Adaptive Composition of Distributed Pervasive Applications in Heterogeneous Environments

STEPHAN SCHUHMANN, KLAUS HERRMANN, KURT ROTHERMEL, University of Stuttgart, Germany
YAZAN BOSHMAF, University of British Columbia, Vancouver, Canada

Complex pervasive applications need to be distributed for two main reasons: due to the typical resource restrictions of mobile devices, and to use local services to interact with the immediate environment. To set up such an application, the distributed components require spontaneous composition. Since dynamics in the environment and device failures may imply the unavailability of components and devices at any time, finding, maintaining, and adapting such a composition is a nontrivial task. Moreover, the speed of such a configuration process directly influences the user since in the event of a configuration, the user has to wait. In this article, we introduce configuration algorithms for homogeneous and heterogeneous environments. We discuss a comprehensive approach to pervasive application configuration that adapts to the characteristics of the environment: It chooses the most efficient configuration method for the given environment to minimize the configuration latency. Moreover, we propose a new scheme for caching and reusing partial application configurations. This scheme reduces the configuration latency even further such that a configuration can be executed without notable disturbance of the user.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms: Algorithms, Design, Management, Measurement

Additional Key Words and Phrases: Adaptivity, Automation, Components, Heterogeneity, Hybrid Configuration, Middleware, Partial Configurations, Pervasive Applications, Service Composition

1. INTRODUCTION

Pervasive Computing focuses on the development of abstractions and concepts for seamless integration of information processing into everyday activities and objects. Due to enormous developments in wireless communication technologies as well as ongoing technological miniaturization, completely new types of mobile end-user devices like smart phones or netbooks have become reality. Moreover, application scenarios like Smart Homes [Helal et al. 2005], ubiquitous cities [Whitman et al. 2008] or even pervasive service ecosystems [Zambonelli and Viroli 2011], providing various services and applications, are emerging. Pervasive Computing aims at enabling seamless connections between these devices and applications, without distracting the users. Ideally, users are not even aware of the involved computer systems.

In pervasive scenarios, the application functionality is normally distributed among several devices since a single device is not capable of executing an entire application. Therefore, a *Pervasive Application (PA)* needs to be configured prior to their execution to ensure that all required functionality is available at execution time. Configuring an application means finding a suitable set of components (resources and services) which can be instantiated concurrently. A *valid* composition provides the functionality required by the application while considering the limited resources in the environment.

Beyond the initial configuration, automatic *adaptation* (i.e., re-configuration) at runtime is needed to find alternative components for those parts of the application that have become unavailable, e.g., due to device failures or user mobility. Configurations and adaptations imply latencies which are experienced by the user as undesired delays. These latencies need to be minimized to make the systems really unobtrusive.

Here, we present a new integrated solution which efficiently and automatically supports various typical Pervasive Computing scenarios. This leads to faster configurations and adaptations in heterogeneous scenarios and, in consequence, to less user distractions. Moreover, an *adaptive mechanism* based on clustering is presented to allow for the automatic switching between different approaches in dynamic scenarios [Schuhmann et al. 2008a]. The resulting system provides seamless application configuration in various homogeneous and heterogeneous environments.

Here, we introduce a concept for automatic caching and re-use of partial configurations assembled in previous configuration processes. Our approach focuses on scenarios with frequently ongoing configuration processes (e.g., the *synchronization of data* between a set of devices). Here, caching component sets with high potential for further re-use (e.g., the *data sources* as well as the *communication interfaces* of the involved devices) and integrating them into future configurations significantly reduces the configuration amount, as it minimizes the number of components that actually need to be configured. In evaluations performed on our system software, average configuration latencies of this new caching approach in typical scenarios are only 9 % higher than in the optimum case where the complete application has been pre-configured before.

The rest of this article is structured as follows: After introducing our system model in Section 2, we present the problem statement for this work in Section 3. In Section 4, we discuss the concepts and algorithms of our comprehensive solution for automatic configuration of distributed applications. This is followed by a presentation of our evaluation results in Section 5. Finally, we discuss related approaches in Section 6 and conclude this article in Section 7 with a brief outlook on our future work.

2. SYSTEM MODEL

Next, we discuss our system model, covering devices, environments, applications, and general requirements we pose on the system software. We derive our main assumptions about realistic application sizes empirically through studies of existing systems.

2.1. Devices and Environments

We focus on Pervasive Computing scenarios that include devices with different properties and computational power. Each device has a unique device ID. In terms of scenario heterogeneity, we distinguish between two different types of involved devices:

— *Resource-poor devices* are usually mobile wearable devices like PDAs or smart phones. Due to their limited computation power, they can slow down the configuration process if too much workload is put on them. Resource-poor devices typically have a low degree of availability, as they are highly mobile and their battery power is strictly limited. Thus, they should preferably not be burdened with computationally intensive tasks. We call environments with only resource-poor devices *homogeneous*.
— *Resource-rich devices* can be stationary infrastructure devices such as desktop PCs or mobile devices with powerful computation resources, e.g., laptops. Due to their increased power, resource-rich devices are perfectly suited for performing computation-intensive tasks such as the calculation of configurations or adaptations. We call an environment *heterogeneous* if at least one resource-rich device is present besides the resource-poor devices. Furthermore, several degrees of heterogeneity are possible, ranging from *weakly heterogeneous* environments where only one resource-rich device is available (e.g., an office with one desktop PC) to *strongly heterogeneous* scenarios with several resource-rich devices (e.g., an auditorium during a conference).

2.2. Applications

Pervasive Applications rely on functionality that is *distributed* among multiple cooperating devices. We assume a component-based application model, i.e., an application consists of *components* and each component instance requires a certain amount of re-

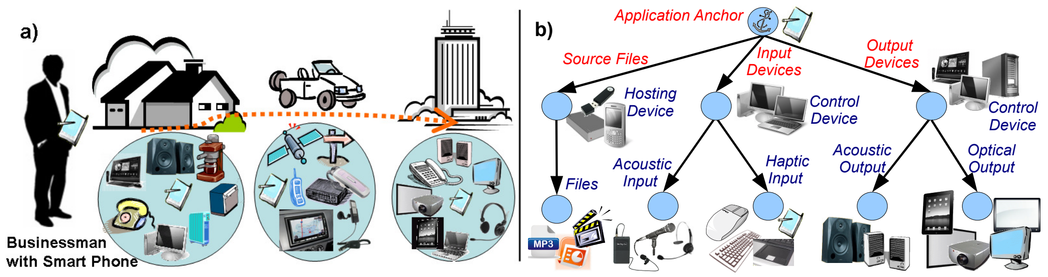


Fig. 1. a) Travelling businessman scenario, b) Distributed Presentation Application

sources. Figure 1a shows an exemplary scenario of a businessman who controls such applications (e.g., home entertainment, car navigation, video conferencing) only using his smart phone, while traveling from his home to his office by car. In such a scenario, the available set of components that may be used by applications changes considerably.

An application is represented by a tree of interdependent components that is constructed by recursively starting the components required by the root instance, the so-called *application anchor*. Figure 1b shows the component tree of an exemplary distributed presentation application in an office environment, where the businessman may hold a talk and present new products to business partners. The application automatically uses the resources available in the vicinity (e.g., displays, microphones, or speakers) as application *components*. A single component has a unique ID and is resident on a specific device such as a laptop or a smart phone.

Dependencies between components as well as resource requirements are described by directed *contracts* which specify the functionality required by the parent component and provided by the child component. In the example shown in Figure 1b, the contract for providing the functionality ”Acoustic Output” connects a control device as parent component (e.g., a laptop) and the speakers as child component.

Since a parent component relies on its child components, it can only be instantiated if all of its children have been instantiated before. If there exist several alternative options for a contract (e.g., several different speaker systems), we call this a *multi-optional* contract. The number of multi-optional contracts is depending on the currently available resources and may change over time in dynamic environments.

Typical application sizes are shown in Table I, divided into applications for homogeneous, heterogeneous, and visionary Pervasive Computing scenarios. Application sizes may widely vary between less than 10 and up to several hundreds of components. The table also shows that applications in heterogeneous scenarios are generally larger than in homogeneous scenarios. We evaluate our system based on applications that have the order of magnitude in size shown in the table.

2.3. System Software

We rely on a *communication middleware* that provides basic services to enable distributed applications and supports different communication models. It has to supply a device registry, maintaining a list of all currently reachable devices. The device registry on a specific device is kept up to date as every device periodically broadcasts heartbeat messages. Through this, global knowledge among all devices is established. As we rely on single-hop environments like conference rooms and a relatively low degree of dynamics where devices typically stay within an environment for some time, we consider to maintain global knowledge as a feasible solution. In our work, we use the BASE middleware [Becker et al. 2003] as it meets the requirements presented here.

Table I. Typical application sizes in homogeneous and heterogeneous environments

Reference	Application Scenario	Typical Size
[Edwards et al. 2002]	Resource sharing among users	10 components
[Grimm 2004]	Travelling consultant scenario	10 components
[Handte et al. 2006]	*Pervasive Presenter*	8 components
[Sadiq et al. 2011]	Service Composition in *Opportunistic Networks*	20 components
Typical (average) size for homogeneous scenarios		**12 comp.s**
[Helal et al. 2005]	*Smart House* with 17 ”hot spots”	30 components
[Cámara et al. 2008]	*Wireless Medical Information System*	20 components
[Fujii and Suda 2009]	Service composition according to User References in Restaurant environment	13 components
[Mukhtar et al. 2011]	Composition of *User Task Services*	15 components
Typical (average) size for heterogeneous environments		**20 comp.s**
[Herrmann et al. 2008]	*Pervasive Workflow (horizontal) adaptation / evolution*	10...50 components
[Ferrari and Mamei 2011]	*City-wide Sports tracking application*	10...100 components
[Zambonelli and Viroli 2011]	*Real-time traffic control; augmented reality services*	$>$100 components
[Zambonelli 2011]	*Pervasive urban crowdsourcing*	$>$1000 components
Typical size for visionary pervasive environments		**100 comp.s and more**

Moreover, we rely on a component system that is executed on top of the communication middleware. To enable application configuration and runtime adaptation, the component system needs to provide event-based signaling mechanisms to detect changes in the availability and quality of specific devices and services. Within this system, the configuration algorithms have to be implemented as assemblers accessing the components and composing them together. We use the system PCOM [Becker et al. 2004] as component system. However, the mechanisms and algorithms presented here are independent from PCOM/BASE and can potentially be used on every middleware system that fulfills the basic requirements discussed above.

3. PROBLEM STATEMENT

In the following, we first discuss the dynamic configuration of distributed PAs in heterogeneous environments (Section 3.1), which represents the main challenge we focus on. Then, we discuss the requirements (Section 3.2) for an adequate solution.

3.1. Adaptive Configuration in Heterogeneous Environments

Configuration denotes the task of determining a valid composition of components that can be instantiated simultaneously as an application. Such a composition is subject to two classes of constraints: *Structural constraints*, given by the functionality that is required by a parent components and needs to be provided by its children for resolving a dependency (e.g., the access to a remote database), and *resource constraints* caused by the limited availability of specific resources (e.g., a single display cannot be used by two applications simultaneously). The complexity of finding a valid configuration arises from the fact that both types of constraints must be fulfilled simultaneously. An application can be started successfully if for each contract a suitable component which satisfies all requirements was found and, thus, all dependencies have been resolved by the *configuration algorithm*.

The problem of configuring an application in a distributed manner represents an NP-complete Distributed Constraint Satisfaction Problem (DCSP) [Handte et al. 2005]. A DCSP is a problem that is defined as a set of objects whose state must satisfy a number of constraints. Backtracking algorithms from the domain of Distributed Artificial Intelligence [Yokoo et al. 1998] represent typical solutions to such a problem.

The *configuration latency* comprises the time between the start of an application and its availability to the user. This latency includes the delays for calculating the configuration and instantiating all application components. Hence, the configuration latency must be minimized to provide a seamless user experience.

The configuration problem has already been solved in homogeneous scenarios by providing decentralized algorithms for peer-based configuration [Handte et al. 2005]. However, decentralized schemes perform suboptimal in *heterogeneous* environments, since they do not exploit the heterogeneity by distributing the configuration tasks in a resource-aware manner among the currently available devices. In this article, we present a comprehensive solution to these problems.

We aspire an *adaptive* way of configuring distributed applications in a twofold way:

(1) Adaptive configuration means that the validity of calculated configurations needs to be maintained even in environments where the availability of specific components dynamically changes, e.g., due to device failures or user mobility. In such situations, components which are part of the current application configuration may become unavailable during application execution. This induces that the respective parts of the configuration have to be *adapted* (i.e., exchanged) at runtime. Therefore, the configuration algorithm has to identify alternative components which provide the same functionality. We focus on this issue in Section 4.1.3.
(2) However, not only the configuration itself, but the algorithm that distributes the configuration load among the available devices has to be chosen in an adaptive way to exploit the degree of heterogeneity in computation power of the involved devices. This requires the provision of several configuration schemes (spanning the complete spectrum from decentralized to centralized approaches) and a mechanism to switch between different approaches. Section 4.2.1 specifically focuses on the adaptive selection of the most suitable configuration in a particular environment.

3.2. Requirements

Solutions to the above discussed challenges have to meet the following requirements:

— **Adaptivity:** As discussed above, dynamic Pervasive Computing environments require the provision of approaches optimized for specific environments. The most suitable configuration algorithm has to be selected adaptively for every configuration process based on the characteristics of the environment.
— **Automation:** Many related projects demand users or application developers to handle configuration and adaptation issues manually. However, providing an *automated* solution where the system software is responsible for determining valid configurations yields systems that are much more transparent to users and developers.
— **Efficiency:** Configuration and adaptation processes induce latencies which users perceive as undesired delays, since the application is not available for a certain time. Thus, a major goal is to achieve *efficiency* by minimizing these delays. Therefore, the configuration schemes have to be aware of the computation power on the available devices and distribute the load uniquely according to the performance of the devices.

4. CONFIGURATION ALGORITHMS

For homogeneous environments, completely decentralized algorithms for peer-based application configuration [Handte et al. 2005] and runtime adaptation [Handte et al. 2006] based on Constraint Satisfaction (cf. Section 3.1) have been presented. These algorithms distribute the task of configuring an application equally among *all* devices and cannot exploit the increased computation resources of specific devices. However, this leads to inefficient configuration processes in heterogeneous environments. The general support of exchangeable configuration algorithms [Handte et al. 2007] increases the range of supported environments, but it does not allow the *adaptive switching* between algorithms in dynamic scenarios with different degrees of heterogeneity.

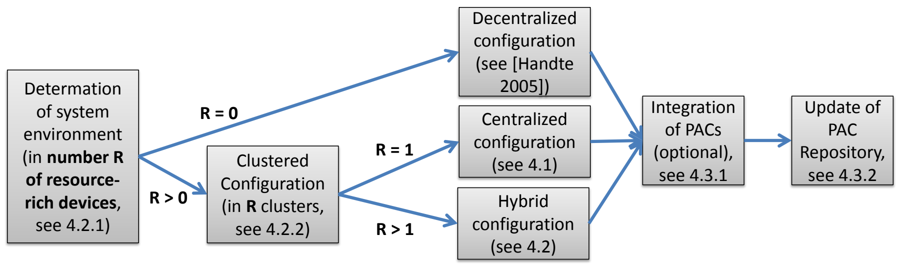

Determation of system environment (in number R of resource-rich devices, see 4.2.1)
R = 0
R > 0
Clustered Configuration (in R clusters, see 4.2.2)
R = 1
R > 1
Decentralized configuration (see [Handte 2005])
Centralized configuration (see 4.1)
Hybrid configuration (see 4.2)
Integration of PACs (optional), see 4.3.1
Update of PAC Repository, see 4.3.2

Fig. 2. Overview of integrated, scneario-aware configuration approach

In this article, we focus on these issues and present an integrated solution which automatically adapts to different environments and efficiently supports all kinds of scenarios. An overview of this solution is presented in Figure 2. We propose an efficient centralized algorithm [Schuhmann et al. 2008b] for weakly heterogeneous environments (e.g., offices) in Section 4.1. In Section 4.2, we discuss a hybrid scheme [Schuhmann et al. 2010] embedded in a clustering framework [Schuhmann et al. 2008a] which represents a generalization of the decentralized and centralized schemes. Finally, in Section 4.3, we introduce a novel approach that re-uses the results of previous configuration processes to reduce the latencies even further.

4.1. Centralized Configuration Approach

In weakly heterogeneous environments with several resource-weak and exactly one resource-rich device, this device can compose configurations much faster than the resource-poor devices, due to its significantly higher computation power. Therefore, we introduce an efficient centralized configuration algorithm called *Direct Backtracking* (DBT, [Schuhmann et al. 2008b]) for such weakly heterogeneous environments.

DBT avoids unnecessary adaptations that arise in many other backtracking algorithms without significantly increasing memory usage. This leads to considerably reduced latencies, as we will show in our evaluations. To perform centralized configuration completely local on the resource-rich device (i.e., without involving remote devices), DBT needs to proactively obtain the relevant resource information of the other devices. We will discuss a framework which supplies this functionality in Section 4.2.2.

4.1.1. General Approach. Like many other backtracking algorithms, DBT proceeds in a depth-first manner from the application root down to the leaf nodes. A *leaf contract* is the contract of a component C_a whose dependencies have not yet been fully resolved by assigning adequate child components. The algorithm tries to find a suitable component C_b satisfying this contract and assigns C_b to C_a. This is performed recursively until DBT either has found a suitable component for each dependency (i.e., it terminates with a valid configuration), or it could not find a component for at least one dependency (i.e., it terminates unsuccessfully). DBT features two intelligent mechanisms to render backtracking processes more efficient, as discussed in the following Sections 4.1.2 and 4.1.3. A more detailed description of DBT is given in [Schuhmann et al. 2008b].

4.1.2. Proactive Backtracking Avoidance. To avoid most resource conflicts and the subsequent backtracking processes from the start, DBT contains a *proactive mechanism* which cautiously selects components of multi-optional contracts. Within multi-optional contracts, components are ordered in a list according to their resource consumption: The component with least resource requirements has highest priority. If there are multiple components with the same resource consumption for a specific dependency, they are additionally ordered according to the ID of the device which is hosting them.

We assume that R is a *resource type* and that a certain amount of resources exists in the system for every resource type. At any time, some of these resources of R are allocated (used by some component) and the remaining resources are free (e.g., in a room, there are 4 screens of which 3 are currently used by some component and 1 is free). For any multi-optional contract, DBT initially selects the first (i.e., the highest-prior) component in its ordered list to be instantiated. If this component consumes the total free amount of R, the algorithm scans the ordered list for alternative components whose instantiation would leave some amount of R unused. Among those alternatives, DBT selects the highest-prior one to decrease the potential for future conflicts.

4.1.3. Intelligent Backtracking. If none of the possible components for a contract C can be instantiated, adaptation via backtracking has to be performed. Backtracking means that the algorithm changes the component assignment for one of the already configured contracts further up in the tree. It replaces components to make them available to the contract that produced the conflict. If more than one candidate for an adaptation exists, DBT performs the *intelligent backtracking* process described below that selects the contract to be adapted carefully to minimize the adaptation overhead.

Let the components that could not be instantiated because of a missing resource of type R form a set $S_1 = \{Cmp_1, Cmp_2, ..., Cmp_i\}$. If C is *not* multi-optional, S_1 includes only one component. DBT determines the set $D_s = \{d_1, d_2, ..., d_j\}$ of containers that host at least one component in S_1, i.e.

$$(\exists k, l : k \in \{1, ..., i\} \wedge l \in \{1, ..., j\} \wedge Cmp_k \in d_l) \Rightarrow d_l \in D_s \quad (1)$$

Let S_2 be the set that contains those multi-optional contracts for which a component is currently instantiated that is resident on one of the containers included in D_s. Moreover, S_2 contains only those components which can be adapted because an alternative component on another container is available. The components in S_2 are sorted in descending order according to the amount of R they consume. This means that the component which consumes the largest amount of R is at the beginning of the list, since its termination causes a considerable deallocation of resources. Proceeding in this order reduces the number of needless adaptations which would have to be revised later.

If only one suitable component exists as a backtracking target, the adaptation of the respective contract is initiated. In case of several suitable backtracking targets which consume an identical amount of R, an additional selection criterion is needed to judge their suitability for backtracking. Since the adaptation of a contract induces the adaptation of its subtrees, DBT selects the component that has the lowest number of subjacent child components to reduce the adaptation. In case of multiple contracts with a subtree of the same size, the algorithm selects the one whose currently instantiated component consumes the largest amount of resources.

If the resource conflict cannot be solved by adapting the first contract in S_2, DBT tries to solve it by adapting the second contract in S_2, and so on. If the conflict cannot be solved by adapting *any* contract included in S_2, then DBT jumps to the parent contract and continues with the intelligent backtracking process from here. If none of the contracts can be adapted to resolve the conflict, the algorithm terminates unsuccessfully and informs the user of this failure which happens due to missing resources.

4.2. Hybrid Configuration Approach

Neither the decentralized nor the centralized configuration approach can be applied efficiently in all perceivable environments: In a homogeneous environment, the decentralized algorithm should be used, while the centralized one should be applied in weakly heterogeneous environments. Moreover, for any setting between these two extremes, it should be possible to adapt the degree of decentralization accordingly.

Therefore, we propose a *hybrid scheme* [Schuhmann et al. 2010] with adaptable degree of decentralization in the following. This approach represents a generalization of the centralized and decentralized approaches and combines their advantages: It enables application configurations to be computed in a distributed fashion *simultaneously* on *multiple* devices, while exploiting the increased power of resource-rich devices.

We first present a simple algorithm to automatically switch between different configuration approaches. Then, we introduce a clustering framework in Section 4.2.2 for identifying resource-rich and resource-poor devices and enable *mappings* between different devices. *Mapping* means assigning a resource-poor device to exactly one resource-rich device which is then responsible for configuring this resource-poor device's components. The clustering framework enables an equal configuration load among the resource-rich devices. Then, we introduce the *Virtual Container* concept in Section 4.2.3, which enables the local emulation of remote devices. This concept increases the efficiency of the configuration in heterogeneous environments, as it enables a proactive loading of configuration information from mapped devices. Finally, we present an exemplary hybrid configuration process in Section 4.2.4.

4.2.1. Efficient Support of Exchangeable Configuration Algorithms. To adapt the degree of decentralization in application configuration, we introduce the *selector* abstraction as a mechanism to dynamically select the most suitable of the provided configuration algorithms in a specific scenario. Whenever a user starts an application, a selector decides based on the number of currently available resource-rich devices which configuration algorithm is chosen: In case of a homogeneous scenario with no resource-rich devices, a decentralized scheme [Handte et al. 2005] is executed on all devices. If exactly one resource-rich device is available, the selector initiates the centralized approach discussed in Section 4.1 on this resource-rich device. In case of multiple resource-rich devices, configurations are calculated using the hybrid algorithm introduced below.

As the information about the current environmental condition (degree of heterogeneity) is provided by the device registry of the middleware which is running on every device, the selectors on all devices have the same global view on the current environmental conditions, i.e., each device knows about the available resource-rich and resource-poor devices. Thus, when a user starts an application, the system software running on her device looks up the currently available devices and sends the information about a pending configuration to the relevant devices determined by the selector.

The selector abstraction is designed in a way that the implementation of additional strategies is easily possible if further configuration schemes are implemented. This represents a flexible solution for supporting an adaptable degree of decentralization in various homogeneous as well as heterogeneous Pervasive Computing scenarios.

4.2.2. Clustering Framework. To maximize the efficiency of configurations, we established a resource-aware mechanism for decentralized configurations, i.e., the distribution of the configuration tasks is based on the computation power of the devices. Therefore, we developed a *Clustering Framework* [Schuhmann et al. 2008a] that uses a distributed algorithm [Basagni 1999] to obtain the *cluster heads* which act as coordinators for the remaining *cluster members*. As clustering is performed prior to any configuration, this process causes additional latencies. Thus, the actual clustering scheme needs to guarantee stable clusters by avoiding re-clustering processes if possible.

To achieve this, the chosen node weight of the clustering algorithm is based on a device's specific computation power – a property which typically does not change over time. To measure its performance for configuration processes, each device initially performs a simple benchmark: It locally calculates a specific application configuration via DBT and, according to the arising latency, assigns itself a cluster weight $w \in [0, 1]$ – the faster the configuration was calculated, the higher the weight is chosen. Only devices with weights above a defined threshold declare themselves as cluster heads and actively calculate configurations. Therefore, we call them *Active Devices (ADs)*, while the resource-poor cluster members represent the *Passive Devices (PDs)*.

Each PD needs to be mapped to exactly one AD. To reduce the risk of possible bottlenecks, the number of PDs mapped to each AD should be balanced, even in case of changing device availabilities. Furthermore, each AD should not be required to know about the mappings at the other ADs. To achieve this, we developed a scheme which establishes balanced clusters in dynamic heterogeneous environments. It builds on the election of ADs and uniquely maps each PD to an AD such that the difference in cluster sizes is at most one. Therefore, a *round robin*-based scheme is used which determines the mappings according to the device IDs. A mapping procedure is initiated by an AD by sending a mapping request to the PD it wants to map. The PD reacts by transmitting its current resource information to the respective AD so that the AD can create a local representation of the remote PD. This scheme is performed in parallel on all ADs, as they map disjoint sets of PDs due to the unique device IDs.

Re-clustering is needed to maintain a balanced load in environments where devices may join or leave the network at any time. Our scheme avoids unnecessary merging and splitting of clusters by simply re-mapping single PDs. Re-clustering comprises four cases: The appearance of a new PD or a new AD, and the disappearance of a PD or an AD. As an example, let us discuss the case of a newly appearing PD here. Then, the round robin distribution of the PDs to the ADs is simply continued: as each AD knows about the number of ADs and PDs and the number of PDs which are mapped to itself, it can determine appropriate to the device IDs if it has to map this new PD to itself or not. For the sake of brevity, we refer to [Schuhmann et al. 2010] for further details.

4.2.3. Virtual Containers. Each cluster head acts as the coordinator for its mapped cluster members and is, thus, responsible for calculating configurations for this cluster. The concept of *Virtual Containers (VCs)* is used for managing the resources used for configurations on the cluster heads, enabling completely local configurations of the components within a cluster on the respective cluster head. Therefore, the cluster heads need to proactively acquire knowledge about the currently available components and resources on the mapped devices. To achieve this, we transmit the resource information of a cluster member to its cluster head. The cluster head then creates a VC which emulates the cluster member device by providing the respective resource and service information. As each cluster member automatically notifies its cluster head via events about changes in its resource conditions, the state of a VC is kept up to date. Through this concept, we decouple the configuration processes from the real devices.

4.2.4. Example. Figure 3 sketches an examplary hybrid configuration process: The scenario consists of two resource-rich devices – a desktop PC (AD 0) and a laptop (AD 1) – and three smart phones (PDs 0 to 2), representing the resource-poor devices. Initially, the cluster structure is established using the round robin scheme. This yields the desktop PC as cluster head for the PDs 0 and 2, and the laptop as cluster head for PD 1 (step 1). In step 2, the PDs transfer their current resource information to their respective ADs. Based on this information, the ADs build the local representations of the mapped PDs within VCs in step 3, yielding two VCs at AD 0, and one VC at AD 1.

When a user starts an application on her mobile device (PD 2) in step 4, the application's resource information is transmitted to AD 0 as the responsible cluster head for PD 2. Subsequently, AD 0 initiates the configuration of the application (step 5). At first, it verifies which of the dependencies can be resolved by components of its own container and the VCs representing its mapped PDs. Then, these dependencies are resolved locally on AD 0 using DBT, as presented in Section 4.1. AD 0 requests AD 1 to resolve the remaining, unresolved dependencies. AD 1 provides AD 0 with the respec-

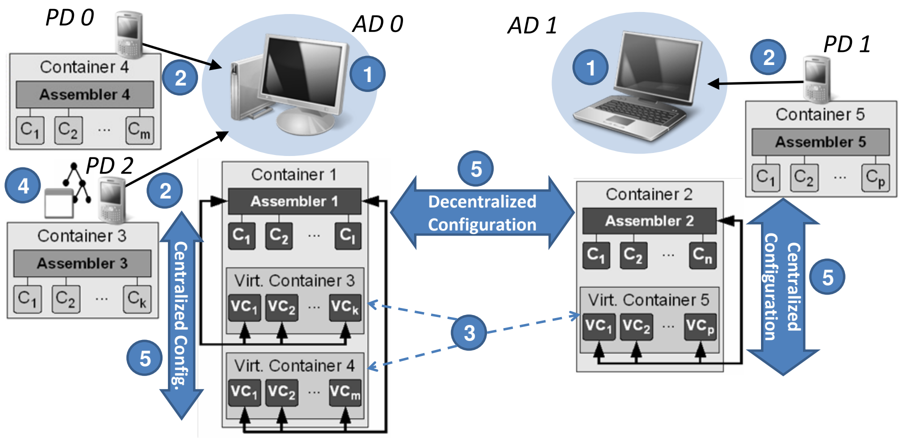


Fig. 3. Hybrid application configuration example

tive fitting components and the devices that host these components. This represents the decentralized part of the hybrid configuration. Subsequently, the complete configuration is constructed by AD 0. After the successful configuration, the PDs whose components are used in the configuration are notified by their cluster head about their component configurations. Finally, the components are initialized, the bindings between the components are established, and the application is successfully started.

4.3. Re-Use of Partial Application Configurations

In many pervasive environments, there is a fixed set of applications, devices and components which are frequently used, e.g. a presentation application in combination with a multimedia system, covering video projectors, microphones and the stationary speaker system at a conference venue (cf. Figure 1b). There, the same set of components is often used in successive configuration processes. Thus, the involved devices undergo a quite similar configuration process whenever an application is launched. Starting the composition from scratch every time not only consumes a lot of time, but also increases communication overhead and burdens the involved devices. These problems can be solved by providing pre-cached component sets which have been used in previous configuration processes. We refer to one of these component sets as a *Partial Application Configuration* (PAC). The components involved within a PAC represent a pre-computed subtree of the complete application tree. To enable their future reuse even in homogeneous scenarios where decentralized configuration is chosen, PACs are stored in a cache on all present devices after they were used in a configuration process. If all components included in a cached PAC are currently available, this PAC can be integrated into a newly calculated configuration without having to configure all the components in the PAC individually. This saves a lot of overhead and, thus, time.

As the cache size is typically limited on mobile devices, only those PACs with the largest expected utility for future configuration processes should be cached. To determine suitable PACs, we present our utility function in Section 4.3.1. Furthermore, a replacement strategy that decides which of the stored PACs are replaced if the cache space is exceeded is discussed in Section 4.3.2. Finally, we focus on issues concerning storage and communication overhead in Section 4.3.3.

4.3.1. PAC Utility Value. The utility value of a PAC is introduced for expressing the expected benefit of using a PAC within a configuration process. The more components of an application tree a PAC covers, the more its use decreases the configuration latency, as a larger number of components does not need to be configured individually. The two most important factors for defining the utility of a PAC for configuration processes are

— *how often* a PAC can be used in configuration processes (i.e., the frequency with which the PAC is usable), and
— the *expected latency reduction* when a PAC is used (i.e., to which extent the PAC covers the application tree).

Both of these factors depend on the size of the PAC and the devices involved in a PAC: Small PACs involve few devices (that have to be present) and are more versatile (can potentially be used in more configuration processes). Moreover, small PACs can be stored in a cache more efficiently: While large PACs may not fit into the cache due to their size, small PACs fit more often. However, large PACs have a higher potential for reducing latencies drastically since many components are already configured. Thus, neither providing *only large* PACs, nor *only small* PACs is obviously desirable.

We use the utility function proposed for web caching [Lee et al. 2001] and for adaptive overlay networks [Dürr and Rothermel 2008], since this function can compactly represent how often and how recently a PAC was used. In our context, this function defines the utility, $u_p(t)$, of a PAC p at time t. Each device that is involved in the configuration process updates the utility of each cached PAC at time t_c when a configuration has finished. The utility $u_p(t) \in [0, 1]$ of PAC p at time $t_c = t' + \Delta t$ of the current configuration (where t' = time of the previous configuration, Δt = time span between the previous and the current configuration) is calculated as follows:

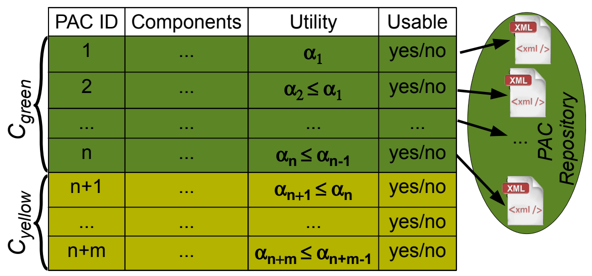
$$u_p(t_c) = \begin{cases} u_p(t') \cdot f(\Delta t) + f(\Delta t), & \text{if PAC } p \text{ was usable} \\ u_p(t') \cdot f(\Delta t), & \text{if PAC } p \text{ was unusable} \end{cases} \tag{2}$$

As proposed by related approaches (e.g., [Bahn et al. 2002]), we use the monotonously decreasing function f(Δt) = $0.5^{\lambda \cdot \Delta t}$ with $\lambda \in [0, 1]$ that defines the influence of reference recency and frequency, leading to a combined Least Recently/Frequently Used (LRFU) policy. With this function, a PAC's utility value is increased if the PAC was usable at time t_c (i.e., all PAC components were available), and reduced if it was not usable.

With the radix of 0.5 in $f(\Delta t)$, switching to pure Least Frequently Used (LFU, $\lambda = 0$) and Least Recently Used (LRU, $\lambda = 1$) strategies is easily possible by adjusting λ. The question of how to choose λ is discussed in the evaluation section.

4.3.2. PAC Cache Maintenance. According to the PAC utility values, the configuration algorithm decides which PACs should be cached for future configuration processes. As the cache space is limited, it has to be used as efficiently as possible. Therefore, we distinguish between two different types of PACs that are relevant for PAC configuration:

— **Green PACs** represent the PACs with highest current utility values. The PAC information that is relevant for configuration processes – the PAC structure, the devices that host the PAC components, and the utility values – is stored in the cache table. Additionally, the complete representation of a PAC in PCOM's specific XML format is stored in the so-called *PAC Repository*. This representation enables the simple inclusion of the PAC into the complete application assembly in a configuration.
— **Yellow PACs** are those which were previously used less frequently, leading to a lower utility value than green PACs. The structure, the hosting devices, and the utility values of yellow PACs are recorded in the cache, but *not* their complete XML information. Thus, a yellow PAC consumes only around 1/6 of the space which the equivalent green PAC would consume. Yellow PACs are not directly usable in configurations, but the information stored in them can be used to quickly create a green



PAC ID	Components	Utility	Usable
1	...	α_1	yes/no
2	...	$\alpha_2 \leq \alpha_1$	yes/no
...	...	...	...
n	...	$\alpha_n \leq \alpha_{n-1}$	yes/no
n+1	...	$\alpha_{n+1} \leq \alpha_n$	yes/no
...	...	...	yes/no
n+m	...	$\alpha_{n+m} \leq \alpha_{n+m-1}$	yes/no

Fig. 4. Cache structure of C with C_{green}, the PAC Repository, and C_{yellow}

PAC: When the utility value of a yellow PAC increases in the future, the respective XML file is created, enabling the direct use of this PAC. Providing yellow PACs reduces cache miss rates, as our evaluations will show.

The remaining PACs – those which are neither green nor yellow – are not relevant for configuration, either because they have never been referenced before, or a long time ago and, hence, they have been removed from the cache in the meantime. Thus, no information is currently stored about them in the tables.

Figure 4 shows that green and yellow PACs are maintained in the so-called PAC Cache C of size $|C|$, which is divided into the two areas C_{green} and C_{yellow} that hold the respective PAC information. C_{yellow} can be regarded as a "waiting area", allowing PACs to increase their utility values due to recent and frequent usage and becoming green PACs in the future. Each entry in C contains the following fields:

— the involved PAC components and the devices which host these components,
— the utility value of the PAC, and
— its current availability, as determined by the conjunction of the availabilities of the involved components. Only currently available PACs are usable in a configuration.

C_{green} is of limited size $|C_{green}| < |C|$ and covers the size of the PAC entries in the cache table as well as the respective XML representations in the PAC repository. The sizes of C_{green} and C_{yellow} sum up to the total cache size $|C| = |C_{green}|+|C_{yellow}|$. A main issue in the evaluations is to find the optimal partitioning of C into C_{green} and C_{yellow} to maximize cache efficiency. Therefore, we introduce the *split factor* $f \in [0,1]$ which determines the relative amount of $|C|$ that is available for C_{yellow} ($|C_{yellow}| = f \cdot |C|$). Reasonable values for f are identified in Section 5.3. Since the PAC utilities change over time, e.g., due to unavailability of components, the utility value of a yellow PAC may exceed the utility value of a PAC stored in C_{green}. We define the Least-Green PAC, $p_{lg}(t)$, as the PAC with the minimum utility value in C_{green} at a specific time t:

$$\forall p \in C_{green} : u_{p_{lg}}(t) \leq u_p(t) \tag{3}$$

The PAC p' with highest utility value in C_{yellow} replaces p_{lg} at configuration time t_c, if

$$u_{p'}(t_c) > u_{p_{lg}}(t_c) \tag{4}$$

Subsequently, the PAC assembly is created and added for the (now green) PAC p', while p_{lg} is moved to C_{yellow} and its assembly is removed. PAC replacements are made until no yellow PAC p' exists for which the condition in Equation 4 is fulfilled.

4.3.3. PAC Overhead. Providing a PAC cache induces overheads concerning disk space and communication (for transferring the PAC Repository to newly arriving devices). For both types of overhead, the size $|C|$ that is reserved for the PAC cache is the relevant parameter. As this parameter highly influences the performance of the PAC con-

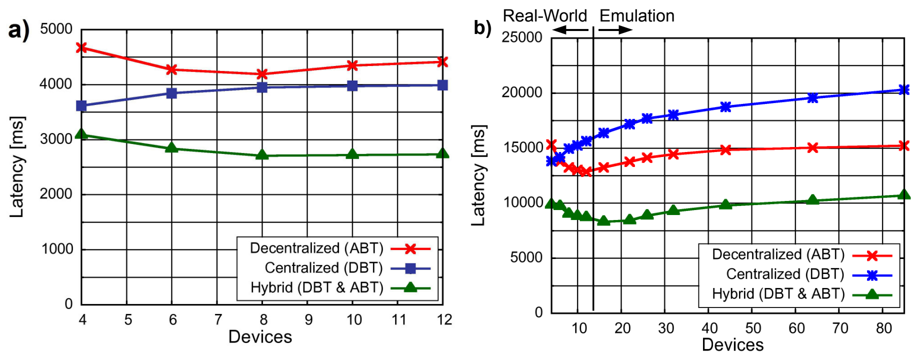


Fig. 5. Hybrid configuration latencies, compared to centralized and decentralized approaches in strongly heterogeneous environment with a) 31 and b) 127 components

cept, we take a closer look on the cache size $|C|$ in Section 5.3.1 and evaluate various cache sizes to find a trade-off between good performance and low overhead.

When using the utility function $u_p(t)$, the utility value of a PAC p decreases monotonically between two consecutive references to p. Thus, two PACs that are not referenced do not change their order in the cache. We only need to compare the utility of a yellow PAC p_y with the utility of p_{lg} when p_y is referenced, ensuring an efficient realization.

5. EVALUATION

5.1. Evaluation Setup

We evaluated our approach using applications with a binary tree structure and tree heights between 2 and 6, yielding applications with 7 up to 127 components, i.e., in the same order of magnitude in size than the applications discussed in Section 2.2. The components represent single functionalities required by a distributed presentation application [Handte et al. 2006] which takes advantage of the input and output components in the surrounding of a lecturer. Each value given in the graphs represents the average of 50 measurements. We implemented the concepts in a real-world prototype of the component system PCOM [Becker et al. 2004] and performed measurements on devices which are typically used in Pervasive Computing environments: Laptops (Pentium Centrino CPU, 1.6 GHz) represent the resource-rich devices, while the resource-poor devices are represented by smart phones (PXA 270 CPU, 528 MHz). We used IEEE 802.11b as standard wireless communication technology and rely on the device and service registry provided by the communication middleware BASE [Becker et al. 2003]. In larger-scale scenarios with more than 12 devices, we used the network emulation simulator NET [Grau et al. 2009] in combination with our system software to provide realistic results.

5.2. Configuration in Weakly and Strongly Heterogeneous Environments

To measure the performance of the hybrid approach in strongly heterogeneous scenarios, we compared its overall latencies with those of the decentralized (ABT) and centralized (DBT) approaches. Here, we used two different application sizes (31 and 127 components), different numbers of devices, and 50 % resource-rich devices in each scenario. Figures 5a (for an application with 31 components) and b (for an application with 127 components) show the overall latencies. As shown on the x-axes, the evaluation of the small application was performed with 4 to 12 devices, while the evaluation of the larger application involved up to 85 devices in the emulations. The figures show that the latencies of hybrid configuration are lowest, thanks to the resource-aware distribu-

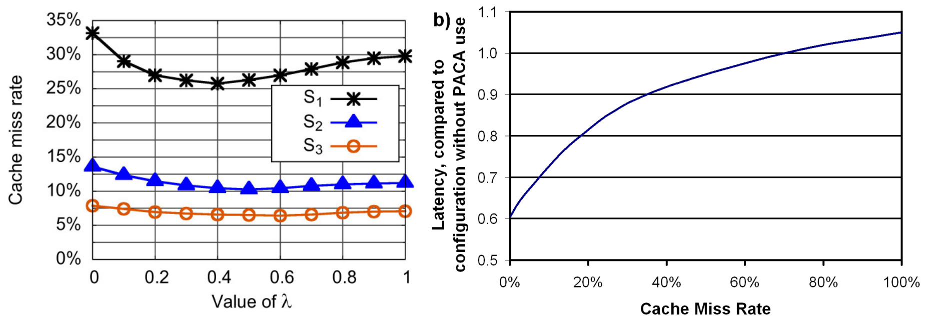


Fig. 6. a) Investigation of optimal static λ values for S_1, S_2 and S_3, b) Correlation between PAC cache miss rate and configuration latency

tion of the configuration tasks among the resource-rich devices, and as calculations can be performed in parallel as multiple resource-rich devices are involved. In the small application scenario, the centralized approach performs better than the decentralized approach, which however reverts in the larger-scale scenario since the available computation resources distributed among the resource-rich devices are not exploited by the centralized approach (here, always one device calculates the *complete* configuration). With an increasing number of devices, latencies of hybrid and decentralized configuration first drop. This happens due to a larger *absolute* number of resource-rich devices which are involved in calculating a configuration, while in centralized configuration, only *one* resource-rich device is always used to calculate configurations. When the total number of devices exceeds 12 (for decentralized configuration) or 16 (for hybrid configuration), the overall latencies slightly rise again, as the increased latencies for establishing the component bindings exceed the reduced latencies for the configuration calculation. The latencies of the centralized approach show continuous growth, as the overhead for the result distribution increases with a rising number of devices. At the same time, the configuration latency remains constant, as the configuration is performed locally on the cluster head.

The hybrid approach outperforms the decentralized approach by 35.7 % (31 components) and by 34.5 % (127 components) on average. Moreover, the hybrid approach outperforms the centralized approach by 26.3 % (31 components) and by 44.1 % (127 components) on average. As the latency reduction is still achieved for large applications with many involved devices, the hybrid approach scales well. More detailed evaluation results concerning latencies at the different stages of the configuration can be found in [Schuhmann et al. 2010].

5.3. Adaptive Configuration in Static and Dynamic Environments

In a first step, we investigate three static scenarios:

(1) a homogeneous environment (scenario S_1), consisting only of resource-poor devices;
(2) a weakly heterogeneous environment (S_2), including one additional resource-rich infrastructure device;
(3) a strongly heterogeneous environment (S_3), consisting of 50 % resource-rich devices and 50 % resource-poor devices.

Additionally, we evaluate the PAC concept in dynamically changing environments. We used sizes of 7 (S_1), 15 (S_2), and 31 (S_3) components.

5.3.1. Evaluation in Static Environment. First, we determine the optimal static values for λ for the LRFU strategy in the different scenarios, i.e., the values where the cache miss rate becomes minimal. In these initial measurements, we set $|C_{yellow}| = 0$. In Figure 6a, it can be seen that neither choosing pure LFU ($\lambda = 0$) nor choosing pure LRU ($\lambda = 1$) leads to the best results. On the one hand, the recency is relevant, as we consider dynamic environments where components may be available only for a limited amount of time. Thus, relying on PACs which have recently shown to be usable makes sense. On the other hand, the frequency also needs to be regarded, as devices which were previously available, but are unavailable now may return again in the future. In this case, it needs to be considered how often PACs have been used before, leading to a higher utility for these PACs. Thus, both the recency and the frequency of a PAC's availability need to be taken into account to minimize the cache miss rate. The optimal λ changes from 0.4 (S_1) to slightly higher values of 0.5 (S_2) and 0.6 (S_3), since the degree of dynamics decreases and, thus, the recency of a PAC's availability becomes more relevant, as the PACs are valid for a longer average period of time in heterogeneous scenarios. We decided to use a static value of $\lambda = 0.5$ in the following measurements, as this leads to a low cache miss rate in *all* three environments.

Next, we are looking for reasonable values for the fixed cache size $|C|$ as well as for the split factor f. For this purpose, suitable overall cache sizes for the different environments were investigated. The user obviously wants to have an application to be configured as fast as possible, which is the case when the cache miss rate is low. However, no cache misses at all can only be achieved with an unbounded cache, which is not possible in practical use, particularly in resource-constrained homogeneous environments. Thus, there is a trade-off between small cache size and low cache miss rate. Figure 6b shows the correlation between the cache miss rate and the expected configuration latency, relative to the latency of a configuration that does not rely on PACs at all. The configuration latency is lowest when no cache misses appear at all, and rises monotonously with rising cache miss rates. In case of cache miss rates close to 1 (i.e., none of the application contracts are covered by a PAC), the configuration latency is slightly higher than standard configuration without PACs, as the contract matching process for the cached PACs requires some additional time in each configuration.

Figure 7 shows the cache miss rate (depicted in z-axis) with variable overall cache sizes $|C|$ (x-axis) and split factors f (y-axis) in the evaluated weakly (a) and strongly (b) heterogeneous scenarios. It can be seen that the cache miss rate is comparatively high in both scenarios in case of strict cache limits and for large values of f. In this case, the cache space for C_{green} becomes the limiting factor, yielding only few PACs that actually fit into the cache. Moreover, the contour lines of the figures (which represent the 2.5 %, 5 %, 7.5 %, and 10 % cache miss rate bounds) show that the more resources the environment has, the smaller the achieved cache miss rates become. For example, if the cache miss rate should not exceed 10 %, then one can infer from Figures 7a - c that a cache size of 400 kB is sufficient to stay below this limit: With $|C| = 400\ kB$, the cache miss rates are 8.6 % (S_2), and 7.3 % (S_3) and, thus, below 10 % in both scenarios. Regarding the choice of f, the cache miss rates become minimal if we choose values of 16.3 % (S_2) and 25.1 % (S_3) for f. The respective data points $X_2(400\ kB, 16.3\ \%, 8.6\ \%)$ and X_3 $(400\ kB, 25.1\ \%, 7.3\ \%)$ are drawn in Figures 7a and b.

5.3.2. Evaluation in Dynamic Environment with Adaptive Parameters. Since static parameters perform suboptimal in dynamic environments, we now focus on adaptive parameters for f and λ that are automatically adjusted to changing environments. For this purpose, we simulate the user mobility using a *General Pareto Distribution*, as it has been found that the periodically changing availability and disavailability of mobile users in several user studies (taken e.g. in University campuses, corporate environments, research conferences) typically follows this propability distribution [Karagiannis et al. 2007]. $|C|$ is set to 400 kB in all experiments since this value provided

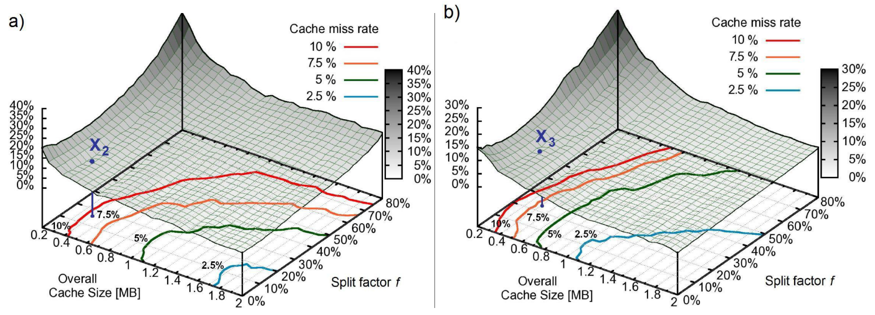


Fig. 7. Distribution of Cache Miss Rate in a) S_2, b) S_3

good performance in all scenarios. Note that adapting $|C|$ would require to allocate or de-allocate memory every time $|C|$ is changed, yielding significant overhead especially on the resource-poor mobile devices. From our experiences with the developed configuration approaches, we found that the ratio s of resource-rich devices (i.e., $s = \frac{\#\ resource-rich\ devices}{\#\ all\ devices}$) available in the environment is the most important factor for the choice of an adequate configuration scheme. Thus, we adapt f and λ based on s.

At first, we focus on an adaptive split factor f. Therefore, we evaluated the arising cache miss rates with various values of s (from 0 % to 40 %) and determined the optimum value for f in each scenario. To find if there are interdependencies between λ and f, we performed these measurements with three different values of λ: 0.25, 0.5, and 0.75. The results of these measurements are shown in Figure 8a. This figure reveals two things: 1.) The results with varying λ values are very close to each other. We infer from these results that the parameters f and λ have little interdependency on each other. 2.) The optimum f values (i.e., the values where the cache miss rate becomes minimal) rise with increasing values of s. This is since an increasing number of resource-rich devices leads to a lower degree of dynamics, yielding less frequent replacements in the cache. Thus, the size $|C_{green}|$ is not that crucial as in higher dynamic scenarios. So, increasing $|C_{yellow}|$ at the cost of $|C_{green}|$ reduces the cache miss rate, as the cache stores more yellow PACs then, which may possibly change to green PACs after some time.

Figure 8b shows the optimal λ values (i.e., the values where the cache miss rate becomes minimal) which we gained by evaluating the same scenarios with the determined adaptive f values and different λ values between 0.0 and 1.0. The optimal λ becomes slightly larger with a rising number of strong devices, since the degree of dynamics decreases and, thus, the recency of a PAC's availability becomes more relevant, as the PACs are valid for a longer average period of time. Typical optimal values for λ are around 0.3 in case of few strong devices ($s \leq 15\ \%$), and rise to around 0.6 as the number of strong devices in the environment increases to $40\ \%$.

Finally, we performed evaluation runs over a time span of 3600 s where we periodically configured an application with period time 50 s, i.e. we performed 72 configurations in total. The availability of each device was randomly and dynamically changed using a Pareto distribution. When a configuration was started, the configuration framework accessed the device registry and determined the number of currently available resource-rich devices. Then, the fitting configuration scheme – decentralized, centralized, or hybrid – was chosen according to the selection strategy presented in Section 4.2.1. We compared the following configuration approaches to each other:

— Configuration without PAC use, i.e., no PAC Repository was established and standard hybrid, decentralized, or centralized configuration was performed.

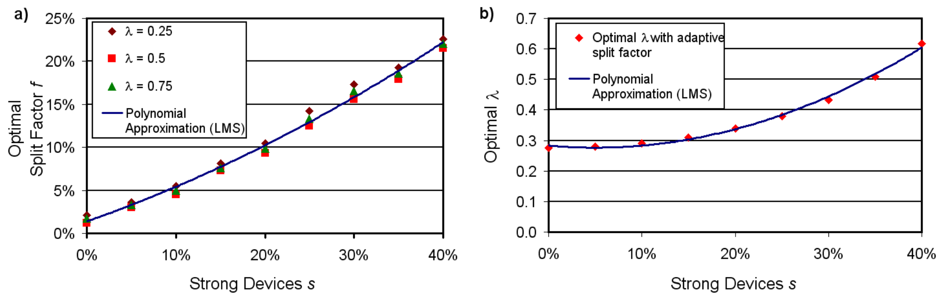


Fig. 8. Investigation of a) optimal adaptive split factor f, b) optimal adaptive λ

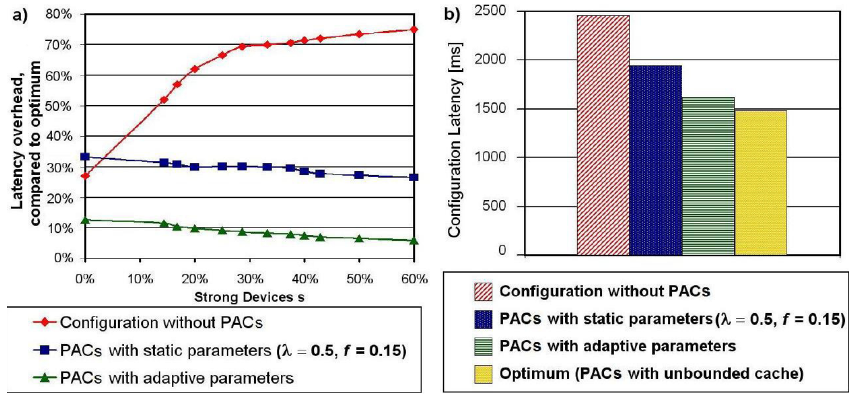


Fig. 9. Latency measurements with Static and Adaptive Cache Parameters: a) Depending on s, b) Average

— Configuration with PACs ($|C| = 400\ kB$) and static parameters ($f = 0.15$, $\lambda = 0.5$).
— Configuration with PACs ($|C| = 400\ kB$) and adaptive f and λ (cf. Figure 8).
— Configuration with PACs with an unbounded cache ($|C| = \infty$), i.e. every used PAC is cached forever in C_{green} and is never replaced, which represents the *optimum* (but infeasible) case for configuration.

Figure 9a shows the latency overhead compared to the optimum case with different values for s. Using PACs with adaptive parameters yields low latencies that are very close to the optimum with only 9 % slowdown. Using static PAC parameters leads to an increased overhead of 31 % compared to the optimum, since the parameters do not adapt to dynamically changing scenarios. Configuration without PACs performs worst with an average overhead of 66 % and can only compete with PAC configuration that uses static parameters when no resource-rich devices are available. Figure 9b shows the overall average latencies. This figure illustrates that PAC use which relies on adaptive parameters is only around 150 ms slower than the optimum case of PAC use in combination with an unbounded cache, despite the 400 kB cache size restriction. When PACs are used with static parameters, the configuration is around 450 ms slower than the fastest possible configuration, so you see that dynamic parameters increase the efficiency of the PAC approach. Standard configuration without PAC use needs around 950 ms more time than the optimum, but works without a cache.

5.4. Summary

In our evaluations, we have shown three significant results:

(1) In weakly heterogeneous environments, a centralized configuration approach with proactive loading of the relevant resource information from other devices reduces the latencies drastically. If the configuration is calculated on the resource-richest device, latencies drop by almost $40\ \%$ compared to decentralized configuration.
(2) In strongly heterogeneous environments, a hybrid scheme reduces the configuration latencies by around $35\ \%$ both compared to centralized and to decentralized configuration. This is because configurations are calculated cooperatively and in parallel by the resource-rich devices, without involving the resource-poor devices.
(3) The re-use of previous configuration results further decreases the configuration latencies in *all* of the regarded scenarios by up to $31\ \%$ when static parameters are used, and even by up to $66\ \%$ when adaptive parameters are used. Adaptive parameters change their values according to changes in pervasive environments.

6. RELATED WORK

Numerous projects such as Olympus [Ranganathan et al. 2005], Gaia [Román et al. 2002] or Aura [Poladian et al. 2004] provide middleware systems for configuration of distributed applications in heterogeneous environments. They supply services for the development of context-aware mobile applications, and represent highly integrated environments supporting stationary as well as mobile devices. Some projects such as PICO [Kalasapur et al. 2007] additionally put the satisfaction of user preferences in their focus [Fujii and Suda 2009; Mukhtar et al. 2011]. Relying on infrastructure support, all of the mentioned systems are however not suited for the use in homogeneous scenarios. Most recent approaches for heterogeneous scenarios introduce self-organizing overlay networks [Al-Oqily and Karmouch 2011] or autonomic applications (AutoHome, [Bourcier et al. 2011]) in service composition. Moreover, some of them aim at coordination frameworks for service composition [Majuntke et al. 2010], applications interacting with public displays [Clinch et al. 2012], semantic quality metrics of service composition [Lecue and Mehandjiev 2011], or trustworthy service composition [Hang and Singh 2011].

In recent years, many projects like MundoCore [Aitenbichler et al. 2007] or RUNES [Costa et al. 2007] aim at providing system support for configuration in homogeneous scenarios. These projects provide automatic application configuration, whereas other peer-to-peer based approaches assign this task to the application programmer or the user (e.g., Pervasive Collaboration [Pering et al. 2009], Medusa [Davidyuk et al. 2010]). All of these projects are suited for various environments. However, they do not benefit from the increased computation power of resource-rich devices, yielding inefficient configurations in heterogeneous environments.

The component system PCOM [Becker et al. 2004], embedded in the research project 3PC [Handte et al. 2012] initially was also developed to provide system support for homogeneous environments. With the extensions discussed here, PCOM now adaptively supports a broad spectrum of pervasive scenarios.

None of the mentioned projects provides concepts for the re-use of previous configuration results. However, this re-use approach has already been introduced in applications for distributed media device control (e.g., OSCAR [Newman et al. 2008]). These systems rely on end-user composition, i.e., they do not provide *automated* configuration by the system, but *manual* composition by the user. Beyond this, systems like CAMP [Truong et al. 2004] or iCAP [Dey et al. 2006] even rely on end-user *programming*, thus by far not achieving the grade of pervasiveness we want to attain.

7. CONCLUSIONS

In this article, a comprehensive set of concepts and mechanisms for the efficient and adaptive composition of distributed applications both in heterogeneous and in homogeneous pervasive scenarios were presented. Different configuration schemes that are specifically suited for efficient application composition in different Pervasive Computing environments were proposed. Furthermore, a concept to enable automatic switching between the different configuration approaches in case of dynamic changes in the environment was presented. Finally, an approach for automatically caching Partial Application Configurations (PACs) and for including them in new configurations was presented. This leads to a large reduction of the configuration overhead.

Our work drastically lowers configuration latencies in the complete spectrum of possible pervasive scenarios: While centralized configuration reduces latencies in weakly heterogeneous environments by 39 % compared to decentralized configuration, hybrid configuration on the subset of resource-rich devices further reduces latencies in strongly heterogeneous environments by around 35 %. The additional use of pre-cached PACs leads to very efficient configuration processes that produce only around 9 % latency overhead compared to the fastest possible configuration where the complete application was pre-configured. In summary, our work leads to much less user distraction and, in consequence, to more seamless configurations processes. Thus, this work represents a large step towards seamless user support, which is essential to make computer systems disappear in the background and achieve true pervasiveness.

REFERENCES

E. Aitenbichler, J. Kangasharju, and M. Mühlhäuser. 2007. MundoCore: A light-weight infrastructure for Pervasive Computing. *Pervasive Mob. Comput.* 3, 4 (Aug. 2007), 332–361.

I. Al-Oqily and A. Karmouch. 2011. A Decentralized Self-Organizing Service Composition for Autonomic Entities. *ACM TAAS* 6, 1, Article 7 (February 2011), 18 pages.

H. Bahn, K. Koh, S. L. Min, and S. H. Noh. 2002. Efficient Replacement of Nonuniform Objects in Web Caches. *IEEE Computer, 35(6)* (2002).

S. Basagni. 1999. Distributed Clustering for Ad Hoc Networks. In *Proceedings of ISPAN'99*. 310–315.

C. Becker, M. Handte, G. Schiele, and K. Rothermel. 2004. PCOM - A Component System for Pervasive Computing. In *Proceedings of IEEE PerCom '04*. Orlando, FL, USA.

C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. 2003. BASE - A Micro-broker-based Middleware For Pervasive Computing. In *Proceedings of IEEE PerCom'03*. 443–451.

J. Bourcier, A. Diaconescu, P. Lalanda, and J. A. McCann. 2011. AutoHome: An Autonomic Management Framework for Pervasive Home Applications. *ACM TAAS* 6, 1, Article 8 (Feb. 2011), 10 pages.

J. Cámara, C. Canal, and G. Salaün. 2008. Multiple Concern Adaptation for Run-time Composition in Context-Aware Systems. *Electronical Notes on Theoretical Computer Science* (2008).

S. Clinch, J. Harkes, A. Friday, N. Davies, and M. Satyanarayanan. 2012. How close is close enough? Understanding the role of cloudlets in supporting display appropriation by mobile users. In *PerCom*. 122–127.

P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. 2007. The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario. In *Proc.s of IEEE PerCom'07*.

O. Davidyuk, N. Georgantas, V. Issarny, and J. Riekki. 2010. MEDUSA: Middleware for End-User Composition of Ubiquitous Applications. *Handbook of Research on Amb. Intell. and Smart Env.s* (2010).

A. K. Dey, T. Sohn, S. Streng, and J. Kodama. 2006. iCAP: Interactive prototyping of context-aware applications. In *Proceedings of Pervasive '06*.

F. Dürr and K. Rothermel. 2008. An Adaptive Overlay Network for World-wide Geographic Messaging. In *Proceedings of IEEE AINA '08*.

W. Edwards, M. Newman, J. Sedivy, T. Smith, D. Balfanz, D. Smetters, H. Wong, and S. Izadi. 2002. Using speakeasy for ad hoc peer-to-peer collaboration. In *Proc.s of CSCW'02*. ACM, 256–265.

L. Ferrari and M. Mamei. 2011. Discovering City Dynamics through Sports Tracking Applications. *IEEE Computer* 44, 12 (2011), 63–68.

K. Fujii and T. Suda. 2009. Semantics-based context-aware dynamic service composition. *ACM TAAS* (May 2009).

A. Grau, K. Herrmann, and K. Rothermel. 2009. Efficient and Scalable Network Emulation Using Adaptive Virtual Time. *Int. Conf. on Computer Communications and Networks* 0 (2009), 1–6.

R. Grimm. 2004. One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing, 3(3)* (2004).

M. Handte, C. Becker, and K. Rothermel. 2005. Peer-based Automatic Configuration of Pervasive Applications. In *Journal on Pervasive Computing and Communications*, Vol. 1. Troubador Publish. Issue 4.

M. Handte, K. Herrmann, G. Schiele, and C. Becker. 2007. Supporting Pluggable Configuration Algorithms in PCOM. In *Proc.s of PerCom'07 Workshops*. IEEE Computer Society, 472–476.

M. Handte, K. Herrmann, G. Schiele, C. Becker, and K. Rothermel. 2006. Automatic Reactive Adaptation of Pervasive Applications. In *Proceedings of ICPS'06*. IEEE Computer Society, 214–222.

M. Handte, G. Schiele, V. Majuntke, C. Becker, and P. J. Marrón. 2012. 3PC: System support for adaptive peer-to-peer pervasive computing. *ACM TAAS* 7, 1, Article 10 (May 2012), 10:1–10:19 pages.

M. Handte, S. Urbanski, C. Becker, P. Reinhardt, M. Engel, and M. Smith. 2006. 3PC/MarNET Pervasive Presenter. In *Demo Session at IEEE PerCom '06*. Pisa, Italy.

C.-W. Hang and M. P. Singh. 2011. Trustworthy Service Selection and Composition. *ACM TAAS, 6(1)* (2011).

S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen. 2005. The Gator Tech Smart House: A Programmable Pervasive Space. *IEEE Computer* 38 (2005).

K. Herrmann, K. Rothermel, G. Kortuem, and N. Dulay. 2008. Adaptable Pervasive Flows - An Emerging Technology for Pervasive Adaptation. In *Proceedings of IEEE SASOW*. IEEE, 108–113.

S. Kalasapur, M. Kumar, and B. A. Shirazi. 2007. Dynamic Service Composition in Pervasive Computing. *IEEE Transactions on Parallel and Distributed Systems* 18 (2007), 907–918.

T. Karagiannis, J.-Y. Le Boudec, and M. Vojnović. 2007. Power Law and Exponential Decay of Inter Contact Times between Mobile Devices. In *Proc.s of MobiCom'07*. Montréal, Québec, Canada.

F. Lecue and N. Mehandjiev. 2011. Seeking Quality of Web Service Composition in a Semantic Dimension. *IEEE TA on Knowledge and Data Engineering* 23 (2011), 942–959.

D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim. 2001. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE TA on Computers* (2001).

V. Majuntke, G. Schiele, K. Spohrer, M. Handte, and C. Becker. 2010. A Coordination Framework for Pervasive Applications in Multi-user Environments. In *Proceedings of the 6th International Conference on Intelligent Environments (IE '10)*. IEEE Computer Society, 178–184.

H. Mukhtar, D. Belad, and G. Bernard. 2011. Dynamic User Task Composition Based on User Preferences. *ACM TAAS, 6(1)* (February 2011).

M. Newman, A. Elliott, and T. Smith. 2008. Providing an Integrated User Experience of Networked Media, Devices, and Services through End-User Composition. In *Proc.s of Pervasive'08*. Springer.

T. Pering, R. Want, B. Rosario, S. Sud, and K. Lyons. 2009. Enabling Pervasive Collaboration with Platform Composition. In *Proc.s of Pervasive'09*. Springer, 184–201.

V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. 2004. Dynamic Configuration of Resource-Aware Services. In *Proceedings of ICSE '04*. IEEE Computer Society, 604–613.

A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. Campbell, and M. Mickunas. 2005. Olympus: A High-Level Programming Model for Pervasive Computing Environments. In *Proc.s of IEEE PerCom '05*.

M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt. 2002. Gaia: A Middleware Infrastructure to Enable Active Spaces. In *IEEE Pervasive Computing*, Vol. 6. Issue 4.

U. Sadiq, M. Kumar, A. Passarella, and M. Conti. 2011. Modeling and Simulation of Service Composition in Opportunistic Networks. In *Proc. of ACM MSWiM'11*. New York, NY, USA, 159–168.

S. Schuhmann, K. Herrmann, and K. Rothermel. 2008a. A Framework for Adapting the Distribution of Automatic Application Configuration. In *Proceedings of ACM ICPS '08*. Sorrento, Italy.

S. Schuhmann, K. Herrmann, and K. Rothermel. 2008b. Direct Backtracking: An Advanced Adaptation Algorithm for Pervasive Applications. In *Proceedings of ARCS 2008*. Springer, 53–67.

S. Schuhmann, K. Herrmann, and K. Rothermel. 2010. Efficient Resource-Aware Hybrid Configuration of Distributed Pervasive Applications. In *Proceedings of Pervasive 2010*. Helsinki, Finland.

K. N. Truong, E. M. Huang, and G. D. Abowd. 2004. CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In *Proceedings of UbiComp '04*.

C. Whitman, C. Reid, J. von Klemperer, J. Radoff, and A. Roy. 2008. New Songdo City – The Making of a New Green City. In *Proceedings of the CTBUH 8th World Congress*.

M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. 1998. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE TA on Knowledge and Data Engineering* 10, 5 (1998).

F. Zambonelli. 2011. Pervasive urban crowdsourcing: Visions and challenges. In *PerCom*. 578–583.

F. Zambonelli and M. Viroli. 2011. A survey on nature-inspired metaphors for pervasive service ecosystems. *Int. J. Pervasive Computing and Communications* 7, 3 (2011), 186–204.