Supporting Pluggable Configuration Algorithms in PCOM

Marcus Handte[*], Klaus Herrmann*, Gregor Schiele[+], Christian Becker[+]
[*] *IPVS, Universität Stuttgart, Germany*
[+] *Universität Mannheim, Germany*
{firstname.lastname}@{ipvs.uni-stuttgart.de|uni-mannheim.de}

Abstract

Pervasive Computing envisions distributed applications that optimally leverage the resources present in their ever-changing execution environment. To ease the development of pervasive applications, we have created a pervasive component system (PCOM). PCOM automates the configuration and runtime adaptation of a component-based application using a built-in distributed configuration algorithm. In this paper, we present an architectural extension that allows switching between different algorithms. This enables PCOM to dynamically select an algorithm that suits the computational resources present in an environment. To validate the extended architecture, we compare the overheads of a distributed and a centralized configuration algorithm in two different environments.

1. Introduction

Pervasive Computing envisions seamless task support through applications that are cooperatively executed by devices invisibly integrated into everyday objects. Due to mobility, the set of devices that is present in an environment is changing continuously. As a result, applications are forced to adapt to the capabilities of their current execution environment.

In order to reduce the complexities arising from the development of distributed applications for dynamic environments, we have created a pervasive component system (PCOM) [1]. Using its component framework, developers are able to create reusable components that explicitly specify their offered functionality and their dependencies with respect to other components and local resources. At runtime, the PCOM component container ensures that each executed component has enough resources and can access all required components. If a required component becomes unavailable, e.g. because the device that hosted it is no longer reachable, the container automatically selects a suitable replacement. Since using a different component can result in resource conflicts, PCOM containers are equipped with an algorithm that performs resource-aware application configuration and adaptation.

In order not to constrain the applicability of PCOM to environments that contain resource-rich devices, the configuration algorithm is fully distributed, i.e. each component container configures and adapts its hosted components. While this approach reduces the computational resources required on a single device, it increases the overall amount of communication required to compute a configuration.

Clearly, in environments where resource-rich devices are available, the overhead introduced by configuration can be reduced by computing the configuration on a single powerful device. However, in order to deal with environments consisting of resource-poor devices, relying solely on a centralized algorithm is not a viable option. Thus, in order to guarantee the optimal performance while maintaining a broad applicability, the system should be able to adapt the degree of distribution depending on the available resources.

In this paper, we present an architectural extension to PCOM that enables the system to switch between configuration algorithms at runtime. This enables the development and usage of configuration algorithms that are optimized for different environments. In order to validate the approach, we have implemented a fully distributed and a centralized algorithm using the new architecture. The comparison of these algorithms shows that our architecture is flexible enough to exploit the performance benefits achievable in different environments without adding intolerable overhead.

The remainder of this paper is structured as follows. In the next section, we briefly describe the configuration process performed by PCOM. In Section 3, we present the goals that guided our design decisions and in Section 4, we describe the architectural extensions to support multiple configuration algorithms. To demonstrate the validity of the extensions and to evaluate them, we present a comparison of a centralized and a

distributed configuration algorithm in Section 5. In Section 6, we describe some related work. Section 7 concludes the paper with on outlook on future work.

2. PCOM Configuration and Adaptation

In the following, we will outline the core concepts of PCOM and the actions required to configure and adapt an application. Thereby, we omit the details of a specific configuration algorithm. Instead, we focus on the tasks performed by the component container. A more detailed description can be found in [1].

PCOM components are atomic with respect to distribution. Each device is equipped with a component container that manages its hosted components. A component specifies its offered functionality and its dependencies using contracts. A contract defines the offered functionality in terms of implemented interfaces. The interface descriptions can be extended with properties to describe non-functional parameters. Similarly, the dependencies on other components are defined in terms of required interfaces and properties. In addition, a component can contractually define resource requirements that must be met by its container in order to use the component. The contractual description of provided and required functionality enables the container to determine whether the offer of a component can be used to satisfy a certain requirement.

The application model supported by PCOM is a tree of components that starts at a root component, the so-called *application anchor*. The lifecycle of the application anchor defines the lifecycle of the application. When a user starts an application anchor, the container resolves its dependencies by recursively starting components that satisfy the specified requirements. As soon as the anchor is stopped, the containers that host components of the application will stop them.

A single component can be used in multiple applications simultaneously. To do this, components are instantiated for each usage and they are removed when they are no longer needed. The number of instantiations is solely limited by the amount of available resources. Since resources can be limited, using one component might prohibit the usage of another one.

A configuration algorithm that is cooperatively executed by the containers of an environment ensures that only valid configurations are started. A valid configuration is thereby defined as a configuration that does not have unresolved dependencies, i.e. for each requirement specified in a contract of a used component there is a component whose offer satisfies the requirement and all resource requirements of the used components can be met by their container.

At runtime, the containers continuously monitor the executed components. If a required resource or a component becomes unavailable, the container will report the invalid configuration to the container that hosts the application anchor. This container will then recursively signal the problem to the remaining components of the application by pausing them. After all components have been paused, the containers will compute a new valid configuration. As soon as a valid configuration has been found, the containers will reconfigure the components that need to be changed and start new components when necessary. Thereafter, the application continues its execution until another adaptation is required or the application is stopped.

3. Architectural Design Goals

In its initial version, the PCOM component containers contained all logic required to configure, start, monitor, pause, adapt, and stop an application. To support switching between different configuration algorithms, we must define the individual responsibilities of a configuration algorithm and the component container as well as their interaction. Clearly, there are numerous possibilities of dividing the responsibilities and defining appropriate interaction schemes. To evaluate different options and to select the most appropriate one, we focus on achieving the following goals:

Resilience to failures: A fundamental design rationale of PCOM is the enforcement of strong guarantees with respect to application configurations. Specifically, PCOM aims at enforcing the invariant that all dependencies of an anchor are recursively resolved with suitable components that have sufficient resources. Enforcing this invariant can greatly simplify component development as a developer does not have to deal with unavailable components and resources. Thus, our paramount goal for the separation of container and algorithm is resilience to this kinds of failures.

Efficiency & minimalism: Separating the configuration algorithm from the component container bears the risk of adding runtime overhead. Since the goal of supporting different algorithms is performance optimization, such overhead is clearly undesirable. Apart from minimizing additional overhead during the configuration, the architectural extensions should also be minimal with respect to code size, in order to be usable on resource-poor devices. Since the extensions facilitate the development of multiple algorithms, minimalism also requires that common functionality is pushed into the container to avoid duplicate implementation.

Simplicity & extensibility: Since the goal of extending the original architecture is supporting multiple

configuration algorithms, the interface of a configuration algorithm should be simple and easy to implement. Furthermore, in order to support various algorithm implementations, the container should be able of providing all information that might be required by a configuration algorithm.

4. Extended Architecture

To achieve the design goals described in the previous section, we split the functionality of the original component container into three distinct services running on top of our communication middleware BASE [2]. All services are equipped with a remote interface that exports their functionality to the other services. Since these services are accessed indirectly through BASE, we can flexibly adapt their distribution.

The first service, the so-called *application manager* is responsible for managing the lifecycle of an application and selecting a configuration algorithm whenever a valid configuration must be computed. Using this service, a user can request that a certain anchor should be started or a running anchor should be stopped.

The second service, the *assembler*, implements the functionality of computing a valid configuration. Note that there might be different assembler implementations available in a given environment, e.g. a centralized and a distributed one. Clearly, using a fully distributed assembler, for instance, requires the availability of an instance of this assembler on each device.

To configure an application, an assembler relies on the inputs of the third system service, the *component container*. The remaining task of this revised service is the management of resources and components as well as the monitoring of hosted components. The resulting shift in the system structure is depicted in Figure 1.

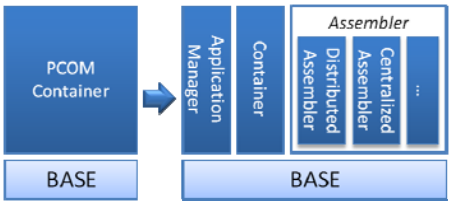

Figure 1 – Architectural Extension

By externalizing the code required to compute a valid configuration in an independent system service with a unified remote interface, we can support different implementations and distributions without changing the container. However, since we can no longer guarantee that a container runs on the same device as the assembler used to configure its components, we must deal with additional failures resulting from remote communication. In order to achieve our goal of resilience, we must specifically ensure that no failure causes the startup of an invalid configuration.

To do this, we design the interaction between the container and the assembler in such a way that the container remains in control during the whole configuration and adaptation process. Therefore, all traversals that start, pause, and stop an application remain the responsibility of the container, while providing the appropriate configuration action for each component is pushed into the assembler. By keeping the traversal in the container we can additionally reduce the amount of functionality that must be implemented in an assembler which supports our desired goal of minimalism.

In the following, we describe the dynamic cooperation of the three system services by walking through the lifecycle of an application. Thereby, it is important to realize that the initial configuration of an application is essentially a special instance of adaptation, i.e. the initial configuration must find a suitable “replacement” for all dependencies of the anchor. Thus, we start our discussion with an application that has been configured and started already.

Figure 2 depicts the sequence of calls required to initialize the adaptation process for an exemplary application consisting of four components (A, B, C, D) running on 4 devices (1, 2, 3, 4) that needs to be adapted because device 4 is no longer reachable.

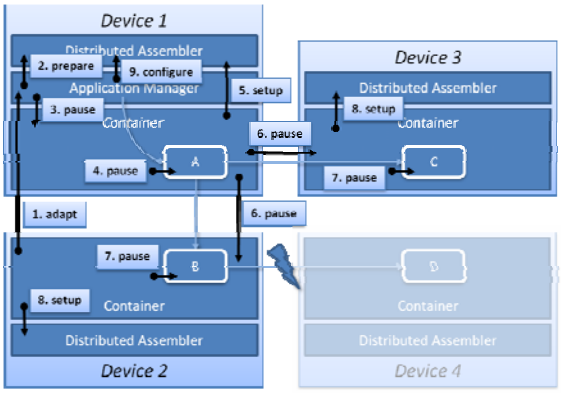

Figure 2 – Distributed Adaptation

When a component used by the application is no longer available the container that hosts the parent component detects this and sends a message to the application manager that started the application (1). The application manager then selects the assembler that shall perform the adaptation and calls a prepare method on the assembler (2). In response, the assembler can prepare its internal data structures and if it requires other assembler instances, it can initialize them. After the call returns, the application manager

signals the container to pause the anchor (3). Thereby, it passes the remote reference of the assembler to the container. The container will then send a pause signal to the anchor (4) and prepare a setup object that describes the anchor, its contractually specified dependencies, and the remote systems that host child components for these dependencies. The setup object is sent to the assembler represented by the remote reference and the assembler returns a remote reference to an assembler for each component declared in the setup object (5). Thereafter, the container sends the pause request to all containers that host children of the anchor. As part of the call, the container passes the remote reference of the corresponding assembler to the container (6). Since the remote references are generated by the assemblers, neither the container nor the application manager must be aware of the distribution degree of the assembler. As soon as initial pause call returns, the application manager sends a configure call to the assembler (9). In response, the assembler will prepare a valid configuration. If one of the remote calls fails, the container detects this and signals the failure as return value of the corresponding pause call. This allows the application manager to restart the process.

When the configure call is sent to the assembler, the assembler has received a setup object for each component of the application. Thus, the assembler can determine which dependencies need to be resolved in order to transform the current invalid configuration into a valid one. To compute a valid configuration, the assembler needs to be able to determine the set of resources that is available on each device and it needs to be able to find the components that can be used to resolve a dependency. To this end, each container offers a remote query interface that provides information about its hosted components and resources.

As soon as the assembler has computed a valid configuration, it returns the configuration to the application manager as return value of the configure call. In order to provide efficient support for different distribution degrees, the result must not necessarily contain the complete configuration. Instead, the data structure used to describe the tree of components can either contain the full configuration data or a reference to the assembler that can provide the configuration data for a certain sub-tree upon request. Using this data structure, the configuration data can be retrieved lazily.

Figure 3 shows the sequence of actions for the previous example required to transform the running invalid configuration into a valid configuration. To do this, the component B running on device 3 is replaced with component B' on device 4. Note that the example assumes that the configuration is stored in each assembler and is returned lazy upon request.

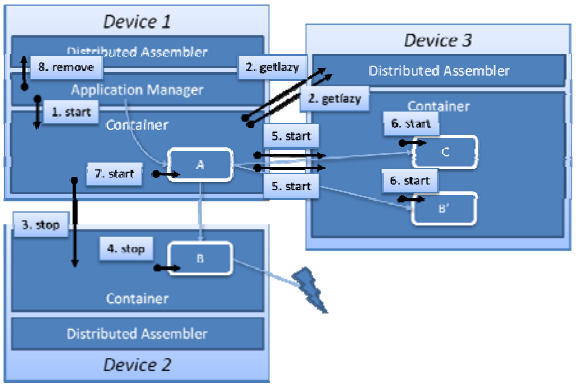

Figure 3 – Distributed Startup

After receiving the configuration, the application manager sends a start request to the anchor (1). Thereby, it passes the configuration data received by the assembler. The container retrieves the configuration for each child component required by the anchor (2). Using the configuration, the container decides whether the child must be reused or replaced. If the child must be replaced, the container first stores its internal state and releases it by sending a stop call (3, 4). Thereafter, the container will send a start call to the container that hosts the new component (5). The start call then contains the state of the stopped component. If the component is reused, the container simply sends a start call to the container of the reused component. As part of the start call, the container sends the configuration for the child and the recursion continues. The return value of a start call signals whether a child has been started successfully. If all start calls for all children of a certain component have returned successfully, the component itself is started and all state is restored (6, 7). After the component and its children have been started recursively, the start call returns the status. If this procedure fails at any point in time, e.g. because a device is no longer available, the start calls will simply return that the startup is not successful. The application manager can then restart the complete adaptation process which pauses the components that have been started. When the start call returns, the application manager sends a remove call to the assembler (8). This allows the assembler to remove all data stored for the application. If the assembler used instances on other devices, it can forward the remove call to release their data, too.

5. Evaluation

To evaluate the runtime overhead of the presented extension, we have measured the time required to perform a local call to a BASE system service through the request broker and the time required for a method

call on a MDA Pro (PXA270, 128MB RAM). A call to a service is 450 times slower, however, the absolute time of 0.12ms and the low number of calls indicate that these costs can be neglected. Furthermore, we have measured the increase in code size induced by the extension. While the size of the original container (126KB) decreased by approximately 8% the overall size for a system equipped with an application manager, a container and an assembler increased by 31KB.

The goal of the presented extension of PCOM is to support a variety of pluggable algorithms that are optimized for different environments. To show that the architecture supports this, we have implemented two greedy assemblers, a fully distributed and centralized one. We have measured the delay introduced by adapting applications consisting of 4-12 components in a resource-rich and a resource-poor environment. The resource-poor environment consists of 4 MDA Pro connected via 802.11b. Each MDA hosts an instance of a distributed assembler and one MDA hosts a centralized assembler. For the resource-rich environment, we replace the MDA hosting the centralized assembler with a Tablet PC (1.8 GHz Pentium M, 1024MB RAM). The applications are binary trees that have been placed onto the devices in such a way that the dependencies of each component must be resolved using a component on a remote device. Figure 4 shows the average adaptation delay of 10 runs with the distributed (DIST) and the centralized assembler (CENT) in the resource-rich (RR) and the resource-poor (RP) environment. The standard deviation lies below 9%.

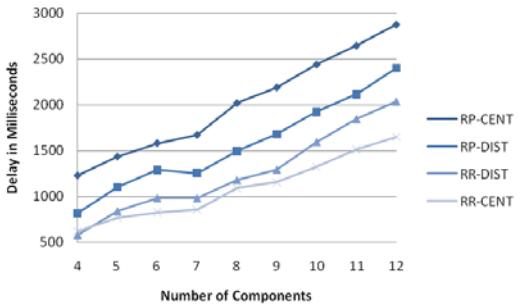

Figure 4 – Adaptation Delay

The values show that replacing the PDA reduces the overall delay, but both algorithms have different associated tradeoffs. In the resource-rich environment, the centralized algorithm outperforms the distributed by 20% for an application with 12 components. In the resource-poor environment, the centralized algorithm introduces an overhead of 20%. While this simple experiment is not a thorough analysis, i.e. it does not consider factors like the application structure, placement and resource conflicts, it shows that the architecture enables significant performance optimizations by supporting pluggable configuration algorithms.

6. Related Work

At the present time, there is no system software for pervasive applications that supports switching between different configuration algorithms at runtime. Many system software solutions [3-5] in the Pervasive Computing domain focus on support for smart rooms and thus, they can guarantee the availability of a resource-rich device. As a result, these systems typically use centralized approaches to solve the configuration problem. With the presented extension of PCOM, we specifically target at system support that is suitable for a broad spectrum of possible application scenarios, ranging from coordinated smart environments to uncoordinated ad-hoc environments of resource-poor devices.

7. Conclusion

In this paper, we presented an extension to the original architecture of PCOM that effectively separates the tasks of computing a configuration from the task of executing an application. This separation enables the development of specialized configuration algorithms for different environments. The evaluation shows that the extended architecture enables significant performance optimizations without introducing intolerable costs. At the present time, we are thoroughly analyzing the effects of various algorithms in different scenarios. The ultimate goal is the development of a set of algorithms and selection strategies resulting in optimal performance.

8. References

[1] Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM – A Component System for Pervasive Computing, Intl. Conference on Pervasive Computing and Communications (PerCom, 04), 2004

[2] Becker, C. Schiele, G., Gubbels, H. Rothermel, K.: BASE – A Micro-broker-based Middleware for Pervasive Computing, Intl. Conference on Pervasive Computing and Communications (PerCom, 03), 2003

[3] Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R., Nahrstedt, K.: A Middleware Infrastructure for Active Spaces, IEEE Pervasive Computing, 1(4), 2002

[4] Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P.: Project Aura: Towards Distraction-Free Pervasive Computing, IEEE Pervasive Computing, 1(2), 2002

[5] Saif, U., Pham, H., Paluska, J., Waterman, J., Terman, C., Ward, S.: A Case for Goal-oriented Programming Semantics, UbiSys Workshop at UbiComp, 2003