PCOM – A Component System for Pervasive Computing

Christian Becker, Marcus Handte, Gregor Schiele, Kurt Rothermel

Institute for Parallel and Distributed Systems (IPVS)
University of Stuttgart, Germany
{Christian.Becker|Marcus.Handte|Gregor.Schiele|Kurt.Rothermel}@informatik.uni-stuttgart.de

Abstract

Applications in the Pervasive Computing domain are challenged by the dynamism in which their execution environment changes, e.g. due to user mobility. As a result, applications have to adapt to changes regarding their required resources. In this paper we present PCOM, a component system for Pervasive Computing. PCOM offers application programmers a high-level programming abstraction which captures the dependencies between components using contracts. The resulting application architecture is a tree formed by components and their dependencies. PCOM supports automatic adaptation in cases where the execution environment changes to the better or to the worse. User supplied as well as system provided strategies take users out of the control loop while offering flexible adaptation control.

1. Introduction

Pervasive Computing is characterized by the interaction of a multitude of highly heterogeneous devices, ranging from powerful general-purpose servers located in the infrastructure, to tiny mobile sensors, integrated in everyday objects. Devices are connected to each other on-the-fly using wireless communication technologies like Bluetooth, IEEE 802.11 or IrDA and share their functionality. A sensor could for instance use a nearby display to present its data to the user.

Developing and executing applications in such environments is a non-trivial task. Apart from the device heterogeneity, the hardware and software resources, i.e. devices and services, available to an application are highly dynamic, due to factors like user mobility, fluctuating network connectivity or changing physical context. This forces applications to adapt themselves constantly to their ever-changing execution environments. User-interaction, e.g. for adaptation control or administrative tasks, should be minimized, thus removing the user from the control loop [12].

To ease application adaptation, we have developed BASE, a flexible middleware for Pervasive Computing environments (see e.g. [1] for details). It provides adaptation support on the communication level by dynamically (re-) selecting communication protocol stacks, even for currently running interactions.

BASE offers no support for adaptation at higher levels, e.g. by automatically reselecting services and devices. Therefore, we have designed and developed PCOM, a light-weight component system on top of BASE. PCOM allows the specification of distributed applications that are made up of components with explicit dependencies modeled using contracts. An application can be executed if all of its components can be executed – either local or remote – meaning that all dependencies between components can be fulfilled. In order to automatically choose alternatives if multiple suitable components are available, strategies are employed. This allows adaptation without prompting the user. The main contribution of this paper is the definition and evaluation of this light-weight component system for strategy-based adaptation in spontaneously networked Pervasive Computing environments.

The remainder of the paper is structured as follows. Next, we will present our system model and briefly sketch BASE. Models for application adaptation are discussed in section 3. The requirements on application adaptation, especially those that are not fulfilled by BASE, are derived in section 4. Section 5 presents the architecture of PCOM, its application model and the mechanisms that enable adaptation. As an indication for the validity of our approach, an evaluation of PCOM, including a comparison of application adaptation in BASE and PCOM is given in section 6. After discussing related work in section 7, we conclude the paper and provide an outlook on future work in section 8.

2. System Model

Our work focuses on spontaneously networked Pervasive Computing environments in which devices are connected on-the-fly, typically using some kind of wireless technology. Such environments are highly dynamic. Connections between devices are not permanent, the topology of the network is constantly changing, and there is no central or coordinating element. We do not assume the presence of a smart environment like Gaia [9], Aura [3] or iRos [5]. Although such an infrastructure could be available at certain times, devices cannot rely on it.

In our system model communication and thus interaction is restricted to devices that are currently reachable by the network (e.g. due to communication technology). As a result, systems in these environments are inherently location-aware as communication is typically spatially limited. The devices have different specializations and resource limitations. Besides resource-poor and specialized devices such as sensor nodes, resource-poor general purpose devices could be present, e.g. PDAs. Also resource rich-devices can either provide a general purpose platform or they can provide single services such as a presentation system.

Due to the lack of a central or coordinating element, applications are dynamically composed of services provided by devices that are part of the currently reachable environment. As an example, consider an instant messaging application that requires an input service such as a keyboard or a touch screen to write messages and an output service to display messages, e.g. a monitor, a video projector or an audio channel. During start up, the application scans the current environment for available services and connects to suitable instances. At execution time, the application uses the services and adapts to changes regarding their availability or quality. Possible adaptations could include for instance the reselection of the output service whenever it becomes unavailable.

2.1.1. BASE. In order to provide basic support for services that enable such applications, we have developed BASE. BASE is written in Java using the Java 2 Micro Edition with the Connected Limited Device Configuration (CLDC). It assists application programmers by providing mechanisms for device discovery and service registration that can be used to locate and access local as well as remote device capabilities and services. Since the availability of services and capabilities can fluctuate in spontaneously networked environments, BASE provides a simple signaling mechanism to determine their availability. Communication protocols and device capabilities can be extended flexibly, since BASE is structured as an extensible micro-broker. This allows the middleware to run on resource-poor devices and benefit from resource-rich devices. In the context of this work, BASE is used as underlying communication middleware, offering communication and discovery on a wide range of devices. More information on BASE can be found in [1] and [2].

3. Adaptation Models

To provide application adaptation support for Pervasive Computing systems, three main levels of support can be distinguished. This classification is similar to the one given in [7].

Manual adaptation: here, adaptation is done by the end user. If an adaptation is performed, the system presents different choices and the user selects the most appropriate one. For the instant messenger described previously, this means that the user has to explicitly select the output or input service used by the application, whenever a used service becomes unavailable or a new service is discovered. Clearly, this is time-consuming and irritating, especially for environments with a high level of dynamism and a large number of different devices and services.

Application-specific automatic adaptation: to lessen the involvement of users, application adaptation should be executed with as little user interaction as possible. This can be realized by shifting the adaptation decision into the application. As a result, the system must support adaptation by signaling changes in the environment and the application programmer has to explicitly handle resource availability on a per-resource base, leading to complex and error-prone adaptation routines. Regarding the instant messenger scenario the programmer must provide routines that reselect the input and output service whenever the used services become unavailable. Such a reselection may be necessary at any point during the usage of a service. Therefore, the code of the application will be cross-cut by adaptation routines that are effectively reducing its readability and maintainability.

Generic automatic adaptation: at the highest level of support, application adaptation is done without stressing users or application programmers. The programmer only specifies the functional and non-functional properties of services required by the application and the user controls the adaptation process by stating adaptation goals. Thereafter, the system monitors service availability and selects the optimal services. The programmer of an instant messenger simply specifies the parameters of the input and output service, e.g. minimum screen resolution, and the user defines the adaptation preferences, e.g. highest available resolution. At runtime, the system automatically tries to find services with an acceptable quality. In cases where multiple services fulfill the requirement, the system performs the selection based on the preferences of the user.

4. Requirements

BASE offers generic automatic adaptation support at the communication layer. With PCOM we aim at providing further generic adaptation support at the application layer. PCOM should enable application programmers to extend the system with application-specific adaptation logic if needed. This enables a rather straight forward specification of application dependencies along with standard adaptation strategies resulting in a simple core system which can be customized to the needs of an application programmer. From these objectives the following requirements can be derived:

Application specification: applications should be specified in terms of their required services. Services should clearly denote their dependencies to other services and the platform. Non-functional properties of the dependencies should be explicitly stated. The composition of an application from services should allow the specification of alternatives in order to support the system to automate adaptation decisions.

Service monitoring: the system has to monitor the availability of services in order to detect currently used services that change their non-functional properties or become unavailable as well as to detect new services.

Strategy based adaptation: the system has to provide means for automatic adaptation of an application. If alternatives of services are present in the current execution environment, strategies decide which service to select. Besides standard strategies, e.g. to optimize energy consumption, user-defined policies should be integrated. At the core of adaptation, the application lifecycle and the lifecycle of single services have to be managed.

Minimalism and extensibility: to meet the resource heterogeneity of Pervasive Computing the resulting system has to be minimal with respect to required resources, e.g. processing power and memory, and it has to be extensible to exploit the advantages of resource-rich devices.

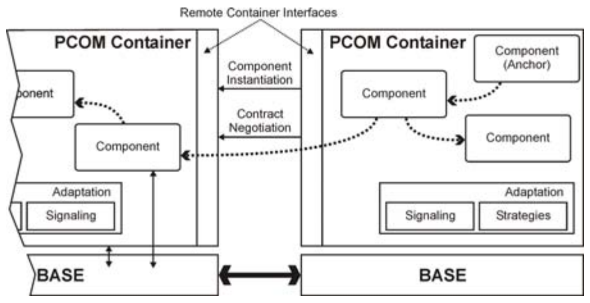


Figure 1: PCOM Architecture

5. PCOM

In the following we will present our component system PCOM (see Figure 1). PCOM provides a distributed application model and supports automatic application adaptation based on signaling mechanisms and adaptation strategies. Applications are composed of interacting entities, so-called components, which dependencies are explicitly specified as contracts. The PCOM container hosts components, manages their dependencies, and thus acts as a distributed execution environment for applications. Each container defines a remote container interface that exports locally available components by their contracts and allows remote containers to negotiate new contracts and access the components. To reuse the communication and discovery capabilities of our middleware BASE, the container is implemented as a single service on top of BASE. As a result, a container is automatically capable of detecting and using other containers.

In the following we will further describe our application model and present components along with their contracts. After that, we discuss application adaptation in PCOM and its realization.

5.1. Application Architecture

Applications in PCOM are composed of components that interact with each other in order to fulfill their dependencies. Components are atomic with respect to their distribution but can rely on local or remote components, resulting in a distributed application architecture.

An application is modeled as a tree of components and their dependencies where the root component (the so-called application anchor) identifies the application. The application tree reflects the dependencies between components where the successors of a component identify its dependencies in order to fulfill the service. PCOM uses a tree as application model, because arbitrary graphs cause several complications. For instance, the multiple use of the same component requires merging probably conflicting requirements. As another example, cycles of the graph could cause infinite loops during the composition of applications.

The life cycle of an application is reflected by the life cycle of its application anchor. Next, we will explain components in more detail, including the modeling of dependencies via contracts and their life cycle.

5.2. Components

Components in PCOM are units of composition with contractually specified interfaces and explicit context dependencies. PCOM's components enclose contracts that describe their offered functionality and requirements regarding the platform and other components. Components are atomic with respect to distribution and may use other components in order to provide their service. Note that PCOM does not regulate the granularity of components. Therefore, the granularity could range from single functionalities to complete applications.

5.2.1. Contracts. Contracts consist of two distinct parts: The first part specifies the corresponding component's requirements on the executing platform, e.g. required libraries or memory. The second part specifies the functionality provided by the component and its dependencies on other components. A dependency between two components has a direction and reflects the fact that one component either requires certain service interfaces (pull) or listens to some events provided by another component (push). Thus, PCOM supports push and pull communication models between components.

In order to describe dependencies, contracts in PCOM specify the service interfaces and the events that are offered and required by a component. Along the syntactical interface specification of events and services that define a functional dependency, non-functional parameters can be added to express further properties, such as a screen-size, energy consumption or performance related parameters. In contrast to the functional specification that is known at compile time, non-functional parameters can vary at runtime and might depend on the offer of components that are used to satisfy the dependencies. Thus, non-functional parameters can be either static or dynamic.

At runtime, contracts in PCOM are represented as object graphs. To ease the specification of these graphs, we use a compiler to transform an XML document into code that creates the desired structure. This representation is used for the comparison of offers and requirements. By applying them, it is possible to determine whether the offer of one component can be used to satisfy the requirements of another component. Due to the possibly large number of comparison operators that is needed to support arbitrary non-functional parameters, the underlying object model provides only a small set of operators that can be extended by application programmers.

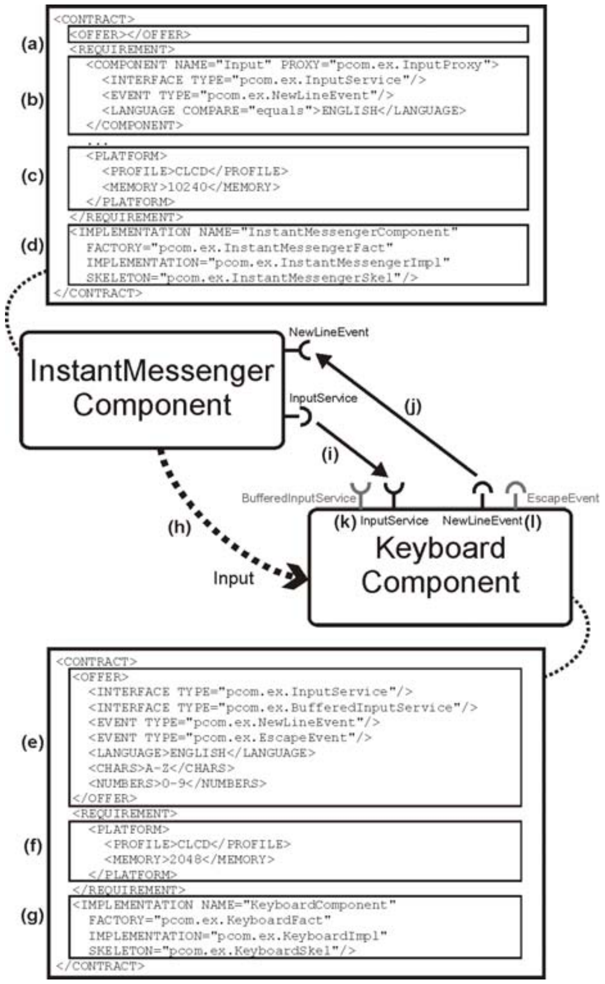

```
<CONTRACT>
  <OFFER></OFFER>
  <REQUIREMENT>
    <COMPONENT NAME="Input" PROXY="pcom.ex.InputProxy">
      <INTERFACE TYPE="pcom.ex.InputService"/>
      <EVENT TYPE="pcom.ex.NewLineEvent"/>
      <LANGUAGE COMPARE="equals">ENGLISH</LANGUAGE>
    </COMPONENT>
    ...
    <PLATFORM>
      <PROFILE>CLCD</PROFILE>
      <MEMORY>10240</MEMORY>
    </PLATFORM>
  </REQUIREMENT>
  <IMPLEMENTATION NAME="InstantMessengerComponent"
    FACTORY="pcom.ex.InstantMessengerFact"
    IMPLEMENTATION="pcom.ex.InstantMessengerImpl"
    SKELETON="pcom.ex.InstantMessengerSkel"/>
</CONTRACT>
```
```
<CONTRACT>
  <OFFER>
    <INTERFACE TYPE="pcom.ex.InputService"/>
    <INTERFACE TYPE="pcom.ex.BufferedInputService"/>
    <EVENT TYPE="pcom.ex.NewLineEvent"/>
    <EVENT TYPE="pcom.ex.EscapeEvent"/>
    <LANGUAGE>ENGLISH</LANGUAGE>
    <CHARS>A-Z</CHARS>
    <NUMBERS>0-9</NUMBERS>
  </OFFER>
  <REQUIREMENT>
    <PLATFORM>
      <PROFILE>CLCD</PROFILE>
      <MEMORY>2048</MEMORY>
    </PLATFORM>
  </REQUIREMENT>
  <IMPLEMENTATION NAME="KeyboardComponent"
    FACTORY="pcom.ex.KeyboardFact"
    IMPLEMENTATION="pcom.ex.KeyboardImpl"
    SKELETON="pcom.ex.KeyboardSkel"/>
</CONTRACT>
```

Figure 2: Exemplary Contracts

5.2.2. Example. Figure 2 shows XML-based contract specifications for an exemplary instant messenger component and a keyboard component. First we will have a look at the messenger's contract. It specifies that the messenger component does not offer any service to other components (a) and that it depends on an input component offering a given service interface and event type (b). Additionally, the messenger's contract states the non-functional requirement that the input component's language must be English (b). Next, the platform dependency declares, that the messenger must be executed by a container that has at least 10 Kbytes of free memory and provides a CLDC (c). The last section of the contract contains information about the component's internals used by the container (d).

In contrast to the messenger's contract, the keyboard component's contract specifies an offer that consists of two interfaces and two events (e). Additionally, the offer also contains non-functional attributes that describe the available keys and the supported language. Apart from the requirements on the platform (f) the keyboard does not have any requirements. Again, the last section of the contract contains information about the component implementation (g).

At runtime, these XML-based contracts are transformed into an object model that allows matching the instant messenger component's requirements with the offer of the keyboard component. As the keyboard offers all required functional and non-functional features, it can be used to satisfy the messenger's dependency. After the components have been combined at runtime (h), the instant messenger component is capable of placing calls to the interface provided by the keyboard component (i) and the keyboard component can send the requested event to the instant messenger (j). The additional interface (k) and event (l) of the keyboard component will never be used.

5.2.3. Component Lifecycle. To consistently embed components into applications, the container defines and manages the lifecycle of components. Conceptually, this lifecycle consists of the two states STARTED and STOPPED. The state transitions are controlled by the container. The container loads a component by first loading the object graph that represents its contract. It then determines whether it can fulfill the component's requirements towards the platform. If they can be satisfied, the container adds the contract to the set of exported contracts. Initially the component rests in the STOPPED state. Once a component is about to be embedded into an application, the container tries to resolve and initialize the component's dependencies by selecting suitable components to fulfill them. This initial resolution of dependencies can be seen as a special case of adaptation. A more detailed description of the selection process is given in subsection 5.3. After all dependencies are fulfilled, the container triggers a transition to the STARTED state. In this state, the component provides its functionality and the container provides signaling and adaptation support. When the state changes to STOPPED, the container releases all resources held by the component.

5.2.4. Contract Exchange and Negotiation. As soon as a component is about to be executed, the container has to determine whether its dependencies – both, functional and non-functional – can be satisfied. In order to find components that can potentially be used to satisfy a dependency, the container sends the requirements to the containers available in the environment. The containers reply with the contractual offers of the components that can fulfill the requirements.

As mentioned earlier, there are non-functional parameters that a component cannot determine without knowing the components that are used to satisfy its dependencies. In order to determine such parameters, PCOM containers also support a negotiation phase that recursively determines the non-functional parameters of a component without starting it. To enable this, containers rely on so-called factories that are representatives for locally installed components. Factories provide the capability to determine the actual value of a non-functional parameter based on the set of components that is currently available. While PCOM provides a simple standard factory, application programmers can provide component-specific factories by declaring them in the component contract's implementation section (see Figure 2 (d)).

The algorithm for contract negotiation is a post-order traversal of the tree of matching offers and requirements, where factories implement the functionality that determines the values of non-functional parameters from the available offers.

5.3. Adaptation

In ever-changing environments, component-based applications have to deal with fluctuating availability and quality of components. Changes regarding the availability and quality of components can either have a positive or a negative impact on the application. This means that the quality of a used component's functionality can either increase or decrease during the execution. Also, used components might become unavailable and new components that could deliver a required functionality might be discovered at any time.

In order to adapt to fluctuations, a component has to have means of detecting changes with respect to quality and availability of other components that either depend on or are required by the component. PCOM defines three signaling mechanisms that detect changes regarding availability and quality.

5.3.1. Signaling Mechanisms. The first signaling mechanism is targeted at the availability of used components. Whenever a used component becomes unavailable, a so-called *communication listener* is notified. Application programmers can register communication listeners for every dependency of a component. As PCOM uses a soft-state lease mechanism to maintain the dependencies between components, the detection of an unavailable component is either a result of an unsuccessful call placed by the using component or by a heart-beat message sent by the runtime system.

The second mechanism detects the availability of new components. In order to receive notifications about components that could potentially be used to replace a currently used component, programmers can define *discovery listeners* for each dependency. Whenever BASE detects a new device, PCOM checks whether the device hosts an instance of PCOM. If a new instance is discovered, PCOM determines whether the new components could be used to replace a dependency of a locally executed component. The comparison of the requirements of a running component and the offer of a newly discovered component is solely based on the static parameters of the offer, significantly reducing the discovery overhead. Once a discovery listener is called, an adaptation strategy can decide, if a full negotiation of the dynamic parameters should be done. Hence, negotiation is performed only if an application may profit from a component change.

The last signaling mechanism provided by PCOM aims at fluctuations in the quality provided by a component. As mentioned above, non-functional parameters can change over time. Therefore, PCOM allows application programmers to specify *contract listeners* that are notified whenever a parameter changes.

5.3.2. Options for Adaptation. Application programmers can use the described signaling mechanisms as hooks to specify their own actions for adaptation or use system provided mechanisms. PCOM offers two generic mechanisms: *execution discontinuation* and *component reselection*. Application programmers are provided with means to implement further options, e.g. modifying contracts or retransmitting messages in case of a transient network partitioning.

The first generic adaptation mechanism is simply the discontinuation of an executed component. Whenever an executed component is no longer able to provide its functionality, it can stop its execution. This will result in an event that is received by the communication listener of the using component. With respect to the application model defined by PCOM, this means that a problem in a component is escalated to the next, i.e. higher, level of the tree. The escalation continues until a component resolves the conflict by either reselecting a component (see below) or applying a user-defined strategy. If the escalation leads to the discontinuation of the application anchor, the execution of the application stops.

The second generic mechanism supports the reselection of components at runtime. This is enabled by two features. First, components specify their dependencies explicitly which allows matching a contractually specified requirement and its corresponding offer. Second, PCOM allows the definition of strategies that prioritize possible components based on user preferences. Therefore, if a component initiates the reselection of a certain dependency, PCOM can automatically determine the possible replacements that match the programmer's requirements. If there are several possible replacements, a user defined strategy is applied to select the best replacement according to the user's current selection goals. Clearly, a simple reselection will only be possible if the corresponding component is stateless. For stateful components, the application programmer still has to provide additional routines that establish the desired state. Nevertheless, the programmer does not have to implement the reselection algorithm and can use the signaling mechanisms to add an application-specific adaptation routine.

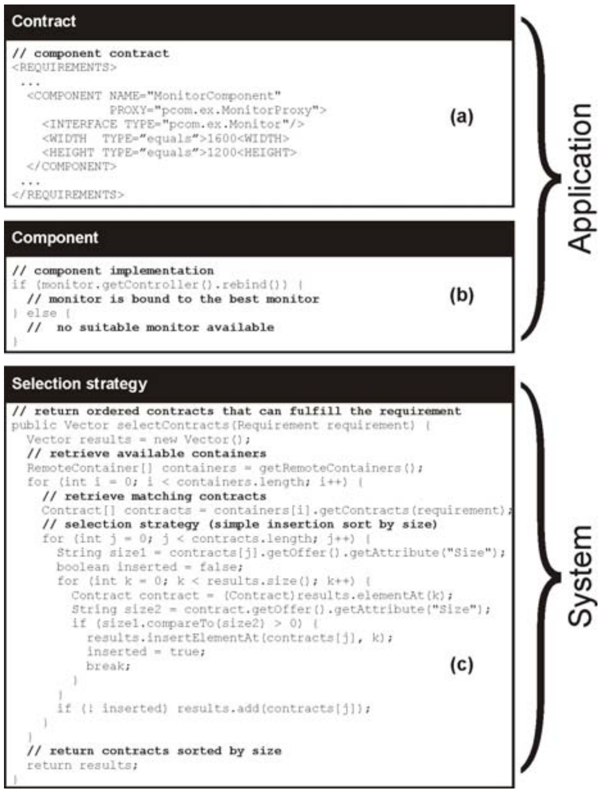
So far we have seen, how PCOM allows for generic application adaptation support via predefined as well as user-supplied strategies. The container realizing PCOM's runtime system resides on top of BASE, our middleware for Pervasive Computing. In the next section we will compare the abstractions provided by PCOM with the support BASE offers. The additional overhead for communication and application adaptation is presented based on measurements.

6. Evaluation

As stated in Section 4 the main requirements on PCOM are application specification and support for strategy-based adaptation. In PCOM these requirements are realized through components with contractually specified dependencies. As shown in Section 5.3, a crucial task for adaptation is the (re-)selection of services. Therefore, we will evaluate the service selection in PCOM and BASE. We compare the necessary tasks of a programmer and the assistance for service selection provided by PCOM and BASE. Next, the time needed for service selection is presented which includes contract evaluation, communication, and component instantiation. Finally, the additional requirements of PCOM regarding remote communication, memory, and computing power are discussed.

6.1. Service Selection

Selecting a service that will be used by an application comprises two fundamental tasks. First of all, an application has to determine the set of services that is available in a given environment. Thereafter, it has to determine the suitability of each service and select the best service possible.



Contract

```
// component contract
<REQUIREMENTS>
 ...
  <COMPONENT NAME="MonitorComponent"
             PROXY="pcom.ex.MonitorProxy">
    <INTERFACE TYPE="pcom.ex.Monitor"/>
    <WIDTH  TYPE="equals">1600<WIDTH>
    <HEIGHT TYPE="equals">1200<HEIGHT>
  </COMPONENT>
 ...
</REQUIREMENTS>
```

Component

```
// component implementation
if (monitor.getController().rebind()) {
  // monitor is bound to the best monitor
} else {
  //  no suitable monitor available
}
```

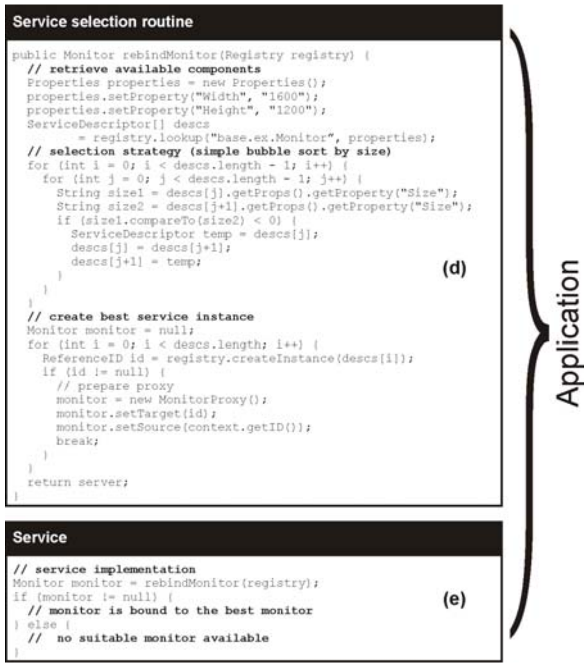
Selection strategy

```
// return ordered contracts that can fulfill the requirement
public Vector selectContracts(Requirement requirement) {
  Vector results = new Vector();
  // retrieve available containers
  RemoteContainer[] containers = getRemoteContainers();
  for (int i = 0; i < containers.length; i++) {
    // retrieve matching contracts
    Contract[] contracts = containers[i].getContracts(requirement);
    // selection strategy (simple insertion sort by size)
    for (int j = 0; j < contracts.length; j++) {
      String size1 = contracts[j].getOffer().getAttribute("Size");
      boolean inserted = false;
      for (int k = 0; k < results.size(); k++) {
        Contract contract = (Contract)results.elementAt(k);
        String size2 = contract.getOffer().getAttribute("Size");
        if (size1.compareTo(size2) > 0) {
          results.insertElementAt(contracts[j], k);
          inserted = true;
          break;
        }
      }
      if (! inserted) results.add(contracts[j]);
    }
  }
  // return contracts sorted by size
  return results;
}
```

Figure 3: Component Selection in PCOM

To allow determining the suitability, BASE and PCOM support non-functional parameters that allow a more detailed description of services. The suitability of a service could recursively depend on the suitability of the services used by it. As mentioned earlier, PCOM supports negotiation of dynamic parameters to model such dependencies. But since BASE does not deal with dynamic parameters, we restricted all parameters used during the evaluation to parameters that are static and thus, do not require negotiation.

Figure 3 shows the units of PCOM that are involved in the component selection process. An application programmer specifies the requirements of a component using a contract (a). At runtime, PCOM provides the application programmer with a handle for each component requested by the contract. Using this handle, a programmer can simply initiate the (re-) selection by calling the rebind-method (b). Typically, this method will be called within one of the listeners discussed above. When a reselection is initiated, PCOM uses contract matching to find suitable components and it uses a strategy to prioritize possible replacements (c). The distinction between contract and strategy separates the requirements that must be met to ensure the desired component behavior from user preferences. Notice, that (a) and (b) are supplied by a programmer, while (c) is a configurable and thus reusable strategy that is integrated in the system.



Service selection routine

```
public Monitor rebindMonitor(Registry registry) {
  // retrieve available components
  Properties properties = new Properties();
  properties.setProperty("Width", "1600");
  properties.setProperty("Height", "1200");
  ServiceDescriptor[] descs
        = registry.lookup("base.ex.Monitor", properties);
  // selection strategy (simple bubble sort by size)
  for (int i = 0; i < descs.length - 1; i++) {
    for (int j = 0; j < descs.length - 1; j++) {
      String size1 = descs[j].getProps().getProperty("Size");
      String size2 = descs[j+1].getProps().getProperty("Size");
      if (size1.compareTo(size2) < 0) {
        ServiceDescriptor temp = descs[j];
        descs[j] = descs[j+1];
        descs[j+1] = temp;
      }
    }
  }
  // create best service instance
  Monitor monitor = null;
  for (int i = 0; i < descs.length; i++) {
    ReferenceID id = registry.createInstance(descs[i]);
    if (id != null) {
      // prepare proxy
      monitor = new MonitorProxy();
      monitor.setTarget(id);
      monitor.setSource(context.getID());
      break;
    }
  }
  return server;
}
```

Service

```
// service implementation
Monitor monitor = rebindMonitor(registry);
if (monitor != null) {
  // monitor is bound to the best monitor
} else {
  //  no suitable monitor available
}
```

Figure 4: Service Selection in BASE

Figure 4 shows how a similar behavior can be implemented using BASE. An application programmer provides a selection routine for the required service that specifies its properties and priorities (d). Whenever a reselection must take place, the application calls this routine (e). In contrast to PCOM, the selection routine provided by the application programmer encapsulates both, service requirements and preferences.

The comparison of these two implementations shows that - from an application programmer's point of view - using a service in BASE is more complex than using a component in PCOM. While application programmers in BASE have to provide the functionality for searching and selecting required services, programmers in PCOM are provided with handles that hide the details of this selection. Instead of providing the specific algorithm that searches and prioritizes components, they simply specify the parameters that denote application-specific requirements and thus, they do not have to reason about user preferences. This means an additional flexibility which would be hard to achieve in a BASE implementation. Note that other, more complex features like contract negotiation or PCOM's signaling mechanisms are even harder to implement on top of BASE because of the lack of dynamic attributes.

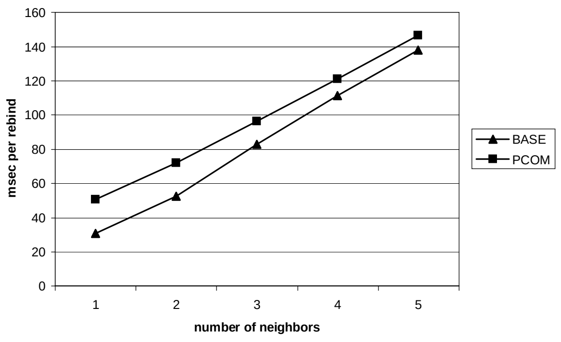


Figure 5: Component vs. Service Selection

Clearly, the extraction of functionality for selection causes an additional performance overhead. To quantify the impact on performance, we measured the time for a reselection in PCOM and in BASE. Figure 5 shows the average time for reselecting a service respectively a component (using the strategies and algorithms described in Figure 3/4) in cases where suitable components (or services in BASE) were available on 1 to 5 remote systems. The measurements have been conducted on PCs (Pentium III/600MHZ) connected with a 100 MBit network in order to show the fundamental effort without experiencing additional delays, such as Bluetooth discovery. The numbers shown in Figure 5 are the result of measuring 10 independent runs with 100 reselections each and varying the number of devices offering services (BASE) and containers (PCOM). To reduce fluctuations as far as possible, we disabled Java's just-in-time compiler. The remaining fluctuations were below 10 percent of the average time of a run and are most likely side-effects of the operating system's scheduler and Java's built-in garbage collector.

The total selection time is determined by the time for obtaining offers from neighbors, choosing an offer, and instantiating the chosen service or component. While the time for obtaining offers and choosing an offer increases linearly with the number of neighbors the instantiation of the chosen offer is constant. The measurements in Figure 5 show that, although reselection in PCOM is slower than in BASE, the relative overhead decreases with the number of neighbors. This is due to the higher cost for instantiating a PCOM component compared to a BASE service. The absolute overhead for a selection of approximately 30 ms however, is unlikely to be a bottleneck for realistic applications.

In addition to these measurements on resource-rich devices we have performed experiments on a JStamp embedded system[1] connected by a 19200 baud serial line. The average selection time was 3300 ms, which still may not impose serious problems, since a constant change of an application configuration, such as switching a monitor, will be annoying to the user.

In summary, comparing service and component selection shows that separating requirements and preferences using contracts and strategies is not for free. Although the overhead is noticeable, we believe that the gained flexibility is worth the performance penalty.

6.2. Communication

In order to compare the communication performance in BASE and PCOM, we measured the cost for a single message transfer using both systems. Our measurements showed that PCOM basically does not induce overhead on calls between components as it does not introduce indirections in the dispatch chain. This in turn is a result of carefully integrating proxies and skeletons of BASE and PCOM.

In terms of general communication overhead, three mechanisms introduced by PCOM require additional remote communication. In contrast to services in BASE, components in PCOM use a soft-state protocol to detect the (un-)availability of components. This protocol transparently exchanges additional keep-alive messages if no other messages have been exchanged during a lease period. These messages represent an additional communication overhead for components

[1] http://www.jstamp.com

that communicate infrequently. The second mechanism that introduces new messages is the discovery listener as it retrieves relevant contracts from devices that have been newly discovered. The last mechanism that requires additional remote communication is the contract listener. It creates a message for every modification of an offer or a requirement that is specified in a contract.

Clearly, all three mechanisms do not only create overhead, but do also provide necessary features. It is conceivable that realistic applications in dynamic environments must rely on soft-state protocols to reduce the amount of wastefully reserved resources. Similarly, components that have changing requirements or offers need to communicate them. Finally, optimization of executed applications requires notification about changes that could have positive impact.

Obviously, all three mechanisms could also be implemented in the application space, but it is questionable whether the possible performance benefit would outweigh the memory and engineering overhead of implementing all mechanisms within each component.

6.3. Resource Overhead

Apart from the cost of single mechanisms, PCOM has additional memory and processing requirements. In terms of memory usage, PCOM adds 30-40KB on top of 90-120KB required by BASE, resulting in a total memory usage of 120-160KB. With respect to processing, component instantiation and contract evaluation as well as all three mechanisms described in the previous section lead to increased requirements. The overhead for comparing contracts and instantiating components has already been discussed in the comparison of service and component selection. The processing requirements for the other mechanisms vary heavily depending on the applications and the environment and thus are hard to quantify.

7. Related Work

We will discuss related work in the areas of component systems, architectures for adaptation and evolution as well as recoverable computing, and pervasive computing.

Component Systems: Szyperski defines components as units of composition with contractually specified interfaces and explicit context dependencies only along with other properties [11]. This definition conforms to our definition introduced in section 5. Existing component systems, e.g. CORBA CCM [6], Enterprise Java Beans [10], conform to this definition by introducing container abstractions to decouple components from the underlying platform and by providing – at least functional – contracts between components via interfaces. Such systems typically provide persistency and transactional behavior and are targeted at enterprise software rather than on resource constrained and dynamic environments, such as Pervasive Computing.

Adaptation Architectures and Recoverable Computing: The self configuration of software is addressed by a number of projects in the research area of application architectures. In contrast to our work, these projects typically consider adaptation to be a rather rare event, caused by errors or changes in the software's mission.

The *Weaves* approach [7] provides a general graph structure to model component dependencies. This leads to complex algorithms and additional specifications to support adaptation decisions. Therefore, this approach is too heavy-weight for resource poor devices and frequent adaptations.

The *recursive restartability approach* [8], proposed in the domain of recoverable computing, uses a tree-based application model quite similar to the PCOM model. Still, this model is specifically designed to allow the restart of failing components. The partitioning of the application follows the encapsulation of restartable units – not units of composition – and the only supported adaptation is a component re-instantiation. PCOMs application model is different in that it models the functional and non-functional properties of inter-component dependencies.

Pervasive Computing: The necessity of application adaptation is realized by a variety of projects that differ widely in their support for adaptation and the abstractions provided to application programmers. The system model considered is often based on smart environments, providing a set of services, such as lookup and persistent storage to devices that connect temporarily or permanently to the smart environment. In contrast to this, our system model does not assume connectivity to a smart environment but spontaneous connectivity to devices in the vicinity.

The *iROS* [5] application model consists of atomic application parts which communicate via an event heap, realized as a tuple space. The event heap decouples distributed parts of an application. If functionality is not present, the request in the event heap is purged using an aging mechanism. Adaptation of applications is implicit, as functionality is only presented to the user if the application receives an answer to its request in the event heap.

One.world [4] is also based on a tuple space to allow communication between distributed parts of an application via events. Applications are composed of nested environments. Environments isolate applications from each other and serve as containers for persistent data. Conquering failure and selective availability is supported by providing mechanisms for application-specific automatic adaptation, such as migration or checkpointing along with persistent storage. Generic automatic adaptation is not supported.

Gaia [9] provides an application model based on a generalized model view controller pattern. An abstract definition of required functionality is mapped to the services available in a distinct smart environment (an active space). A coordinator component ensures that the application is executed as long as their integral parts are available. Adaptation is mainly considered to happen when a user moves to another active space and the matching of non-functional parameters is solely used to create a mapping between them.

The application model of *Aura* [3] provides a high level, user oriented task scheduler. Like PCOM, Aura aims at providing generic automatic adaptation support, but assumes a variety of services, e.g. remote communication, distributed file system, between remote Aura environments. PCOM is intended for environments, where this cannot be assured.

8. Conclusion

In this paper we have presented PCOM, a lightweight component system supporting strategy-based adaptation in spontaneous networked Pervasive Computing environments. Using PCOM, application programmers rely on a component abstraction where interdependencies are contractually specified. The resulting application architecture is used for strategy-based adaptation of applications. Our results so far are promising. Based on our middleware BASE, PCOM adds only little memory overhead and basically no runtime overhead on communication. Overhead is introduced by the instantiation of components resulting in higher reselection time. However, this overhead decreases with the number of involved nodes. We conclude that providing a component abstraction along with generic adaptation support is possible with reasonable overhead even for resource-restricted devices.

Besides evaluating PCOM on a variety of different devices and communications technologies in our lab, we are currently evaluating PCOM's abstractions by developing further and more complex applications. From the gained experiences, we expect to identify additional generic adaptation mechanisms. Furthermore, we are working on generic adaptation mechanisms that will allow the reselection of stateful components. In the near future different adaptation strategies will be developed and evaluated using our system.

References

[1] C. Becker, G. Schiele, H. Gubbels, K. Rothermel, "BASE - A Micro-broker-based Middleware For Pervasive Computing", *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communication*, pp. 443-451, Fort Worth, USA, March 2003

[2] C. Becker, G. Schiele, "Middleware and Application Adaptation Requirements and their Support in Pervasive Computing", *Proceedings of the 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems at ICDCS*, pp. 98-103, Providence, USA, May 2003

[3] D. Garlan, D. Siewiorek, A. Smailagic, P. Steenkiste, "Project Aura: Towards Distraction-Free Pervasive Computing", *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 22-31, April-June 2002

[4] R. Grimm, T. Anderson, B. Bershad, D. Wetherall, "A system architecture for Pervasive Computing", *Proceedings of the 9th ACM SIGOPS European Workshop*, pp. 177-182, Denmark, September 2000

[5] B. Johanson, A. Fox, T. Winograd, "The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms", *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 67-74, April-June 2002

[6] Object Management Group (OMG), "CORBA Component Model V3.0", *formal/2002-06-65*, 2002

[7] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software", *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54-62, May-June 1999

[8] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft, "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies", *UC Berkeley Computer Science Technical Report UCB//CSD-02-1175*, March 2002

[9] M. Román, R. Campbell, "Gaia: Enabling Active Spaces", *Proceedings of the 9th ACM SIGOPS European Workshop*, Denmark, pp. 229-234, September 2000

[10] SUN Microsystems, "Enterprise Java Beans Specification", *http://java.sun.com/products/ejb/docs.html*, 2003

[11] C. Szyperski, "Component Software - Beyond Object-Oriented Programming", *Addison-Wesley*, 1998

[12] R. Want, T. Pering, D. Tennenhouse, "Comparing Autonomic and Proactive Computing", *IBM Systems Journal*, vol. 42, no. 1, pp. 129-135, January 2003