the message, the message name and the status of the message (send, receive or eoa). Finally, the time probe samples the system time and attaches it to the event information. The probe then puts this information in a non-blocking data structure that is read out by the contract monitoring subsystem.

It is important to note that the use of time probes also influences the timing behaviour of the application. Therefore, this probe needs to perform its computations in constant time and in a non-blocking fashion.

**Contract monitoring:** The contract monitoring system is responsible for collecting events coming from the various probes. These events are then sent to the contract verification algorithms that are registered within the monitoring system. For each type of contract there is a corresponding contract verification algorithm. Currently, two such algorithms have been developed, one for each type of timing contract.

Contract violations are logged in a file, which can be analyzed after the execution of an application. In a future release of the Contract Monitor module, we intend to include online violation feedback functionality. This will enable a DRACO application to reflect and adapt to non-functional constraint violations of its constituting components.

**Contract Monitor Module**

The Contract Monitor module is responsible for monitoring a DRACO application. It contains the necessary functionality for event gathering and contract monitoring, and has to be loaded before the startup of a DRACO application. Initially, the Monitoring module reads in a *probe file* and a *contract file*.

The probe file contains \([\text{ComponentName}].[\text{PortName}].[\text{MessageName}].[\text{HookType}].[\text{OccurrenceNumber}]\) entries indicating which events are to be monitored. The contract file contains information about deadline and periodicity contracts that have to be monitored, including references to the involved probes.

The Monitor module uses several reflection and interception points, that are built into the DRACO component system, in order to perform its activities. It registers itself with the Connector Manager since it has to be notified when a connector is created to which a contract has been attached. If such a connector has been created, the Monitoring module will insert the necessary time probes in the message...
chains associated to the connected ports. Additionally, the Monitoring module notifies its monitoring subsystem that a new contract needs to be monitored.

6 Related Work

Other interesting approaches on the use of reusable components in embedded systems can be found in [12–14]. Although they have some additional application-domain specific features, each of these systems is based on the wiring of independent components. In PECOS, focus is on the use of components in heavily resource constrained devices. Components communicate through ports that are implemented as shared variables. The PECOS component model ([12]) is strongly formalized (including the behaviour of a component), and is intended for use in hard real-time systems. Both in DESS ([13]) as in Koala (developed by the EPSRC and Philips to enable the reuse of components in consumer electronics [14]) components have provided and required interfaces. DESS interfaces are denoted by means of the lollipop notation without a specific port concept whereas koala components specify their interfaces using an Interface Definition Language. Both DESS and Koala allow components to be decomposed into connected subcomponents thus supporting the construction of hierarchical components.

With Conus, Kramer and Magee pioneered in the field of component-oriented evolution. In [8] they introduced the concept of quiescent nodes for safe removal of a component from the running system. Using provisions available in the Chorus Operating system, Hauptmann and Waisel achieve deterministic timing behaviour during updates ([15]). Just like the SEESCOA component system, it uses the concept of ports to reroute messages between objects. A very flexible approach can be found in meta-architectures (e.g. [16,17]). Through reification of object-oriented concepts (class, method-call, . . . ), a meta-model is built on top of the application. By changing this meta-model, structural changes of the application are possible. Other systems focus specifically on dynamic Java. In [18] the default Java class loader is extended so that class definitions can be replaced and objects or dependent classes can be updated. More theoretical work on dynamic updating is done by Gupta ([19]) and Hicks ([20]). A complete survey of this domain is beyond the scope of this paper (we refer to [19–22] for more complete overviews).

A lot of research has already been done on the specification and static verification of timing constraints (examples are Real-Time Logic [23] and timed MSC’s [24]). Less efforts have been done till now concerning the dynamic verification of constraints. One particularly interesting approach is the one used by [25] and [26], in which RTL formulas are monitored at runtime by a generic verification algorithm. RTL is a powerful formalism, but not all RTL formulas can be monitored efficiently at runtime. Our approach is not based on a formalism like RTL; instead a template-based approach is used in combination with efficient contract-specific verification algorithms.

7 Future work

Although the DRACO runtime system is fully implemented, the preprocessor is not. Implementation of this preprocessor will receive high priority in the near future, since development of components is dramatically simplified using the SEESCOA constructs. Having a fully functional preprocessor at our disposal also allows us to port more complex existing component oriented software (e.g. the camera surveillance system [6]) to DRACO with minimal effort. These more complex cases will then allow us
to further evaluate the performance of DRACO in real life situations.

Next to the continuing work on the runtime environment itself, DRACO will particularly be used as a base platform for further research on component oriented development for embedded systems. Issues that will be investigated are state transfer during a live update of a component, performance and resource monitoring and component mobility. As such, many extension modules (among which the two proposed modules, but also a distribution module for instance) will be implemented. Additional complications may need to be resolved when different extension modules are being used at the same time (e.g. updating a component in the distributed case, ...).

8 Conclusion

In this paper we have introduced DRACO, a modular component runtime environment intended for embedded devices which adheres to the SEESCOA component methodology. First, a detailed overview of the DRACO architecture was given and the 6 core modules were discussed in detail. In order to preserve its small footprint, DRACO can be extended with external modules that implement additional features as distribution, runtime contract monitoring or dynamic updating. These extension modules can influence with the message delivery process using message handlers. The power of this technique has been illustrated with two example modules that have been worked out in detail. In addition, a trivial component was developed to illustrate the SEESCOA component constructs and a mapping from the SEESCOA language to Java was worked out.

References


Evolution Management and Process for Real-Time Embedded Software Systems

CRuMB Design

Deliverable D.2.4-D.2.5 Appendix B
Written by Joris Gorinsek, Andrew Wils & Stefan Van Baelen, K.U.Leuven,
Edited by Stefan Van Baelen, K.U.Leuven

12 December 2003
Version 1.0
Status final
Public Version
This document is part of the work of the EUREKA Σ! 2023 – ITEA 00103 project EMPRESS. Copyright © 2002-2003 EMPRESS consortium.

Authors/Partners:

<table>
<thead>
<tr>
<th>Partner</th>
<th>Author</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>K.U.Leuven</td>
<td>Stefan Van Baelen</td>
<td><a href="mailto:Stefan.VanBaelen@cs.kuleuven.ac.be">Stefan.VanBaelen@cs.kuleuven.ac.be</a></td>
</tr>
<tr>
<td></td>
<td>Joris Gorinsek</td>
<td><a href="mailto:Joris.Gorinsek@cs.kuleuven.ac.be">Joris.Gorinsek@cs.kuleuven.ac.be</a></td>
</tr>
<tr>
<td></td>
<td>Andrew Wils</td>
<td><a href="mailto:Andrew.Wils@cs.kuleuven.ac.be">Andrew.Wils@cs.kuleuven.ac.be</a></td>
</tr>
</tbody>
</table>

Document History:

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Editor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Dec 2003</td>
<td>1.0</td>
<td>S. Van Baelen, K.U.Leuven</td>
<td>Public version based on internal version 1.4</td>
</tr>
</tbody>
</table>

Filename: D2.4_D2.5_v1.0_Appendix_B_Public_Version.doc
1 Introduction

1.1 Problem Description

Embedded hardware devices are getting multi-modal and multi-functional. MP3 players and mobile phones are turning into PDA’s and vice versa. However, the resources of these devices remain limited, and for flexible software platforms to operate safely and robustly, they must be resource-aware. Moreover, runtime evolution of resource-critical services relies even more on the resource knowledge of the underlying platform. Enter CRuMB, a Component-based Resource Monitor and Broker for Ubiquitous computing systems.

CRuMB has been designed to complement current and new component based middleware systems, with the following two goals in mind:

add robust resource-aware behaviour to component based systems: Dynamic component systems cannot predict up front what component configurations will be formed and if the system is able to accommodate enough resources. CRuMB ensures that every configuration is a safe one. Each component’s QoS (Quality of Service) constraints are made explicit and guaranteed by the underlying system

enable run-time evolution of QoS behaviour: a component’s QoS constraints should be flexible in order to maximise the number of services the hardware can run, while keeping an adequate QoS for each one

Current state of the art component based systems either offer no safe and fine-grained resource management or no safe dynamic evolution. Indeed, the embedded systems industry is still sceptical about anything that is dynamic, and research is only slowly moving towards more dynamic real-time systems.

CRuMB’s modular design allows the use of pluggable modules such as the Dynamic updating module which enables the robust run-time evolution of software components. Chapter 6 describes a component middleware extension for dynamic life-cycle management of components. With CRuMB, one can deploy, remove and update components at runtime without interfering with global system behaviour. These structural reconfiguration actions allow changes to the following:

- Location of components: services can be moved from device to device to follow its user
- Implementation of components: bug-fixes and enhancements can be installed at runtime
- How components interact, messages or RPCs

This allows a long lived service to maintain its availability despite changes in its execution environment

1.2 Target

CRuMB targets embedded and soft-realtime systems that have adequate resources to serve as a component platform. These systems act as a server for services that are packaged in loosely coupled components. Components are built to be easily migrated, interchanged and combined. Contemporary PDA’s, residential gateways, desktop PC’s as well as future ubiquitous computing devices are potential candidate component servers.

Amongst the resource requirements services (and thus components) pose are:
**Timing:** certain code fragments need to be executed within a certain time frame. Typically, this involves a single, often recurring task for a service, e.g. an engine control loop or video stream decoding

**Memory:** memory is needed to accommodate code, data and runtime variables.

**Bandwidth:** a recurring task can need to reserve a portion of the aggregate bandwidth of the system.

**Storage:** the ability to write data to persistent media at a certain rate.

**Power:** a side-effect of must resources is the battery drain they cause

**Cost:** another side-effect, e.g. electricity or bandwidth costs

The 3 resources CRuMB is aiming at most are CPU power, memory and bandwidth. We find that these constraints are the most difficult to tackle, yet few efforts exist to create general broker for them. On the other hand, CRuMB also endeavours to have its support for resource constraints to be easily plugged in. This way, we want to gradually extend CRuMB to deal with more types of resources, and offer an API for others to do the same.
2 Scenarios of use

We are convinced that the use of resource contracts greatly contributes to the flexibility and robustness of many types of software. To illustrate this, we present a number of example applications and scenarios in which CRuMB would increase the overall user satisfaction.

2.1 A graphic intensive application

Multimedia services are becoming more and more important in portable devices. User interaction is increasingly done using CPU-intensive, yet bandwidth-friendly vector graphics. An example of such a protocol is Macromedia Flash. This pseudo-standard can bring down even high-performant desktop PC’s; if configured to use all anti-aliasing and other techniques. A service on an embedded device wishing to display such graphics, should instead negotiate a resource contract in which a quality level is fixed. E.g. the animation could be fullscreen or not, the lines could be anti-aliased, and so on. Depending on the policy CRuMB uses, the system can automatically decide which services should be prioritised. This way, the user is always enjoying the highest quality level on his system.

2.2 A resource aware spelling checker

This example shows how non-realtime applications could also benefit from resource contracts. Editing documents is in itself not a very resource-consuming task. Adding intelligent behaviour to a word processor however, is. An example of such a resource-consuming task is the spelling or grammar checker. This activity often runs as some sort of background process in modern word processors. Desktop-sized systems usually have the computing power required to do real-time spelling checking. Even then however, the user could load a large document, upon which the checker starts analysing the entire document, reserving resources and reducing the responsiveness of other applications and tasks.

The bottom line here, is that the importance of the spelling checker needs to be established, and its priority and behaviour should be adjusted according to the importance and available resources. E.g. the spelling checker could be configured to do run-time checking at startup. Addition of new services could force the checker to run at less-than-real-time speed, or even to suspend its operations entirely. Naturally, when the user requires a spelling check, or as more resources get freed, the system is configured to allocate more resources to the checker.

2.3 Automatic Security Updates

Users are increasingly confronted with leaks and security bugs in their software. Every week, at least one component of a desktop system needs to be replaced to fix these, often requiring a reboot of the system. For a user of a ubiquitous device, often operating in “hostile” environments, these security fixes are at least as important as they are for a normal desktop user. Of course, that user would expect no or minimal change in the device’s behaviour while installing security fixes. Using CRuMB, this update can be negotiated and scheduled at a convenient moment. Some services may have to decrease their quality level when the update is installing, yet the user suffers no real inconvenience.
3 Resource declarations and contracts

The primary use of resource declarations and contracts is to quantitatively measure and guarantee a component’s quality of service. CRuMB requires components to be resource-aware, which means they know about the resources they require. More specifically, CRuMB requires components to declare the nature, quantity and point in time particular resources are needed.

To effectively describe resources in an abstract and platform-dependent manner, we propose the use of resource pools, each pool representing the availability of one particular resource. When declaring resource usage up-front, we will differentiate between the declarations and reservations of resources. Declarations describe the resource usage of the component’s interface methods, whereas reservations are negotiated agreements about the component’s requirements.

3.1 Declaring Resource Usage

In order to guarantee that each components gets the resources they need, CRuMB needs the following information from them:

- what resources are needed if a component is used e.g. the method x takes 50 ms of CPU time to complete (declarations)
- when components will be used e.g. method x will or can be executed every 100 ms (constraints)
- how is a component used e.g. method x needs to be executed within 70 ms (reservations)

Resource declarations describe the resource requirements of all interfaces the component exports. This information is necessary when other components want to make use of the interfaces. To encourage low-coupling among components, a component needs not provide resource declarations for the components it uses. However, it will need to include which and how much their interface methods will use those components. To illustrate this, consider figure 1. It shows how resource use depends on a chain of resource declarations. The example depicted is an application opening an audio network stream: it requests frames from the network, and decodes them. Network delays are ignored, and the frame size is considered the same as the packet size. Each component provides as much resource usage information as possible, split up in “internal” use figures (e.g. the network use in the fetchPacket method), and external calls. If CRuMB knows that openStream calls both decodeFrame and fetchPacket methods, it can infer the size of the CPU and network resource “slots” needed to schedule the periodic openStream method.

The described resource declarations are formalised in 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 are a form of information towards the component system; they will enable us to calculate the resource usage of each call. The information in figure 1 captures the information in these contracts in an informal way. Note that this information can and should be mostly automatically inferred from the design artefacts and benchmarks.
3.2 Describing Reservations

Next, we can establish concrete agreements on how the component is to be used in the future. If a system is to guarantee correct QoS behaviour for all its components, it needs to know up-front what to expect. Until a contract is agreed on, no guarantees can be made. E.g., a video stream component can be asked to download a video at a rate of 300Kb/s. If a contract is present for such a stream, the download can proceed. If not, a new contract can be negotiated; this negotiation may fail. A contract, however, can be made as early as deployment time. That way the system can reserve the appropriate resources beforehand.

One can split up component use resource costs in:

Deployment, initialisation and default operating costs. Resources are needed to load the component, negotiate interfaces, and start and initialise 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. Examples of typical use could be:

- a one shot action
- guarantees for an incoming call (reserve-ahead)
- a periodical call

Costs per additional use. These unanticipated requests can seriously destabilise a carefully tuned component configuration; hence, they are scheduled and admitted depending on available resources on a per-call basis.

A (component use) contract describes 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.
Consider the earlier discussed example. Figure 2 shows an example contract proposal for using the Streamer component (it aims at getting 20 frames/second done). On a system with a 5MBit/s bandwidth without any load, the highest quality level can easily be selected; it would only take 4ms network time and 10ms CPU time, the (worst-case) sum of which lies far below the 50ms limit. Depending on the load of the system, the system capabilities (e.g. another CPU and/or bandwidth) and policy of CRuMB however, the contract may or may not be approved, or a lower quality level may be selected. If necessary, the client can offer a new proposal, restarting the negotiation.

![Component use: streamer.openStream Periodic(50ms) { streamer.getFrame } streamer.closeStream](image)

Figure 2: An example of a component use contract

### 3.3 Negotiating Contracts

As mentioned above, it is the component use contract that is subject to negotiation. To offer a adequate level of flexibility, the following mechanisms are used during the negotiation process:

- a component can offer a propositional contract to the system. In case the contract is refused, a new contract can be offered. This enables a component to consider other algorithms (e.g. consuming less resources or favouring other ones).
- to minimise the number of negotiation steps, a component may also specify its constraints in ranges: e.g. the deadline of a message is preferably 20 ms, but the information is still useful after 50 ms.

### 3.4 Dealing with non-contracted components

In a resource-aware system, it is essential to “sand-box” non-resource-aware components, to keep them from disturbing the carefully negotiated resource constraints. Normally, they would only be allocated resources when others do not need them. It could be wise to split the resource pools, meaning both parties could benefit from a fixed set of e.g. half the resources, where the resource-aware party enjoys all earlier mentioned benefits. Typically, the resource-aware components will be too important to give them only a part of the available resources.
4 CRuMB architecture

CRuMB is designed to be modular and dynamically pluggable into existing component systems. This modular design allows for easy integration into new component systems and enables the application developer to fine-tune the component middleware for his application by using only those parts of CRuMB he requires. Figure 3 shows how CRuMB is situated in a more general system architecture.

![Diagram of CRuMB architecture](image)

**Figure 3: General overview of a system running CRuMB**

Due to the nature of system resources, CRuMB may need access to the HAL (Hardware Abstraction Layer) and OS levels of the system. Therefore, it sits alongside with the component middleware, instead of running on top of it. CRuMB also provides an extra layer between the component middleware and the component instances, yet it does not require all components to be aware of CRuMB. Indeed, non-resource aware components can still function with CRuMB running. Because of this positioning, CRuMB can negotiate and interface directly with the resource-aware components while minimising interference with, and adaption of the existing component middleware and components.

In figure 4 we zoom in to a general overview of CRuMB’s main components.

Depending on the requirements asked of CRuMB, not all internal components may be installed. In fact, only the contract manager and resource abstraction layer are mandatory. The contract broker, responsible for negotiating new contracts, can be left out in systems.
where all contracts can be established at design time. The contract monitor, may be left out to speed up the execution of trusted and thoroughly tested components. Finally, the contract scheduler is not needed when using a fixed, static scheduling strategy. This flexibility with regard to the configuration of CRuMB allows the embedded software developer to tune his resource-aware component system to his needs.

The following section provides a more detailed overview of CRuMB’s key components.
Figure 4: General overview of a system running CRuMB (detail)
5 Tasks

CRuMB’s core architectural building blocks all deal with resource-awareness and are described in section 5.1. They are complemented with descriptions of what we expect of component middleware platforms in section 5.2.

5.1 Resource related components

5.1.1 Resource Abstraction Layer

This component provides a resource abstraction layer: it reifies resources and monitors their availability. The resource abstraction layer can be queried for information on at least three types of resources: CPU time, memory and bandwidth usage. Also, it can inform other components of changes in the resource capabilities. Because of the diverse nature of resources typically available in an embedded system it is not feasible to monitor all resources at once in one “generic” resource abstraction layer. Therefore we will implement a different resource abstraction layer subcomponents for each type of resource we handle.

A resource abstraction layer typically hooks itself into the messaging system to be able to account when resources get consumed. However, the resource abstraction layer itself should not present a significant resource hog. In fact, it can be used without the contract monitor, only serving information about resource capacities and resource changes.

The resource abstraction layer contains subcomponents for each type of resource.

5.1.2 Contract Monitor

The contract monitor’s task is to verify that components don’t use more resources than allowed by their contract. If a component violates its resource contract, appropriate actions are taken depending on the used policy. Appropriate actions could be: stop component, inform messaging system, increase or decrease component priority, … To detect violations, the contract monitor can cooperate closely with the Message Scheduler, to detect which message violates a certain contract; or with the resource abstraction layer to retrieve information on the actual resource usage a thread or process.

The contract monitor is resource-specific: we distinguish at least three different subcomponents:

- **TimingContractMonitor**: This kind of monitor continuously checks the temporal properties of the running application. It detects contract violations when components spend more time using the CPU than specified in its contract. It also detects when the combined load on the CPU reaches a certain upper bound (this is used to ensure that the component runtime system and the monitoring have a reserved quota of CPU time to do their thing).

- **MemoryContractMonitor**: Monitors the memory usage of each component and detects a violation when a component requests more memory than it is allowed to use. This can be a combination of heap memory, reserved memory, etc.

- **BandwidthContractMonitor**: Monitors how much bandwidth a certain component is using. Again, violations of resource contracts will be detected.

The ContractMonitor relies on:

- **Contract Manager**: this component provides the ContractMonitor with the necessary data on the amount of resources a certain component is allowed to consume.
5.1.3 Contract Broker

This is one of the most important components in the CRUMB architecture since it allows components to dynamically negotiate and renegotiate their QoS needs in the form of contracts. A negotiation of contracts can be initiated by a component (for instance when it starts missing deadlines and would like to be allowed to use some more processing time), by the runtime system itself (for instance when a change in the component composition requires a reconfiguration in the use of resources), or by the resource abstraction layer (for instance when the available resources itself change).

To accomplish its task, the contract broker makes intense use of the contract manager and contract scheduler to gather the information needed to negotiate about contracts. The contract broker should not know about resource specific data.

The contract broker’s negotiation heavily depends on the used policy. For further details on the negotiation process itself, we refer to section 3.

This component relies on:

**Contract Manager:** The contract broker needs to know which contracts are currently active in the system. This information is provided by the contract manager. Also, the contract broker uses the contract manager to check if a proposed contract is feasible in the current configuration.

**Component Instances:** The contract broker negotiates with each component instance in order to establish or change a resource contract.

5.1.4 Contract Scheduler

The contract scheduler calculates priorities for component messages so that they can be scheduled properly.

Calculating schedules is based on certain heuristics: for the reservation of CPU time we can rely on rate monotonic analysis as a heuristic, for memory and bandwidth reservations we rely on simpler heuristics that do take in account the type of resource being reserved. For instance, when a reservation is requested for Immortal memory, this will be treated differently than a reservation for Scoped memory\(^1\).

Internally, this component is realised in terms of three subcomponents, one for each type of resource we address.

Relies on:

**resource abstraction layer:** The Contract Scheduler queries the resource abstraction layer to gather information about the total available amount of resources. (for instance: it can ask the resource abstraction layer how much memory is available to distribute)

**Contract Manager:** The Contract Scheduler relies on the information about current resource reservations that is provided by the Contract Manager to calculate a global schedule.

\(^{1}\)We use the terminology for memory as defined in the Realtime Specification for Java (RTSJ).
5.1.5 Contract Manager

This component holds a bookkeeping of all contracts that are currently active in the system. Because of this, the contract manager had a total image of theoretical component resource consumption. This is in sharp contrast to the contract monitor, which measures actual component consumption and which has a total image of the resource consumption of the running application at the current point in time.

This component is basically a repository of all contracts that are currently active in the system, with an interface that allows for the execution of queries such as the total reservations for a certain resource or the total reservation of a certain resource by a certain component etc . . .

A second important task is the feasibility analysis of existing and future resource reservations. Given a certain contract, this component will check if the reservations in terms of CPU time, memory and bandwidth are possible by adding up the current reservations of all contracts and checking if the proposed reservation can be safely allowed. To accomplish this, the Contract Manager uses methods similar to those used in the Contract Scheduler. Also, the Contract manager depends on subcomponents for each type of resource. Relies on:

resource abstraction layer: The Contract Manager queries the resource abstraction layer to gather information about the total available amount of resources. (for instance: it can ask the resource abstraction layer how much memory is available to distribute)

5.2 Middleware components

This collection of components provide the functionality which is typically found in standard component middleware. They provide the basic functionality needed to enable the use of components as basic building blocks of a software system: component life-cycle management, component composition management and an inter-component messaging system. We rely on the features provided by these middleware components to implement our resource broker system upon. Note however that we don’t rely on the specific implementation of the basic functionality. This allows us to be flexible with respect to the actual component middleware being used\(^2\).

5.2.1 Configuration Manager

The Component Configuration Manager is responsible for the management of the composition of components which makes up a component based application. The Configuration manager is also the link to the outside world: it provides a shell (or a network port) that is used by the application programmer to perform some of the following operations:

- installation/deployment of a new component
- re-negotiation of expected QoS behaviour
- removal of an existing component
- (live-) updates

\(^2\)This doesn’t mean we aim at developing a system that is plug-and-play compatible with all component middleware out there. Plugging CRuMB into another component middleware will require changes to CRuMB as well as some minor changes to the component middleware, allowing CRuMB to hook into the component middleware.
Upon receipt of a request through this shell (or via the network), the Configuration Manager activates a number of subscribed components (such as the life cycle manager, the dynamic updater, . . .) that check different aspects of the request. These aspects include security, policy, and of course, resource behaviour.

The Configuration Manager is responsible for the component composition. This implies that it has knowledge of all components and interfaces available in the composition. The Configuration Manager provides a Lookup service that matches component names to actual interfaces of component instances. It also keeps track of inter-component connections: which components can exchange messages. The restrictions on the actual communication between those components, however, is not the responsibility of the configuration manager: that information is captured in a contract and is controlled by the contract managing facilities of CRuMB.

Relies on:

**Dynamic Updater:** update requests are forwarded to this component.

**Component Life Cycle Manager:** request for the installation of a new component or the removal of an existing one are sent to this component.

### 5.2.2 Message Scheduler

The messaging system is responsible for the dispatch of all periodic and non-periodic messages exchanged between all user components. To realise this it maintains a pool of threads which are assigned to a certain component when a message has been sent to it.

When a message to a component is sent, it is queued for delivery in the Message Scheduler. Messages can be queued in different queues according to the priority they have been assigned. The Message Scheduler uses a simple priority-based FIFO scheduler to deliver the queued messages according to their priority. After this, delivering a message to a component comes down to selecting a free thread (with the right priority) from the thread pool and assigning it to the destination component for execution. When the message has been executed, the thread is reclaimed and added to the thread pool again.

This mechanism of message dispatching implies that no component will have its own dedicated thread. Note however that this does not imply that there can be no concurrency in the component-based application: several threads can be active at the same time. Section 7.4 provides more information on this.

Relies on:

**Contract Scheduler:** The Message Scheduler uses the message priorities determined by the Contract Scheduler for the timely delivery of messages. However, when no Contract Scheduler is present, the Message Scheduler will still deliver messages, since messages without a set priority are handled as if they have the lowest priority.

### 5.2.3 Component Life-cycle Manager

The Component Life-cycle Manager is responsible for the management of the life-cycle of component instances. This component’s tasks include the loading of new components, the instantiation of new component instances, the connection of interfaces to component instances and the removal of component instances. The component life-cycle manager obeys the commands of the configuration manager (and the dynamic updater in case of a component update request).

This component does not rely on other components to perform its tasks.
5.3 Policies

The reconfiguration actions are all heavily dependent on the general desired behaviour of the system. This behaviour can be captured in policies. These policies determine how CRuMB will reserve or free resources or how the resources in the negotiation step will be assigned to components. Policies therefore again allow the application developer to fine tune the behaviour of CRuMB to his specific needs. CRuMB policies are configured at system startup time, but can be changed at runtime.

We explain the use of policies in CRuMB by illustrating how some of the most important decisions are affected by policies.

5.3.1 Allocation of Resources

One of the main uses of policies is to determine how free resources are allocated. Giving each component as much as it asks will ensure good performance of the deployed components, but can pose problems for future configuration changes. “Playing it safe” on the other hand can affect performance right now, but may produce better results when a component is added in the future. These 2 ideas boil down to the following two resource (re)allocation strategies:

**greedy allocation**: the system either aggressively allocates resources for new services. This is very useful for systems where resource shortage does not occur often or where the configuration is very unlikely to change.

**conservative allocation**: the system always makes sure there is a pool of free resources to accommodate for unexpected changes in configuration.

This policy is mainly enforced by the contract manager.

5.3.2 Reallocation Strategies

When the amount of available resources decreases, contracts will be re-considered and possibly re-negotiated. The policy here depend on the following factors:

- importance of the component
- “absolute” resource consumption (freeing the resources a heavy process consumes could help a lot of smaller ones)
- possibilities of reducing the resource consumption

This helps decide what components to stop or reduce. This policy is mainly enforced by the contract manager.

5.3.3 Violation

Another important issue is how the system deals with contract violations. Depending on the policy a violation may result in a new contract with more or less reserved resources:

**renegotiate contract with less resources**: Since the component has taken more than its share (and has possibly endangered other components in doing so) it might be a good decision to force it to use less resources in order to allow the other components to "catch their breath" again. In the mean time, some or all messages can be thrown away or processed at a low priority.
renegotiate contract with more resources: Instead of killing a component that has taken more than its share one could argue that it is better to let it finish its job as quickly as possible to minimise the impact on the other components and to avoid having to restart the current computation and doing double work. Therefore one might allocate more resources to a component in order to allow it to finish its job.

emergency stop: Upon violation of its contract a component could be stopped completely.

emergency stop of the system: In order to guarantee the safety of the complete system, it could be necessary to bring the whole system to a halt when a component violates its contract.

change implementation: Another policy regarding violating contracts could be to force the component to change its implementation in order to avoid violating the contract again in the future. This could be combined with the "emergency stop" policy.

The policy is enforced primarily by the resource abstraction layer, which will often delegate the violation to the contract broker, if present.

6 CRuMB plug-in: Runtime-Evolution

This chapter shows how middleware extensions such as support for the dynamic updating of components and a security framework can work using CRuMB. A dynamic-update extension is explained, consisting of 2 components: a Dynamic Updater and State Convertor.

6.1 Dynamic Updater

The Dynamic Updater is invoked by the Composition Manager each time an update request is received. When this request is received the dynamic updater checks how much resources the update will consume by inspecting the update contract associated with the new component. It then checks if the update is feasible given the current resource reservations.

If the update is feasible, the Dynamic Updater instructs the message scheduler that the component to be replaced must be made quiescent. When the component has reached a quiescent state, the dynamic updater instructs the component life cycle manager to load the new component and it instructs the state converter to convert the state of the old version to the new version (this can also be done offline). The new version of the component is then initialised using the converted state, the connections are rerouted from the old component to the new version and the life cycle manager is instructed to remove the old component.

If the update is not feasible, the Dynamic Updater asks the Contract Broker to renegotiate the existing contracts with all components to start a graceful degradation of services in order to free up resources. If, after that degradation step, the update still isn’t feasible, it is rejected. Otherwise it is executed as mentioned above.

After successfully updating a component, the Contract Broker is asked to start a renegotiation of all contracts. This is necessary to return the system to a normal state if there had been a graceful degradation of services or if the new component has a different resource consumption profile than the old component version.

Relies on:

Contract Manager: Registers the update contract when it was accepted as feasible.
**Contract Scheduler:** Checks if an update contract is feasible given the current resource reservations.

**Contract Broker:** When graceful degradation is needed, the contract broker is instructed to renegotiate all contracts. Also, after each successful update a renegotiation phase is started through the Contract Broker.

**Component Life Cycle Manager:** Relies on this component to load the new version of the component and remove the old version when the update was completed.

**State Converter:** Uses this component to convert the state of the old component to the new component.

### 6.2 State converter

This component is essential in the dynamic updating module: it allows the new component to pick up where the old component stopped. To ensure that the new component can do so, it must know in what state the old version was when it was replaced. Also, since the new version is most likely changed internally when compared to the old version, it will have a (slightly) different representation of its internal state. Therefore the old state must not only be transferred to the new version, it must also be converted.

To convert the state of an old component to a newer one, we first extract the state of the old component to an abstract state representation (using the getState interface call that each component is obliged to implement), the actual conversion is performed on the abstract state representation and the new version is initialised using the abstract state.

Of course converting states can be a complex and computationally intensive problem. Therefore we allow it to take place offline, on another machine, enabling us to skip the state conversion step and immediately loading the new component version.
7 Scenarios

This section explains how CRuMB deals with what we designate the primitive reconfiguration steps: deployment, removal and reallocation. Finally, we will shortly discuss the scheduling and monitoring of resources.

7.1 Deployment

Before a component is deployed, its resource declarations are passed to CRuMB. Depending on the current configuration, 3 outcomes are possible.

- no contract is possible; the deployment is refused
- the deployment is accepted, a contract is created allocating resources depending on the policy of CRuMB and available resources
- the deployment is accepted; however, other components had their contracts changed to accommodate the new component (albeit temporarily). Again, this depends on the policy.

Let us zoom in on the interactions of a successful negotiation.

1. A request for reconfiguration is made. The Configuration Manager delegates this to the Life-Cycle manager.
2. The Contract Manager has registered itself to validate the requests of the Life-Cycle manager, and is thus called by the latter.
3. The Contract Manager, being notified of the configuration request, checks for a contract validating the request. It cannot find such a request, and instructs the Broker to negotiate one.
5. A contract gets registered and the necessary scheduling parameters are calculated.
6. The configuration action is authorised.
7. The Life-Cycle manager instructs the Message Scheduler to execute a number of deployment messages.

Finally, the component instance is created, connections are created to other components and the Configuration manager is informed of the successful instantiation of the component. When receiving this message, the configuration manager will add the component instance in the component lookup service.

7.2 Removal

Removal of a component causes existing contracts to be re-evaluated. If the policy permits it, freed resources get redistributed.

The following interactions take place:
1. A request for reconfiguration is made. The Configuration Manager delegates this to the Life-Cycle manager.

2. The Contract Manager has registered itself to validate the requests of the Deployer, and is thus called by the latter. The Contract Manager, being notified of the configuration request, checks for a contract validating the request. The contract specified the declarations for the component destructor. The contract is marked as "Ended" (except for a possible clause involving the Destructors), but it is retained for some time.

3. The configuration action is authorised.

4. The deployer instructs the Message Scheduler to execute a number of removal messages.

The component instance is then removed, connections are removed and the Configuration manager is informed of the successful removal of the component upon which the component instance is removed from the component lookup service.

7.3 Reallocation

Reallocation is the process of re-distributing resources over some or all components. There are 2 possible causes for reallocation:

- at the request of a component, e.g. the user demands more of a particular application
- due to environmental changes, e.g. bandwidth is cut off

Whatever the cause, reallocation is a very important reconfiguration step and, essential to keep the system running. If necessary, reallocation triggers a renegotiation of one or more involved contracts, resulting in one or more of the following actions:

- components get more resources
- components have to give up some resources
- components are stopped

It is the responsibility of the contract broker, with the help of a resource policy, to decide upon the proper actions and re-establish the proper contracts.

7.4 Scheduling messages

We assume that all messages between components are handled by the Message Scheduler. Another constraint we put on components is that they should not create threads of their own. Recurring and/or simultaneous actions should be declared in the resource contracts and will be scheduled accordingly. This way the Message Scheduler can assign threads to execute messages as needed (see 5.2.2). The Contract Scheduler uses the Resource Contracts to determine the deadlines of the messages and to advise the Message Scheduler of the proper scheduling priorities.

Components can be easily implemented in a thread-declarative way. We propose an extra “system” interface on each component, which the component only can call. This way, one can separate threads from the functionality. Figure 7 shows a resource-aware component and its interfaces.
The aforementioned scheduling constraints will often force changes upon existing component middleware, as we lift up scheduling from the OS level to be controlled on a higher level. Converse, and more important, we need to make sure that resource-aware components could also be run on a not-resource-aware platform. Two possible ways of dealing with this are:

- Insert a minimal CRuMB implementation on not-resource-aware platform. The advantage here is that CRuMB can make the best decisions about the behaviour of the component.
- Package the components with their own thread, when run on a non-resource-aware platform. The thread-box makes use of the above-mentioned system interface and enables autonomous behaviour of the component.

### 7.5 Monitoring A Resource

The Contract Monitor relies heavily on the Message Scheduler: upon receiving a message to be scheduled and/or the correct execution of a message, the Scheduler informs the Monitor. The latter can check if the message took too long or if the message is not forthcoming from a contract clause.
Figure 5: Deployment of a new component
Figure 7: A resource-aware component