Efficient Resource-Aware Hybrid Configuration of Distributed Pervasive Applications

Stephan Schuhmann, Klaus Herrmann, Kurt Rothermel

University of Stuttgart, Institute of Parallel and Distributed Systems (IPVS),
Universitätsstr. 38, 70569 Stuttgart, Germany
{schuhmann, herrmann, rothermel}@ipvs.uni-stuttgart.de

Abstract. As the size and complexity of Pervasive Computing environments increases, configuration and adaptation of distributed applications gains importance. These tasks require automated system support, since users must not be distracted by the (re-)composition of applications. In homogeneous ad hoc scenarios, relying on decentralized configuration schemes is obviously mandatory, while centralized approaches may help to reduce latencies in weakly heterogeneous infrastructure-based environments. However, in case of strongly heterogeneous pervasive environments including several resource-rich and resource-weak devices, both approaches may lead to suboptimal results concerning configuration latencies: While the resource-weak devices represent bottlenecks for decentralized configuration, the centralized approach faces the problem of not utilizing parallelism. Instead, a hybrid approach that involves only the subset of resource-rich devices is capable of rendering configuration and adaptation processes more efficiently. In this paper, we present such a resource-aware hybrid scheme that effectively reduces the time required for configuration processes. This is accomplished by a balanced-load clustering scheme that exploits the computational power of resource-rich devices, while avoiding bottlenecks in (re-)configurations. We present real-world evaluations which confirm that our approach reduces configuration latencies in heterogeneous environments by more than 30% compared to totally centralized and totally decentralized approaches. This is an important step towards seamless application configuration.

1 Introduction

The Pervasive Computing research area focuses on the development of abstractions and concepts for seamless integration of information processing into everyday activities and objects. In such environments, resources are normally scattered among the devices and any single device is not capable of executing an entire application. Thus, distributed applications need to be configured prior to their execution to ensure all required functionality is available. Configuring an application means finding a set of components which can be instantiated at the same time. Thus, application configuration is also known as *composition* of the required resources and services. Furthermore, this composition has to fulfill the structural constraints given by the required application functionalities,

Published in P. Floréen et al. (Eds.): Pervasive 2010, LNCS 6030, pp. 373-390, 2010.

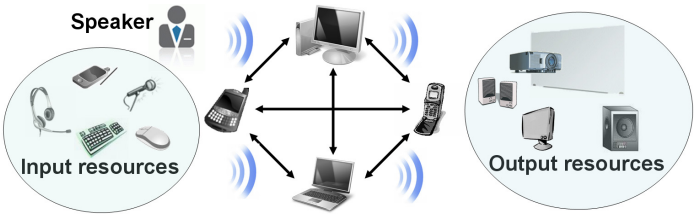

Fig. 1. Distributed presentation application

while considering the limited resources in the pervasive environment. Moreover, automation is needed to make this complex process transparent for the user. As an example, consider a conference environment depicted in Figure 1 where a speaker wants to give a presentation. For this purpose, the available input resources (e.g., keyboards, microphones, touch screens) and output resources (e.g., video projectors, loudspeakers) have to be leveraged by the distributed presentation application, as demonstrated in [14]. Further typical applications provide the easy sharing of files and resources like printers or webcams with other users (Casca, [9]) or the flexible and generic control of devices and services in home media networks (OSCAR, [20]). The actual composition of the application (called *configuration*) has to be calculated by configuration algorithms on the available devices. Furthermore, automatic runtime *adaptation* (or *re-configuration*) is necessary due to the dynamism in pervasive environments. Adaptation denotes the task of finding alternative components for those parts of the application that have become invalid, e.g. due to device failures. As distractions are highly undesirable during application execution, our main goal is to perform (re-)configuration processes as fast as possible.

Two fundamentally diverse approaches for configuration and adaptation of distributed applications exist, namely *decentralized* and *centralized* configuration. Decentralized approaches focus on mobile ad hoc networks [7],[10],[13] and calculate configurations in a cooperative fashion on all devices, as relying on central instances is not feasible there. While this approach increases the robustness of the configuration process, it implies extensive communication between the devices. Moreover, it disseminates the configuration tasks equally among all devices, not exploiting resource-rich devices in heterogeneous environments.

In such scenarios, centralized configuration on the fastest device can speed up the configuration process, as it exploits the additional computation power and avoids network communication. As a typical example, [29] presents an efficient centralized approach for weakly heterogeneous Smart Environments, featuring exactly one resource-rich device and several resource-weak devices. To distinguish between the different devices and classify them, [29] uses a combined-metrics clustering strategy to establish a cluster structure consisting of one *cluster head* – the single resource-rich device – which is responsible for centralized configuration calculations, while all other devices are the *cluster members* that remain inactive during configurations. As the cluster head needs to

acquire knowledge of the currently available resources on its cluster members, the *Virtual Container* concept is introduced: A Virtual Container (VC) is a local representation of a remote device on the cluster head and contains the resource information that is relevant for configuration. This enables local access to the remote configuration logic for the cluster head, allowing a strict decoupling of the (re-)configuration processes from the real devices. As the available resources of devices may change over time, each device automatically notifies the cluster head about changes in its resource condition, which then updates the corresponding VC to keep the resource information consistent. After successful configuration, the component bindings that are based on the application structure are established between parent and child components. For runtime adaptation of a configuration, it is sufficient to recalculate only those parts of the configuration that require changes, and re-establish the respective bindings. This scheme represents an efficient solution for weakly heterogeneous Smart Environments. However, such a centralized approach introduces a single point of failure and prevents the parallel calculation of configurations. Furthermore, the other devices' resource information has to be transferred to the configuration device in advance to enable efficient configuration on the resource-rich device. Centralized and decentralized approaches are compared in more detail in [19].

Many typical real-world pervasive scenarios are highly heterogeneous: They feature *several* resource-rich infrastructure devices like servers or desktop PCs as well as small mobile devices such as smart phones or PDAs, like in the auditorium scenario presented in Figure 1. For such environments, we propose a *hybrid configuration approach* in this paper. This approach represents a generalization of the existing centralized and decentralized approaches. It relies on a clustering scheme and enables the application configuration to be computed by multiple resource-rich devices simultaneously, which eliminates the single point of failure that is common in centralized approaches. The resource-weak devices stay passive during the hybrid configuration process. Thus, computational bottlenecks within the calculations are avoided, giving our approach an advantage over fully decentralized approaches. Moreover, our extended clustering mechanism allows the clusters to compute compositions independently from other clusters in the environment. Hence, this hybrid approach reduces the configuration latencies by more than 30% in heterogeneous environments, as our evaluations show.

This paper is structured as follows: After discussing related projects in the next section, we present our system model in Section 3. Afterwards, we introduce our efficient hybrid configuration approach, which is the main contribution of this paper. Then, we present our evaluation results in Section 5 to show the viability of our approach. Section 6 concludes and gives an outlook on future work.

2 Related Work

2.1 Application Configuration and Service Composition

Many current projects deal with component systems for Pervasive Computing. Speakeasy [9] and OSCAR [20] represent examplary systems allowing users to

create compositions of devices, media and services based on their current context. Pering et al. [24] present a composition framework to enable user-centric collections that combine mobile components together for carrying out a user task. However, these projects rely on user interaction during configuration processes and do not provide algorithms for the automatic composition of application configurations, which is the main focus here.

Projects like Gaia [26], Aura [31], iRoom [17], or Matilda's Smart House [18] provide a middleware for automatic configuration in Smart Environments. They support developers by providing services for the development of context-aware mobile distributed applications. These systems represent highly integrated environments and support various stationary and mobile devices. However, they are not suited for the use in pure ad hoc environments, as they rely on an existing infrastructure. For environments with a higher degree of dynamics, a more recent version of Gaia called Olympus [25] was presented that uses semantic descriptions to automate the mapping process.

In contrast, projects such as Mobile Gaia [7], RUNES [8] or P2PComp [10] target at pure ad hoc networks. While these projects provide automatic configuration, other peer-to-peer based approaches assign this task to the application programmer (e.g., one.world [12]). For highly dynamic environments, Paluska et al. [23] present an indirect specification via goals to refrain from specifying a single configuration. They provide an extensible mechanism to manage users' system runtime decisions and scan the vicinity for techniques that satisfy the user's goals. All of these projects do not rely on a supporting infrastructure, but they also do not exploit the increased computation power of resource-rich devices, yielding suboptimal efficiency in Smart Environments.

MobiGo [30] and PCOM [4] represent systems that support efficient automatic configuration in various environments. While PCOM provides decentralized [13] and centralized [29] configuration algorithms for complex component-based applications, MobiGo focuses on service level virtualization and migration.

Standard component systems like CORBA [21] or Enterprise Java Beans [32] offer persistency and transactional behavior. However, they rather focus on enterprise software than on resource-constrained dynamic pervasive environments. Infrastructures such as Jini [2] or UPnP [16] deal with service discovery in spontaneous networks. Though, they do not provide system support for automatic application configuration and adaptation, which is required here.

Hybrid configuration approaches have already shown to be efficient in other research areas like the distribution of scientific dataflows [3], Web Service composition [5] or large-scale Grid computing systems [11]. With the hybrid scheme presented in this paper, the complete spectrum of possible pervasive environments is covered, giving this system a distinct advance over the related projects that focus on application configuration in specific pervasive environments only.

2.2 Load-balancing Clustering Schemes

Many related approaches aim at balancing the load among nodes. MANET-based schemes [34] like DEEC [27] or DLBC [1] balance the load in infrastructure-less

scenarios to extend the overall network lifetime. Thus, these schemes equally distribute the load among *all* nodes. In addition, schemes like AMC [22] focus on highly dynamic mobile devices and multi-hop connections. Thus, the merging and split-up of clusters are common actions, yielding low cluster stability. In contrast, we only want to balance the load between the subset of resource-rich infrastructure devices to minimize the (re-)configuration latencies, with as few re-clustering processes as possible. As this infrastructure is typically continuously available, the respective subset of resource-rich devices is rather static. In the area of web clusters, scheduling algorithms try to balance the load distribution on the servers to increase the loading capacity of the cluster [6]. However, these schemes do not consider aspects like mobility or node failures. Hence, they do not provide the re-clustering strategies needed here and are not suited to solve our problem of balancing the configuration load between the resource-rich devices.

3 System Model

3.1 Application Model

For this work, we presume a component-based software model, i.e. an application consists of several *components* which are resident on specific devices and require a certain amount of resources. An application is represented by a tree of interdependent components that is constructed by recursively starting the components required by the root instance. Interdependencies 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. A parent component may have an arbitrary number of dependencies. Further details can be found in [4].

3.2 Underlying System

We especially focus on heterogeneous pervasive environments in this paper, consisting of resource-rich devices like PCs or laptops as well as resource-poor mobile devices like smart phones or PDAs. The number of components per device is not restricted. Devices have a unique system identifier (SID) and may become unavailable at any time, e.g., due to mobility or device failures, causing the unavailability of their components. All devices use standard wireless communication technology, e.g., Bluetooth or WiFi, and have a direct, bidirectional communication link to each other, which is usually the case in typical pervasive scenarios like offices or home entertainment. Furthermore, the underlying middleware is supposed to maintain a registry containing all devices in the vicinity with information about their services and properties.

3.3 Problem Statement

We focus on automatic configuration and adaptation of distributed applications in heterogeneous pervasive scenarios. In a *configuration* process, a specific configuration algorithm tries to resolve all application dependencies by finding a

suitable composition of components. Such a composition is subject to two classes of constraints: *Structural constraints* describe what constitutes a valid composition in terms of functionalities. *Resource constraints* are a result of the limited resources. For example, in the presentation application introduced in Figure 1, a structural constraint is that the video projector can only be used if the computer to which the projector is connected to is also available. A resource constraint could be that there needs to be at least one loudspeaker as acoustic output device. An application is successfully configured if all dependencies were resolved and the bindings between the components were established. The *configuration latency* comprises the time between the start and the availability of the application to the user. This latency includes the delays caused by calculating a valid configuration and instantiating all application components. Our goal is to minimize the configuration latency in order to provide a seamless user experience.

Re-configuration processes, or *adaptations*, become necessary if devices whose components are part of the current application configuration become unavailable. Then, alternative components have to be found that can provide the same functionality. Generally, an adaptation represents a special case of a configuration where only those parts of the application need to be recalculated that are no longer valid. So, the same algorithms are used for configuration and adaptation.

4 Hybrid Configuration Management

4.1 Approach and Challenges

Both the totally decentralized and totally centralized approaches have advantages, but also drawbacks that prevent an efficient configuration in *all* possible pervasive environments. Our hybrid approach combines the best properties of these two approaches to minimize the configuration latency. For this purpose, only the resource-rich devices actively calculate application configurations. We call these devices *Active Devices (ADs)* in the following. Contrary to this, the resource-weak devices only provide information about their available resources and services, prior to configuration processes. They stay passive during the configuration, so we call them *Passive Devices (PDs)*.

In a hybrid configuration process, initially, the AD and the PD roles need to be assigned to the devices in the environment since the configuration of each PD has to be calculated by one AD. We call this assignment of a PD to an AD a *mapping*. Subsequently, the ADs need to obtain the configuration-specific information from their mapped PDs. Finally, a hybrid configuration algorithm is necessary which calculates valid configurations on the ADs and distributes the configuration results to the PDs. Details are presented below.

4.2 Cluster Formation and Maintenance

Initially, a suitable subset of devices for calculating configurations has to be discovered to exploit the device heterogeneity efficiently. To reduce the risk

of possible bottlenecks, the component algorithms' configuration load should be balanced between the ADs and maintained even in case of changing device availabilities. Furthermore, each AD should not be required to know about the mappings at the other ADs.

Resource-Aware Cluster Formation The following scheme establishes multiple stable clusters in heterogeneous environments with several resource-rich devices. These devices automatically become the cluster heads (and, hence, the ADs) if a resource-aware clustering strategy like in [29] is used. Our new scheme balances the configuration algorithm's load among these ADs such that a) they are not overloaded and b) the configuration is parallelized to reduce the latencies.

We assume there are m ADs A_i with cluster indices (CIDs) $i \in \{0, \ldots, m-1\}$ and n PDs P_j with indices $j \in \{0, \ldots, n-1\}$. Initially, each AD assigns itself a CID i according to its SID, i.e. the AD with lowest SID (of all ADs) assigns itself CID $i = 0$, and the AD with highest SID gets CID $i = m - 1$. The same holds for the PDs that assign themselves CIDs j according to their SID.

There is an overhead for each AD consisting of the efforts needed to retrieve its mapped PDs' resource information, calculate its mapped PDs' components' configuration and send the configuration results back to them. This overhead highly depends on the number of PDs within its cluster. Thus, if the mapping of PDs to ADs is balanced, each AD takes the responsibility for about the same amount of configuration work. This establishes the load balance among the ADs that is important to reduce the configuration latency. To achieve this, each AD has to map at least $\lfloor \frac{n}{m} \rfloor$ PDs to itself. If $n \; modulo \; m = z > 0$, the ADs $0, \ldots, z-1$ need to map one additional PD to ensure all PDs are mapped to an AD. This leads to the so-called *Balancing Condition* that has to be fulfilled at each AD:

$$mapped(A_i) = \begin{cases} \lfloor \frac{n}{m} \rfloor + 1, & i < n \; modulo \; m \\ \lfloor \frac{n}{m} \rfloor, & i \geq n \; modulo \; m \end{cases}, \tag{1}$$

where $mapped(A_i)$ is the number of PDs that need to be mapped to AD A_i. The fulfillment of this condition is verified on each AD, initially on startup of the device and whenever the number of ADs or PDs changes. For the actual mapping, a simple round robin scheme is used where each AD maps every m-th PD, starting with A_0 that maps P_0, P_m, P_{2m}, and so on.

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. They just need to know their own CID i and the number of ADs and PDs, which can be looked up in the device registry.

For clarification, let us consider an exemplary scenario consisting of three ADs A_0 to A_2 and eight unmapped PDs P_0 to P_7. Using the described cluster formation scheme, A_0 maps P_0, P_3, and P_6. Furthermore, A_1 maps P_1, P_4, and P_7, and A_2 maps P_2 and P_5. The arising cluster structure is shown in Figure 2a.

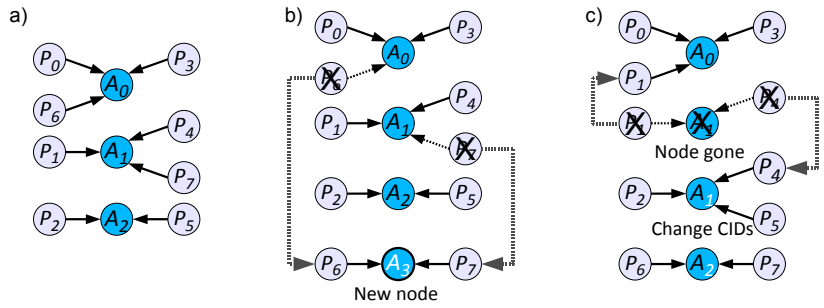

Fig. 2. a) Initial mapping, b) Remapping: A_3 appeared, c) Remapping: A_1 disappeared

Cluster Maintenance Re-clustering is needed to maintain a balanced load in dynamic environments. 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. If a new device appears, it assigns itself the lowest free CID within its class, e.g., if it is an AD and there are m other ADs with indices $0, \ldots, m-1$ present, it assigns itself index $i = m$. Each device decrements its CID if another device from the same class (i.e., AD or PD) with a lower CID disappears.

If a new PD appears, the AD with cluster index $i = (n \textit{ modulo } m)$ maps this device. Thus, the round robin distribution of the PDs to the ADs is continued. After this mapping, each device increments the number n of PDs.

If a new AD D appears, D needs to map $\lfloor \frac{n}{m} \rfloor$ devices to itself[1] so that all ADs still have a similar fraction of PDs. The re-mappings are executed in the following way: Initially, D needs to remap a PD from the AD that has the maximum number of mapped PDs (and the highest index i, in case of multiple options), which yields the AD with index $i = [(n\text{-}1) \textit{ modulo } (m\text{-}1)]$ due to the round robin scheme. D sends a remapping request to the corresponding AD which then notifies its mapped PD with highest CID that this PD has to be remapped to D. This remapping process is repeated $\lfloor \frac{n}{m} \rfloor$ times, whereas the ADs whose PDs are re-mapped by D are chosen by a round robin scheme, as shown in line 5 of Listing 1.1. Hence, the Balancing Condition is still fulfilled on all ADs after these re-mappings. Nodes that appear during an ongoing configuration process – ADs as well as PDs – are not considered within this configuration yet, but starting with the next one. Consider the example from Figure 2b where AD A_3 appeared. At first, A_3 calculates it needs to remap $\lfloor \frac{8}{4} \rfloor = 2$ PDs. According to line 5 from Listing 1.1, A_3 finds out it needs to remap one PD from device $(8\text{-}1\text{-}0) \textit{ modulo } 3 = 1$, and one PD from device $(8\text{-}1\text{-}1) \textit{ modulo } 3 = 0$. Then, A_3 sends remapping requests to these ADs. Thus, A_1 notifies P_7 (its mapped PD with highest index) to remap to A_3, and A_0 notifies P_6 to remap to A_3. Now, the Balancing Condition is fulfilled again. If multiple devices appear at almost the same time, the problem of race conditions during the mapping process may

[1] Here, D is already included in the number m of ADs

arise and potentially lead to inconsistent mappings. To analyze the seriousness of this problem, we did multiple real-world tests where we started two devices timely close to each other and regarded the arising mappings. We found out that inconsistent mappings started to emerge when the time span between two subsequent appearances of new devices fell below 30 ms. Thus, every new device waits for 50 ms for potential other new devices before it starts its mapping process. Proceeding like this, inconsistencies did not appear anymore.

```
1  request_remappings(){
2      mapped := 0;
3      remappings := floor(n/m);
4      while (mapped < remappings) {
5          remapId = (n−1−mapped) modulo (m−1);
6          send_remap_request_to_AD(remapId);
7          await_resource_info();
8          mapped++;
9      }
10 }
```

Listing 1.1. Reclustering process executed by a newly appearing AD

If a PD P_j disappears, all ADs need to decrement the number n of PDs, and P_j's mapping needs to be removed at the AD A_j to which it was mapped. Additionally, A_j verifies if the Balancing Condition is still fulfilled. If this is not the case, A_j sends a remapping request to the AD with index $k = n \ modulo \ m$. Then, A_k notifies its mapped PD with highest CID that this PD needs to be remapped to A_j. The chosen PD finishes this remapping by sending its resource information to A_j. Additionally, if P_j disappears during an ongoing configuration process, A_j recognizes those parts of the application which were provided by P_j's components and selects alternative components for them, if available.

Finally, the case of a disappearing AD A_x remains. If A_x was the last available AD, then each PD notices that the cluster structure is dissolved, and the decentralized configuration approach is chosen in future configuration processes. Otherwise, remapping processes are necessary: Each PD that recognizes that its cluster head A_x is gone broadcasts a so-called *Unmapped Message* to notify the other nodes that it is currently unmapped and needs to be remapped to another AD. If an AD A_y notices the disappearance of A_x, it at first checks if A_x had a lower CID than itself. In this case, it decrements its CID. Then, A_y needs to calculate the number of required remappings ($remap(A_x)$) for itself: In order to fullfil the Balancing Condition, A_y needs to remap at least $\lfloor \frac{n}{m} \rfloor$ devices, *minus* the number of its currently mapped devices ($mapped(A_y)$). As before, if A_y recognizes that there are some remaining unmapped devices, i.e., $n \ modulo \ m = z > 0$, the ADs with indices $0, \ldots, z-1$ need to remap one additional device. Subsequently, each AD broadcasts how many remappings it will perform. Then, each AD waits for a certain time T_1 to gather all remapping and unmapped messages. The value of T_1 has to be large enough to cover the whole gathering process. Otherwise, T_1 expires without all messages having

been received, causing inconsistencies and potentially thrashing effects in the remapping processes. However, as too large values of T_1 unnecessarily increase the time for (re-)clusterings, T_1 must also not be chosen too high. A reasonable compromise is to determine the average time a gathering process takes in typical scenarios, and add some additional time to be on the safe side. Further information about the value we chose for T_1 is given in Section 5.1. After this waiting time, each AD knows which PDs are unmapped, and how many PDs the other ADs will remap. Finally, the remappings are performed according to the indices of the involved ADs and PDs: AD A_0 with lowest CID 0 maps the $remap(A_0)$ unmapped PDs with lowest CIDs, i.e. the unmapped PDs with CIDs $0, \ldots, remap(A_0) - 1$. AD A_1 with the second lowest CID maps the $remap(A_1)$ PDs with next higher indices, i.e. $remap(A_0), \ldots, remap(A_0) + remap(A_1) - 1$, and so on up to the AD with highest CID which maps the unmapped PDs with highest CIDs. In the special case of a disappearing AD A_x during an ongoing configuration process, those parts of the application which were calculated by A_x are no longer available, making a remapping of the PDs that were mapped to A_x and a subsequent restart of the configuration process inevitable. This increases the arising latencies. However, a disappearing infrastructure-device exactly at a configuration process is quite unlikely and should happen rather seldom.

As an example, consider Figure 2c where AD A_1 disappeared, leaving P_1 and P_4 unmapped. Now, A_2 and A_3 decrement their CIDs and become A_1 and A_2, as an AD with lower CID disappeared. According to the previously described scheme, A_0 and A_1 need to remap one additional device because of their low indices, i.e. $\lfloor \frac{8}{3} \rfloor$ - 2 + 1 = 1 PD, while A_2 needs to remap $\lfloor \frac{8}{3} \rfloor$ - 2 = 0 PDs. As A_0 has a lower CID than A_1, A_0 remaps the unmapped PD with lower CID (i.e., P_1), and A_1 remaps P_4 as the unmapped PD with higher CID. Again, the Balancing Condition is fulfilled after these remapping processes.

4.3 Hybrid Application Configuration and Result Distribution

The hybrid configuration is calculated in a parallel and cooperative fashion on the subset of ADs. The configuration of each PD's components is performed locally on the AD it was mapped to. Therefore, the created VCs are used (cf. Section 1). This reduces the communication overhead during the configuration compared to decentralized configuration. Moreover, the PDs are not involved in these calculations. This avoids that the resource-constrained PDs become computational bottlenecks, and it conserves their (usually limited) energy resources.

An adapted version [13] of Asynchronous Backtracking [33] is used for the cooperative configuration on the ADs. This decentralized configuration algorithm enables the concurrent configuration of components and utilizes the available parallelism. It performs a depth-first search in the tree of dependencies. In case a dependency cannot be fulfilled, dependency-directed backtracking is used. Furthermore, for the local configuration of the PDs' components on the ADs, we use an efficient centralized algorithm called Direct Backtracking [28]. This algorithm features a proactive mechanism to avoid backtracking in many situations, and an intelligent backtracking mechanism to handle conflict situations more

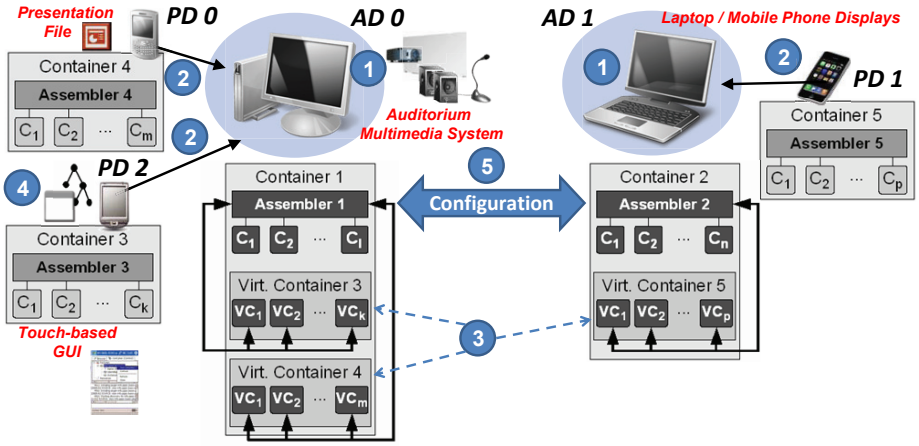

Fig. 3. Hybrid application configuration example

efficiently. After a successful configuration, the ADs distribute the configuration results among their PDs to notify them about which of their components were chosen. The respective messages are rather small, as they only contain the relevant information about the chosen components on the recipient PDs. The average message overhead per application component is only 9 kB. Finally, the component bindings are established, yielding the application execution.

4.4 Exemplary Configuration Process

Resuming the scenario from the introduction, Figure 3 shows an examplary environment where a distributed presentation application needs to be executed. When a speaker wants to give a presentation, the configuration algorithm needs to automatically find suitable components for the distributed application on these devices. For instance, if a speaker wants to switch between the slides using the touchscreen of his mobile phone (PD 2), a touch-based graphical user interface needs to be provided on this device. Moreover, all presentation files may potentially be resident on a remote device, like the conference organizer's smart phone (PD 0). The speaker also needs supporting input and output devices such as the auditorium's multimedia system covering video projector, loudspeakers and a microphone, which are connected to the stationary PC (AD 0). As some auditors may potentially be sitting far from the presentation screen, it might be more convenient for them to have the slides displayed on their own mobile devices, e.g. their laptop (AD 1) or even their mobile phone (PD 1).

Initially, the cluster structure is established using the presented 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. On the basis of this information, the ADs build the local representations of the mapped PDs within

Virtual Containers in step 3. This leads to two VCs at AD 0, and one VC at AD 1. Then, a user wants to start an application on his/her mobile device, PD 2. Thus, the information about the application start is transmitted to AD 0 as the responsible cluster head for PD 2 (step 4). Subsequently, AD 0 initiates the configuration of the application, which is shown in step 5. At first, it verifies which of the dependencies can be resolved by components of its local container and the Virtual Containers, representing its mapped PDs. For the remaining unresolved functionalities, AD 0 requests AD 1 to resolve these dependencies. AD 1 provides AD 0 with the corresponding information about the fitting components. Subsequently, the complete configuration is constructed by AD 0. After successful configuration, the PDs whose components are used in the configuration are informed by their cluster head about their component configurations. Finally, the required components are initialized, the bindings between the components – as negotiated within step 5 – are established, and the application is executed.

5 Evaluation

5.1 Experimental Setup

For our real-world evaluations, we used six laptops[2] and six smart phones[3]. The laptops became cluster heads (ADs) because of their high computation power, while the smart phones became cluster members (PDs) and were equally distributed among the cluster heads. In all scenarios, we used the 802.11b Ad Hoc mode in combination with broadcast messages between the devices. The configuration process was initiated by invoking the application anchor on one of the smart phones. Apart from the real-world experiments, we also performed extensive evaluations on the Network Emulation Testbed (NET, [15]) to evaluate the scalability of our approach in larger scenarios with up to 85 devices. In these evaluations, we emulated the same wireless network as in the real-world evaluations. To find a suitable value for the parameter T_1 for gathering the *unmapped* and *remapping* messages (cf. Section 4.2), we performed 50 measurements to identify the time it takes to gather this information from the other devices. The average time to receive all of these messages was 0.57 s. Furthermore, the gathering process never took longer than 0.83 s, even in large scenarios. As a precaution, we initialized T_1 with a slightly increased value of 1 s for the evaluations. Consequently, we did not face any thrashing effects or race conditions in the remapping processes during any of the taken evaluations. In the shown graphs, each measurement represents the average of 50 evaluation runs. Standard deviations were below 15 % in all cases and below 10% in 90% of all cases.

We used the PCOM [4] system for our evaluations. The evaluated application represents a binary tree of depth 6, i.e., it consists of $k = 127$ components. Additionally, we measured the configuration latencies in a smaller scenario with a binary tree of depth 4, i.e. $k = 31$, to verify our results in a smaller scale. In the

[2] ThinkPad T41p, Intel Centrino CPU, 1.6 GHz, 1 GB RAM
[3] T-Mobile MDA, PXA 270 CPU, 520 MHz, 128 MB RAM

evaluations, the laptops got an increased number of resources compared to the smart phones (factor 2 to 5, randomly chosen for each laptop) to consider that they are usually much more resource-rich. We evaluated the hybrid scheme in comparison to the totally decentralized and centralized approaches to show the advantage over these standard approaches. We measured the message overhead and the latencies that arose at the various stages of the configuration: initial cluster formation and re-clustering processes, the preconfiguration process, the actual configuration as well as an adaptation process where only 50 % of the components needed to be adapted, the distribution of the configuration results, and the binding of the components.

5.2 Communication Overhead Measurements

Figure 4 shows the message overhead at the various stages of the configuration. In these graphs, "Hybrid-x" represents the hybrid approach with x ADs (laptops), where $2 \leq x \leq 6$. The remaining devices (PDs) were the smart phones.

In the preconfiguration process (Figure 4a), an average overhead of 53 kB per device and configuration process arises for the centralized and hybrid schemes, since these schemes need to build the cluster structure and to transmit the configuration-specific information for the VCs. For hybrid configuration, this overhead arises only at every PD, as they need to transmit their resource information to their cluster head. This leads to a reduced overhead compared to the centralized scheme. The decentralized scheme does not use preconfiguration.

Figure 4b shows the message overhead for the actual configuration. In centralized configuration, the device where the application was started initially transmits the application information to the cluster head. The resulting overhead only depends on the application size, i.e. the involved components. As we used a fixed application with 127 components, the overhead was static with 183 kB in total per configuration process. The hybrid approach's message overhead mainly depends on the number of involved ADs, as only they calculate configurations. Thus, a rising number of available PDs does not have an impact on the message overhead. The message overhead for decentralized configuration increases with a rising number of involved devices, as all devices have to communicate with each other. However, this overhead converges for a larger number of involved devices, since the per-device-overhead decreases due to a lower number of components per device. The centralized approach's distribution overhead (Figure 4c) and the component binding overhead (Figure 4d) converge for the same reason.

As the devices piggyback the configuration results during the decentralized configuration process, no further messages are needed for result distribution, as it can be seen in Figure 4c. Compared to the centralized approach, the piggybacking increased the overhead during the actual configuration by 403 kB, but reduced the result distribution overhead by 1418 kB on average. In centralized configuration, the cluster head broadcasts the *complete* composition, yielding high communication overhead. In hybrid configuration, the cluster heads only need to notify their PDs about which of their components were chosen. Thus, the hybrid approach's overhead rises linearly with the number of PDs.

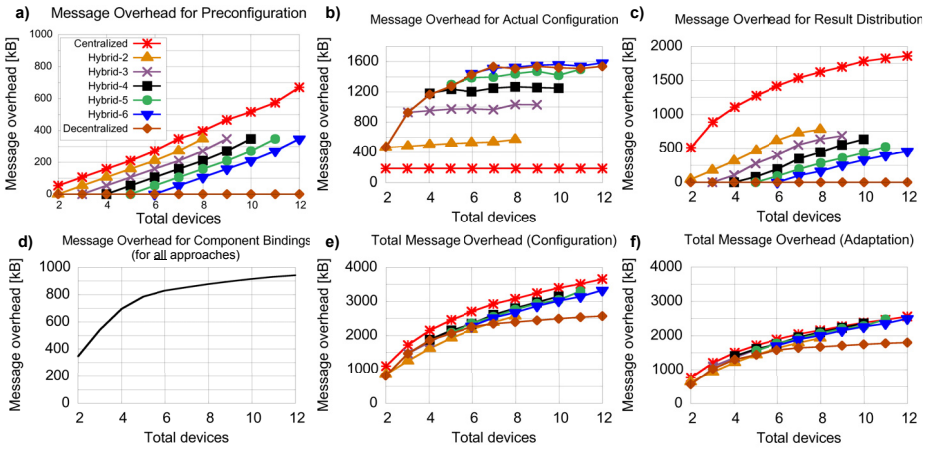

Fig. 4. Message overhead at the different stages of one configuration process ($k = 127$)

The overhead for establishing the component bindings (Figure 4d) is the same for all configuration schemes, as it is independent from the actual configuration. This overhead rises with a rising number of involved devices, since bindings between components on different devices are likely to emerge more often then.

Figure 4e shows the total message overhead for one configuration process as the sum of all overheads. The decentralized approach scales best due to the result piggybacking at the configuration process. Its total message overhead converges with a rising number of involved devices due to the almost constant overhead for actual configuration and no further distribution overhead (cf. Figures 4b and 4c). The centralized approach performs worst because of a high overhead for preconfiguration and result distribution. The hybrid approach produces an average overhead at all stages of configuration, yielding a moderate total overhead and showing its applicability concerning message overhead.

Regarding adaptation, the total message overhead is shown in Figure 4f. Compared to configuration, the overheads for the centralized and decentralized schemes were reduced by 30 %, as only parts of the application needed to be recalculated and distributed. The message overhead of the hybrid scheme decreased by 25 % only, as the remapping messages needed to be sent, too. Thus, the hybrid and centralized schemes produce about the same adaptation message overhead, while the decentralized schemes' overhead is around 22 % lower.

5.3 Configuration Latency Measurements

We compared the overall latencies of all three approaches in two heterogeneous scenarios ($k = 31$, $k = 127$) with differing device numbers and 50 % resource-rich devices in each scenario. Figure 5 shows the total latencies. The real-world evaluations were performed with 4 to 12 devices, and the emulations in the large-scale scenario with $k = 127$ with up to 85 devices, where each laptop holds two

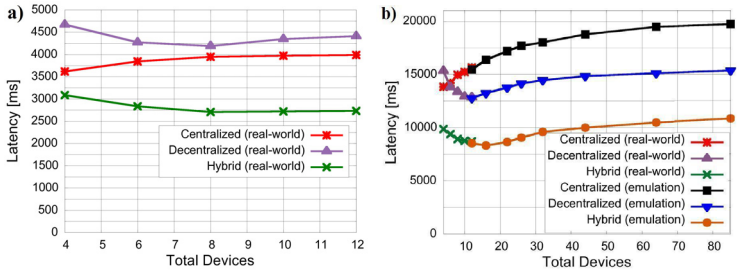

Fig. 5. Total configuration latencies: a) $k = 31$, b) $k = 127$

resources and each smart phone holds one resource. Increasing the number of devices above 85 would not lead to changing results, since some of the devices would not hold any resources then. Figure 5b shows that the latencies for the hybrid and the decentralized approach at first drop with a rising number of devices. This happens because of an increasing *absolute* number of resource-rich devices that are involved in configuration calculations, while in centralized configuration, *only one* resource-rich device is always used to calculate configurations. When the total number of devices exceeds 12 (distributed) or 16 (hybrid) devices, the overall latencies start to slightly increase again, as the latencies for establishing the component bindings grow stronger than the latencies for the configuration calculation drop. The latencies of centralized configuration show continuous growth, as the latencies for distribution and establishment of the bindings increase with a rising number of devices, while the configuration latency remains constant. It can be seen that the hybrid approach outperforms the decentralized approach by 35.7 % ($k = 31$) and by 34.5 % ($k = 127$) on average, and the centralized approach by 26.3 % ($k = 31$) and by 44.1 % ($k = 127$), respectively. The emulation results point up the hybrid approach's scalability, as latency reduction still holds with large applications and many involved devices.

For clarification, Figure 6 shows the latencies at the different configuration stages in a specific scenario with $k = 127$, four ADs and up to six PDs. The clustering of devices produces a negligible latency of below 30 ms per PD, as you can see in Figure 6a. Re-clustering processes due to dynamics take a constant time of 1.1 s more than the initial clustering, mainly because of the chosen value of 1 s for T_1 (cf. Section 5.1). The loading of the resource information increases linear with an overhead of 400 ms per device. The clustering and resource information loading latencies are *not* included in the overall latencies in Figures 6e and 6f, as they are performed once *prior* to the configuration. However, the re-clustering latency is included in the overall adaptation latency shown in Figure 6f.

Regarding the latency for the configuration process itself (Figure 6b), the centralized approach performs best, as the resource-richest device locally calculates the configuration. The decentralized approach is significantly slowed down due to the fact that the resource-limited devices are involved in the calculations. Another factor is the immense communication overhead of the decentralized ap-

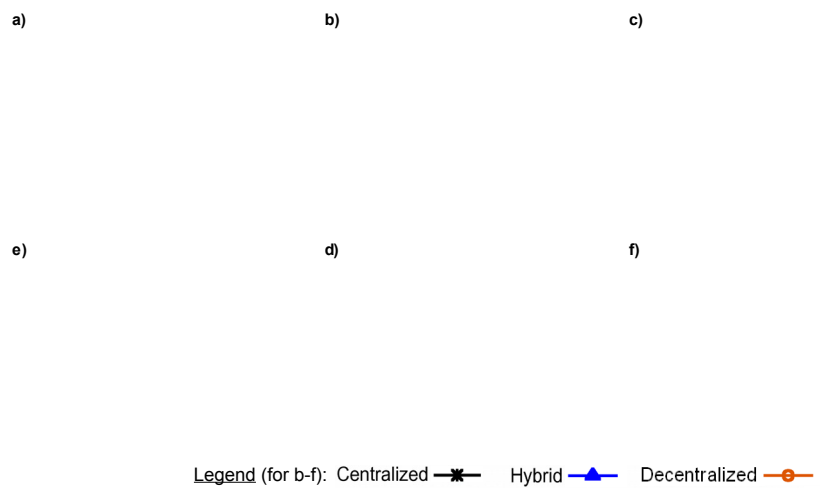

Fig. 6. Latencies at the different stages of the configuration process ($k = 127$)

proach (cf. Figure 4b). In the hybrid approach, only the resource-rich devices perform the calculation, but message exchanges between them still take time. Thus, its latencies are slightly above the centralized scheme's latencies.

Figure 6c shows the latency to distribute the configuration results. The centralized scheme has the highest latency, as the single configuration device needs to distribute the complete configuration (cf. Figure 4c). In contrast, the other approaches have already piggybacked information about configured components in the configuration messages, in case of decentralized configuration even between *all* devices. Thus, these approaches have much lower distribution latencies.

The initialization of the component bindings (Figure 6d) comprises the sum of the import of the received configuration results and the establishment of the respective component links. Since message overhead and delay for the result distribution are much higher for the centralized approach, as seen in Figures 4c and 6c, the configuration import is responsible for a big fraction of the latency, especially on the resource-weak devices. The establishment of the links is performed in the same way by all approaches and, hence, takes the same amount of time.

Figure 6e shows the total latencies as sum of the latencies from Figures 6b-d. The centralized approach is slowest due to its increased result distribution and component binding overhead. The decentralized scheme performs 14 % better on average, although the resource-weak devices are involved. The hybrid approach avoids the drawbacks of the other schemes and performs fine in all configuration stages. Thus, it outperforms the decentralized scheme by 34.2 % and the centralized scheme even by 40.7 % on average. Regarding the total latencies for an adaptation process (Figure 6f), the advantage of the hybrid approach decreases to 20.4 % compared to decentralized and to 30.2 % compared to the centralized scheme, due to the additional re-clustering overhead (cf. Figure 6a).

6 Conclusions and Outlook

We presented a hybrid approach for configuring distributed pervasive applications. This approach efficiently exploits the available computation resources in heterogeneous environments. Since this hybrid scheme is a generalization of the pure centralized and decentralized approaches, it covers the complete spectrum of pervasive scenarios, which has not been achieved by related projects yet.

Our approach is based on the formation of clusters with balanced configuration load for the resource-rich devices. These devices represent the active devices during configuration calculation processes, while the resource-weak devices remain passive to avoid bottlenecks in the configuration process. Single points of failure are avoided due to the parallel execution of the configuration calculations on the active devices. The hybrid approach automatically adjusts its degree of decentralization to the available resources in the network. In our evaluations, we proved that our approach reduces the configuration latencies by more than 30 % on average compared to decentralized and centralized approaches. Moreover, the evaluations on a network emulation cluster showed that these results also hold in larger scenarios. The reduced configuration time strongly helps to increase users' acceptance for pervasive systems and represents a large step towards seamless automatic application configuration.

Our next step is to reduce the hybrid approach's communication latencies by exploiting the application structure and local component dependencies at the clustering processes. Moreover, we want to use idle periods at the ADs to precalculate partial configurations and store them for future configuration processes.

References

1. Amis, A. D., Prakash, R.: *Load-Balancing Clusters in Wireless Ad Hoc Networks.* In Proc. IEEE Asset 2000, 2000.
2. Arnold, K., O'Sullivan, B., Scheifler, R., Waldo, J., Wollrath, A.: *The Jini Specification.* Addison Wesley, 1999.
3. Barker, A., Weissman, J. B., van Hemert, J.: *Eliminating The Middleman: Peer-to-Peer Dataflow.* In Proc. ACM HPDC '08, 2008.
4. Becker, C., Handte, M., Schiele, G., Rothermel, K.: *PCOM - A Component System for Pervasive Computing.* In Proc. IEEE PerCom '04, 2004.
5. Benatallah, B., Sheng, Q. Z., Dumas, M.: *The Self-Serv Environment for Web Services Composition.* In IEEE Internet Computing, 7(1), 2003.
6. Cardellini, V., Colajanni, M., Yu, P. S.: *Dynamic Load Balancing on Web-server Systems.* In IEEE Internet Computing, 3(3), 1999.
7. Chetan, S., Al-Muhtadi, J., Campbell, R., Mickunas, M. D.: *Mobile Gaia: A Middleware for Ad-hoc Pervasive Computing.* In Proc. IEEE CCNC '05, 2005.
8. Costa, P. et al.: *The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario.* In Proc. IEEE PerCom'07, 2007.
9. Edwards, W. K. et al.: *Using Speakeasy for Ad Hoc Peer-to-Peer Collaboration.* In Proc. ACM CSCW '02, 2002.
10. Ferscha, A., Hechinger, M.,Mayrhofer, R., Oberhauser, R.: *A Light-Weight Component Model for Peer-to-Peer Applications.* In Proc. ICDCS '04 Workshops, 2004.

11. Graupner, S., Andrzejak, A., Kotov, V. E., Trinks, H.: *Adaptive Service Placement Algorithms for Autonomous Service Networks.* In Proc. ESOA '04, 2004.
12. Grimm, R.: *One.world: Experiences with a Pervasive Computing Architecture.* In IEEE Pervasive Computing, 3(3), 2004.
13. Handte, M., Becker, C., Rothermel, K.: *Peer-based Automatic Configuration of Pervasive Applications.* In Proc. IEEE ICPS '05, 2005.
14. Handte, M., Urbanski, S., Becker, C., Reinhard, P., Engel, M., Smith, M.: *3PC/-MarNET Pervasive Presenter.* In Proc. IEEE PerCom '06, 2006.
15. Herrscher, D., Rothermel, K.: *A Dynamic Network Scenario Emulation Tool.* In Proc. ICCCN '02, 2002.
16. Jeronimo, M., Weast, J.: *UPnP* Design by Example*, Intel Press, 2003.
17. Johanson, B., Fox, A., Winograd, T.: *The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms.* In IEEE Pervas. Computing, 1(2), 2002.
18. Lee, C., Nordstedt, D., Helal, S.: *Enabling Smart Spaces with OSGi.* In IEEE Pervasive Computing, 2(3), 2003.
19. Liu, D., Law, K. H., Wiederhold, G.: *Analysis of Integration Models for Service Composition.* In Proc. ACM WOSP '02, 2002.
20. Newman, M. W., Elliott, A., Smith, T. F.: *Providing an Integrated User Experience of Networked Media, Devices, and Services Through End-User Composition.* In Proc. Pervasive 2008, 2008.
21. Object Management Group (OMG): *CORBA Component Model V3.0*, 2002.
22. Ohta, T., Inoue, S., Kakuda, Y.: *An Adaptive Multihop Clustering Scheme for Highly Mobile Ad Hoc Networks.* In Proc. IEEE ISADS '03, 2003.
23. Paluska, J. M., Pham, H., Saif, U., Chau, G., Terman, C., Ward, S.: *Structured Decomposition of Adaptive Applications.* In Proc. IEEE PerCom '08, 2008.
24. Pering, T., Want, R., Rosario, B., Sud, S., Lyons, K.: *Enabling Pervasive Collaboration with Platform Composition.* In Proc. Pervasive 2009, 2009.
25. Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R. H., Mickunas, M. D.: *Olympus: A High-Level Programming Model for Pervasive Computing Environments.* In Proc. IEEE PerCom '05, 2005.
26. Román, M., Hess, C. K., Cerqueira, R., Ranganathan, A., Campbell, R. H., Nahrstedt, K.: *Gaia: A Middleware Infrastructure to Enable Active Spaces.* In IEEE Pervasive Computing, 1(4), 2002.
27. Safa, H., Mirza, O., Artail, H.: *A Dynamic Energy Efficient Clustering Algorithm for MANETs.* In Proc. IEEE WIMOB '08, 2008.
28. Schuhmann, S., Herrmann, K., Rothermel, K.: *Direct Backtracking: An Advanced Adaptation Algorithm for Pervasive Applications.* In Proc. ARCS, 2008.
29. Schuhmann, S., Herrmann, K., Rothermel, K.: *A Framework for Adapting the Distribution of Automatic Application Configuration.* In Proc. ACM ICPS '08, 2008.
30. Song, X., Ramachandran, U.: *MobiGo: A Middleware for Seamless Mobility.* In Proc. IEEE RTCSA 2007, 2007.
31. Sousa, J. P., Garlan, D.: *Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments.* In Proc. IEEE/IFIP WICSA, 2002.
32. SUN Microsystems: *Enterprise Java Beans Specification, Java Specification Request (JSR) 220 Final Release*, http://java.sun.com/products/ejb/docs.html, 2003.
33. Yokoo, M., Durfee, E. H., Ishida, T., Kuwabara, K.: *The Distributed Constraint Satisfaction Problem: Formalization and Algorithms.* In IEEE Transactions on Knowledge and Data Engineering, 10(5), 1998.
34. Yu, J. Y., Chong, P. H. J.: *A Survey of Clustering Schemes for Mobile Ad Hoc Networks.* In IEEE Communications Surveys and Tutorials, 7(1), 2005.