Automated Composition of Adaptive Pervasive Applications in Heterogeneous Environments

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Stephan Andreas Schuhmann

aus Heilbronn am Neckar

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr. phil. nat. Christian Becker
Mitprüfer:	Prof. Dr. rer. nat. Frank Leymann
Tag der mündlichen Prüfung:	29. Oktober 2012

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2012

Contents

List of Figures

List of Tables

Abstract

Distributed applications for Pervasive Computing represent a research area of high interest. Configuration processes are needed before the application execution to find a composition of components that provides the required functionality. As dynamic pervasive environments and device failures may yield unavailability of arbitrary components and devices at any time, finding and maintaining such a composition represents a nontrivial task. Obviously, many degrees of decentralization and even completely centralized approaches are possible in the calculation of valid configurations, spanning a wide spectrum of possible solutions. As configuration processes produce latencies which are noticed by the application user as undesired waiting times, configurations have to be calculated as fast as possible.

While completely distributed configuration is inevitable in infrastructure-less Ad Hoc scenarios, many realistic Pervasive Computing environments are located in heterogeneous environments, where additional computation power of resource-rich devices can be utilized by centralized approaches. However, in case of strongly heterogeneous pervasive environments including several resource-rich and resource-weak devices, both centralized and decentralized approaches may lead to suboptimal results concerning configuration latencies: While the resource-weak devices may be bottlenecks for decentralized configuration, the centralized approach faces the problem of not utilizing parallelism. Most of the conducted projects in Pervasive Computing only focus on one specific type of environment: Either they concentrate on heterogeneous environments, which rely on additional infrastructure devices, leading to inapplicability in infrastructure-less environments. Or they address homogeneous Ad Hoc environments and treat all involved devices as equal, which leads to suboptimal results in case of present resource-rich devices, as their additional computation power is not exploited.

Therefore, in this work we propose an advanced comprehensive adaptive approach that particularly focuses on the efficient support of heterogeneous environments, but is also applicable in infrastructure-less homogeneous scenarios. We provide multiple configuration schemes with different degrees of decentralization for distributed applications, optimized for specific scenarios. Our solution is adaptive in a way that the actual scheme is chosen ba sed on the current system environment and calculates application compositions in a resource-aware efficient manner. This ensures high efficiency even in dynamically changing environments.

Beyond this, many typical pervasive environments contain a fixed set of applications and devices that are frequently used. In such scenarios, identical resources are part of subsequent configuration calculations. Thus, the involved devices undergo a quite similar configuration process whenever an application is launched. However, starting the configuration from scratch every time not only consumes a lot of time, but also increases communication overhead and energy consumption of the involved devices. Therefore, our solution integrates the results from previous configurations to reduce the severity of the configuration problem in dynamic scenarios.

We prove in prototypical real-world evaluations as well as by simulation and emulation that our comprehensive approach provides efficient automated configuration in the complete spectrum of possible application scenarios. This extensive functionality has not been achieved by related projects yet. Thus, our work supplies a significant contribution towards seamless application configuration in Pervasive Computing.

Deutsche Zusammenfassung

Automatische Komposition adaptiver verteilter Anwendungen in heterogenen Umgebungen

1. Einleitung

Der Forschungsbereich des Pervasive Computing wird charakterisiert durch die Interaktion vieler heterogener Geräte, welche von leistungsstarken Serverinfrastrukturen bis zu winzigen, in Alltagsgegenstände integrierten mobilen Sensorknoten reichen. Hierbei können Geräte durch standardisierte drahtlose Kommunikationstechnologien wie Bluetooth oder Infrarot miteinander kommunizieren. Ebenso können die Geräte lokale Funknetze aufbauen, beispielsweise gemäß dem weit verbreiteten IEEE 802.11 Standard. Die in einer bestimmten Umgebung verfügbaren Geräte stellen dabei ihre Funktionalität anderen in der Nähe befindlichen Geräten zur Verfügung.

Die Entwicklung von Anwendungen für dynamische Pervasive Computing Umgebungen stellt eine nichttriviale Aufgabe dar. Durch Gerätemobilität, schwankende Netzwerkverbindungen oder sich ändernde physikalische Kontexte variieren die zur Verfügung stehenden Hardware- und Softwareressourcen typischerweise häufig. Die von einer Anwendung benötigten Ressourcen werden darüber hinaus meist nicht von einem einzigen Gerät erbracht, sondern die Anwendungslogik ist verteilt auf mehrere Geräte. Die Anwendungen stellen somit sowohl funktionale wie auch strukturelle Anforderungen an die Ressourcen. Des Weiteren verfügen Geräte lediglich über beschränkte Ressourcen, welche noch dazu für eine bestimmte Anwendung nicht verfügbar sein können, da sie momentan von einer anderen Anwendung genutzt werden. Als Konsequenz hieraus müssen Anwendungen vor ihrer tatsächlichen Ausführung zunächst *konfiguriert* werden, um sicher zu stellen, dass die von der Anwendung benötigten Ressourcen auch wirklich zur Verfügung stehen. Darüber hinaus müssen Anwendungen in der Lage sein, sich während ihrer Ausführungszeit ständig an wechselnde Ausführungsumgebungen anzupassen. Da die Konfigurationsvorgänge von den Anwendungsnutzern als störende Unterbrechungen wahrgenommen werden, müssen diese möglichst effizient durchgeführt werden, um die dabei entstehenden Latenzen zu minimieren und dem Nutzer somit eine möglichst hohe Dienstgüte zur Verfügung zu stellen.

In der Vergangenheit beschränkten sich die meisten verwandten Projekte bei der Konfiguration verteilter Anwendungen lediglich auf eine bestimmte Art von Anwendungsumgebung. Deshalb konnten Konfigurationen entweder nicht universell durchgeführt werden, da eine unterstützende Infrastruktur zwingend erfordert wurde. Oder die Konfigurationen liefen in ressourcenreichen Umgebungen ineffizient ab, da die Verteilung der Konfigurationsaufgaben nicht gemäß der Leistungsfähigkeit der Geräte angepasst wurde. Da Pervasive Computing Umgebungen typischerweise hochdynamisch sind, ist die effiziente Unterstützung lediglich einer bestimmten Umgebung inakzeptabel, um eine adäquate Dienstqualität zu erreichen. Außerdem sind viele Umgebungen hochgradig heterogen hinsichtlich der Fähigkeiten der enthaltenen Geräte, da sie sowohl ressourchenschwache mobile Geräte als auch ressourcenstarke Infrastrukturen mit deutlich höherer Leistung enthalten.

Daher wird in dieser Arbeit ein neuer Ansatz verfolgt, welcher dynamisch wechselnde Umgebungen effizient unterstützt, indem er die Geräte in Gruppen anordnet und die Konfigurationsaufgaben zwischen den einzelnen Gruppen gleichmäßig aufteilt. Dadurch werden Flaschenhälse bei der Konfiguration vermieden, gleichzeitig aber eine parallele Berechnung von Konfigurationen ermöglicht. Um dieses Ziel zu erreichen, wurden im Rahmen dieser Arbeit neue Konfigurationsalgorithmen entwickelt, welche sowohl die zur Verfügung stehenden Ressourcen als auch die Anforderungen der Anwendungen sowie die Anwendungsnutzung in vergangenen Konfigurationen berücksichtigen. Außerdem wird ein Verfahren integriert, welches die tatsächlich gewählte Konfigurationsmethode in einer bestimmten Umgebung entsprechend anpasst. Dadurch wird die Umgebungsheterogenität, die durch Geräte mit unterschiedlicher Leistungsfähigkeit entsteht, sinnvoll ausgenutzt. Die unterstützende Systemsoftware wurde dabei so konzipiert, dass sie zum einen minimal bezüglich ihrer Ressourcenanforderungen ist, um ressourcenschwache Geräte zu unterstützen. Gleichzeitig wurde ihre Softwarearchitektur flexibilisiert, um die effiziente Ausnutzung von Ressourcen auf leistungsstärkeren Geräten zu ermöglichen.

Neben der automatischen, in Abhängigkeit von der aktuellen Umgebung vorgenommenen Anpassung des Konfigurationsverfahrens wird in dieser Arbeit ein Verfahren vorgestellt, welches durch die Ausnutzung der Ergebnisse früherer Konfigurationsvorgänge folgende Konfigurationsprozesse insbesondere in Szenarien mit geringer Dynamik, aber auch in allen anderen Umgebungen optimiert. Bei diesem Verfahren werden partielle Konfigurationen, welche innerhalb einer Konfiguration genutzt wurden, in einem Zwischenspeicher gesichert und für zukünftige Konfigurationen bereitgestellt. Wenn diese Anwendungsteile sich dann in zukünftigen Konfigurationsvorgängen als integrierbar erweisen, da sie die geforderte Funktionalität bereitstellen und momentan nicht von anderen Anwendungen genutzt werden, so müssen für diese Anwendungsteile keine Berechnungen durchgeführt werden. Stattdessen werden diese Anwendungsteile einfach aus dem Zwischenspeicher geladen. Dadurch wird der Berechnungsaufwand reduziert, und die bei der Konfiguration auftretenden Latenzen sinken, was eine nahtlosere Konfiguration ermöglicht. Dieses Verfahren wurde unabhängig von konkreten Konfigurationsalgorithmen entwickelt und kann somit in sämtlichen Umgebungen, die durch die bereitgestellten Konfigurationsalgorithmen unterstützt werden, eingesetzt werden. Die entsprechend entwickelten Konzepte und Mechanismen werden in den folgenden Abschnitten detaillierter diskutiert.

2. Systemmodell und Problemstellung

In dieser Arbeit wird zwischen zwei verschiedenen Arten von Geräten unterschieden: Bei den *ressourcenschwachen* Geräten handelt es sich typischerweise um mobile tragbare Geräte wie Mobiltelefone oder Smart Phones. Dahingegen sind *ressourcenstarke* Geräte deutlich leistungsfähiger, da sie üblicherweise Standard-PC-Hardware verbaut haben. Ressourcenstarke Geräte können mobil (z.B. Laptops), aber auch stationär (z.B. Arbeitsplatzrechner) sein und sind durch ihre erhöhte Leistung besonders geeignet, um innerhalb einer Konfiguration berechnungsintensive Aufgaben zu übernehmen. Durch die Zusammensetzung von Geräten dieser beiden Gerätetypen sind Pervasive Computing Umgebungen mit verschiedenen Graden an Heterogenität denkbar: Während manche Umgebungen lediglich ressourcenschwache Geräte enthalten und homogen hinsichtlich des Leistungsspektrums der Geräte sind (so genannte *Ad Hoc* Umgebungen), weisen infrastruktur-gestützte Umgebungen, in denen neben ressourcenschwachen auch eine Anzahl ressourcenstarker Geräte zur Verfügung stehen, eine höhere Heterogenität auf.

Für diese Arbeit wird ein komponenten-basiertes Anwendungsmodell angenommen, in welchem eine Anwendung aus mehreren Komponenten besteht, wobei jede Komponenteninstanz eine bestimmte Menge an Ressourcen benötigt. Anwendungen werden als Baum von untereinander abhängigen Komponenten betrachtet. Die gesamte Anwendung wird ausgeführt, nachdem passende Kompontenen für jede Komponentenabhängigkeit entsprechend ihrer Position in der Baumstruktur ermittelt und anschließend rekursiv gestartet wurden. Die Wurzelkomponente des Baums wird hierbei als Anwendungsanker bezeichnet und befindet sich auf dem Gerät, von welchem aus die Anwendung gestartet wird. Jede Komponente befindet sich jeweils auf genau einem Gerät, die Anwendungslogik ist somit auf die anwesenden Geräte verteilt. Innerhalb des Anwendungsbaums ist jede Komponente eindeutig durch eine rekursiv aufgebaute Kennung adressierbar. Typische Anwendungsgrößen für verteilte Pervasive Computing Anwendungen wurden durch eine empirische Studie ermittelt und führten zu Anwendungen, die aus 8 bis 34 Komponenten bestehen. In dieser Arbeit werden daher Anwendungen in dieser Größenklasse untersucht.

Es wird außerdem angenommen, dass eine unterstützende Systemsoftware zur Verfügung steht, die zum einen von den verwendeten Kommunikationstechnologien abstrahiert, zum anderen für die automatische Konfiguration der Anwendungen verantwortlich ist, sodass diese für den Anwendungsnutzer transparent und automatisch abläuft. Die Systemsoftware soll dabei zu jeder Zeit jedem Gerät eine konsistente Sicht auf die momentan zur Verfügung stehenden Geräte sowie die von diesen Geräten angebotenen Dienste bereitstellen. Exemplarische Systeme, welche die gestellten Anforderungen erfüllen, sind die Systemplattform BASE [BSGR03] und das Komponentensystem PCOM [BHSR04].

Ausgehend von den oben beschriebenen Annahmen hinsichtlich der verwendeten Geräte, Anwendungen und unterstützenden Systemsoftware lässt sich die Problemstellung herleiten, die dieser Arbeit zu Grunde wird. In homogenen Umgebungen wurde das verteilte Konfigurationsproblem bereits in vorherigen Arbeiten gelöst, indem das Problem der Konfiguration auf das verteilte Bedingungserfüllungsproblem abgebildet wurde und ein Algorithmus aus dem Bereich der verteilten künstlichen Intelligenz zur Problemlösung angepasst wurde [Han09]. Allerdings ist die komplett verteilte Anwendungskonfiguration suboptimal in heterogenen Umgebungen, da die Berechnung auf sämtliche Geräte verteilt wird und somit leistungsschwache Geräte zum Flaschenhals der Konfiguration werden können. Daher muss in einer bestimmten Umgebung eine geeignete Teilmenge an Geräten gefunden werden, welche die Konfigurationsaufgaben unter sich aufteilen. Daraus ergeben sich Fragestellungen wie die automatische Unterscheidung von ressourcenstarken und ressourcenschwachen Geräten, die Verteilung der Konfigurationslast unter den Konfigurationsgeräten, die Bereitstellung entsprechender Konfigurationsverfahren, oder die automatische Auswahl eines in einer bestimmten Umgebung passenden Verfahrens. Innerhalb dieser Arbeit werden Lösungen zu sämtlichen dieser Fragestellungen vorgeschlagen.

Im Rahmen dieser Arbeit werden fünf Anforderungen an die Lösungen dieser Probleme gestellt. Zunächst muss die vorgeschlagene Lösung *adaptiv* sein, also in der Lage sein, sich automatisch an wechselnde Anforderungen und dynamische Umgebungen anzupassen. Darüber hinaus muss die Berechnung von Konfigurationen *automatisch* erfolgen, um die Transparenz gegenüber dem Anwendungsnutzer aufrecht zu erhalten. Außerdem soll die Lösung gültige Konfigurationen so schnell wie möglich berechnen, um die Anwendung dem Nutzer möglichst ohne Verzögerung bereit zu stellen, somit also *effizient* sein. Die Berechnung von Konfigurationen muss darüber hinaus *ressourcen-abhängig* geschehen, um die Heterogenität der Geräte auszunutzen und berechnungsintensive Aufgaben lediglich an die stärksten Geräte zu verteilen. Nicht zuletzt müssen die *Ergebnisse vorangegangener Konfigurationen berücksichtigt* und, sofern möglich, in zukünftige Konfigurationsprozesse integriert werden. Dadurch wird die gesamte Konfigurationslast für alle involvierten Geräte verringert und somit die zur Verfügung stehenden Berechnungskapazitäten in einer *ressourcenschonenden* Weise verwendet.

3. Verwandte Arbeiten

Zu Beginn dieser Arbeit existierten bereits verschiedene Arbeiten, welche sich mit der Konfiguration von verteilten Anwendungen des Pervasive Computing beschäftigten. Die Projekte können gemäß deren Forschungschwerpunkten im Wesentlichen in zwei Gruppen eingeteilt werden: Verfahren zur Anwendungskonfiguration in homogenen mobilen Ad Hoc Netzen, und Verfahren zur Anwendungskonfiguration in infrastrukturbasierten heterogenen Umgebungen. Eine zusätzliche Diversifizierung kann durch die Unterscheidung, ob die Konfiguration und Adaption automatisch vom System oder manuell durch den Anwendungsprogrammierer oder Benutzer vorgenommen werden muss, erreicht werden.

Infrastrukturen wie *Oxygen* [Rud01], *iRoom* [JFW02], *Gaia* [RHC+02], *Olympus* [RCAM+05] oder *MEDUSA* [DGIR11] konzentrieren sich auf Fragestellungen, die sich durch die Integration von Rechnersystemen in infrastruktur-gestützten Umgebungen ergeben. Bei der Konzeption dieser Infrastrukturen ging man allerdings davon aus, dass bestimmte Rechnersysteme ständig für notwendige Koordinationsaufgaben zur Verfügung stehen, daher sind diese Systeme nicht in Ad Hoc Umgebungen anwendbar.

Projekte wie *Weaves* [OGT+99], *Aura* [SG02], *P2PComp* [FHMO04], *Mobile Gaia* [CAMCM05] oder *RUNES* [CCG+07] benötigen zwar keine speziellen Infrastrukturen und können demnach in sämtlichen Umgebungen eingesetzt werden. Allerdings nutzen diese Systeme zusätzlich vorhandene Rechenressourcen von leistungsstarken Infrastrukturgeräten nur ineffizient, da sie keine Unterscheidung zwischen den Geräten vornehmen, sondern die Rechenlast gleichmäßig zwischen allen Geräten aufteilen. Dadurch werden in heterogenen Umgebungen nur suboptimale Ergebnisse hinsichtlich der Geschwindigkeit der Konfigurationsprozesse erzielt.

Die im Rahmen dieser Arbeit verwendete Systemsoftware, welche aus der Kommunikationsmiddleware *BASE* [BSGR03] und dem Komponentensystem *PCOM* [BHSR04] besteht, kann in verschiedensten Umgebungen verwendet werden. Allerdings unterstützt sie Umgebungen, welche als heterogen hinsichtlich der verfügbaren Rechenressourcen betrachtet werden können, nicht effizient, da ihr Fokus bisher vor allem auf komplett verteilter Anwendungskonfiguration in infrastrukturlosen Umgebungen lag.

Andere Systeme für Ad Hoc Umgebungen basieren hingegen nicht auf vollautomatisierter Anwendungskonfiguration, sondern legen die Verantwortung hierfür entweder in die Hände des Anwenders (z.B. *Speakeasy*, [ENS+02], *OSCAR* [NES08]) oder des Anwendungsprogrammierers (z.B. *one.world*, [Gri04]).

Aus diesem Überblick wird deutlich, dass keines der bisher existierenden Projekte eine effiziente automatische Anwendungskonfiguration *sowohl* in homogenen Ad Hoc Umgebungen *als auch* in heterogenen infrastrukturbasierten Szenarien bereitstellt. Darüber hinaus nutzt keines der genannten Systeme die Ergebnisse vergangener Konfigurationen für zukünftig anstehende Konfigurationsprozesse, um die Konfigurationslatenzen sowie die Konfigurationslast der involvierten Geräte weiter zu verringern. Daher war ein wesentliches Ziel dieser Arbeit, erstmals im Forschungsbereich des Pervasive Computing eine effiziente automatische Unterstützung von Anwendungskonfigurationen in verschiedensten, dynamischen Umgebungen bereitzustellen.

4. Ein hybrider Ansatz zur automatischen Anwendungskonfiguration

Basierend auf einem in einer vorigen Arbeit [HBR05] entwickelten komplett dezentralen Algorithmus wird in diesem Kapitel der Weg über einen komplett zentralen Ansatz [SHR08b] hin zu einem hybriden Ansatz [SHR10], welcher die Vorteile der dezentralen und zentralen Algorithmen vereint, beschrieben. Diese Ansätze erlauben besonders in heterogenen Umgebungen eine effizientere Konfiguration als der dezentrale Ansatz. Außerdem wird ein Rahmenwerk [SHR08a] präsentiert, welches eine automatische Anpassung des Grades der Verteiltheit in den Berechnungen ermöglicht und somit eine optimierte Konfiguration selbst in dynamischen Szenarien erlaubt.

Dezentrale Konfiguration in homogenen Umgebungen

Handte et al. stellten 2005 einen dezentralen Algorithmus vor, welcher besonders für homogene Umgebungen geeignet ist, da er die Konfigurationslast gleichmäßig unter allen verfügbaren Geräten verteilt [HBR05]. Dieser Ansatz basiert auf Algorithmen aus dem Forschungsbereich der verteilten künstlichen Intelligenz [YDIK98]. Durch den Verzicht auf eine zentrale Instanz ist die Anwendbarkeit in sämtlichen Umgebungen, vor allem in einfachen Ad Hoc Umgebungen, gewährleistet. Allerdings ist dieser Ansatz ineffizient in Umgebungen, welche zusätzlich rechenstarke Geräte bereitstellen, da die Verteilung der Konfigurationslast nicht gerätespezifisch angepasst werden kann.

Folglich ist für solche Umgebungen ein Konfigurationsverfahren erforderlich, das die zusätzlichen Rechenressourcen effizient ausnutzt. Um mögliche Flaschenhälse bei der Konfiguration zu vermeiden, werden bei diesem Ansatz die ressourcenschwachen mobilen Geräte wie PDAs oder Smartphones von der Berechnung ausgeschlossen. Sie müssen lediglich im Voraus Informationen über die von ihnen zur Verfügung gestellten Komponenten bereitstellen. Der dezentrale Ansatz dient dabei in den vorgenommenen Evaluationsmessungen als Referenz.

Zentrale Konfiguration in schwach heterogenen Umgebungen

Die einfachste Konfigurationsmethode in heterogenen Umgebungen besteht darin, die Konfiguration komplett zentral auf dem ressourcenstärksten Gerät in der Umgebung berechnen zu lassen. Daher werden zunächst existierende zentrale Backtracking-Algorithmen aus dem Bereich des maschinellen Lernens untersucht und ein erweiterter Algorithmus namens *Direct Backtracking (DBT)* entworfen [SHR08b]. Dieser Algorithmus baut auf dem auf Tiefensuche basierenden *Synchronous Backtracking (SBT)* [YDIK98] Algorithmus, einer zentralen und synchronen Variante des verteilten Asynchronous Backtracking Algorithmuses, auf. Zur Beschleunigung des Konfigurationsprozesses wird DBT um zwei zusätzliche Mechanismen erweitert: *proaktive Adaptionsvermeidung* und *intelligentes Backtracking*.

Die *proaktive Adaptionsvermeidung* wird in der Weise ausgeführt, dass während eines Konfigurationsvorgangs im Falle verschiedener zur Verfügung stehender Komponenten, die dieselbe geforderte Funktionalität bereitstellen, diejenige Komponente ausgewählt wird, welche die minimale Anzahl an Ressourcen auf dem Gerät, durch das sie bereitgestellt wird, verbraucht. Hierdurch wird die Wahrscheinlichkeit zukünftiger Ressourcenkonflikte auf dem entsprechenden Gerät während einer Konfiguration minimiert, da die Ressourcen schonend vergeben werden. Dadurch sinkt die Anzahl nötiger Adaptionsvorgänge während einer Konfiguration.

Bei bestimmten Anwendungskonstellationen kann es dennoch vorkommen, dass Adaptionen durchgeführt werden müssen. Dann muss die Menge der für eine Anwendung ausgewählten Komponenten durch einen Adaptionsvorgang angepasst werden. Hierbei muss berücksichtigt werden, dass die Adaption einer Komponente zusätzlich die Adaption anderer Komponenten nach sich ziehen kann, um weiterhin sämtliche struktulle Bedingungen der Anwendung zu erfüllen. Um diese Adaptionen mit geringst möglichem Aufwand durchzuführen, wurde ein *intelligenter Backtracking-Mechanismus* eingeführt. Dieser adaptiert diejenige Komponente, welche den geringst möglichen Adaptionsaufwand verursacht und darüber hinaus möglichst wenig zusätzliche Adaptionen nach sich zieht.

Ein Rahmenwerk zur automatischen Anpassung des Grades der Konfigurationsverteilung

Um den Grad der Verteilung des Konfigurationsvorgangs optimal an die momentane Anwendungsumgebung anzupassen, müssen die in der Umgebung befindlichen Geräte entsprechend vorbereitet werden. Hierfür wurde ein Rahmenwerk entwickelt [SHR08a], welches automatisch und proaktiv, d.h. *vor* tatsächlich stattfindenden Konfigurationsprozessen, die für die Konfiguration relevanten Ressourceninformationen von entfernten Geräten überträgt, um diese Aufgaben nicht mehr zur Konfigurationszeit erledigen zu müssen. Dadurch wird die tatsächliche Konfigurationszeit gesenkt und somit die Effizienz der Konfiguration erhöht. Durch das entwickelte Rahmenwerk werden insbesondere drei Herausforderungen von heterogenen Umgebungen adressiert:

- *Die automatische Anpassung des Verteilungsgrades der Konfigurationsberechnung:* Um verschiedene Umgebungen effizient zu unterstützen, ist die automatische Auswahl eines passenden Konfigurationsalgorithmus nötig. Dafür ist ein Verfahren nötig, welches zunächst basierend auf der Art der verfügbaren Geräte den Umgebungstyp – ressourcenschwache Ad Hoc-Umgebung oder ressourcenstarke Infrastrukturumgebung – ermittelt. Im Falle einer Ad Hoc-Umgebung wird anschließend der verteilte, im Falle einer Infrastrukturumgebung der zentrale Konfigurationsalgorithmus ausgewählt. Dieser Selektionsmechanismus ist so konzipiert, dass er einfach um die Unterstützung zusätzlicher Konfigurationsmethoden erweitert werden kann.
- *Die automatische Ermittlung der Geräte, welche in ressourcenreichen Umgebungen die Berechnung der Konfiguration übernehmen:* Hierfür dient ein auf

Knotengruppierungen (engl. *Clustering*) basierendes, in die Systemsoftware integriertes Rahmenwerk. Um die Gruppenstruktur zu ermitteln, wird ein verteilter Algorithmus von Basagni et al. (*Distributed Clustering Algorithmus*, [BCFJ97]) verwendet, als eigentliches Gruppierungskriterium werden die vorhandenen Rechenressourcen auf den vorhandenen Geräten gewählt. Das ressourcenreichste Gerät in der Umgebung wird dann zum Gruppenführer (engl. *Clusterhead*) bestimmt und somit verantwortlich für die Konfigurationsberechnung der benachbarten Geräte gemacht.

- *Das Erlangen konfigurationsspezifischer Informationen durch den Gruppenführer:* Zur Realisierung eines effizienten Konfigurationsvorgangs auf dem Gruppenführer, muss dieser zunächst die für die Konfiguration relevanten Informationen der vorhandenen Geräte – im Wesentlichen deren aktuelle Ressourcen- und Kompontenverfügbarkeiten – bestimmen. Hierfür dient ein Verfahren, durch welches dem Gruppenführer die relevanten Informationen über vorhandene Ressourcen automatisch durch Analyse von Veränderungen der Gruppenstruktur im Voraus zur Verfügung gestellt werden. Geräte, deren Ressourcenverfügbarkeit sich ändern, teilen dem Gruppenführer dabei automatisch ihre aktualisierte Ressourcenlage mit. Der Gruppenführer baut beim Empfang dieser Informationen eine interne Repräsentation der entfernten Geräte in Form von sogenannten *virtuellen Containern (VCs)* auf. Der Assembler greift bei einem Konfigurationsprozess dann lokal auf diese virtuellen Container zu. Die Erzeugung der virtuellen Container stellt einen der Konfiguration vorgelagerten Prozess dar. Hierdurch sinkt der tatsächliche Aufwand der Konfiguration, da während der Konfiguration keinerlei Kommunikation zwischen den Geräten notwendig ist.

Hybride Konfiguration in stark heterogenen Umgebungen

Um das komplette Spektrum zwischen zentraler Konfiguration in schwach heterogenen Umgebungen und vollständig verteilter Konfiguration in homogenen Umgebungen abzudecken, ist ein erweitertes Verfahren nötig, welches die Vorteile der verteilten Konfiguration – generelle Anwendbarkeit in sämtlichen Umgebungen, keine Single-Point-of-Failure Problematik – mit denen der zentralen Konfiguration – effiziente Nutzung ressourcenstarker Geräte, geringer Kommunikationsaufwand – in stark heterogenen Umgebungen vereint.

Aufbauend auf den bisher konzipierten Ansätzen musste daher ein Verfahren entwickelt werden, welches eine optimierte hybride Anwendungskonfiguration ermöglicht, die teilweise dezentral und teilweise zentral abläuft [SHR10]. Dafür wird zunächst eine Teilmenge aller Geräte bestimmt, welche dann für die restlichen Geräte die Konfiguration ihrer Komponenten und Ressourcen übernehmen. Um die Effizienz des Verfahrens zu garantieren, sollen unter Nutzung des Gruppierungs-Rahmenwerks lediglich die ressourcenstarken Geräte wie Laptops, Desktop-PCs oder Server aktiv in die Konfiguration eingebunden werden. Hierfür muss zunächst der eingeführte Mechanismus, welcher basierend auf der aktuellen Umgebung den passenden Konfigurationsalgorithmus auswählt, erweitert werden, um in Umgebungen mit mehreren ressourcenstarken Geräten eine Konfigurationsberechnung auf genau diesen Geräten zu ermöglichen.

Anschließend muss sichergestellt werden, dass eine eindeutige Abbildung der ressourcenschwachen Geräte auf ressourcenstarke Geräte erfolgt, welche dann lokal die ihnen zugeordneten ressourcenschwachen Geräte mittels der oben genannten virtuellen Container emulieren und diese in den Konfigurationsprozess einbinden. Damit eine ausgeglichene Konfigurationslast zwischen den ressourcenstarken Geräten erreicht wird, wird ein Verfahren integriert, das gemäß dem bekannten *Round-Robin*-Schema die Gerätezuordnung vornimmt. Hierdurch wird sichergestellt, dass jedem starken Gerät annähernd gleich viele schwache Geräte zugeordnet werden und somit die Konfigurationslast gleichverteilt wird. Um dynamische Umgebungen zu berücksichtigen, wird zusätzlich eine automatische Aktualisierung der Geräteabbildungen integriert. Zur automatischen Erkennung von Änderungen der Geräteumgebung wird hierbei ein von der Systemsoftware bereitgestellter Mechanismus verwendet. Im Falle neu hinzugekommener oder nicht mehr verfügbarer ressourcenstarker und -schwacher Geräte wird durch eine Neugruppierung der Geräte die ausgeglichene Konfigurationslast auf den starken Geräten beibehalten, sodass auch dynamische Umgebungen berücksichtigt werden.

Abschließend ist ein erweiterter Konfigurationsalgorithmus nötig, bei dem die ermittelten ressourcenstarken Geräte jeweils zentral die Teilkonfigurationen für die ihnen zugeordneten Geräte berechnen, um anschließend dezentral untereinander die ermittelten Teilkonfigurationen auszutauschen und somit die Gesamtkonfiguration der Anwendung zu bestimmen.

Evaluation der entwickelten Verfahren

Bei vergleichenden Messungen zur Ermittlung der Leistungsfähigkeit der einzelnen Verfahren wurde zunächst in einer schwach heterogenen Umgebung, in welcher lediglich ein starkes Gerät zur Verfügung steht, der entwickelte zentralisierte *Direct Backtracking*-Konfigurationsalgorithmus mit dem verwandten *Synchronous Backtracking*-Algorithmus hinsichtlich der zu erwartenden Konfigurationslatenz verglichen. Dabei zeigte sich, dass der neue Algorithmus aufgrund seiner fortschrittlichen Mechanismen deutlich leistungsfähiger als der verwandte Algorithmus ist. In den Messungen ergab sich durch Nutzung der erweiterten Mechanismen zur proaktiven Adaptionsvermeidung sowie intelligentem Backtracking für Direct Backtracking ein im Vergleich zu Synchronous Backtracking signifkant beschleunigter Konfigurationsprozess, welcher die Konfigurationslatenz bis auf unter 10 % der entsprechenden Latenz von Synchronous Backtracking senkte.

Anschließend wurde, wiederum in einem schwach heterogenen Szenario, der zentrale Algorithmus mit dem in einer vorigen Arbeit entwickelten verteilten Algorithmus verglichen, um zu überprüfen, ob die zentrale Konfiguration durch lokale Berechnungen auf dem stärksten Gerät die Konfigurationseffizienz tatsächlich steigert. Bei den Messungen zeigte sich, dass der zentrale Ansatz die Latenzen im Durchschnitt um fast 40 % reduziert, da er durch Ausnutzung des Vorkonfigurationsprozesses sowie der Heterogenität der Umgebung eine effiziente zentrale Konfiguration ermöglicht. Die maximale Reduzierung der Konfigurationslatenz betrug sogar 84 %.

Daraufhin wurde der hybride Ansatz in stark heterogenen Umgebungen mit den zentralen und verteilten Ansätzen verglichen. Dabei stellte sich heraus, dass durch die parallele Berechnung, die lediglich auf den ressourcenstarken Geräten abläuft, die Latenzen nochmals um mehr als 25 % verringert werden können.

In den vergleichenden Messungen konnte also gezeigt werden, dass in Umgebungen mit verschiedenen Graden an Heterogenität jeweils unterschiedliche Konfigurationsansätze zu den geringsten Konfigurationslatenzen führten. So war in homogenen, ressourcenschwachen Umgebungen die verteilte Anwendungskonfiguration am Schnellsten, während in schwach heterogenen Umgebungen die zentralisierte und in stark heterogenen Umgebungen die hybride Anwendungskonfiguration zu besten Ergebnissen hinsichtlich der auftretenden Konfigurationslatenz führten. Durch diese Ergebnisse wurde schließlich die entwickelte einfache Verteilungsheuristik bestätigt, welche abhängig vom Grad der Heterogenität der Umgebung die automatische Auswahl der zu diesem Szenario passendsten Konfigurationsmethode durchführt.

5. Partielle Anwendungskonfigurationen

In vielen typischen Szenarien wie Hörsälen oder Konferenzräumen werden häufig identische Anwendungskomponenten auf denselben Geräten wiederverwendet. Diese Komponenten beschreiben sogenannte *partielle Anwendungskonfigurationen (PACs)*. Allerdings wurden diese durch die zuvor beschriebenen Ansätze nicht genutzt, weshalb in solchen Fällen trotzdem jeweils der komplette Konfigurationsprozess für jede einzelne Komponente durchgeführt werden musste. Durch Berücksichtigung der Ergebnisse voriger Konfigurationsprozesse kann die Konfigurationslatenz jedoch effektiv verringert werden. Dafür ist ein Verfahren erforderlich, um automatisch partielle Anwendungskonfigurationen zu bestimmen und zu speichern, effektiv an die in der Umgebung vorhandenen Geräte zu verteilen und automatisch in zukünftige Konfigurationsvorgänge einzubinden [SHRB13].

Die im Rahmen dieser Arbeit verwendeten PACs werden von unten nach oben aufgebaut, also von den Blattknoten des Anwendungsbaums ausgehend hin zur Wurzel des Baums. Daher sind bei jeder Komponente, die Teil einer PAC ist, auch sämtliche Kindskomponenten in der PAC enthalten, wodurch keine unaufgelösten Abhängigkeiten durch diese PAC in den Konfigurationsprozess eingebracht werden. PACs, die dieser Methode folgen, sind *umgebungsspezifisch* und vor allem in solchen Szenarien sinnvoll, in denen immer dieselben Komponenten einer bestimmten Umgebung verwendet werden. Umgebungsspezifische PACs bieten damit einen hohen Nutzwert für heterogene Umgebungen mit geringer Dynamik und führen nicht zu unaufgelösten Abhängigkeiten im Konfigurationsprozess. Hierdurch wird der Rekonfigurationsaufwand in den meisten Fällen deutlich reduziert.

Zur Speicherung dieser partiellen Anwendungskonfigurationen wird ein auf XML basierendes Verfahren verwendet, welches nach einer erfolgreichen Konfiguration automatisch sämtliche entstehende Teilkonfigurationen lokal speichert. Um diese

Teilkonfigurationen anschließend zu verteilen und die Konsistenz unter den vorhandenen Geräten zu gewährleisten, werden unter Nutzung der Kommunikationsmechanismen der verwendeten Systemsoftware die erzeugten PACs per Broadcast an die Geräte in der Umgebung gesendet. Mit Hilfe der Überwachung der Umgebung durch die Systemsoftware kann der Grad der Verteilung der PACs außerdem automatisch angepasst werden, um die Anwendbarkeit dieser Konfigurationen auch in dynamischen Umgebungen und bei komplett verteilter Anwendungskonfiguration zu ermöglichen.

Außerdem müssen die bisher verwendeten Konfigurationsalgorithmen erweitert werden, damit sie für gespeicherte partielle Konfigurationen automatisch die Verfügbarkeit der von ihnen verwendeten Komponenten überprüfen und diese dann in den Konfigurationsprozess einbeziehen können. Hierfür wird zunächst eine initiale Bewertung der neu zu speichernden partiellen Konfigurationen definiert. Basierend auf vorhergegangenen Konfigurationen und Adaptationen werden anschließend Schlüsse für zukünftige Konfigurations- und Adaptionsprozesse gezogen und der Nützlichkeitswert jeder partiellen Anwendungskonfiguration für zukünftige Anwendungskonfigurationen, basierend auf der zur Konfigurationszeit verfügbaren Anwendungskomponenten, dynamisch angepasst. Die Konfigurationsalgorithmen wurden in der Art erweitert, dass sie bei der Konfiguration von Komponenten zunächst überprüfen, ob gespeicherte Teilkonfigurationen genutzt werden können, da sämtliche in den Teilkonfigurationen enthaltene Komponenten der Anwendung aktuell zur Verfügung stehen. Nur wenn keine nutzbaren Teilkonfigurationen integriert werden können, muss eine gültige Belegung der jeweiligen Komponenten neu berechnet werden.

Schließlich werden Messungen durchgeführt, die den Ansatz der partiellen Anwendungskonfiguration in die Konfigurationsberechnung einschließen. Dadurch kann in sämtlichen homogenen wie heterogenen Umgebungen die Latenz reduziert werden, wobei die Leistungsfähigkeit dieses Ansatzes in stark heterogenen Umgebungen mit geringer Dynamik am höchsten ist, da die Verfügbarkeit der einzelnen Komponenten sich hier am wenigsten ändert und die im Zwischenspeicher gehaltenen partiellen Konfigurationen somit am längsten nutzbar bleiben.

In abschließenden Messungen wird gezeigt, dass der Ansatz der partiellen Anwendungskonfigurationen mit adaptiven Parametern nur um 9 % höhere Konfigurationslatenzen verursacht als der optimale Fall, in welchem der Zwischenspeicher von unbeschränkter Größe ist und somit sämtliche jemals benutzten partiellen Konfigurationen dauerhaft zur Verfügung stehen. Die absoluten Latenzen pendeln sich in diesem Fall bei lediglich 1,5 Sekunden ein, was im Vergleich zur Konfiguration ohne partielle Konfigurationen (rund 2,5 Sekunden) die Dienstqualität deutlich erhöht.

6. Prototyp

Die entwickelten Verfahren und Mechanismen wurden in eine prototypische Implementierung der Systeme BASE [BSGR03] und PCOM [BHSR04] integriert und mit diesen Systemen evaluiert. Die Systemplattform BASE wurde hierfür um einen

Dienst erweitert, der den automatischen Abruf der Konfigurationsklassen von entfernten Geräten ermöglicht. Außerdem wurde ein verteilter Ereignisdienst integriert, durch den ein Gerät automatisch über Änderungen an der Ressourcenlage anderer Geräte informiert wird. Dies ist nötig, um bei der lokalen Konfiguration entfernter Komponenten die Gültigkeit der aktuellen Ressourcenverfügbarkeit zu gewährleisten.

Das Komponentensystem PCOM wurde an mehreren Stellen erweitert: Zunächst einmal wurde das Rahmenwerk integriert, welches die automatische Gruppierung von Geräten abhängig von bestimmten, erweiterbaren Gruppierungsstrategien ermöglicht. Zur lokalen Emulation von entfernten Geräten wurde das Konzept der virtuellen Container realisiert, welche mit Hilfe der neuen Dienste von BASE die aktuelle Ressourcenlage entfernter Geräte abbilden, ohne Anfragen an diese stellen zu müssen. Dadurch wird eine komplett lokale Konfiguration ermöglicht. Zur Unterstützung verschiedenster Umgebungen wurde außerdem ein Auswahlverfahren implementiert, welches den Grad der Dezentralisierung bei der automatischen Konfiguration abhängig vom Grad der Heterogenität in der Umgebung anpasst. Durch neu entwickelte zentrale und hybride Konfigurationsalgorithmen (sogenannte *Assembler*) werden dabei verschiedenste heterogene Umgebungen effizient unterstützt. Die Ausnutzung vergangener Konfigurationsvorgänge wird mittels der Bereitstellung eines Zwischenspeichers beschränkter Größe und eines Speicherersetzungsverfahrens, welches für die Erhaltung der nützlichsten Konfigurationen innerhalb des Speichers sorgt, ermöglicht. Die zuvor entwickelten Konfigurationsverfahren mussten dabei nur geringfügig angepasst werden, um die Ausnutzung der Ergebnisse vergangener Konfigurationen zu ermöglichen.

Zur Evaluation großer Anwendungen und hochdynamischer Umgebungen wurde neben dem Prototyp noch ein ereignisdiskreter Simulator von PCOM in der Weise erweitert, dass die Verfügbarkeit und Nichtverfügbarkeit von Geräten realitätsgetreu nachgebildet werden kann. Außerdem wurden zentralisierte Konfigurationsverfahren in diesem Simulator implementiert, um deren Effizienz in verschiedenen Szenarien einfach vergleichen zu können. Um die Leistungsfähigkeit der entwickelten Konzepte beispielhaft darzulegen, wurden auf dem Prototypen die in den Kapiteln 4 und 5 präsentierten Messungen in verschiedenen homogenen sowie heterogenen Umgebungen vorgenommen.

7. Ausblick

Für zukünftige Arbeiten bietet sich die Erforschung erweiterter Verfahren zur intelligenten automatischen Vorberechnung und Verteilung von Konfigurationen und Teilkonfigurationen an. Dafür können spekulativ ermittelte (Teil-)Konfigurationen, die ausgehend von der aktuellen Ressourcenlage in berechnungsfreien (engl. *idle*) Zeiten vorgenommen werden, verwendet werden. Außerdem bietet es sich an, das bestehende Komponentenmodell zu flexibilisieren, um alternative Konfigurationen zu ermöglichen und somit die Anzahl gültiger Konfigurationen zu erhöhen. Diese erweiterten Untersuchungen sollen zu einer weiteren Reduktion des Kommunikationsaufwandes und der Konfigurationslatenz sowie einer Erhöhung der Anwendungsverfügbarkeit führen.

1 Introduction

In this chapter, we lay the foundations for this work. Therefore, we first present the emergence of the Pervasive Computing research area in Section 1.1. Then in Section 1.2, we discuss distributed Pervasive Applications as the aspect of Pervasive Computing which is most important for this work. These applications need to be configured and possibly adapted at runtime, as Section 1.3 states. Following, we give a motivation for this work in Section 1.4, and present the focus and our contribution in Section 1.5. Finally in Section 1.6, we give an overview of the remainder of this thesis.

1.1. Pervasive Computing

The founding manifesto of Pervasive Computing was a ground-breaking work by Mark Weiser who observed that "the most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it." [Wei91, Wei99]. In his work, Weiser defines Pervasive Computing as "the method of enhancing computer use by making many computers available throughout the physical environment while making them effectively invisible to the user." Thus, the essence of Pervasive Computing is the creation of environments saturated with computing and communication, yet gracefully integrated with human users.

The road towards Pervasive Computing can be described as an evolution in three steps [Sat01], starting with *Distributed Systems* [TS06] in the 1970s, then turning to *Mobile Computing* [Sch03] within the 1990s and finally, evolving to *Pervasive Computing* at the beginning of the 21^{st} century.

Distributed Systems have already covered many areas that have become fundamental for Pervasive Computing, such as *remote communication* [BN84], *fault tolerance* by transaction processing [GR92], *high availability* through consistency [DGMS85], *access to remote information*, e.g. by distributed databases [Sat90], or *security* through encryption techniques [NS78]. In a second evolutionary step, the emergence of fully functional laptops and wireless networks lead to the new field of Mobile Computing, which added many important technological improvements such as *location sensitivity* [WFG92], *mobile networking* [BPT96] or *energy saving techniques* [FS99]. Finally, the consecutive technological miniaturization yielded the appearence of completely new types of sensors (e.g., digital compasses, GPS sensors, proximity sensors) and devices (e.g., PDAs, Smart Phones, netbooks). This enabled the introduction of *Smart Spaces* [SBK06] and the invisibility in means of complete disappearence of the technology from the user's mind [WB97], leading to the age of *Pervasive Computing*.

In recent years, computing has mobilized itself beyond the desktop PC. Beyond this, models for Pervasive Computing have evolved [SM03] which particularly address four system components:

- **Devices** now not only cover traditional input and output devices such as keyboards or speakers, but also wireless mobile devices such as pagers, PDAs or smart phones. Furthermore, everyday objects like cups [GBK99] or furniture [IIS+03] are becoming more and more smart [Mat03] by equipping them with sensors and wireless technology.

- **Networking technologies** have to be redesigned to support an evergrowing number of people and devices that participate in a wireless network. Furthermore, these networks now cover types of devices which have never been used in networks before, like vending machines, toasters, or refridgerators. It is supposed that one day, a billion people will interact with a million e-businesses via a trillion of interconnected intelligent devices [Amo01].

- **Applications** in Pervasive Computing are much more environment-centric than mobile applications. This is established by providing location information services [CBW03] to the users, which may be working in outdoor (e.g., GPS-based [AK06]) or even indoor scenarios (e.g., WLAN-based [SHR+08c]). For instance, in the domain of pervasive healthcare [Var07], patients are automatically monitored by scattered sensors in their environment. Further typical application scenarios are e-learning [NGL+09], ambient assisted living [SDFGB10], home entertainment [BFM+06, NES08] or disaster operations [CLM+08, CCG+07].

- **Social contexts** have strongly changed the application landscape with the rise of Web 2.0 technology [O'R05]. In recent years, a multitude of new sociocentric applications have been developed. Thus, the integration of social contexts has become an additonal important key aspect for pervasive applications [BB02].

Along with these research fields, various new kinds of challenges arise, covering scalability, heterogeneity, integration, invisibility, context-awareness and context-management [SM03]. To overcome these challenges, *system support* by means of frameworks and middleware platforms has been developed in the past years. Like in traditional distributed and mobile computing, middleware platforms introduce a logical abstraction layer between the hardware and the software. Through this, they interact with the networking kernel on the user's behalf and keep the users immersed in the pervasive computing space. Thus, they provide a uniform homogeneous software development environment on physically heterogeneous hardware systems. The primary objectives of middleware systems are to foster application portability, distributed application component interoperability, and facilitate software component integration. Moreover, the middleware wants to abstract from specific technologies and increase the quality of service for the users. Therefore, such platforms try to hide the heterogeneity of underlying layers. Common middleware platforms include CORBA [Vin97], DCOM [Ses98], Java RMI [Dow97], J2EE [SSJ02] and others. Recent research additionally provides a unique programming model for application scenarios like nomadic computing or human-centric computing. Some of these projects are Cooltown [KBM$^+$00], Oxygen [Rud01], or Aura [SG02].

Besides the term Pervasive Computing, another notion which has emerged since the early 1990s for these new types of disappearing computing systems is *Ubiquitous Computing*. These terms are considered to be more or less equivalent [CPFJ04], [WP05], so we also use them synonymously.

1.2. Pervasive Applications

From its beginning, the research area of Pervasive Computing was about applications. Pervasive applications are more environment-centric than Web-based applications or mobile computing [SM03]. Thus, pervasive applications can be characterized by the following main properties:

- **Distributed nature:** In most typical Pervasive Computing scenarios, a single device cannot provide the entire functionality required by an application because of limited resources. Hence, a main characteristic of a pervasive application is the fact that the required functionality is *distributed* among multiple devices. In consequence, these devices have to collaborate while maintaining distribution transparency to the user. To achieve this, the systems automatically *configure* the application without user interaction to find a suitable *composition* of resources that provide the functionality required by the application. McKinley et al. [MSKC04] provide a survey on adaptive system composition.
- **High degree of dynamics:** Since the corresponding runtime environments for Pervasive Applications usually include mobile devices, there is a high fluctuation in the presence of the devices whose resources provide functionalities for the applications. Obviously, this influences the availability of the resources to the applications, as specific resources may become unavailable at any time and, thus, also during the execution of an application. Therefore, pervasive systems have to specifically take care of these dynamic changes in the environment by *adapting* parts of the application [HHS$^+$07].
- **Resource constraints:** Many Pervasive Computing scenarios often have to deal with strictly limited resources, especially in sensor network scenarios [KNK05], or in Ad Hoc scenarios where people meet spontaneously with their wireless mobile devices [HBR05]. Hence, application developers, but also

system software developers need to consider these *resource constraints* in their design decisions.

Due to the mentioned main characteristics of Pervasive Applications, finding an application model for Pervasive Computing is very challenging and was first addressed by Banavar et al. [BBG+00]: According to the authors, the lifecycle of an application consists of three different parts: *Design-time* (when the user creates the application), *load-time* (when the system configures the application composition) and *run-time* (when the end-user executes the application). At *design time*, the application programmer has to identify interaction elements, specify an abstract service description language, create a task-based model for the program structure and integrate a navigation model to identify the application's programming model. Moreover, one has to think about the development methodology to realize the application from a set of requirements. Tang et al. [TYZ+11] have discussed challenges and presented solutions to enable rapid Pervasive Application design. The *load-time* of a pervasive application is typically much more dynamic than in traditional applications. Therefore, the available applications and services in a specific environment have to be dynamically discovered and, among these, a suitable set of components has to be determined. Furthermore, the user interface has to be dynamically selected and adapted from a set of available interfaces [Sch10]. At *run-time* of an application, the environment has to be monitored to detect service or resource changes, to notice disconnections, e.g. because of problems with the wireless interfaces. For such situations, failure detection as well as recovery mechanisms have to be established to identify and understand the arising problems.

Pervasive Applications are integrated within their physical environment and are aware of their location. They are independent of underlying architectures or other specific software or hardware that is used [DR07]. Recent research in Pervasive Computing covers the development of customizable pervasive applications [WHKB06], decentralized bootstrapping [KWSW07], or the formal specification and verification of pervasive applications [CP09, DBGW10]. Another important issue deals with the increase of security in mobile wireless and pervasive systems, e.g., via Intrusion Detection Systems [ZLH03, SV08]. Furthermore, some works even enlighten the vision of complete future pervasive cities like New Songdo [WRvK+08].

1.3. Configuration and Adaptation of Pervasive Applications

Due to their distributed nature, applications in pervasive computing environments need to be configured *prior* to their execution, as the availability of the needed resources may change over time. Application configuration, as part of the middleware, tries to make optimal decisions about which services on which devices have to be used by a specific application. A valid configuration covers a set of components available in the current execution environment that fulfills the requirements posed by the application. To achieve true pervasiveness, this configuration should be calculated automatically without user intervention to enable technology abstraction. In order to additionally increase the quality of service for the user, new configurations should be calculated and installed as fast as possible.

However, it may also happen during execution of an application that specific resources are no longer available and, thus, parts of the application fail due to resource shortages. Typical situations in which this may happen are:

- Users leaving the environment with their mobile devices
- Device failures, e.g. a desktop PC crashes and has to be re-booted
- Mobile devices that are running out of battery

In such situations, runtime support for the *adaptation* – also called *re-configuration* – of these applications is required. This means alternative resources need to be found that provide an adequate functionality. As it is possible that these alternative resources are currently providing the functionality of another part of the application, adaptation processes oftenly also require a re-configuration of additional parts of the application, enforcing the need for efficient adaptation algorithms. These algorithms have to analyze the application structure and the currently available resources and services and decide depending on this information about the contracts that have to be adapted.

A formal and more precise problem statement concerning configuration and adaptation of distributed applications is given in Section 2.2.1.

1.4. Motivation

In Pervasive Computing, different types of systems have emerged over the past years. Initially, most projects focused on developing an infrastructure which provides basic services and resources, e.g. input and output sources like keyboards, touchscreens, graphical user interfaces or speaker systems. However, in recent years, system support for Ad Hoc scenarios without any infrastructure has also become an important research field.

While the first class of environments require services that form an underlying infrastructure, Ad Hoc environments do not rely on any existing infrastructure at all; all devices are spontaneously connected. Thus, Ad Hoc scenarios are typically highly homogeneous with respect to the computation capabilities of the involved devices. In such scenarios, completely decentralized configuration approaches [HBR05] have shown to be perfectly suited: they distribute the configuration load equally among all involved devices, do not rely on specific devices or services to be available and, thus, are generally applicable.

However, many typical application scenarios for Pervasive Computing include additional powerful devices, which may form a stationary infrastructure (e.g., desktop PCs, servers) or a mobile backbone (e.g., laptops with high performance). Due to the significantly increased computation resources of these devices compared to the small mobile devices, the respective scenarios can be characterized as *heterogeneous* with respect to the computational performance of the involved devices. A typical representation for such a heterogeneous scenario is an auditorium environment that features powerful devices like a stationary presentation PC or the laptops of auditors, as well as mobile devices such as smart phones or PDAs. In heterogenenous scenarios, totally decentralized configuration approaches are obviously also usable, but they do not exploit the device heterogeneity efficiently, as they distribute the computation tasks equally among powerful *and* weak devices. This leads to sub-optimal results concerning the arising latencies for configuration and adaptation processes [HHS+07]. Thus, alternative configuration approaches that distribute the configuration load according to the capabilities of the present devices may increase efficiency.

Most of the currently existing research projects solely focus on *one* specific class of scenarios: They rather concentrate on pure Ad Hoc scenarios, like the projects P2PComp [FHMO04] or Mobile Gaia [CAMCM05] do, or they rely on mandatory infrastructure support, such as the projects iRoom [JFW02] or Gaia [RHC+02]. However, none of these projects provides efficient support for *both kinds* of application scenarios and different degrees of heterogeneity. Hence, the currently existing approaches have restricted flexibility in dynamically changing scenarios.

Thus, an advanced approach is required which supports a broad spectrum of environments, ranging from homogeneous peer-based scenarios without any powerful devices, up to strongly heterogeneous infrastructure-based environments with many powerful devices. Moreover, the current environmental conditions have to be monitored and an automatic adaptation of the currently fitting configuration approach is needed. Additionally, it is worthwhile to investigate mechanisms that additionally exploit the results of previous configuration processes, as it is very likely that specific services and components are frequently used in a specific scenario. This is especially the case in infrastructure-based environments where a set of devices – the stationary infrastructure – is supposed to be always available.

1.5. Focus and Contribution

As we want to provide automated system support for a broad spectrum of pervasive environments in this thesis, we only concentrate on systems that supply automated system-level configuration and adaptation: Infrastructure-based *heterogeneous environments*, and infrastructure-less *homogeneous environments* which are also known as *Ad Hoc environments*.

The contribution of this thesis is twofold: Firstly, we comprehensively discuss an efficient and resource-aware hybrid configuration scheme for highly heterogeneous environments. And secondly, we introduce a scheme that automatically caches partial compositions used in previous application configurations for their future re-use to reduce the configuration load, independent from the actually chosen configuration algorithm.

In a first step, we present a new centralized configuration algorithm called Direct Backtracking (DBT). This algorithm is tailored to weakly heterogenenous environments like offices where one additional powerful device, e.g., a desktop PC or a laptop, is available besides several mobile devices with low computation power. Direct Backtracking is employed on this powerful device to configure Pervasive Applications in a completely local manner, i.e., without the need of communication to remote devices during the configuration process. DBT features two new innovative mechanisms in backtracking: *proactive backtracking avoidance*, and *intelligent backtracking*. Proactive backtracking avoidance tries to avoid conflict situations within application configuration right from the beginning and reduces the number of situations in which adaptations are really needed. The intelligent backtracking mechanism handles the remaining conflict situations, i.e., adaptations which cannot be avoided, without producing thrashing effects (repetitive unnecessary reconfigurations). In our evaluation, we show that *Direct Backtracking* shows vastly improved performance compared to other approaches and, therefore, enables fast centralized configurations and adaptations even in very large Pervasive Applications.

Following, we present an approach to allow the automatic adaptation of the degree of decentralization for the application composition algorithms. The presented scheme enables the efficient support of both infrastructure-based and Ad Hoc environments. It represents an important step towards hybrid application configuration by enabling the automatic selection of exchangeable configuration algorithms. To provide particular support of infrastructure-based heterogeneous environments, we introduce a new concept called *Virtual Container (VC)*. This concept enables the local emulation of remote devices and allows proactive and local access to the relevant configuration logic on these devices. This enables centralized application configuration on a distinguished device without the necessity of remote validation of the obtained configuration. In order to identify powerful devices that are suited for performing centralized configuration and act as coordinators for the other devices, we introduce a clustering scheme. Moreover, we provide several strategies to access the required distributed configuration logic for further reduction of the configuration latency, and we present an advanced algorithm which uses the Virtual Containers and the provided access strategies. As the remote information is obtained prior to configurations by this algorithm, the configuration latency as well as the communication overhead during configurations is drastically reduced.

Centralized configuration algorithms like Direct Backtracking represent an efficient solution in environments where one resource-rich device is available. They only require the devices to communicate their current resource situation. However, they introduce a single point of failure and prevent the parallel calculation of configurations like in decentralized approaches. Contrary to this, decentralized approaches increase the robustness of the configuration process, but imply extensive communication between all devices *during* configuration. Moreover, they disseminate the configuration tasks equally among all devices and do not efficiently exploit the increased computation power of specific devices in heterogeneous environments. Thus, we present an efficient and resource-aware *hybrid configuration* approach which combines the beneficial properties of both the centralized and the decentralized approaches and enables a broad range of efficient configuration solutions with different degrees of decentralization in heterogenenous environments with multiple resource-rich devices. This approach represents a generalization of the existing centralized and decentralized approaches. It relies on an advanced clustering scheme and enables the application configuration to be computed on multiple powerful devices simultaneously, which eliminates the single point of failure that is common in centralized approaches. The resource-poor devices stay passive during the hybrid configuration process. Thus, computation bottlenecks within the calculations are avoided, giving our approach an advantage over fully decentralized approaches. Moreover, the advanced clustering mechanism enables a balanced configuration load among the powerful devices and allows the clusters to compute compositions independently from other clusters in the environment. As we provide cluster maintenance methods which react to device changes in the environment, we sustain the balanced configuration load even in dynamically changing environments. Hence, this hybrid approach reduces the configuration latencies by more than 30 % in heterogeneous environments, as our evaluations show.

Besides our efforts to realize a hybrid configuration scheme for a broad spectrum of different scenarios, we propose a novel approach to increase the efficiency of all of the developed configuration approaches that reduces the number of components which actually need to be configured. We refer to this concept as *Partial Application Configuration (PAC)* of Pervasive Applications. This approach is based on the fact that in many typical ubiquitous environments, there is often a fixed set of devices and services that are used by distributed applications. If many identical component sets are frequently used in subsequent configuration calculations, the involved devices undergo a quite similar configuration process whenever an application is launched. However, starting the composition from scratch every time not only consumes a lot of time, but also increases communication overhead and energy consumption of the involved devices. Our approach takes care of such consecutively ongoing configuration processes by caching component sets with high potential for further re-use, distributing them among the configuration devices, and integrating them into future compositions. This minimizes the number of components that actually need to be configured and reduces the latencies for decentralized, hybrid and centralized configuration further. To enable their future re-use, PACs are stored in a cache of limited size after they were used in a configuration. Additionally, we provide a replacement strategy that decides based on the recency and the frequency of their usage which PACs are removed from cache if the cache space is exceeded. Our approach checks, whether at configuration time all components included in a PAC are currently available to make this PAC usable. Then, the cached PAC is automatically integrated into the current composition and only those components which are not included in any of the cached PACs have to be configured in the conventional way. In our evaluations, we show that when PACs are used, the configuration latencies are drastically reduced compared to standard configuration approaches.

1.6. Overview

The remainder of this dissertation thesis is structured as follows: In Chapter 2, we introduce our system model, present an exemplary scenario, and discuss our problem statement and the requirements for this work. In Chapter 3, we give an overview of the projects and algorithms which are most related to this thesis. Subsequently, in Chapter 4, we discuss the way towards a hybrid application configuration scheme for strongly heterogeneous Pervasive Computing environments. Therefore, we present new centralized and hybrid approaches for application configuration and compare the different approaches in real-world evaluations which show that our new schemes significantly outperform the previously used schemes. Chapter 5 discusses the PAC concept to re-use the results of previous configuration processes which is based on caching previously used parts of an application configuration. This concept supports a replacement strategy for outdated cache entries, as well as the support of dynamic scenarios. Next in Chapter 6, we present the overall architecture of the used system software and discuss the extensions we made to implement the developed concepts. Finally, Chapter 7 concludes this thesis, giving at first a summary and then identifying open challenges and enhancements for future research.

2 Requirements and Assumptions

In this chapter, we lay the foundations for this work. Therefore, we first present our system model in Section 2.1, covering devices, environments, applications, and the required system software. Then in Section 2.2, we discuss the problem that is addressed within this thesis: the dynamic configuration of distributed applications in heterogeneous environments. At last, we deduce the requirements we pose on our solutions.

2.1. System Model

As this work especially focuses on the exploitation of the device heterogeneity, we start with a discussion of the devices and the Pervasive Computing environments that are formed by these devices in Section 2.1.1. The applications themselves represent another important aspect for service composition, so with argue the structure and typical sizes of distributed applications in Section 2.1.2. Finally in Section 2.1.3, we take a closer look on the system software we are relying on.

2.1.1. Devices and Environments

We particularly focus on Pervasive Computing scenarios that include devices with different properties and computation power. Each device has a unique device ID[1]. Furthermore, we suppose that all devices, which host components of a distributed application, are in direct communication range, i.e., we rely on one-hop communication environments. Conference rooms, offices, auditoriums, or living rooms represent typical exemplary application scenarios. Moreover, we assume that all devices act in a *cooperative* way, i.e., they are trustworthy and provide their computation power

[1]For this thesis, we are relying on randomly chosen 32 bit identifiers. Thus, the probability that two arbitrary devices get the same ID is neglectable with only around 0.00000002 %

and resources to the other nodes in order to enable a distributed application execution with a high quality of service. In terms of scenario heterogeneity, we distinguish between two different types of involved devices:

- *Resource-poor devices* are usually represented by mobile devices, such as Smart Phones or PDAs. Due to their limited computation power, they can slow down the complete configuration if too much workload is put on them. Resource-poor devices, which we will also call *weak* devices within this work, typically have a low degree of availability, as they are highly mobile and their battery power is strictly limited. Thus, resource-poor devices should preferably not be burdened with computationally intensive tasks. We call environments with only resource-poor devices *homogeneous*.
- *Resource-rich devices* are usually stationary infrastructure devices such as desktop PCs, but may also be mobile devices with high computation power, such as laptops. Due to their increased computation resources, resource-rich devices are perfectly suited for performing computationally 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, reaching from environments where only one resource-rich device is available (e.g., an office scenario with one desktop PC) up to scenarios with various resource-rich devices (e.g., an auditorium during a conference). In this thesis, resource-rich devices are also called *powerful* devices.

Obviously, relying on only two different types of devices concerning the computation power represents a rather coarse-grained approach. However, it represents a simple yet powerful way to enable a resource-aware distribution of the configuration load among the present devices, as we will show in Chapter 4. We will go into details concerning the determination of the device roles in Section 4.1.2, where we describe the cluster-based approach that we have chosen.

2.1.2. Application Model

As mentioned in the introduction, one main characteristic of a pervasive application is the fact that the required functionality is *distributed* among multiple devices which have to collaborate. As a typical example for Pervasive Computing, Figure 2.1a shows a scenario of a businessman who controls distributed applications only using his Smart Phone, thereby travelling from his home by car to his office. As you can see, the available set of components that may be used by applications varies widely, reaching from home entertainment devices and white goods to car entertainment and outdoor navigation up to office components like video projectors, headsets or tablet PCs.

We assume a component-based application model, i.e., an application consists of several *components* and each component instance requires 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 a root instance, the so-called application *anchor*. Figure 2.1b shows the component tree of

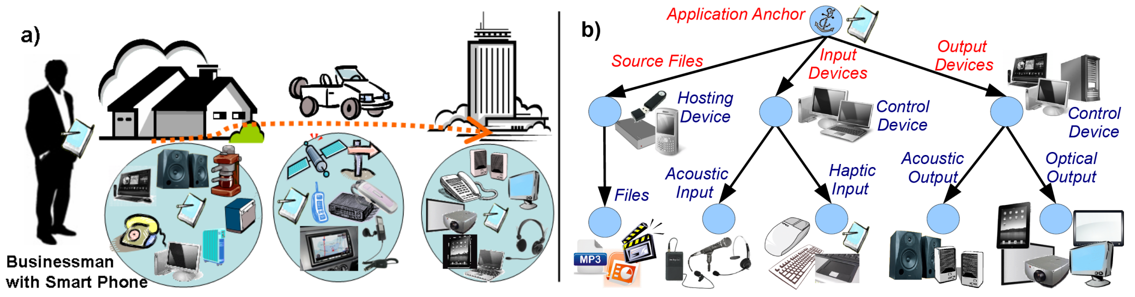


Figure 2.1.: a) Travelling businessman scenario, b) Distributed Presentation Application in businessman scenario

an exemplary distributed presentation application in an office environment, where the businessman may present a new product to business partners. The application automatically uses the resources available in the vicinity as the required input and output functionalities, e.g. display, microphones, or speakers.

A single component is resident on a specific device which is represented by a *container* that carries a unique identifier, called *device ID*. The number of components per container is not restricted. Interdependencies between components as well as resource requirements are described by directed *contracts* which specify the functionality required by the parent instance and provided by the child instance of this contract. A parent instance may have an arbitrary number of child instances and, thus, contracts. Every instance can uniquely be identified within the application tree structure by its *Instance Identifier (IID)*. An IID is built up in a top-to-down manner following the levels of the application tree up to the respective instance. This enables the identification of all ancestors of an instance within the application tree. The root instance of the application is called *anchor* and has IID [0]. For every level down the tree, the instance ID is determined by taking the direct ancenstor instance's IID and attaching the ordinal number of the instance within all of its siblings, ordered from left to right. As an example, consider the concrete application structure with IIDs of the distributed presentation application introduced in Figure 2.1b, which is displayed in Figure 2.2. Its application anchor (IID [0]) requires the provision of three child contracts (ctc. 0 to 2) to provide instances for the source file's input device (IID [0][0]), the acoustic and haptic input device (IID [0][1]), and the acoustic and optical output device (IID [0][2]). Furthermore, each of these three contracts relies on additional child contracts, e.g., the acoustic and haptic input device instance needs one acoustic input instance (IID [0][1][0]) and one haptic input instance (IID [0][1][1]). The structure represented here is given by the application developer and is independent from the provision of specific components, but solely forces the functionality that has to be provided by a component.

If there exists more than just one component that provides the required functionality for a contract, the parent component can choose among several *options*. In the following, this is called a *multi-optional contract*. Obviously, the number of multi-optional contracts is relying on the currently available resources and may change over time in dynamic environments. As long as the anchor component of an application is executed, its container ensures that dependencies are recursively

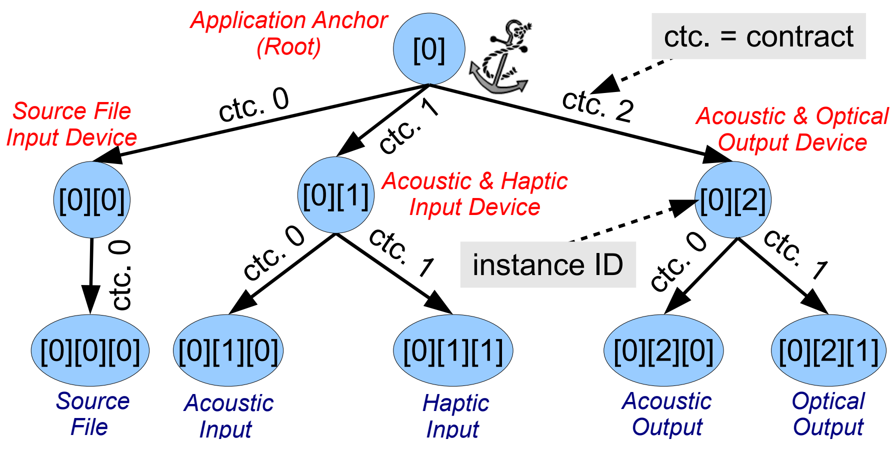


Figure 2.2.: Exemplary application structure with instance IDs (IIDs)

resolved by binding adequate components to them. Since a parent component relies on its child components, it can only be instantiated if all of its children have been instantiated previously. An application is successfully started if all dependencies have been resolved so that for each contract, a suitable component which satisfies all requirements could be found.

In a specific environment, several *components* are available which represent the resources that can potentially fulfill the requirements given by the contracts and, thus, become instances of the application. Each component is resident on a specific device which is represented by a *container*.

A component which fulfills the requirements of a specific contract and is used within a configuration can uniquely be identified within the application tree by its *component identifier (CoID)*. Like for the instance IDs, the CoID of a component is built up in a top-down manner following the levels of the application tree unto the respective component. However, as it is possible that there exist multiple components which can fulfill the requirements of a specific contract, the CoID additionally includes a parameter representing the *option* for this component among all components that provide the required functionality of this contract. This enables the identification of all predecessors of a component. The root component of the application is called anchor and has the component ID [0,0]. For every level down the tree, the component ID is determined by taking the parent component's ID and attaching a tuple $[a, b]$, where a represents the ordinal number of the *dependency* of the parent component that is resolved by this component, and b represents the *option* (meaning the index number of the alternative components) for that dependency. If there is only one possible component for a dependency, b becomes 0. Otherwise, the different alternatives are labelled with increasing option numbers. To clarify this, regard Figure 2.3 which shows the previously introduced distributed presentation application with known application structure, but extended by the available components. The contract which covers the acoustic input devices, for instance, is multi-optional, as there are two different options that fulfil the application's requirements: using a microphone (CoID [0,0][1,0][0,0], i.e. option 0 for that dependency) or, alternatively, using a headset (CoID [0,0][1,0][0,1], i.e. option 1) that is also available in the

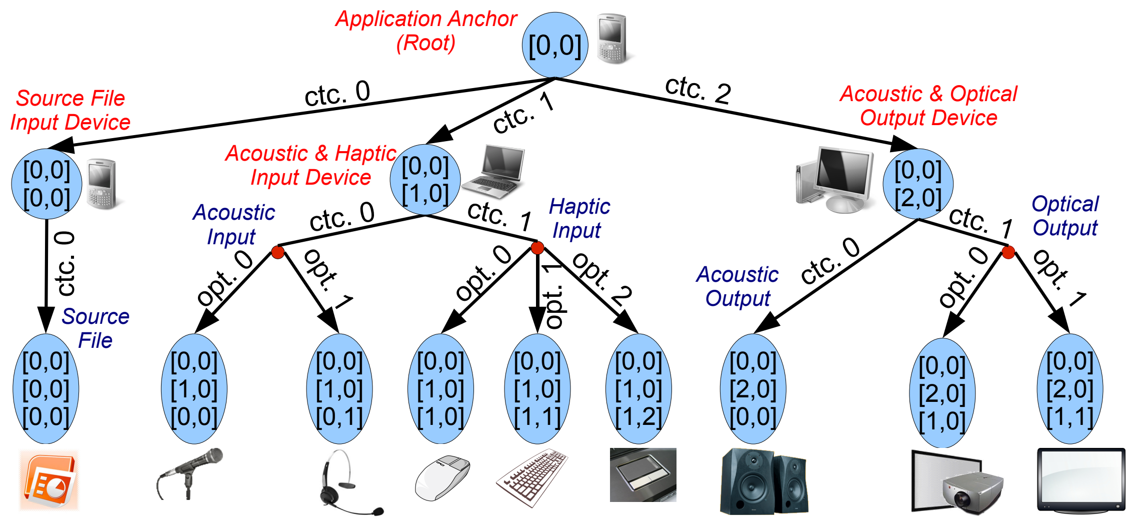


Figure 2.3.: Extended application structure with component IDs (CoIDs) of components which have been selected within a configuration process

environment. The application developer may define priorities between the different options relevant for the configuration algorithm, i.e. using a video projector leads to a higher quality for the application execution than using the flat screen. Further multi-optional contracts cover the instance for the haptic input with optional usage of a mouse (CoID [0,0]1,0][1,0]), a keyboard (CoID [0,0][1,0][1,1]), or a touch pad (CoID [0,0][1,0][1,2]) as well as the optical output, where the configuration algorithm can choose between a video projector (CoID [0,0][2,0][1,0]) or a flat screen TV (CoID [0,0][2,0][1,1]).

To get a feeling about typical application sizes in the mentioned scenarios, we investigated the applications used in related research projects. Typical applications and their sizes are shown in Table 2.1, divided into applications for Ad Hoc scenarios, and applications for infrastructure-based scenarios. One can see that applications for Ad Hoc scenarios typically involve mobile users (e.g., in navigation or tracking scenarios), while infrastructure-based scenarios are usually indoor environments with a lower degree of dynamics, such as medical support, health care, or residents' support by providing smart homes. The applications listed here cover 8 up to 34 components. It can also be seen that applications in Ad Hoc scenarios are typically smaller than those created for infrastructure-supported environments. We rely on applications in the same order of magnitude in the evaluations within this work.

2.1.3. System Software

For the work presented here, we rely on a *communication middleware* which provides basic services to enable distributed applications and supports different communication models. Moreover, the communication middleware supplies a registry which contains all currently reachable devices. Through this registry, global knowledge among all devices is established. Usually in mobile networks, global knowledge is critical, as it may cause inconsistencies among the different devices, e.g. because

Reference	Application Scenario	Typical Size(s)
[CAMCM05]	Indoor/Outdoor Navigation	8 components
[Gri04]	Travelling consultant scenario	10 components
[ENS+02]	Resource sharing among users	10 components
[OGT+99]	Tracking application	8 components
[OGT+99]	Cargo routing logistics scenario	12 components
[Sai03]	Typical O2S system application	13 components
[BHSR04]	Evaluation of distrib. application	8/15/17 comp.s
[HUB+06]	Pervasive Presenter	8 components
Average size for ad hoc scenarios		**10.1 comp.s**
[JFW02]	Smart Room	20 components
[CCS08]	Wireless Medical Info System	20 components
[LNH03]	Smart Flat for elderly people	30 components
[Tan01]	Smart Board & connected display	12 components
[Tan01]	Smart Board & intelligent chair	19 components
[Tan01]	Multi-user Smart Board	34 components
[HMEZ+05]	Smart House with 17 "hot spots"	30 components
Average size for infrastructure scenarios		**23.5 comp.s**

Table 2.1.: Typical application sizes in ad hoc environments and infrastructure-based scenarios

of network failures or very high dynamics. However, as mentioned in Section 2.1.1, we rely on single-hop environments. Furthermore, these environments are characterized by a relatively low degree of dynamics, as devices typically stay within an environment for at least several minutes, as we will show in Section 5.7.2. Thus, we consider the access on global knowledge to be uncritical here.

An exemplary system for Pervasive Computing environments which supports the requirements discussed here is BASE [BSGR03]. This communication middleware has been developed in the Peer-to-Peer Pervasive Computing (3PC, [HSM+12]) project. BASE provides basic services to enable distributed applications and features a plug-in architecture to support different communication models such as RPC via exchangeable plug-ins. Beyond this, BASE supplies a device registry (as mentioned above), maintaining a list of all currently reachable devices. The device registry is kept up to date as every device periodically broadcasts heartbeat messages.

Beyond the mentioned communication middleware, we rely on a *component system* that is executed on top of the communication middleware and provides system support at the application level. 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. Moreover, the system needs to be capable to supply algorithms which enable the automatic configuration and adaptation of distributed applications based on the information provided by the communication middleware and the component system itself.

PCOM [BHSR04] represents an exemplary system that fulfills the requirements posed here. This component system initially was developed as a middleware for self-organizing software systems in Mobile Ad Hoc Networks (MANETs) without supporting infrastructure devices. However, the PCOM middleware provides automatic adaptation on system level. PCOM supports a wide range of end-user devices and can integrate additional infrastructure devices, but it does not take special care of their computation power and, hence, does not exploit available resources efficiently in many scenarios. Within the system, devices are represented by *containers*: They host components and manage their dependencies, so they act as a distributed application execution environment. Containers re-use the discovery and communication capabilities of the communication middleware. More details about the specific algorithms and mechanisms as well as the system architecture can be found in Section 3.2.2.

2.2. Problem Statement

In the following, we first discuss the dynamic configuration (Section 2.2.1) of distributed Pervasive Applications in heterogeneous environments, which represents the main challenge we focus on in this work. Based on this discussion, we derive the requirements for an adequate solution in Section 2.2.2.

2.2.1. Dynamic 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*, describing the functionality that is required by the parent and provided by the children of a specific component for resolving a dependency (e.g., the access to a remote database), and *resource constraints* due to limited resources (e.g., a single display cannot be used by two applications simultaneously). The complexity of finding a configuration arises from the fact that both types of constraints must be fulfilled coevally. An application is successfully started if all dependencies have been resolved by a *configuration algorithm* such that for each contract, a suitable component which satisfies all requirements was found.

The problem of configuring an application in a distributed manner can be represented as an NP-complete Distributed Constraint Satisfaction Problem (DCSP) [HBR05]. Generally, a DCSP is a mathematical 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 [YDIK92] represent typical solutions to this problem. More details on Distributed Constraint Satisfaction will follow in Sections 3.1 and 4.1.1.

All of the algorithms considered within this work follow a depth-first search approach. This means that an algorithm proceeds from the top to the bottom of the tree and, within a sublevel of the tree, from left to right. Whenever an algorithm has found a suitable component for a contract dependency, this component is added to the so-called *assembly*, which represents the configuration that has been found so far.

Dynamic 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 configurations have to be *adapted* at runtime. Then, the corresponding contracts that are conflicted have to be identified by the configuration algorithm, and alternative components have to be found which can provide the same functionality. Therefore, a currently instantiated component of another contract has to be stopped to free resources, and an alternative component that fulfills this contract's requirements with less resource consumption has to be instantiated afterwards. If it is not considered whether the adapted contract requires the same type of resource as the conflicted contract, it is possible that many adaptations are needless since they do not solve the problem. Then, the number of necessary adaptations increases and leads to an additional configuration latency. This undesired effect is called *thrashing* and has to be avoided by providing efficient configuration algorithms.

While the *configuration latency* comprises the time between the user's application start and the availability of the application to the user, the *adaptation latency* covers the time span between the unavailability of specific components until the re-execution of the application after alternative components have been found. Configuration and adaptation latencies include the delays for (re-)calculating the configuration and instantiating all application components. Both latencies should be minimized to provide a seamless user experience even in dynamically changing scenarios.

The configuration and adaptation problems have already been solved for homogeneous scenarios by providing decentralized algorithms for peer-based application composition (e.g., [HBR05, HHS$^+$07]). However, decentralized schemes perform suboptimal in *heterogeneous* environments, since they do not exploit the scenario heterogeneity by distributing the configuration tasks in a resource-aware manner among the currently available devices. To provide such efficient heterogeneity support, issues like distinguishing resource-rich from resource-poor devices, distributing the load in a unique manner among several devices, or the provision of configuration schemes tailored to specific scenarios and the automatic selection of a fitting scheme have to be regarded. In this article, we present solutions to these issues.

2.2.2. Non-functional Requirements

Besides the functional requirement of providing a valid composition in dynamic environments, there are several non-functional requirements that an adequate solution has to fulfill. According to the challenges discussed above, we pose the following non-functional requirements to configuration and adaptation processes:

- **Adaptivity:** Pervasive Computing scenarios are characterized by a high degree of dynamics, as they involve mobile devices with limited battery capacities. Thus, the execution environment of an application may change dynamically. As different environments promote different configuration approaches, the provision of approaches optimized for specific scenarios is mandatory. Moreover, the most suitable approach has to be selected automatically. This means that the chosen configuration algorithm needs to be *automatically adapted* in dynamically changing scenarios.

- **Automation:** Many related projects demand users or application programmers to handle configuration and adaptation issues. However, providing an *automated* solution where only the system software is responsible for determining valid configurations yields Pervasive Computing systems that are much more transparent to users and application developers.

- **Efficiency:** Configuration and adaptation calculations induce latencies which users perceive as undesired delays, as the application is not available before these processes are completed. To sustain the user's interest for distributed applications, these distractions should be as low as possible. Thus, a major goal is to achieve *efficiency* by minimizing the arising configuration latencies.

- **Resource-Awareness:** In heterogeneous environments, the computation resources on the available devices differ significantly. To exploit this heterogeneity effectively and avoid bottlenecks for the configuration, solutions have to be *aware of the computation resources* of the involved devices. Crucial tasks within a configuration should be performed by the resource-richest devices, as they can perform these tasks much faster than slow devices.

- **Resource Conservation:** Many scenarios involve a fixed set of applications and devices which are frequently used. As the involved devices undergo a similar configuration process whenever an application is launched, starting a composition from scratch every time unnecessarily wastes computation resources on the configuration devices. Instead, the *results of previous configuration processes* have to be analyzed, cached, and automatically integrated into future configurations to reduce the complexity of the configuration problem in terms of the number of components that have to be configured.

3 Related Work

In this chapter, we give an overview of those projects and research efforts dealing with issues which are also highlighted in this thesis. Initially, we describe algorithms that solve Constraint Satisfaction Problems in Section 3.1. Following in Section 3.2, we present related research projects that also focus on service composition in Pervasive Computing. Here, we distinguish between projects which rely on heterogeneous infrastructure-based environments (Section 3.2.1), and projects which concentrate on homogeneous mobile Ad Hoc scenarios (Section 3.2.2). As clustering a group of nodes is used in this thesis to separate different classes of devices, we give a summary of related work on clustering frameworks in Section 3.3. Finally in Section 3.4 of this chapter, we discuss projects that try to exploit the results gained from previous configuration runs, which is also one of the main issues here.

3.1. Algorithms for Solving Constraint Satisfaction Problems

As presented in Section 2.2.1, the adaptation of tree-based applications where the functionality is distributed among the devices in the environment can be mapped to a Constraint Satisfaction Problem (CSP, [YDIK92]). CSPs appear in many areas such as artificial intelligence, operational research or hardware design. They can be solved either in a *centralized* and synchronous way where one specific device locally calculates the complete configuration, or in a *distributed* and asynchronous way where all present devices are included in the calculations. Furthermore, there can be distinguished between two different methods to solve CSPs: *search algorithms*, and *consistency algorithms* [ZM91]. Moreover, search algorithms can further be divided into *backtracking algorithms* and *iterative improvement algorithms*. This leads to six different classes of algorithms to solve CSPs, as shown in Figure 3.1.

Consistency algorithms are pre-processing procedures that are invoked *before* the actual search. They aim at reducing futile backtracking. Path consistency [Mon74],

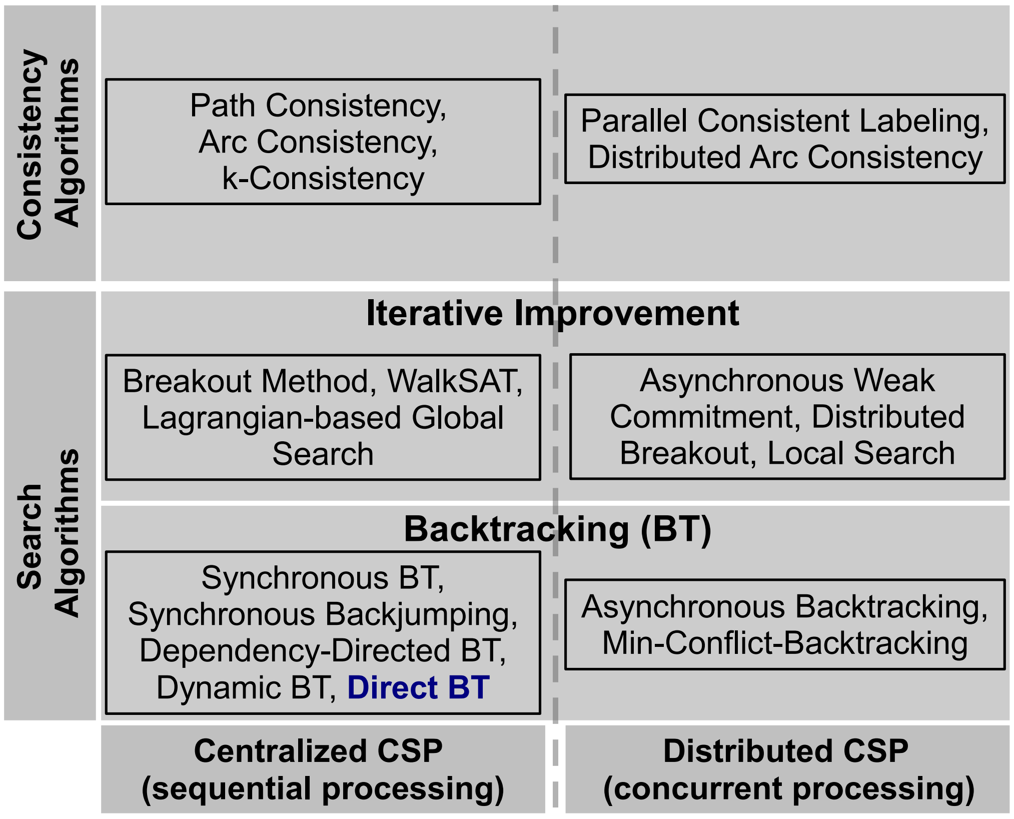

Consistency Algorithms
Path Consistency, Arc Consistency, k-Consistency
Parallel Consistent Labeling, Distributed Arc Consistency
Search Algorithms
Iterative Improvement
Breakout Method, WalkSAT, Lagrangian-based Global Search
Asynchronous Weak Commitment, Distributed Breakout, Local Search
Backtracking (BT)
Synchronous BT, Synchronous Backjumping, Dependency-Directed BT, Dynamic BT, Direct BT
Asynchronous Backtracking, Min-Conflict-Backtracking
Centralized CSP (sequential processing)
Distributed CSP (concurrent processing)

Figure 3.1.: Classification of algorithms to solve CSPs

arc consistency [Mac77, MH86, Bes94], k-consistency [Fre78] as well as parallel consistent labeling [SH87] or distributed arc consistency [ND98, Ham02] are popular efficient consistency algorithms. Because of their pre-processing nature, consistency algorithms are not applicable in dynamic pervasive environments where the availability of specific components may change at any time and, hence, *runtime* configuration is needed. So, we put the focus on search algorithms in the following.

In case of a fully distributed application configuration, the CSP problem can be solved by distributed algorithms from the research domain of Artificial Intelligence [YDIK98]. When using distributed algorithms for solving CSPs, all of the involved *agents* perform their search procedures concurrently while communicating information on their search processes with each other. Exemplary distributed algorithms from the class of backtracking schemes are min-conflict backtracking [MPJL92] or asynchronous backtracking [YDIK98]. Moreover, there exist solutions which are based on *iterative improvement algorithms* [HY05]. Since these algorithms are hill-climbing search algorithms, they are occasionally trapped in local-minima. Local-minima represent states violating some constraints, but the number of constraint violations cannot be decreased by changing any single variable value. To escape from local minima, these algorithms typically provide heuristics like the min-conflict heuristic [MPJL92] which changes a variable value so that the number of violated constraints is minimized. Thus, a mistake can be revised without conducting an exhaustive search, that is, the same variable can be revised again and again.

Therefore, these algorithms may be efficient, but their completeness cannot be guaranteed. Typically used distributed improvement algorithms are *asynchronous weak-commitment search* [Yok94], *distributed breakout* [YH96], or *local search* [HY02].

Decentralized algorithms have a common drawback: they cause huge communication overhead for resolving dependencies between components. In case of n devices that are involved in the application configuration, the worst-case amount of messages to be sent is $\mathrm{O}(n^2)$. In centralized algorithms, every involved device has to send only one message to the configuration device to inform this device about its available resources, which leads to a worst-case message amount of only $\mathrm{O}(n)$. Furthermore, distributed algorithms do not take special care of potential strong devices that are present in heterogeneous environments.

Thus, we concentrate on centralized algorithms from now on. Synchronous approaches for centralized CSPs that belong to the class of iterative schemes are the *Breakout* method [Mor93], *WalkSAT* [SKC94], or *Lagrangian-based global search* [SW98]. However, they suffer from the same local minima problem than their distributed versions do.

Backtracking algorithms are complete algorithms that do not get stuck in local minima, so we only focus on them in this thesis. A survey of these algorithms is given by Andrew Baker [Bak05]. The simplest approach imaginable is to generate every possible composition until one is found that satisfies all of the constraints given by the application. This algorithm performs an *exhaustive search* and is obviously very inefficient, as it does not check any of the constraints until it arrives at the bottom of the application tree.

An improved centralized backtracking algorithm which is not relying on exhaustive search is *Synchronous Backtracking (SBT)* [BM04]. SBT executes a depth-first search in the application tree. The only difference between Synchronous Backtracking and the exhaustive search algorithm is in the location of the consistency check: After Synchronous Backtracking assigns a value to a variable, it checks at once whether the partial assignment still satisfies the constraints. If it does not, the procedure immediately moves on to the next value for this variable. Because of the consistency check of these partial assignments, SBT significantly reduces the number of possible configurations at an early stage. However, SBT has one huge drawback concerning adaptations: Since it does not consider the reason for backtracking and tries to adapt the first possible multi-optional contract (cf. Section 2.1.2), it suffers from thrashing. Thrashing represents the problem of wasting time while exploring portions of the search space that cannot possibly contain any solutions because the respective contracts address different functionalities. Thrashing leads to an enormous overhead and, thus, increased latency, especially if the application is huge and contains many multi-optional contracts.

Several approaches to reduce this thrashing effect have been developed. Since SBT lacks mechanisms to recognize the reasons for a failure, an advanced algorithm called *Synchronous Backjumping (SBJ)* [Gas77] tries to keep track of the reasons that led to a backtrack operation. In case of a necessary backtracking process, i.e., when a contract could not be instantiated by any component, SBJ at first computes the so-called *conflict set* that represents the subset of all contracts which are currently resolved by a component that is also needed to fulfill the conflicting contract. This means SBJ searches for multi-optional contracts that depend on the same kind of resource as the contract that was the reason for backtracking. SBJ does not adapt contracts that are independent of the backtracking cause. This helps to reduce thrashing, but it cannot avoid it completely since SBJ does *not* keep previous intermediate results for subsequent adaptations. Furthermore, SBJ relies on a *stack* during backtracking, causing additional computation and storage overhead.

Dependency-Directed Backtracking (DDB) [SS77] solves the problem of thrashing by storing a set of so-called *nogoods* which are partial configurations without a solution for the complete application. Every time a backtrack is about to occur, DDB learns a new nogood from the current conflict set and the current partial assignment. This nogood states that it is not possible for every dependency in the conflict set to simultaneously have their current assignments. The set of nogoods is used when the current partial assignment is checked both against the original constraints and the nogoods that have been learned so far. Therefore, it can avoid infeasible solutions subsequently. While DDB avoids thrashing, its main drawback is its enormous memory consumption: Every time the algorithm has to perform a backtrack, it learns a new nogood and adds it to the nogoods set. Because of this ever-increasing set of nogoods on the stack, the space complexity of DDB is comparable to its time complexity. Since the Constraint Satisfaction Problem is NP-hard, both complexities may be exponential. Such backtracking procedures which use exponential space are often too expensive for practical use [Bak05]. One possibility to reduce this extraordinary memory waste is to retain only those nogoods up to a certain size k. This approach is called k-order learning [Dec90] and uses only polynomial space, but may introduce thrashing again because of the limited set of nogoods that can be stored.

Dynamic Backtracking (DyBT) by Ginsberg [Gin93] is an algorithm which outperforms all of the previously discussed approaches since it removes thrashing completely without excessive waste of memory. Similar to SBJ and DDB, this algorithm immediately moves to a point which conflicts with the latest assignment in case of a conflict. DyBT neither forgets intermediate values, nor is it memory-intensive since it does not rely on a stack. It is an iterative algorithm that stores a set of so-called *culprits* which represent forbidden assignments. DyBT does not only retain the chosen value for a contract C, but also the culprit set of C. If the instantiation of a component for a contract fails, the algorithm can easily decide which of the formerly configured contracts conflict with this contract, and directly jump back to them. Proceeding like this, DyBT achieves to produce only polynomial space overhead [Bak05]. However, DyBT changes the order of contracts to resolve conflicts. This is not an option in our problem of tree-based application configuration since parent-child relationships and differences in resource consumption are encoded in the order of contracts. This order needs to be preserved to ensure useful configurations. So, DyBT cannot be used for the configuration of tree-based Pervasive Applications.

Our algorithm Direct Backtracking, as discussed in Section 4.2, proceeds similar to DyBT in general, but in addition, it also adapts the adherent subtree of a component during an adaptation process. Thus, Direct Backtracking does not need to perform any changes in the order of components. Moreover, DBT provides two advanced mechanisms to reduce the number of adaptations and render the remaining adaptations more efficiently.

3.2. Overview of Service Composition Frameworks

Projects for composition of services and applications in Pervasive Computing can be classified according to Christian Becker [Bec04] along the two axes *scenario support* – either with or without infrastructure support – and *level of composition* – either automatically by the system or manually by the application programmer or application user. This leads to the four classes shown in Figure 3.2. As also shown in the figure, various projects have been presented for each of these classes. The figure shows that the scope of this thesis covers automated composition both in infrastructure-less and infrastructure-based scenarios, i.e. *two* of the introduced system classes, giving our system an advance over most of the related projects.

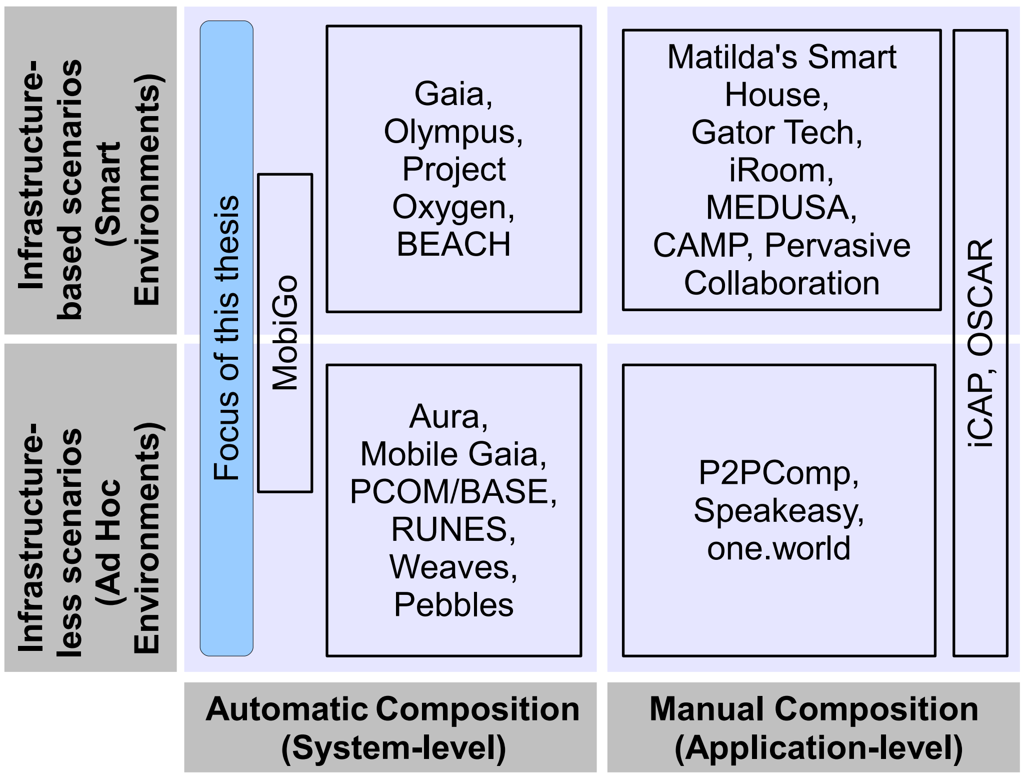

Infrastructure-based scenarios (Smart Environments)
Infrastructure-less scenarios (Ad Hoc Environments)
Focus of this thesis
MobiGo
Gaia, Olympus, Project Oxygen, BEACH
Matilda's Smart House, Gator Tech, iRoom, MEDUSA, CAMP, Pervasive Collaboration
iCAP, OSCAR
Aura, Mobile Gaia, PCOM/BASE, RUNES, Weaves, Pebbles
P2PComp, Speakeasy, one.world
Automatic Composition (System-level)
Manual Composition (Application-level)

Figure 3.2.: Classification of related projects in service and application composition

In the following, we first discuss the most related systems for infrastructure-based environments, beginning with the projects Gaia, Olympus and Oxygen that provide automated composition, and then switching to systems that provide manual composition, namely Matilda's Smart House, Gator Tech and iRoom.

Then, we switch to systems for infrastructure-less Ad Hoc scenarios, again beginning with systems that perform automated composition, which are Aura, Mobile

Gaia, and the previous version of the systems PCOM and BASE at time the works for this thesis started. Finally, we discuss the systems P2PComp, Speakeasy and one.world that perform manual composition.

Further pervasive service composition projects which are included in the classification, but are not described in detail here are the following:

- The *BEACH* [Tan01] project provides a middleware system that supports automated composition in infrastructure-based scenarios, without supporting Ad Hoc scenarios.
- The projects *RUNES* [CCG+07], *Weaves* [OGT+99] and *Pebbles* [Sai03] also supply automated composition, but they rely on infrastructure-less Ad Hoc scenarios and lack efficiency in heterogeneous scenarios.
- *MobiGo* [SR07] supports both infrastructure-less and infrastructure-based scenarios, but puts its focus rather on service migration and virtualization. Furthermore, it defines only a fixed set of environments called "spaces", while we dynamically adapt to environments with various degrees of heterogeneity.
- *MEDUSA* [DGIR11] and *Pervasive Collaboration* [PWR+09] rely on end-user composition in infrastructure-based scenarios, thus not providing the aspired automatic composition.
- *CAMP* [THA04], *iCAP* [DSSK06] and *OSCAR* [NES08] even rely on end-user *programming*, which contradicts the paradigm of pervasiveness. While iCAP and OSCAR do not pose any requirements on an existing infrastructure and support both infrastructure-based and infrastructure-less scenarios, CAMP relies on infrastructure-based Smart Home environments.

We refer the interested reader to the respective references for further details concerning these systems.

3.2.1. Service Composition in Infrastructure-Based Environments

Gaia [RHC+02] is a project which provides a Corba [Vin97]-based middleware for resource-rich environments, *Gaia OS*. It supports developers by providing services for the development of user-centric, resource-aware, multi-device and context-sensitive mobile distributed applications. Gaia represents a highly integrated environment that supplies runtime adaptation and supports various kinds of devices, such as audio devices, video cameras, or even fingerprint sensors. The system is transparent to the user, but yet regards security aspects. Gaia distinguishes between two kinds of spaces: *physical spaces* and *active spaces*. A physical space (Figure 3.3a) is a geographic region with limited and well defined physical boundaries, containing physical objects, heterogeneous networked devices, and users performing a range of activities. Contrary to this, an active space (Figure 3.3b) is a physical space coordinated by a responsive context-based software infrastructure that enhances the ability of mobile users to interact and configure their physical and

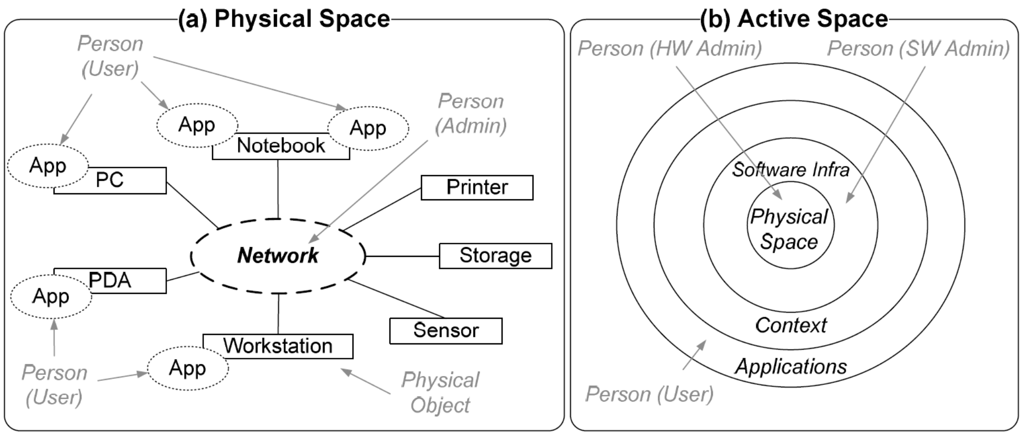


Figure 3.3.: a) Physical spaces and b) active spaces in Gaia [RHC+02] and Olympus [RCAM+05]

digital environment seamlessly. A basic requirement of active spaces is to support the development and execution of user-centric mobile applications.

For environments with a higher degree of dynamics, the developers of Gaia presented a new high-level programming model named **Olympus** [RCAM+05]. Olympus uses semantic descriptions and ontological hierarchies to automate the mapping process and specify active spaces at an abstract, high level. Therefore, Olympus comes with an associated framework which takes care of resolving virtual entities into actual active space entities. This resolution is based on constraints that are specified by the developer, the resources available in the current space, space-level policies and the space's current context. Furthermore, Olympus' framework uses a utility model to find the most suitable entity that is available in a space to perform a specific task.

While Gaia and Olympus support stationary as well as mobile devices, they are however not applicable in infrastructure-less Ad Hoc environments, as they rely on specific system infrastructure that is strictly required.

The **Oxygen** project at the Massachusetts Institute of Technology (MIT) proposes a programming paradigm called *goal-oriented programming* [Sai03]. In Oxygen, goals represent user requirements that have to be met by the system. However, contrary to procedure calls, goals do not provide an implementation, but rather may have various alternative implementations that are selectable at runtime, called *techniques*. Goals are satisfied by dynamically assembling a set of generic components, so-called *pebbles*, to implement the high-level function that is encoded by the goal. Pebbles represent lightweight, policy-neutral distributed components, conforming to a standardized API whose abstraction layers – the *planning layer* covering the goals, the *abstraction layer* which represents the composites, and the *component layer* with the pebbles – are shown in Figure 3.4. Every pebble typically implements a single function. Pebbles are designed to be standalone components with well-defined, explicit ports of communication with other components. By focussing on a single operation, pebbles are easily reusable in many different applications.

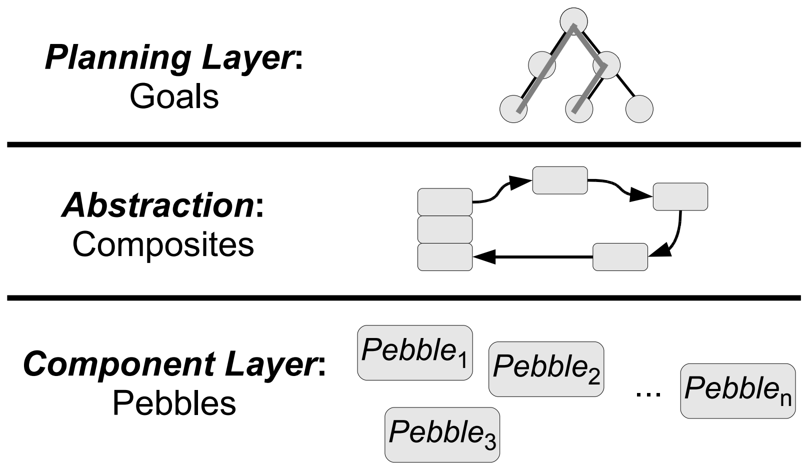


Figure 3.4.: O2S [PPS+08] abstraction layers with Pebbles [Sai03] component API

As a part of the Oxygen project, the system software *O2S* [PPS+08] has been developed for highly dynamic environments. O2S represents an indirect specification via the above mentioned goals to refrain from specifying a single configuration. It provides an extensible mechanism to manage users' system runtime decisions and scan the vicinity for techniques that satisfy the user's goals. O2S enables a model in which an application programmer specifies the behavior of an adaptive application as a set of open-ended decision points. The respective system allows hierarchical decomposition of applications through technique-based sub-goals and provides a framework for declaring dependencies between sub-goals to programmers. By technique-based goal resolution, O2S performs a hierarchical decomposition of goals down to the technique-level. Same as the projects mentioned before, the Oxygen system requires infrastructure support.

Another project which depends on infrastructure devices is **Matilda's Smart House** [LNH03] which focuses on Pervasive Healthcare support. This multidisciplinary project at the University of Florida explores the use of emerging Smart Phones and other wireless technologies to create a smarter environment that allows elder people with disabilities to monitor, interact with, and control their surroundings. The system creates a lucent environment for users via mobile sensors and end-user devices. Matilda represents a research robot with an onboard computer and a vest fitted with location sensors used for indoor location tracking research. Matilda's Smart House uses the Open Services Gateway initiative (OSGi) framework [MK01] as an extensible software infrastructure that can adapt to environmental changes such as introducing and integrating new devices and services. This project also supports remote monitoring and administration by family members and caregivers.

The OSGi Service Platform is an execution environment for remotely deployed services that provides added services to the Gateway Operator which combines and verifies services received from service providers, and delivers them to the customer. OSGi provides platform independence, application independence, multiple service support, security, simplicity and multiple network technology support. OSGi's service platform provides abstractions allowing a multitude of different communication technologies. Figure 3.5 illustrates the Service Platform framework. The *Execution Environment* is the Java runtime environment, building on the *Operating System* and the *Hardware*. The *Modules* layer represents a class loading model that is based on Java, but with added modularization. The *Life Cycle* layer allows bundles to be dynamically installed, started, stopped, updated, and uninstalled. The *Service Registry* layer controls the services and allows for a cooperation model between bundles. The *Services* layer maintains services grouped by bundles, and provides functionality required by the user. Finally, the *Security* layer provides basic application security and spans across all upper layers.

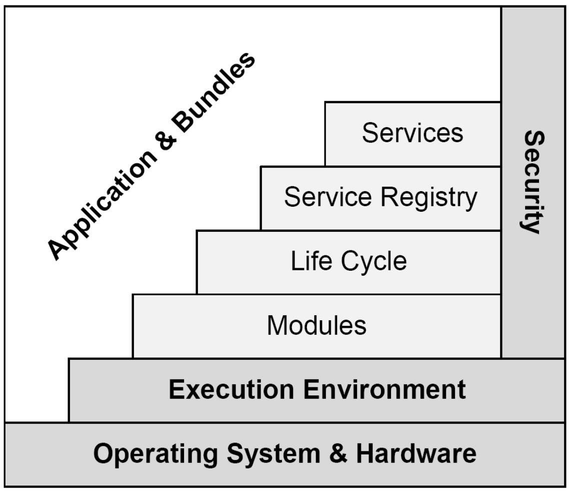


Figure 3.5.: The OSGi [MK01] Service platform framework, used within Matilda's Smart House [LNH03] and Gator Tech [HMEZ+05]

After creating Matilda's Smart House, researchers from the University of Florida realized that the outcome of this project resulted in some impressive demonstrations, but not something people could actually live in. Thus, they designed the second-generation **Gator Tech Smart House** [HMEZ+05] to outlive existing technologies and be open for new applications that researchers might develop in the future. The Gator Tech Smart House represents a new programmable space that consists of 17 ”hot spots” that can sense specific contexts and the house's residents within an environment. The house's ”hot spots” enable mappings between the physical world and remote monitoring and intervention services.

The **iRoom** project [JFW02] aims at an increased user experience of distributed visualization, as it focuses on integrating high-resolution displays into application configuration. iRoom originates from the *Interactive Workspaces* project at Stanford University. The main component of this project is a software infrastructure called *interactive Room Operating System (iROS)* that provides services for the implementation of distributed applications by the use of infrastructure devices. iRoom supports the automatic adaptation of graphical user interfaces depending on the available devices, e.g., PDAs or laptops.

Figure 3.6 shows iROS' component structure. iROS has three main subsystems that are designed to address the three user modalities of moving data, moving control, and dynamic application coordination: The *Event Heap* stores and forwards

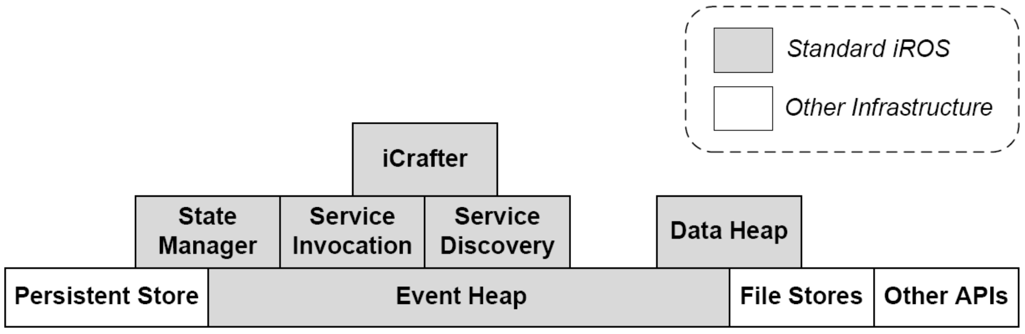


Figure 3.6.: iROS [JFW02] component structure

messages called events. Each event is a collection of *name-type-value* fields. It provides a central repository to which all applications in an interactive workspace can post events. The *Data Heap* facilitates data movement by allowing any application to place data into a store associated with the local environment. The data is stored with an arbitrary number of attributes that characterize it. The *iCrafter* system provides a system for service advertisement and invocation, along with a user interface generator for services. iCrafter services are similar to those provided by systems such as Jini [ASW$^+$99], except that the invocation happens through the Event Heap.

Summing up this paragraph, it became obvious that all of the systems and projects discussed here are capable of strongly supporting users in resource-rich, heterogeneous environments. However, due to their strictly required stationary infrastructure support, these systems lack the flexibility for their use in Ad Hoc environments, which is one of the main goals of this work.

3.2.2. Service Composition in Infrastructure-Less Ad Hoc Environments

The project **Aura** [SG02] provides a highly integrated environment for Ad Hoc pervasive computing that consists of various modules. Aura features an architecture where user tasks become first class entities in such a way that user proxies, or *Auras*, use models of user tasks to set up, monitor and adapt computing environments proactively. Aura's architectural framework represents the central component of Project Aura, the campus-wide ubiquitous computing effort at the Carnegie Mellon University in Pittsburgh. Aura enables mobile users to make the most of pervasive computing environments, while shielding those users from managing heterogeneity and dynamic variability of capabilities and resources. This is achieved by three key features: Firstly, user tasks are represented explicitly and autonomously from a specific environment. Secondly, user tasks are represented as coalitions of abstract services. Thirdly, environments are equipped to self-monitor and re-negotiate task support in the presence of runtime variation of capabilities and resources. Besides mobile devices such as laptops, Gaia seamlessly integrates existing infrastructure to support users in their daily work. Therefore, a context observer recognizes changes in the current user context and reports them to the task manager which exploits this information to adapt the system to the changed conditions. Figure 3.7 shows an abstract view of Aura's architectural framework. The *Task Manager* (Prism) encapsulates the concept of a personal aura. It aims to minimize user distractions in the face of four kinds of changes: environmental changes, task changes, context changes, and user mobility. The major idea behind Prism is to provide platform independence for user task descriptions. *Service Suppliers* provide the abstract services that tasks are composed of. The *Context Observer* provides information about the physical context and reports events in the physical context back to Prism and the Environment Manager. Finally, the *Environment Manager* component represents a gateway to the environment: it is aware of the available suppliers, and where their services can be deployed. It also encapsulates the mechanisms for distributed file access.

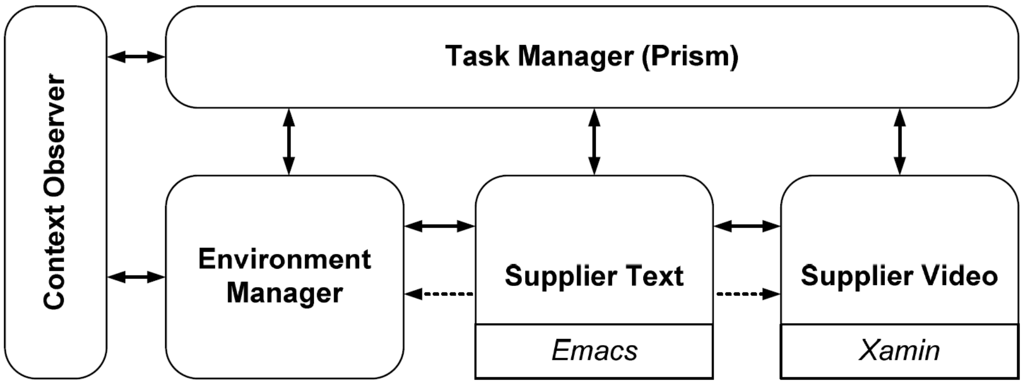


Figure 3.7.: Abstract architectural framework of Aura [SG02]

More recent works within Project Aura focus on the access to people's context and location information [JS03, Hen05] or the dynamic configuration and task-based adaptation of resource-aware services [PSGS04, SPG$^+$06].

Mobile Gaia [CAMCM05] is a successor of Gaia and, like the above discussed project Aura, represents a middleware for Ad Hoc Pervasive Computing environments. It is a service-based middleware that integrates resources of various devices. Mobile Gaia manages several functions such as forming and maintaining device collections and sharing resources among devices, and it enables seamless service interactions. It also provides an application framework to develop applications for the device collection. The application framework decomposes the whole application into smaller components that can run on different devices in this collection. Mobile Gaia considers pervasive environments as a cluster of personal devices that can communicate and share resources among each other. This cluster is referred to as *personal active space* and has a coordinator device and zero or more client devices.

Mobile Gaia is composed of a set of core services that manage the device cluster. These services enable the devices in the cluster to share resources and data seamlessly. Figure 3.8 shows Mobile Gaia's architecture. The core services for communication, component management, service development and several services, e.g., for discovery or security, make up the *Mobile Gaia Kernel*. On top of the kernel is Mobile Gaia's *Application Framework* which provides automated patterns for multi-

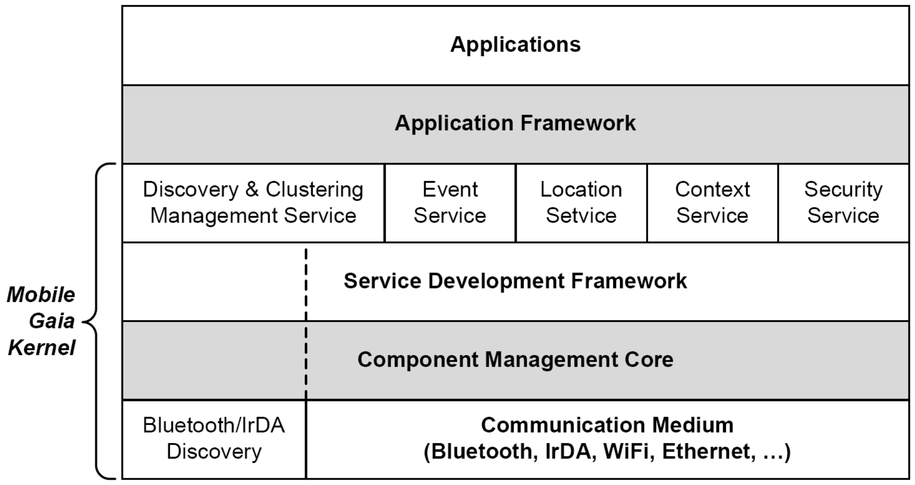

Applications
Application Framework
Discovery & Clustering Management Service
Event Service
Location Setvice
Context Service
Security Service
Service Development Framework
Component Management Core
Bluetooth/IrDA Discovery
Communication Medium (Bluetooth, IrDA, WiFi, Ethernet, ...)
Mobile Gaia Kernel

Figure 3.8.: Mobile Gaia architecture [CAMCM05]

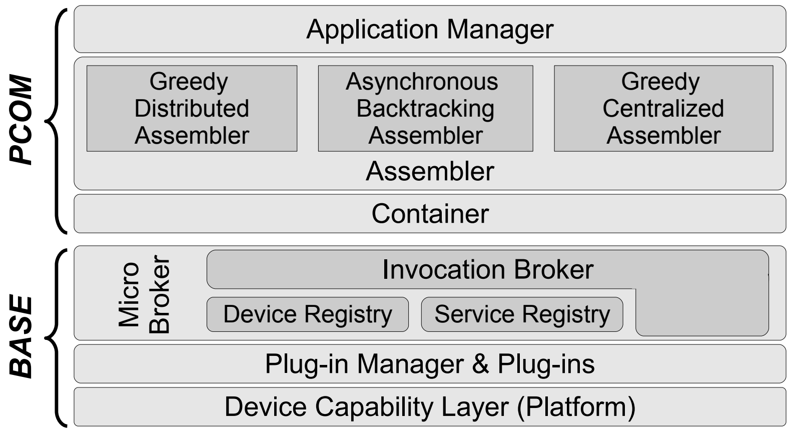

PCOM
Application Manager
Greedy Distributed Assembler
Asynchronous Backtracking Assembler
Greedy Centralized Assembler
Assembler
Container
BASE
Micro Broker
Invocation Broker
Device Registry
Service Registry
Plug-in Manager & Plug-ins
Device Capability Layer (Platform)

Figure 3.9.: Initial layered architecture of BASE [BSGR03] and PCOM [BHSR04] with developed configuration assemblers

device support, runtime adaptation, mobility, or context-awareness. Moreover, the application framework facilitates the creation and management of distributed applications and their components.

The system **PCOM** [BHSR04] which we rely on for the works in this thesis represents another system that was created for the use in peer-to-peer based Ad Hoc environments. PCOM relies on the **BASE Micro-Broker** [BSGR03] to provide automatic adaptation of protocols on the communication layer. Figure 3.9 gives an overview of the initial PCOM/BASE system architecture at time the research for this thesis started.

The *BASE Micro-Broker* was developed to support automatic configuration and adaptation of communication protocols at runtime. This enables a very stable and flexible communication platform. BASE provides distribution-independent access to the offered services and decouples the application from the underlying communication protocols. The main component of BASE is the *invocation broker* which delegates method calls to the corresponding services on the mobile devices and, thus, realizes BASE's core functionality. Furthermore, BASE manages the devices which can currently be reached through its *device registry* and the services that are available on a device through its *service registry*.

The automatic configuration of the supported protocols is possible by the *plug-in manager*, as the protocols are outsourced in plug-ins which represent the entities that are capable of receiving invocation. Plug-ins are loaded and configured at runtime. Typical representations of plug-ins are transport protocols, interoperability protocols, or service discovery protocols. The plug-in concept allows the BASE micro-broker to support a wide spectrum of end-user devices, reaching from simple micro controllers to mobile phones, standard desktop PCs or even servers. The only requirement a device has to meet is the presence of a JVM [LY99] that supports the *Connected Limited Device Configuration version* (CLDC, [Sun03]) profile. Plug-ins typically involve interaction with the underlying operating system or directly with the hardware to offer access to a device capability or transport. The *device capability layer* as BASE's lowest layer represents the platform of a device with its supported hardware and software, e.g., a WLAN antenna, a GPS sensor, or an XML library.

Since BASE does not offer adaptation support at higher levels, the component middleware *PCOM* was developed to provide a runtime environment for distributed applications. PCOM enables automatic runtime configuration and adaptation at application level without user intervention. In PCOM, application components are executed within a *container* that provides an interface to the PCOM middleware. Moreover, a PCOM container offers and manages basic services for the components. Thus, containers act as a distribution execution environment for components. They are implemented as a single service on top of BASE. The automatic configuration of components is performed by the so-called *assemblers* which enable access to components prior to their instantiation and, thus, decouple the configuration processes from the lifecycle management of the components. The lifecycle management of applications and components is realized by the *application manager* that enables to start, stop or configure distributed applications. Therefore, it uses the containers and assemblers.

Initially, PCOM was designed to provide system support in spontaneous Ad Hoc scenarios. Therefore, a configuration approach based on Distributed Constraint satisfaction [YDIK92] was presented [HBR05]. This approach features an algorithm which is based on Asynchronous Backtracking [YDIK98]. To avoid that specific devices have to be present in environments with strict resource constraints, this algorithm configures applications in a completely decentralized manner.

After the development of this initial configuration approach, an algorithm was developed which enables the re-configuration of running applications that may be needed due to device failures or mobility of users [HHS^{+}07]. In this work, the component model is modified so that fully automatic adaptation is possible. Moreover, a simple, but yet powerful cost model is presented to capture the complexity of specific adaptations. Additionally, an online optimization heuristic is discussed that extends the distributed configuration algorithm [HBR05] in a way that it is able to switch to a configuration with low costs whenever the current application configuration needs to be adapted.

In a subsequent work [HHSB07], the PCOM container was extended in a way that it enables the support of exchangeable configuration algorithms to optimize the configuration processes in environments where additional infrastructure devices may be available. Therefore, two non-complete algorithm which select resources in a greedy manner were presented: a completely decentralized one called *Greedy Distributed Assembler*, and a completely centralized one called *Greedy Centralized Assembler*. In the evaluations, the authors could show that this new architecture enables significant performance optimizations by providing pluggable configuration algorithms. However, an automatic switching between different configuration approaches is not provided.

Most recent works on PCOM and BASE cover the development of a dynamic conflict management system (COMITY, [TSB07]) that detects and resolves conflicts according to the user's preferences [TSB09], the context-based coordination of resources in multi-application environments [Maj10, MSS+10], and the introduction of modular plug-ins into BASE to increase flexibility and code re-use via efficient runtime composition of plug-ins [HWS+10]. However, none of these works cover an efficient exploitation of scenario heterogeneity or the re-use of data about historic configuration processes, which we focus on in this thesis.

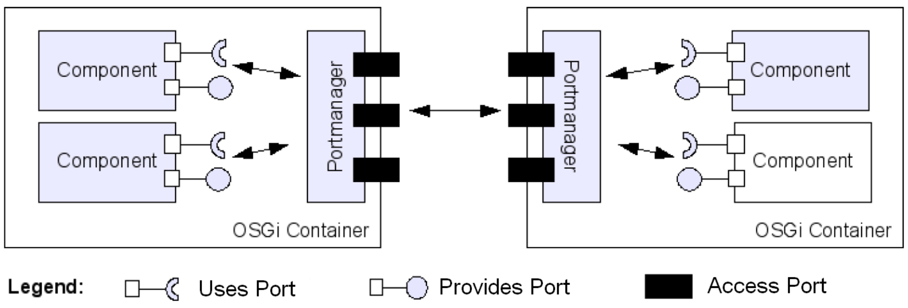


Figure 3.10.: Ports concept of P2PComp [FHMO04]

P2PComp [FHMO04] is a project to provide context-aware mobile devices by a Peer-to-Peer (P2P) pervasive computing middleware. P2PComp's lightweight software component model addresses the development needs of peer-based mobile applications. It provides an abstract, flexible, and high-level communication mechanism among components via a *ports* concept that is displayed in Figure 3.10 and consists of three different types of ports: *Access Ports* represent communication endpoints of a communication channel between two components and abstracts from the next-lower protocol layer. Single peers can supply services via *Provide Ports* as well as consume them via *Uses Ports*. Moreover, services can migrate between containers, and services are ranked to support Quality of Service (QoS) choices. P2PComp's lightweight container realization leverages the OSGi platform and can utilize various P2P communication mechanisms such as JXTA [Gon01].

Speakeasy [ENS+02, NIE+02] is an Ad Hoc peer-to-peer framework that offers seamless flexibility in terms of discovery protocols, network usage, and data transport. In the Speakeasy framework, any entity that can be accessed over a network is cast as a component. Components are discrete elements of functionality that may

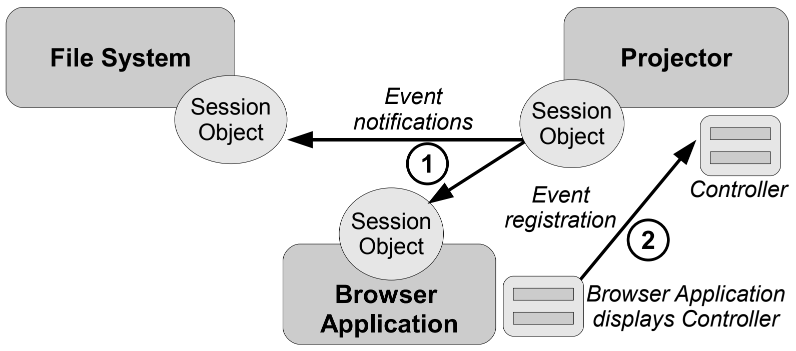


Figure 3.11.: Typical proceeding in a Speakeasy application [NIE+02]

be interconnected with other components or used by applications. Components may represent devices such as printers or projectors, services such as search engines or file servers, or information such as files and images. Applications engage in a discovery process to find components ”around” them in the network. These may be components that are running locally (that is, on the same machine as the application itself), on nearby machines (potentially discovered using some proximity- based networking technology such as Bluetooth), on remote intranets or the Internet itself. Speakeasy applications do not use a fixed set of discovery protocols. Instead, applications dynamically adopt new discovery protocols as their underlying networks change. Likewise, Speakeasy applications do not depend on fixed data exchange protocols or data types. Instead, they acquire new behaviour as they interact with the components around them. Speakeasy applications use these interfaces to access component functionality. Because the set of interfaces is fixed, any new component can be used by any existing application that understands the interfaces.

Figure 3.11 shows an exemplary Speakeasy application where a user controls a projector for showing contents on the projector's screen. In a first step, the *Session Object* of the projector notifies the session objects of the other present components (*file system* and *browser application*) that may be accessed using a *controller*. In the second step, the browser application registers an event at the projector that enables the user to locally access the controller of the projector.

Speakeasy provides a platform for extensible peer-to-peer computing that can be applied to a number of domains where flexibility is paramount, including Pervasive Computing and collaboration. Still, the system itself is not architected exclusively for collaboration, but can be considered as a good framework for building collaborative applications. In Speakeasy applications, the user is responsible for choosing the components needed for configuration.

The **one.world** [Gri04] system is an architecture that provides an integrated and comprehensive framework for building pervasive applications. It targets applications that automatically adapt to highly dynamic computing environments, and it includes services that make it easier for developers to manage constant changes. One.world puts the responsibility of application configuration and adaptation on the application developer through providing a set of APIs and tools to manage this task. Figure 3.12 gives an overview of one.world's system architecture.

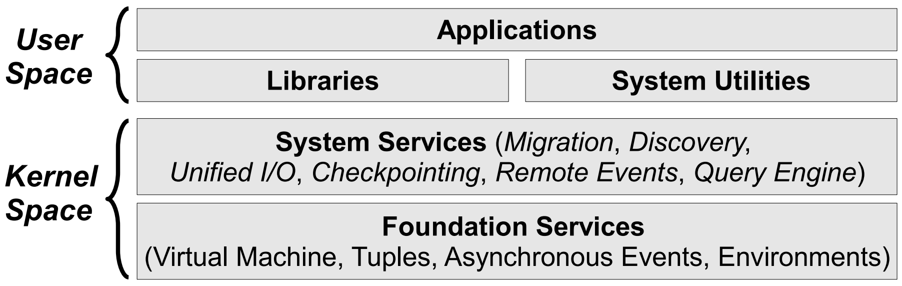

User Space
Kernel Space
Applications
Libraries
System Utilities
System Services (Migration, Discovery, Unified I/O, Checkpointing, Remote Events, Query Engine)
Foundation Services (Virtual Machine, Tuples, Asynchronous Events, Environments)

Figure 3.12.: System architecture of one.world [Gri04]

To overcome the limitations of distributed systems, one.world identifies three requirements to provide system support for pervasive applications: Firstly, system support must embrace contextual changes, not hide them from the applications. Secondly, system support must encourage Ad Hoc composition and not assume a static computing environment with just a few interactions. Thirdly, system support must facilitate sharing between applications and between devices.

One.world's architecture is centered on meeting these three requirements. It employs a classic user/kernel split: *Foundation* and *system services* like discovery, migration, or unified I/O run in the kernel, while *applications*, *libraries*, and *system utilities* run in user space. One.world's foundation services provide the basis for the architecture's system services, which in turn serve as common building blocks for pervasive applications. The four foundation services are a *virtual machine* to ensure that applications and devices are composable, *tuples* which define a common model for all data to simplify data sharing, *asynchronous events* for all communication concerning changes in the runtime context, and *environments* that host running applications and isolate them one from another.

3.3. Clustering Frameworks

When introducing our approach for adapting the distribution of automatic application configuration in Section 4.3, we will present a clustering framework to elect the most powerful device as cluster head, being responsible for centralized configuration calculations. Moreover, for our hybrid configuration scheme, we will extend this framework to support multiple cluster heads in strongly heterogeneous environments and provide a balanced configuration load among the cluster heads. Clustering schemes that try to achieve a balanced load among several nodes have widely been used in different research areas for many years. Here, we present the most related approaches. Figure 3.13 gives an overview of the discussed schemes in the various areas.

The election of specific nodes to become a coordinator for a group of nodes via clustering is a common subject in the research area of Mobile Ad Hoc Networks (MANETs) [YC05]. Many of these related approaches also aim at balancing the load among nodes. Schemes like the *Weighted Clustering Algorithm* (WCA, [CDT02]),

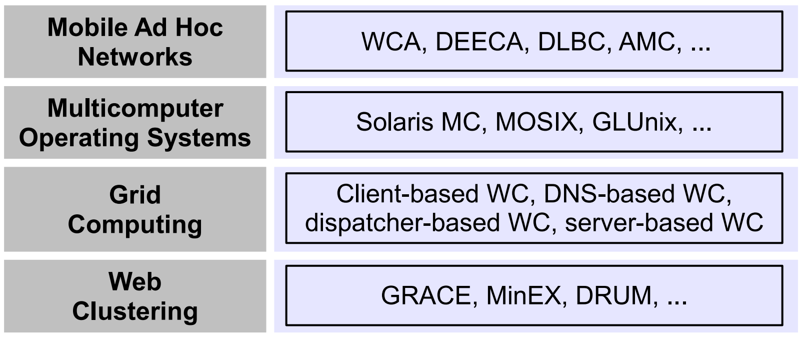

Mobile Ad Hoc Networks
WCA, DEECA, DLBC, AMC, ...
Multicomputer Operating Systems
Solaris MC, MOSIX, GLUnix, ...
Grid Computing
Client-based WC, DNS-based WC, dispatcher-based WC, server-based WC
Web Clustering
GRACE, MinEX, DRUM, ...

Figure 3.13.: Load-balanced clustering approaches in different research areas

the *Dynamic Energy Efficient Clustering Algorithm* (DEECA, [SMA08]) or *Degree-Load-Balancing Clustering* (DLBC, [AP00]) 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 *Adaptive Multihop Clustering* (AMC, [OIK03]) 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 strong 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 strong devices is rather static.

In some approaches for MANETs, clustering is often used to save the energy of mobile and battery-constrained nodes. This is achieved by determining clusters such that cluster members can be deactivated to save energy. Typical energy-efficient schemes are GAF [XHE01], Span [CJBM02], or SANDMAN [Sch07], where the latter especially focuses on Pervasive Computing environments. These approaches are usually used for routing and try to minimize the influence of deactivating nodes to the network's connectivity. However, the respective clustering schemes strictly focus on energy-efficiency, which differs from our goal of minimizing the latencies of configuration processes by exploiting the scenario heterogeneity.

In *Multicomputer Operating Systems* [BK87], load-balancing approaches have been introduced to achieve dynamic work distribution and load sharing in high performance computing, where some form of remote execution and process migration is used. Some of the proposed systems are Solaris MC [KBM+96], MOSIX [BL98], or GLUnix [GPR+98]. Like the schemes that were proposed for MANETs, the approaches discussed here have goals which are completely different to ours.

In the research area of *Web Clusters*, scheduling algorithms like Dynamic Weighting Scheduling (DWS, [QCH08]) try to balance the load distribution on the servers to increase the loading capacity of the cluster [CICY99]. The respective clustering approaches can be classified whether they are *client-based* (e.g., [BBM+97]), *DNS-based* (e.g., [CCY99]), *dispatcher-based* (e.g., [HGKM98]), or *server-based* (e.g., [BCLM99]). Furthermore, many balanced-load clustering schemes have been developed in the research area of *grid computing*, as grid computing environments represents highly heterogeneous, dynamic and shared environments and, thus, widely differ from most traditional distributed or parallel systems. Exemplary projects which provide load-balancing clustering schemes for grid computing are GRACE [BAG01], MinEX [DHB02], or DRUM [Fai05]. A broader survey is given by Li and Lan [LL05]. However, the mentioned schemes for web clusters or grid computing 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 strong devices.

3.4. Re-Utilization of Previous Configurations

Most of the systems discussed in Sections 3.2 do not provide concepts for the re-use of previous configuration results, so we only discuss those systems here which supply the respective functionality.

The re-utilization approach has already been introduced in the project OSCAR [NES08], which deals with applications for distributed media device control. OSCAR builds on the Speakeasy system [NIE+02] and provides flexible and generic control of services and devices in home media networks. OSCAR represents an end-user composition tool which provides flexible and generic control of services and devices for home media networks that builds upon user experience goals. These goals are extended by allowing the construction of *reusable* compositions for common tasks. In OSCAR, it is supposed that most users will find regular patterns of connections that they want to re-use on a regular basis. OSCAR provides a *setup* concept. Setups describe how components are to be found, selected, and connected together in order to carry out a routine activity. Each setup consists of slots for sources and destinations. A slot contains a query in order to generate a set of candidate components. Moreover, each slot includes a selection rule dictating how active components are selected in case of alternative components at invocation time. While OSCAR supports the re-use of compositions, it relies on *end-user* composition, i.e., OSCAR does not provide automated configuration by the system as we do, but only manual composition by the user.

A solution quite similar to OSCAR is ICrafter [PLF+01] from Stanford University. ICrafter is a sub-system of iROS [JFW02] and represents a framework for services and their user interfaces in a class of pervasive computing scenarios to allow user interaction with the services in their environment using various input devices. ICrafter allows the reuse of User Interface (UI) templates across workspaces. Additionally to manual composition, it also allows automatic generation of UIs, giving ICrafter an advance over OSCAR. However, as already mentioned in Section 3.2, iROS is only usable when a specific infrastructure is available, and it only provides automatic creation of UIs, while the overall service composition has to be done manually.

In the area of home entertainment, *universal remotes* [OPID06] such as the Logitech Harmony [Log13] or Philips Prestigo [Phi10] provide a single point of control for interacting with multiple devices. Most of these remotes also allow the storage of specific profiles to easily allow their future re-use. However, these devices are limited in the way that their initial set up is tedious. Moreover, they cannot adapt to new devices without being explicitly told, e.g. by cumbersome user programming on a PC. Furthermore, the present universal remotes cannot create Ad Hoc connections among networked services; they can only activate connections among devices that are already hard-wired together, which restricts their applicability to rather static scenarios. In contrast, mobile computers can control arbitrary numbers of device instances [OD07].

The provision of configured sets of services is also a typical issue of Web Services [ACKM03]. Web Service compositions compose high-level functionalities (e.g., "Collect sensor data") by services that have lower-level functionalities. Many schemes and patterns [YP02, HY04, MF04, PA05] have been proposed to enable reusable compositions in Web Services. The compositions are reused by applying them to orchestrate different component services. However, the mentioned papers and articles do not present strategies and mechanisms for the automatic provision of a limited set of high quality compositions in dynamic environments. Lamparter et al. [LAS07] provide a strategy which automatically selects dynamic service compositions according to a quality ranking using ontologies, but they still do not present a replacement strategy for memory-restricted environments or deal with highly-dynamic pervasive scenarios.

4 A Hybrid Approach for Automatic Application Configuration

With the concepts and algorithms presented here, we want to cover a broad spectrum of possible pervasive scenarios with various degrees of heterogeneity. Decentralized approaches (e.g., [FHMO04, HBR05, CAMCM05]) do not pose any requirements on the scenarios and, thus, are generally usable. However, decentralized configuration may yield inefficient configurations in heterogeneous environments, as these approaches balance the configuration load among *all* nodes, but do not exploit the increased computation resources of specific infrastructure devices. This leads to increased configuration latencies [SHR08b].

Thus in this chapter, we first present an extensive discussion of our design rationale in Section 4.1. Subsequently, we present an efficient centralized configuration algorithm in Section 4.2. Then, we introduce a concept to support various configuration algorithms and allow the automatic switching between them in Section 4.3. Following in Section 4.4, we present an advanced hybrid configuration approach for strongly heterogeneous environments. Then, we show in Section 4.5 by evaluation that our new concepts, mechanisms, and algorithms perform well in realistic scenarios. Finally in Section 4.6, we summarize and give a broader discussion on the achievements made by our contributions.

4.1. Design Rationale

In this section, we describe the rationale for a framework to enable an automatic adaptation of the degree of decentralization. Moreover, we focus on the support of a wide spectrum of possible Pervasive Computing scenarios by tailored configuration algorithms. First in Section 4.1.1, we depict the path from supporting only completely decentralized configuration that involves all present devices towards an adaptable degree of (de-)centralization. This adaptable scheme covers centralized and decentralized schemes as well as hybrid approaches that lie between those two "extreme" approaches. Then in Section 4.1.2, we discuss *clustering schemes* as a typical mechanism to distinguish between different device roles, which is important for our centralized and hybrid configuration approaches, as they distribute the configuration tasks only among a selected subset S of all devices. Finally in Section 4.1.3, we argue for the introduction of a pre- configuration process to automatically and proactively retrieve the relevant configuration information by the devices included in S.

4.1.1. Towards Hybrid Application Composition

In this section, we first formally introduce the configuration problem. Then, we sketch the way from decentralized configuration via centralized configuration to a hybrid approach, providing highest efficiency and best adaptiveness by combining the beneficial properties of the decentralized approach and the centralized approach.

As mentioned in Section 2.2.1, *configuring* an application denotes the task of determining a set of components that can be instantiated at the same time. These components have to provide the functionality required by the application while considering both structural and resource constraints. The configuration task is resolved by a *configuration algorithm* which matches the offered component functionality with the application requirements for each contract. Whenever the algorithm fails to find such a component for a contract (e.g., due to lack of resources), an *adaptation* process has to be initiated to resolve this conflict.

In our context, the application configuration problem can be mapped to an NP-complete Distributed Constraint Satisfaction Problem (DCSP), as Handte et al. have shown [HBR05]. A DCSP is a special case of a Constraint Satisfaction Problem (CSP) in which the variables are distributed among automated agents. Agents communicate by sending messages with finite, though random delivering delays. Generally, a CSP consists of n variables $X = \{x_1, x_2, \cdots, x_n\}$, whose values are taken from finite, discrete domains $D = \{D_1, D_2, \cdots, D_n\}$, respectively, and a set of m constraints $C = \{C_1, \ldots, C_m\}$ on their values [YDIK98]. A constraint is defined by a predicate. That is, the constraint $p_k(x_{k1}, \cdots, x_{kj})$ is a predicate that is defined on the Cartesian product $D_{k1} \times \cdots \times D_{kj}$. This predicate is true iff the value assignment of these variables satisfies this constraint. Solving a CSP is equivalent to finding an assignment of values $x = \{x_1 \in D_1, \ldots, x_n \in D_n\}$ to all variables such that all constraints are satisfied. If we map the components to the discrete domains D, the dependencies between components to the variable set X, and the resource dependencies to the constraints C, we obtain the following configuration problem statement: The DCSP (D, X, C) consists of n structural dependencies $X = \{x_1, \ldots, x_n\}$ and m resource dependencies $C = \{C_1, \ldots, C_m\}$ between the component domains $D = \{D_1, \ldots, D_n\}$. Then, the following equation needs to be fulfilled to solve the application configuration problem:

$$\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\} : x_i \in D_i \wedge C_j \subseteq D_{i_1} \times ... \times D_{i_k} \qquad (4.1)$$

The *configuration latency* comprises the time between the user's application start and the availability of the application to the user. This latency includes the delays for calculating the configuration, distributing the configuration results among the present devices, and instantiating all determined application components. In this work, we aim at minimizing the configuration latency to provide a seamless user experience.

Decentralized configuration which relies on backtracking algorithms from the research area of Distributed Artificial Intelligence [YDIK98], as discussed in the previous paragraph, is widely used because of its universal applicability in different environments, without relying on specific instances or devices to be available. The decentralized configuration process is shown in Figure 4.1: Configuration begins when a user wants to start a distributed application on his or her mobile device (step 1). In the shown example, an application is started on the mobile device with id 3. Following, the other devices are notified about the requirements that have to be met to execute the application, and the devices try to cooperatively resolve all dependencies (step 2). After a valid application composition was found, the determined component set is transmitted back to device 3 in the third step. In the example, the devices 1 and 4 provide partial solutions of the application, while device 2 cannot provide any of the required functionality. Finally in step 4, the bindings between the components on the devices 1, 3 and 4 that were determined within the configuration process are instantiated. Now, the successfully configured application can be executed. The configuration latency represents the sum of the times which steps 1 to 4 take.

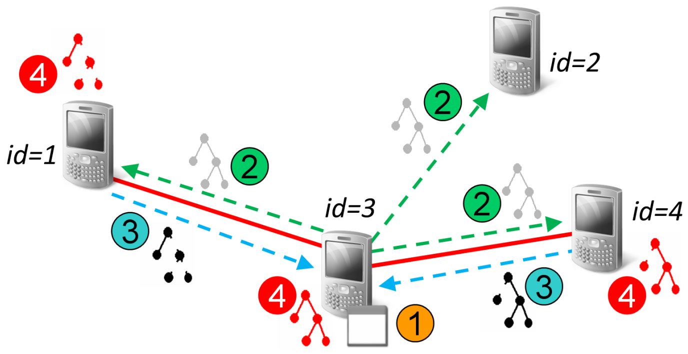


Figure 4.1.: Decentralized configuration process

When a single node fails during configuration, the other devices can notice this by the communication middleware because of missing messages from this node, and another node can adopt the part of the configuration of the failed device. Thus, decentralized configuration, as shown in Figure 4.1, is still efficiently applicable if some device fails because it distributes the configuration tasks among various devices. The decentralized approach is *particularly suited* for the use in *homogeneous* scenarios and shows *good scalability* in such environments, as it equally distributes the configuration tasks among all devices. However, decentralized configuration introduces a drawback in *heterogeneous environments* with additional powerful devices: there, weak devices may become bottlenecks for decentralized configuration due to their reduced computation power.

In order to additionally *exploit the increased performance* on powerful devices in *heterogeneous environments*, we aspire a new approach where the configuration calculation is performed in a **completely centralized** manner on the most powerful device in the environment to keep the configuration latencies as small as possible. In this approach, a particular node has to be found that performs this centralized calculation. We suggest to use clustering for this issue and discuss this topic in more detail soon. As an example for centralized configuration, regard the scenario shown in Figure 4.2a. Here, the desktop PC is significantly more powerful than every other device, as identified in step 1. This device then needs to collect configuration-relevant data to create an internal representation of the application tree that comprises dependencies, components, and incorporated devices with their available resources before the actual calculation of a valid configuration. Thus, the cluster members automatically transfer their current resource and component conditions to the selected powerful device in step 2.

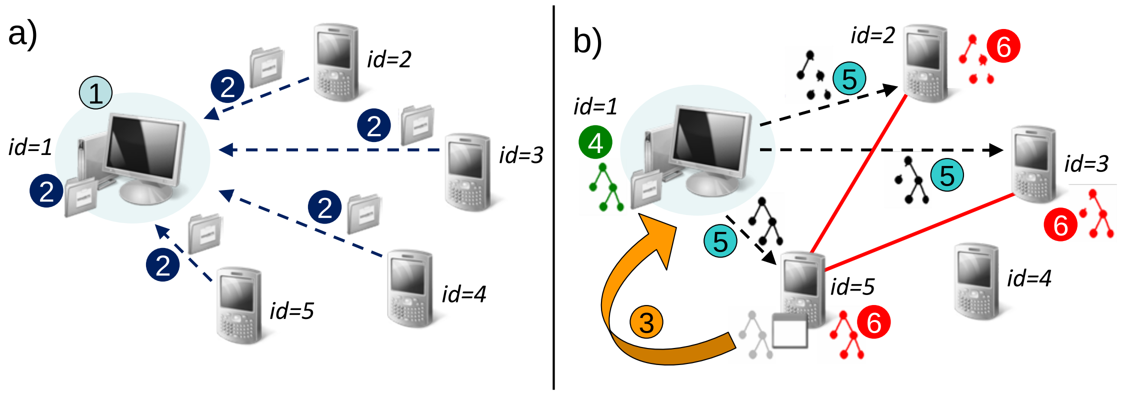


Figure 4.2.: Centralized configuration process: a) Selection of configuration device and retrieve of resource information, b) Centralized configuration and distribution of results

Now, when a user wants to start an application, as noted in step 3 of Figure 4.2b, the most powerful device is notified about this application start by transferring the application information from the user's mobile device 5 to device 1. Following in step 4, device 1 calculates a valid configuration completely locally using a centralized configuration algorithm. If the configuration process was successful, device 1 distributes the configuration results amongst the involved devices in step 5. In the example, device 1 informs devices 2 and 3 about which part of the application tree they have to provide. Furthermore, device 1 notifies device 5 (as the device where the application is executed) about all involved components and the devices that host these components. Finally, the found components are instantiated in step 6 and the application is successfully executed. This approach aims at reducing the noticeable configuration latencies in heterogeneous environments since the available additional computation power is efficiently used and communication is not needed during configuration, as the resource information of each device has already been transmitted in step 2 (see above). We will present our approach for centralized configuration in more details in Section 4.2.

Summing up the centralized approach's advantages, it reduces the noticeable configuration latencies in heterogeneous environments and does not produce any message overhead during configuration. Furthermore, as the resource-weak devices are not involved in the actual configuration (they only need to communicate their resource situation whenever it changes, *independent* from configuration processes), their computational burden during configuration is lowered. However, the centralized approach also has some general drawbacks: The message overhead before and after the configuration is increased because of the communication between the cluster head and its cluster members. Moreover, the centralized approach scales quite bad as it does not perform parallel calculations, but instead puts all computational burden on *one single* device. Additionally, the cluster head may represent a potential single point of failure: if it fails during configuration, another node has to be elected as cluster head and needs to configure the application again from scratch, which significantly increases the configuration latency. The biggest drawback, however, is that centralized configuration is not generally applicable, as it relies on the permanent availability of one powerful device. However, this cannot be guaranteed in highly-dynamic Ad Hoc scenarios and induces many re-clustering processes there.

Concerning the degree of distribution in the configuration, the decentralized and centralized configuration approaches represent the extreme cases: Decentralized configuration distributes the load *equally among all devices*, thus being perfectly suited for homogeneous Ad Hoc scenarios where *only* weak mobile devices are available. As the opposite extremum, centralized configuration does not distribute the load at all, but puts the complete configuration calculations on *one single device* – the one with most computation power. Because of this, centralized configuration is optimized for environments with one additional powerful device. Our design rationale for a **hybrid configuration** is to provide a tradeoff that partially relies on decentralized and partially relies on centralized configuration. Thus, such a hybrid approach should combine the best properties of these two configuration approaches: Only the powerful devices like laptops, servers, or desktop PCs should calculate valid configurations in a cooperative fashion. This reduces the risk of possible bottlenecks due to the mobile devices' weak computation power. The weak devices only have to provide their resource information and receive the results of the configuration from the powerful devices. This significantly helps to conserve their resources during configuration processes. The main goal posed to this configuration approach is the further reduction of the latencies in the whole spectrum of possible Pervasive Computing scenarios.

Figure 4.3 sketches the hybrid configuration scheme we aspire. Different to the previously described approaches, hybrid configuration is neither performed by *all devices* (decentralized), nor by *one specific device* (centralized), but by a *selected subset* of all devices, the set of powerful devices. Therefore, the hybrid scheme initially has to determine the specific devices (step 1), e.g., by the aid of a clustering scheme (cf. Sections 4.1.2, 4.3.2). Then, each weak device has to be mapped to exactly one powerful device, which hereby takes the responsibility for configuring the weak device's components (step 2). Subsequently, the powerful devices calculate the configuration for their mapped devices in a decentralized manner, and distribute the configuration results to their mapped devices, respectively.

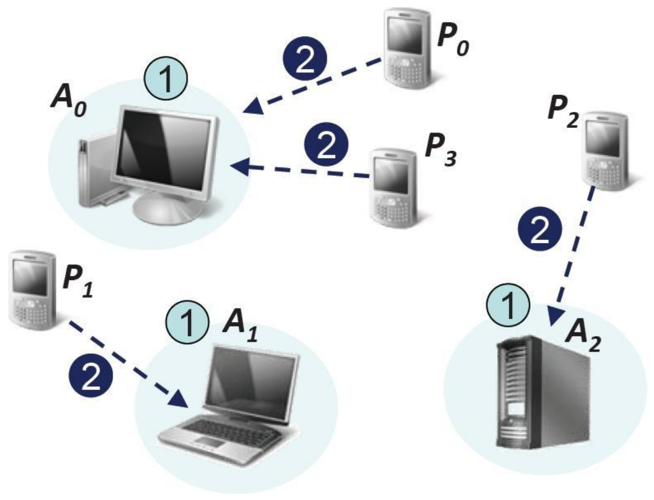


Figure 4.3.: Aspired hybrid configuration process: 1. Selection of configuration devices (A_x), 2. Retrieve of resource information from weak devices (P_y) for hybrid configuration

As shown in Table 4.1.1, which represents our comparison of the properties of the three discussed configuration approaches, only a hybrid approach is efficiently applicable both in homogeneous and heterogeneous environments and represents a tradeoff between pure decentralized configuration on *all* devices, and pure centralized application configuration on a *single* resource-rich device.

4.1.2. Role Determination using Clustering

As discussed in the last section, centralized and hybrid configuration distinguish between two different roles in configuration: the one class of the devices, the *powerful devices*, are actively calculating configurations, while the second class of the devices, the *weak devices*, remain passive during the configuration and only provide their resource information prior to the configuration. In centralized configuration, there is obviously only one active configuration device, while hybrid configuration has to deal with several active configuration devices. Thus, before centralized or hybrid configuration can take place, one or more dedicated devices have to be identified which perform the actual configuration and act as coordinators for the configuration process.

The election of specific devices – also called *nodes* in our context – to become coordinators for a group of nodes is a common subject in the research area of Mobile Ad Hoc Networks (MANETs). This problem is typically solved by the introduction of *clustering schemes* which introduce different device roles and form a hierarchy of disinguished groups of nodes. A lot of schemes for organizing nodes to clusters exist. Clustering scheme can be classified according to their overall goal, whereas typical goals of common schemes are to achieve a low maintenance overhead, a high energy efficiency, or a balanced load among the cluster heads. Furthermore, clustering schemes either rely on a central server which coordinates the clustering

Property	Distributed	Centralized	Hybrid
Efficiency in homogeneous environments	High (+)	Low (-)	High (+)
Efficiency in heterogeneous environments	Low (-)	High (+)	High (+)
Message overhead before/after configuration	Low (+)	High (-)	Medium (o)
Message overhead during configuration	High (-)	None (+)	Medium (o)
Burden of mobile devices	High (-)	Low (+)	Low (+)
Pre-configuration process possible	No (-)	Yes (+)	Yes (+)
Scalability in target scenario	Good (+)	Bad (-)	Okay (+/o)
Single point of failure	No (+)	Yes (-)	No (+)

Table 4.1.: Comparison of general properties of distributed, centralized and hybrid application configuration

process, or they are performed in a distributed manner without a central instance. Yu and Chong [YC05] give a survey on existing approaches and classes of MANET clustering schemes. Furthermore, we discuss clustering schemes in our Related Work section 3.3.

A clustering scheme divides the nodes in a MANET into different virtual groups based on certain rules. Under a cluster structure, mobile nodes are typically assigned a different status or function. Generally, most clustering schemes introduce three different node roles: *cluster heads*, *cluster members*, and *cluster gateways*. A *Cluster Head (CH)* represents an exclusive node within a cluster and usually serves as a local coordinator within its cluster. The cluster head is connected to every other node within its cluster and is responsible for tasks such as intra-cluster transmission arrangement or data forwarding. In this work, the cluster heads are *the only nodes within a cluster* which are responsible for calculating configurations. Thus, they take the responsibility for selecting the components of the other cluster nodes within a configuration process. Therefore, it is vital that the cluster heads have high computation power to enable highly efficient configuration processes. A *Cluster Member (CM)* is usually also called an ordinary node and represents a non-cluster head node without any inter-cluster links. In this work, the cluster members are represented by the resource-poor devices and are not actively involved in the configuration processes. They only have to allow the cluster heads to access their current resource conditions. Finally, *Cluster Gateway (CG)s* represent the non-clusterhead nodes with inter-cluster links, so they can access neighboring clusters and forward information between clusters. Figure 4.4 shows an exemplary established cluster structure consisting of three cluster heads with their transmission ranges, and several cluster members and cluster gateways.

Cluster gateways are only relevant in environments where some nodes are not in transmission range to every other node. According to our environment model

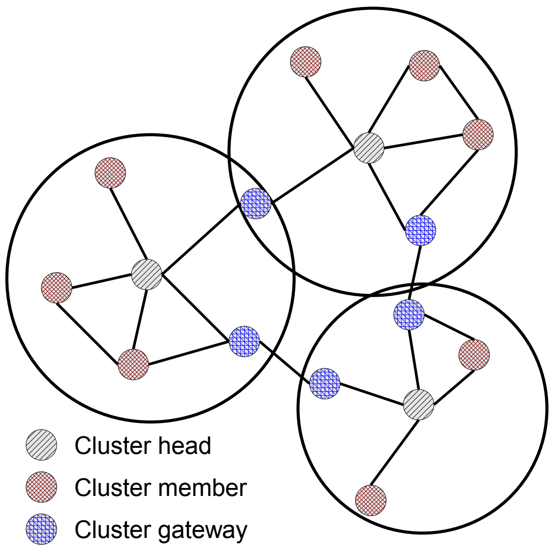


Figure 4.4.: Typical cluster structure, as discussed by Yu and Chong [YC05]

presented in Section 2.1.1, we rely on one-hop environments where each device is directly connected to any other device participating in the application configuration, so we do not need cluster gateways here.

Two key attributes that designers have to consider when creating a clustering scheme for environments where the availability of devices dynamically changes [KDLS08] are the selection criteria for CHs and CMs (also called the initial *cluster formation*), and the provision of a repair mechanism in case of changing availability of nodes (also called the *cluster maintenance*). The first phase of cluster formation is accomplished by selecting some nodes as cluster heads that act as coordinators of the clustering process. Then, a cluster is formed by associating a cluster head with some of its neighbors that become its cluster members. In this phase, clustering schemes assume that there are no changes in the environment, i.e., nodes are stationary, and no new devices appear or available devices disappear. Once the nodes are partitioned into clusters, the assumption that nodes are stationary is released and the cluster maintenance organizes the cluster structure in the presence of dynamic changes in the environment. Obviously, this may induce re-organization of the cluster structure, called *re-clustering*. During re-clustering processes, the network (and, thus, the applications) cannot rely on the cluster organization, making clustering only a feasible solution if re-organizations are not needed too frequently. Relying on the user mobility discussed in Section 5.7.2, mobile devices are typically available for several minutes, while stationary devices have very high availability. Thus, relying on clustering is practicable in this work.

As we want to maximize efficiency by exploiting the device heterogeneity in Pervasive Computing environments, the properties of the available devices have to be investigated *prior* to cluster formation to find a suitable set of cluster heads that represent the configuration devices. This obviously has to be done in a distributed manner to retain applicability in Ad Hoc environments, so we do not consider centralized clustering approaches and only focus on distributed clustering schemes here.

Moreover, as Pervasive Computing environments are typically very dynamic, the provision of cluster maintenance strategies for the different undesired events that may appear at system runtime (e.g., the failure of a specific node and the consecutive breakdown of the corresponding configuration entity) require special attention. Therefore, we will discuss the possible erronous situations that may appear, and provide solutions to each of these situations when introducing our clustering schemes in Sections 4.3 and 4.4.

As mentioned before, a lot of schemes for organizing devices to clusters exist [YC05], focusing on different goals such as finding a connected set of dominating nodes [WL99], providing low cluster maintenance costs [CWLG97], mobility-awareness [BKL01], energy-efficiency [RSC02], or load-balancing [OIK03]. However, we want to provide a clustering scheme in a one-hop environment which satisfies three main goals: Firstly, the scheme should select the cluster heads in a resource-aware manner, i.e., only powerful devices are elected as cluster head and actively involved in configuration processes. Secondly, to avoid bottlenecks in the configuration phase, each weak device has to be mapped to exactly one strong device in a way that enables a balanced configuration load among the powerful devices. And thirdly, as clustering causes additional latencies, the provided clustering scheme should guarantee stable clusters and particularly avoid many reclustering processes. Relying on computational resources typically yields high cluster stability, as these resources normally do not change for the short term. So far, none of the previously presented schemes provide this functionality, so we discuss some convenient basic schemes here and adapt these schemes correspondingly to satisfy our needs.

Two classical clustering schemes which represent the base for our new scheme are *Lowest ID Clustering* (LID, [LG06]) and *Highest Connectivity Clustering* (HCC, [GTCT95]). Both schemes rely on unique device IDs, but select the cluster heads in a different manner: While LID elects those devices which have the *lowest ID* within their neighborhood as cluster heads, HCC chooses those devices as cluster heads which have the highest *number of neighbors* (also called the highest *degree of connectivity*).

```
INPUT: $G = (V, E)$: network, $w$: weights, $N[v]$: set of neighors of node $v$
OUTPUT: $\{C_i\}_{i \in I \subset V}$: clustering
begin
    i:=0;
    **while** V $\neq \emptyset$ {
      i := i + 1;
      // *Pick the node with the lowest ID*
      // *among those with maximum weight:*
      $v := min\{u \in V : w_u = max\{w_z : z \in V\}\}$;
      $C_i := \{v\} \cup N[v]$;
      $V := V \setminus C_i$;
    }
end
```

Listing 4.1: Generalized Clustering Algorithm (GCA) [BCFJ97]

Both LID and HCC can be generalized by Basagni's *Generalized Clustering Algorithm* (GCA, [BCFJ97]), which is shown in Listing 4.1. Here, the network graph G is represented by nodes V and edges E, representing bidirectional transmission links between two nodes. GCA assigns each node V_i a cluster weight $w_i \in [0, 1]$ and then elects the node with highest cluster weight within its neighborhood as cluster head. Weight-based algorithms are suited for expressing preferences on which nodes are better suited to be cluster heads. The generalization of LID and HCC to GCA can be performed by assigning the cluster weights as reciprocals of the (w.l.o.g. positive) devices' IDs id_i (in LID), or the degree of connectivity dc_i (in HCC):

$$\text{LID: } w_i = \frac{1}{id_i} \tag{4.2}$$

$$\text{HCC: } w_i = \frac{1}{dc_i} \tag{4.3}$$

To allow its use in distributed environments, a clustering framework needs to rely on a completely distributed clustering scheme to obtain the cluster heads that act as a coordinator for the cluster members within configuration processes. The distributed version of the Generalized Clustering Algorithm called *Distributed Mobility-Adaptive Clustering Algorithm* (DMAC, [Bas99]) represents such a distributed scheme. DMAC is executed on each node and enables the initial set up as well as the maintenance of cluster organizations, since it is able to *adapt* to changes in the network topology, e.g., due to mobility of the nodes or node additions and/or removals.

In DMAC, a node decides its own role, i.e., *cluster head* or ordinary *cluster member*, solely knowing its current *one-hop* neighbors. DMAC assumes that a message sent by a node is received correctly within a finite time by all of its neighbors. Furthermore, it assumes that each node knows

- its own unique device ID,
- its cluster weight,
- its role (if already selected), and
- the weight and the role of all of it neighbors (which is, in our case, the role of *all* nodes).

DMAC is a message-driven algorithm where a generic node v communicates via two different types of broadcast messages: `CH(v)` notifies all neighboring devices that this node obtains the cluster head role, while a `join(v, u)` message indicates that this node will be part of the cluster whose cluster head is node u. The DMAC algorithm is executed at each node in such a way that a node v decides its own role, i.e., cluster head or cluster member, based on its own weight and the weights of its neighbors. Initially, only nodes u with the biggest weight in their neighborhood will broadcast a `CH(u)` message to all of their direct neighbors to maintain the cluster head role, and wait for other nodes in the neighborhood to join their cluster. On receiving such a `CH(u)` message, a node v decides to join the neighboring cluster head u' with biggest weight and, thus, sends a `join(v, u')` message. If all of the neighbored nodes of v with a higher weight have sent a join message indicating they have joined another cluster head, then v itself becomes an additional cluster head and sends a `CH(v)` message. In the following, v waits until all of its neighbors have sent a join message and either joined its own cluster, or the cluster of another node. Then, v exits the algorithm. The cluster formation is finished when all nodes have exited the algorithm. A more detailed description of the standard DMAC algorithm is given by Basagni [Bas99].

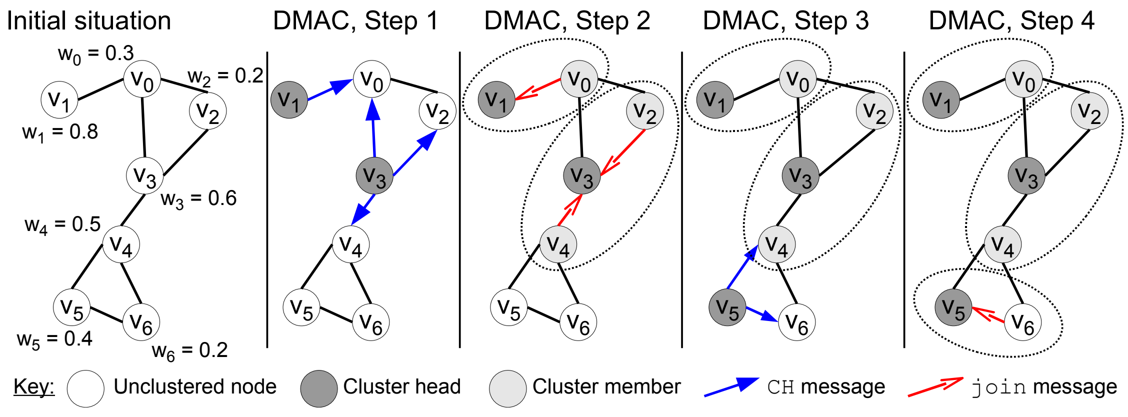


Figure 4.5.: Exemplary cluster creation with DMAC

As an example, consider the initial situation in Figure 4.5 with seven unclustered nodes v_0 to v_6 and the given edges between the nodes representing direct wireless connectivity. In the first step, nodes v_1 and v_3 notice they have highest weight within their neighborhood and, thus, send a `CH` message. In step 2, the node v_0 joins v_1, as v_1 has higher weight than v_3. Furthermore, v_2 and v_4 join v_3, and the nodes v_0 to v_4 finish the DMAC algorithm. Next in step 3, v_5 notices that v_4 (which has higher weight) has joined v_3. Now, v_5 is the node with highest weight within its neighborhood that is unclustered, and broadcasts a `CH` message. Finally in step 4, v_6 joins v_5, and the cluster structure is completed.

We decided to choose DMAC as general clustering algorithm, as it has been proven [Bas99] that DMAC

- is correct,
- terminates at each node (i.e., the node having selected its role) after the maximum of $D + 1$ steps in a network with diameter[1] D. In our work, this yields the maximum number of only two steps.
- requires each node to send *exactly one* message (a `CH` or a `Join` message),
- has a low message complexity of $O(n)$ in an environment with n nodes,
- guarantees that each node belongs to *exactly one* cluster, and
- does not produce clusters with neighboring cluster head.

[1]The diameter of a network determines the maximum number of hops between two arbitrary nodes.

We present the *clustering framework* we developed to enable the automatic selection of the cluster head and cluster member roles in Section 4.3.2. This clustering framework represents the basis for the automatic adaptation of the degree of decentralization in the application configuration. For centralized configuration, it only introduces one single cluster around the most powerful device. Later in Section 4.4, we extend the clustering strategy to enable multiple clusters for hybrid configuration with a balanced configuration load among the powerful devices, and we present cluster maintenance strategies to keep the balanced configuration load even in dynamically changing environments.

4.1.3. Introduction of a Pre-Configuration Process

In order to clarify the configuration process and the arising latencies, one needs to take a closer look on the interactions that take place within a configuration. To show the differences between decentralized configuration in homogeneous Ad Hoc environments and centralized configuration in infrastructure-supported heterogeneous environments, let us take a look on the interaction model of a typical decentralized configuration scheme, which is exemplarily shown in Figure 4.6.

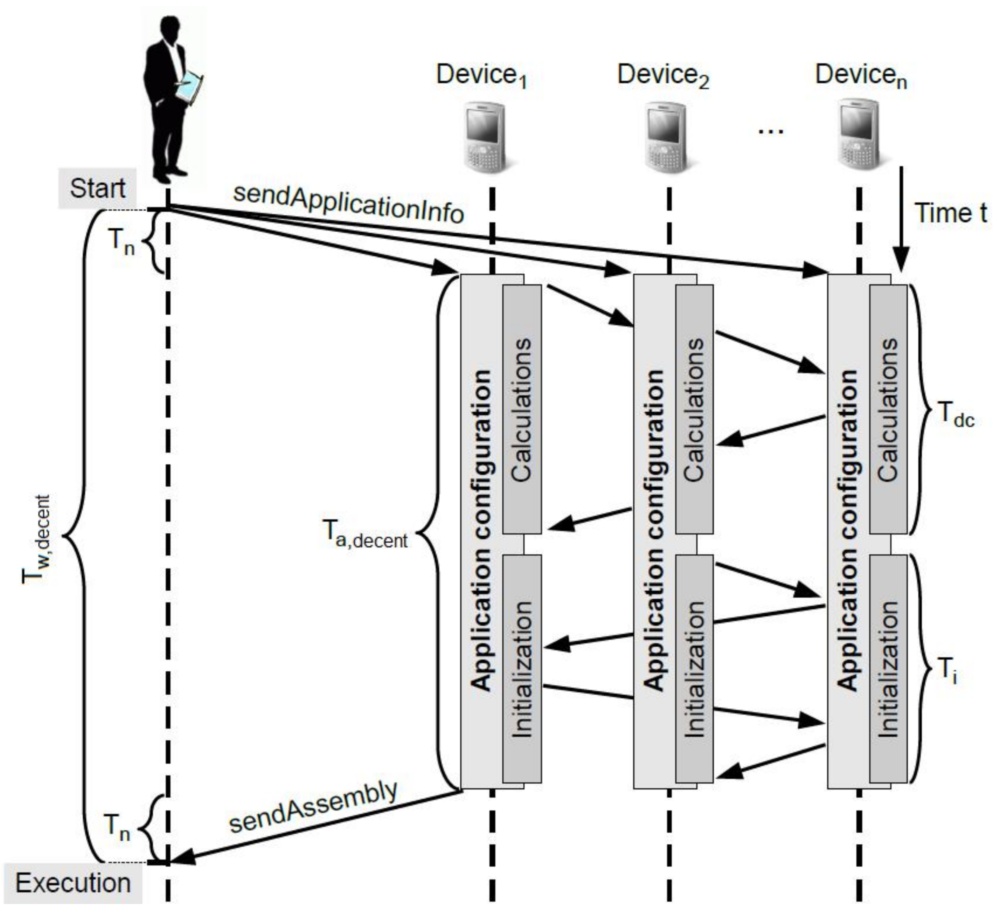


Figure 4.6.: Interaction diagram of decentralized configuration

A configuration process is initiated when the user starts an application on his or her mobile device. Then, the relevant information about the application, i.e. the application structure and information about the dependencies that have to be resolved, is automatically transmitted to all present devices. The arising latency is heavily depending on the wireless network and is denoted by T_n (*network latency*). In the following, the present devices cooperatively calculate a valid configuration in a distributed manner, i.e., using the algorithm proposed by Handte et al. [HBR05]. The corresponding latency is called *distributed configuration latency*, or T_{dc}. After the devices have found a valid composition, the initialization of the bindings between the different components is started, yielding the *initialization latency* T_i. Thus, the application configuration process (covering calculations and component initialization) lasts for $T_{a,decent} = T_{dc} + T_i$. Finally, the complete composition is transmitted to the user's device, causing again latency T_N as this transmission depends on the wireless network. Now, the application is executed and available to the user. Consequently, the overall *waiting time* $T_{w,decent}$ of a user which covers the time span between the user's application start and the can be expressed by the following equation:

$$T_{w,decent} = 2 \cdot T_n + T_{a,decent} = 2 \cdot T_n + T_{dc} + T_i \tag{4.4}$$

(Waiting time of decentralized configuration)

Switching from decentralized to centralized configuration, the corresponding interaction model changes, as Figure 4.7 shows. When the user starts an application that is to be configured in a centralized manner, then at first the most powerful device has to be found. Subsequently, the cluster around this device needs to be established, which takes the additional *cluster time* T_c. In the next step, the cluster head requests the resource information of the other devices in order to enable the local calculation of a composition on the powerful device. The time between the cluster head's request and the receipt of all resource information is denoted by the *resource retrieve time* T_r. After the weak devices have sent their resource information to the cluster head, this device calculates a valid composition, without involvement of the other devices. We denote the respective centralized configuration latency by the *centralized configuration time* T_{cc}. Then, the powerful device sends the configuration information to the other devices, which initialize the respective component bindings then. The final step is the same as in the decentralized approach: The assembly is sent back to the user's device, and the application is executed. Like before, we can easily determine the arising waiting time for the user as follows, where $T_{a,cent}$ denotes the changed latency for the centralized configuration process:

$$T_{w,cent} = 2 \cdot T_n + T_{a,cent} = 2 \cdot T_n + T_c + T_r + T_{cc} + T_i \tag{4.5}$$

(Waiting time of centralized configuration)

The local calculation on the powerful device does not involve message communication with the other weak devices. Furthermore, it is supposed that the powerful device calculates valid compositions faster than the weak devices, due to its significantly increased computation power, i.e. $T_{cc} < T_{dc}$. However, centralized configuration introduces the times T_c to establish the cluster and T_r to gather the resource information from the weak devices. So, the centralized configuration is only faster than the decentralized configuration (i.e., $T_{w,cent} < T_{w,decent}$) if $T_c + T_r + T_{cc} < T_{dc}$.

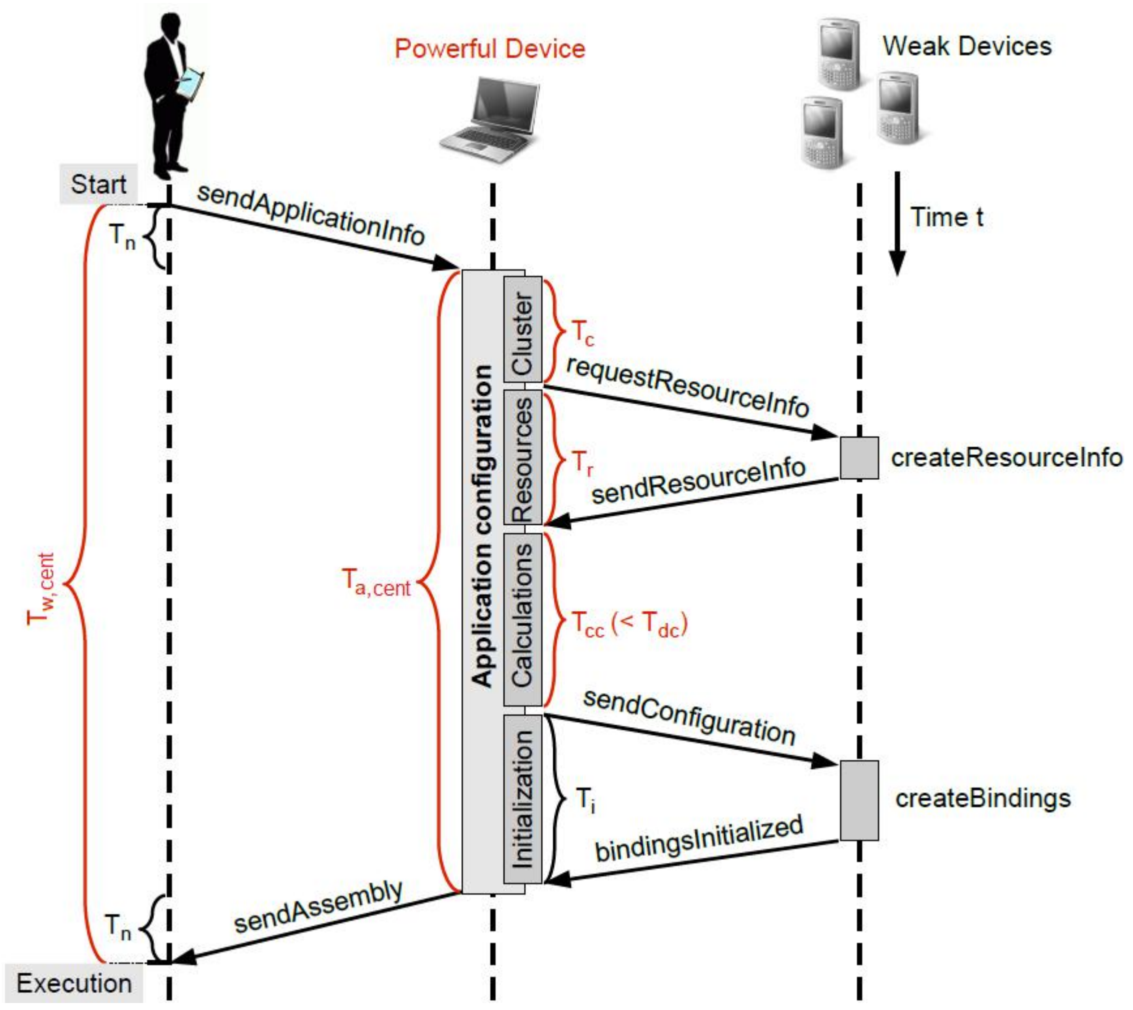


Figure 4.7.: Interaction diagram of centralized configuration

As the centralized configuration as shown above needs a powerful device to retrieve resource information (which introduces the additional time T_r), a main focus for a centralized configuration approach is to increase its efficiency by speeding up specific parts of the configuration. Therefore, we evaluate the arising latencies from Figure 4.7 according to their potential for speed ups:

- As the times T_n for transmitting information about the application and found compositions mainly depend on the bandwidth of the network connection, reducing these latencies is only possible if the wireless technology is changed, e.g., from IEEE 802.11b (11 Mbit/s) to 802.11g (54 Mbit/s) or 802.11n (up to 600 Mbit/s). We do not follow this approach here.

- As the interaction model in Figure 4.7 shows, the times T_c for establishing the cluster and T_r for retrieving the resource information of the weak devices are included in the overall latency T_w. However, the powerful device does not need to wait for the user's application start to request the resource information, but may already request this information *prior* to the configuration, i.e. when the weak devices come into communication range. The same holds for the retrieval of the resource information. So, an approach to increase efficiency of centralized configuration is to accomplish some configuration-relevant tasks *before* the user actually starts an application. We follow this approach in Section 4.3.5 and call the premature cluster establishment and resource retrieval a *pre-configuration process*.

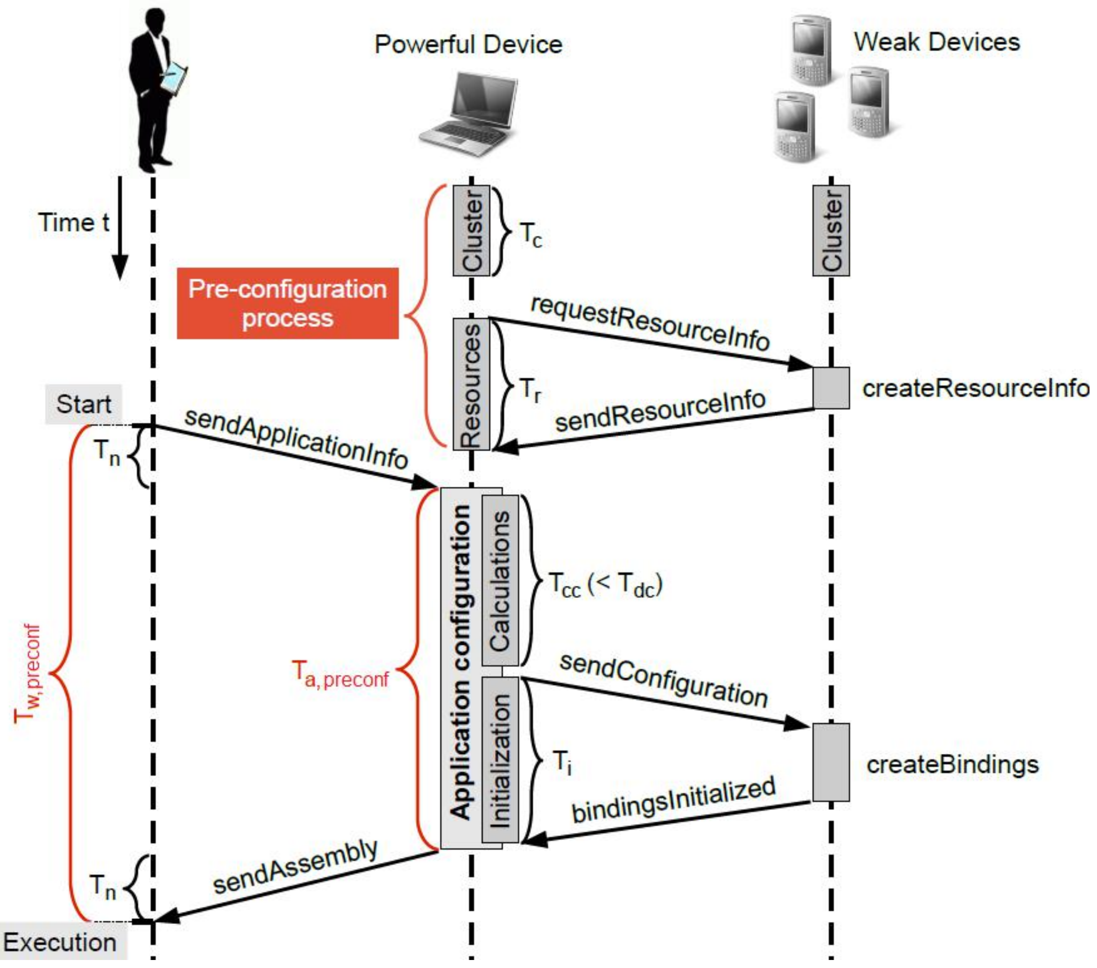


Figure 4.8.: Interaction diagram of centralized configuration with pre-configuration process

- The time T_{cc} for the actual configuration depends solely on the centralized configuration algorithm. Therefore, it is important to have an efficient configuration algorithm to be executed on the powerful devices, without involving thrashing effects or unnecessary subsequent adaptations within a configuration. For this purpose, we present *Direct Backtracking* as an efficient centralized algorithm in Section 4.2.

- Initializing the bindings between those components which have been determined as valid components is only depending on the determined composition and the location of the components among the present devices. Reducing this binding overhead by finding special configurations that involve many neighboring components on the application tree which are resident on the same device is left for future work, as Section 7.2.1 will briefly discuss.

So, the main challenge to reduce the overall latencies is to provide a mechanism that automatically loads the weak devices' resource information to the powerful devices, prior to a configuration, as Figure 4.8 shows. This reduces the overall waiting time to

$$T_{w,preconf} = 2 \cdot T_n + T_{a,preconf} = 2 \cdot T_n + T_{cc} + T_i \tag{4.6}$$

(Waiting time of centralized configuration with pre-configuration process)

So, centralized configuration that involves a pre-configuration process is always faster than centralized configuration without pre-configuration (i.e., $T_{w,preconf} < T_{w,cent}$) when no configuration information needs to be obtained at configuration time. Moreover, centralized configuration is also faster than decentralized configuration (i.e., $T_{w,preconf} < T_{w,decent}$) if $T_{cc} < T_{dc}$, which is – in compliance with the discussion we led above – typically the case in heterogeneous environments. To achieve such an efficient centralized configuration, we will take a closer look on how to proactively and automatically retrieve remote resource information by the powerful device in Section 4.3.5.

As introduced above, a hybrid configuration scheme has to combine the advantages of both decentralized and centralized configuration. Figure 4.9 shows the interaction diagram of the hybrid approach. As main difference to the centralized scheme, there are *several* powerful devices involved in the configuration. Like in centralized configuration, the powerful devices also perform a pre-configuration process, covering the establishment of several clusters (contrary to the centralized approach where only *one* cluster is established), and the retrieval of the resource information of the *mapped* weak devices. Furthermore, another difference arises at the configuration calculation stage, where the powerful devices cooperatively calculate a valid configuration. As this configuration is performed *in parallel* and *only* on the resource-rich devices without involvinh the weak devices, the configuration latency T_{hc} of the hybrid configuration is supposed to be lower than the configuration latency T_{cc} of the centralized configuration.

In consequence, the waiting time $T_{w,hybrid}$ for an application user when hybrid configuration with pre-configuration is used can be denoted as follows:

$$T_{w,hybrid} = 2 \cdot T_n + T_{a,hybrid} = 2 \cdot T_n + T_{hc} + T_i \tag{4.7}$$

(Waiting time of hybrid configuration with pre-configuration process)

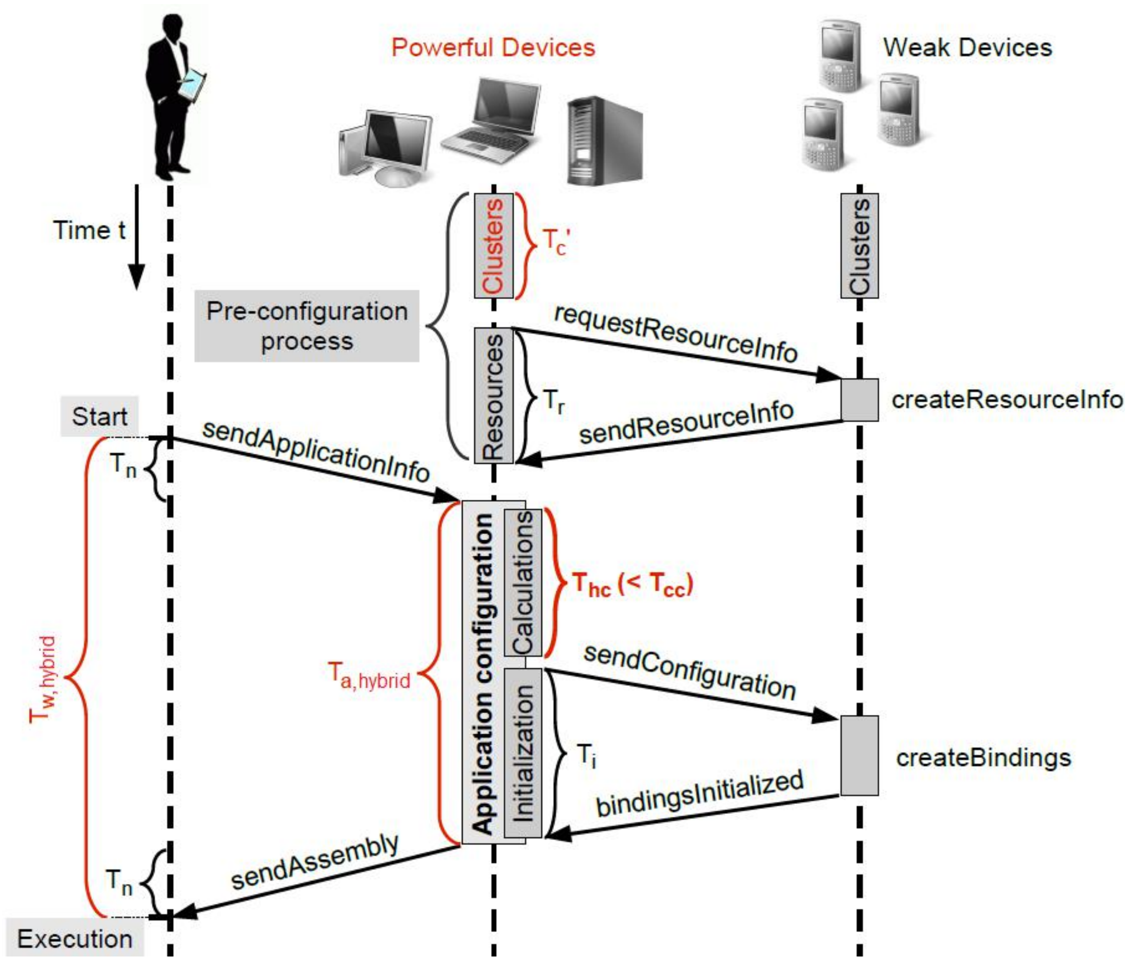


Figure 4.9.: Interaction diagram of hybrid configuration with pre-configuration process

Comparing $T_{w,hybrid}$ to the waiting time $T_{w,preconf}$ of centralized configuration with pre-configuration, the hybrid approach is faster iff $T_{hc} < T_{cc}$. In our evaluations in Section 4.5.3, we will prove this by comparing the latencies of decentralized configuration to those of centralized and hybrid configuration with implemented pre-configuration processes.

4.2. Centralized Application Configuration

In weakly heterogeneous environments with *exactly one* powerful device, we aspire to put the complete burden of a configuration to this device which then completely locally calculates a configuration using a centralized configuration algorithm. To execute centralized configuration processes in an efficient manner, we now present an algorithm called *Direct Backtracking (DBT)* [SHR08b], which is generally described in Section 4.2.1. Furthermore, we present two specific mechanisms of the algorithm to ensure its efficiency in application configuration and adaptation: *Proactive backtracking avoidance*, as described in Section 4.2.2, eliminates most of the adaptation processes from the start, while *intelligent backtracking* (Section 4.2.3) helps to render the remaining adaptation processes more efficiently. Furthermore, after having performed centralized configuration using Direct Backtracking, the configuration device distributes the found configuration assembly to those devices whose components are involved, and to the user's device where the application was started. Finally, after creating the bindings between the found components and the respective devices, the application is successfully executed.

We assume that a single device which performs centralized configuration and adaptation collects the relevant data beforehand. Therefore, the clustering mechanism which we describe in Section 4.3.2 is used. Thus, the configuration device can create an internal representation of the application tree that comprises dependencies, components, and incorporated devices with their available resources. After the configuration is complete, the configuration results have to be distributed to the respective devices whose resources represent components of the configuration.

While we solely focus on the actual centralized configuration calculation here, we will present a general framework to obtain the required knowledge about the resources and services as well as distributing the found configurations in a distributed environment in Section 4.3.

4.2.1. Approach

Like many other backtracking algorithms, DBT proceeds in a depth-first search manner: It starts with the root of the application tree and descends the contracts of the tree, from left to right, until it arrives at a leaf contract of the tree. Then, it tries to find a suitable component for that dependency using contract matching (i.e., it compares the demand of the parent component to the provision of the child component), and returns the identified child component to the parent component. 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). Figure 4.10 shows the depth-first ordered proceeding of DBT and one exemplary configuration in the application tree shown in Figure 2.2. In the following, we will show the DBT approach in detail on the basis of this example.

The DBT algorithm only needs to have access to the following information in order to be executed:

- The complete application tree with the interdependencies between the components in terms of contracts, and
- the available resources on the present devices that may provide the required functionalities to fulfill specific contracts.

The listings in this section illustrate DBT in Java-like pseudo code. DBT is method-driven and uses, besides several helper methods, six different main methods which are called within the configuration: `start`, `create`, `started`, `stopped`, `backtrack`, and `terminate`.

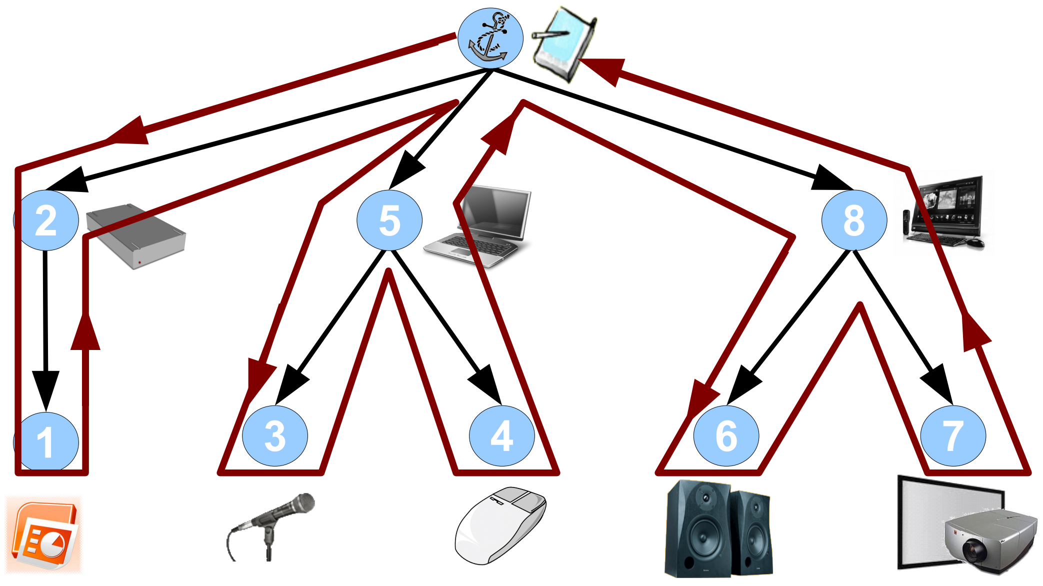


Figure 4.10.: General approach of Direct Backtracking

```
start(Component anchor) {
    Assembly ass = new Assembly();
    create(anchor.getChild(0, determOption()), ass);
}
```

Listing 4.2: Initial `start` method of DBT

The `start(anchor)` method (Listing 4.2) initiates the configuration process at the application anchor component, `anchor`. It creates a new assembly where each component that is determined within the configuration is added. Then, it calls a `create(cmp, ass)` method to resolve the first dependency of the anchor by finding a suitable child component `cmp` which then is added to the assembly `ass`. In this regard, the respective component `cmp` is determined by the helper method `getChild(i, j)`, which returns the `j`-th option of the `i`-th dependency of the current component. Initially, `i` is set to zero (as we want to resolve the first dependency of the anchor), and the selection of an option in case of a multi-optional contract is performed within the helper method `determOption` which tries to decrease the number of conflict situations that make a backtracking process inevitable. This fundamental mechanism of DBT implemented within `determOption` is called *proactive backtracking avoidance* and is described in detail in Section 4.2.2.

Figure 4.11 shows the start of the configuration process in the exemplary distributed presentation application. The anchor has instance ID (IID) [0] and is unique (i.e., there is only one suitable component on the device of the user who started the application), so the respective component ID (CoID) is [0,0] and the configuration starts within the `start([0, 0])` method. Then, a suitable component option for the child instance with IID [0][0] is determined within the `determOption` function. Let us suppose there is only one component – a hard disk where the presentation source file is stored – that fulfills the needs of the instance with IID [0][0], so this

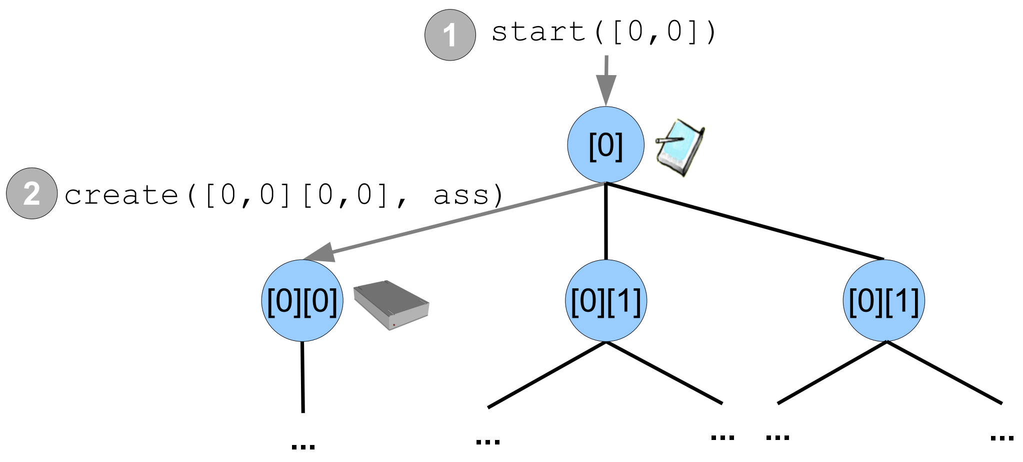


Figure 4.11.: Execution of initial `start` method and subsequent `create` method

contract is not multi-optional. Thus, the `create` method is called with parameters `[0,0][0,0]` (as CoID) and the current assembly `ass`, including only the anchor component so far.

```
create(Component comp, Assembly ass) {
    if (enoughResourcesAvailableForInstantiation())
        if (leafReached()) {
            ass.addComponent(comp);
            started(comp.getParent(), ass);
        }
        else
            create(comp.getChildComp(0, determOption()), ass);
    else
        stopped(comp.getParent(), ass);
}
```

Listing 4.3: `create` method of DBT

In the next step, the `create` method, as illustrated in Listing 4.3, is executed by the algorithm, trying to fulfill the needs of the current contract. If there are enough resources available to resolve a dependency (checked within the helper method `enoughResourcesAvailableForInstantiation` in line 2), two possibilities exist:

1. The component to be started represents an inner node, i.e., no leaf of the tree. In this case, the `create` function is called recursively for the first child contract to instantiate the components on the next lower level of the tree, thus following the depth-first search approach. In Figure 4.12, the steps 2 and 3 of the algorithm represent this situation: After having found the hard disk where the source file is stored as suitable component for the instance with IID [0][0], the configuration is continued with the first (and, in this case, only) child instance of [0][0], the instance with IID [0][0][0]. Thus, the `create` method is called recursively, but this time with a changed IID of [0][0][0] as first parameter.

2. The component to be started represents a leaf of the tree. In this case, the component is added to the current assembly, and the `started` method is called subsequently to indicate this instantiation of a leaf component to its parent component, which is obtained by `comp.getParent`. In Figure 4.12, when the `create([0,0][0,0][0,0])` method is called as third step of the algorithm, a leaf of the tree is reached, and as the found component – the presentation source file – fits the requirements of the application, the `started` method (which we will describe in the following) is called with the parent contract (with CoID [0,0][0,0] as parameter), indicating the successful configuration of the respective contract to the parent node.

The `started(comp, ass)` method, as displayed in Listing 4.4, indicates to the parent component `comp` of a contract that its child component (the caller of this method) was successfully started, and includes the updated assembly `ass`. Then, the parent component `comp` first checks if there are additional unresolved dependencies. In this case, the parent component tries to resolve the first of these remaining unresolved dependencies by calling the `create` method recursively, however this time with an incremented parameter for the dependency parameter (line 3). If there are no more unresolved dependencies available for this instance, then the algorithm proves if it has reached the anchor component. In this case, the configuration is complete and the `terminate` method is called with the included assembly `ass` to create the bindings between the components. Otherwise, the assembly is extended by the found component `comp`, and the `started` method is called again, with the parent component (`comp.getParent`) and the extended assembly as parameters. As an example for the recursive call of the `started` method, consider steps 4 and 5 of the algorithm in Figure 4.12, which indicate that components for the respective contracts have been found.

```
 1 started(Component comp, Assembly ass) {
 2   if (unresolvedDependenciesAvailable()) // resolve next dependency
 3     create(comp.getChild(comp.getDependency() + 1, determOption()), ass);
 4   else // no unresolved dependencies left
 5     if (comp == [0,0]) { // anchor reached?
 6       terminate(ass); // terminate successfully with found assembly
 7     }
 8     else { // go one level up in the tree
 9       ass.addComponent(comp);
10       started(comp.getParent(), ass);
11     }
12 }
```

Listing 4.4: `started` method of DBT

So far in the exemplary scenario, each dependency could successfully be instantiated, as a suitable component which provides all of the required functionality was found, respectively. However, in case that not enough resources are available to instantiate a contract at the moment, then the `stopped` function displayed in Listing 4.5 is called to indicate a resource conflict situation to the parent component. In

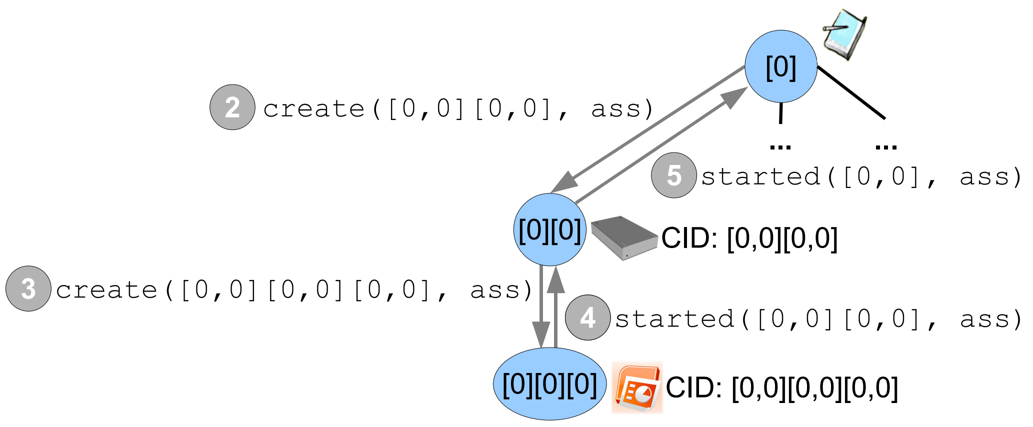

[0]
2 create([0,0][0,0], ass)
... ...
5 started([0,0], ass)
[0][0]
CID: [0,0][0,0]
3 create([0,0][0,0][0,0], ass)
4 started([0,0][0,0], ass)
[0][0][0]
CID: [0,0][0,0][0,0]

Figure 4.12.: Recursive calls of `create` method, and following recursive calls of `started` method

this case, DBT at first calls the helper method `alternativeComponentsAvailable` and checks if there are additional components that can potentially resolve this contract. If `alternativeComponentsAvailable` resolves to true, then the `create` method is called again, with the next component option as parameter. If no further option exists, a `backtrack` process has to be initiated. This function at first stops the adherent subtree of the selected backtracking component instance `toStop`, then stops the component `toStop` itself, and instantiates another component `toStart` (which is found by the already discussed helper method `determOption` and the previously stopped subtree components). `toStop` has been determined before within the helper method `findBacktrackContract` (line 5) that implements the intelligent backtracking process which is further described in Section 4.2.3. Afterwards, DBT re-tries to instantiate the component `comp.getOption` (line 11) which could not be instantiated previously.

```
 1 stopped(Component comp) {
 2     if (alternativeComponentsAvailable())
 3         create(comp.getChild(comp.getDependency(), determOption()));
 4     else { // backtrack needed
 5         Contract btContract = findBacktrackContract(); // intelligent backtracking
 6         if (btContract == null)
 7             terminate(null); // terminate unsuccessfully
 8         Instance toStop = btContract.getInstance();
 9         Component toStart = btContract.determOption();
10         backtrack(toStop, toStart);
11         create(comp.getChild(comp.getDependency(), comp.getOption()), ass);
12     }
13 }
```

Listing 4.5: `stopped` method of DBT

As an example, consider the situation illustrated in Figure 4.13a. At first in steps 6 and 7, the `create` method is called to resolve the dependencies for the

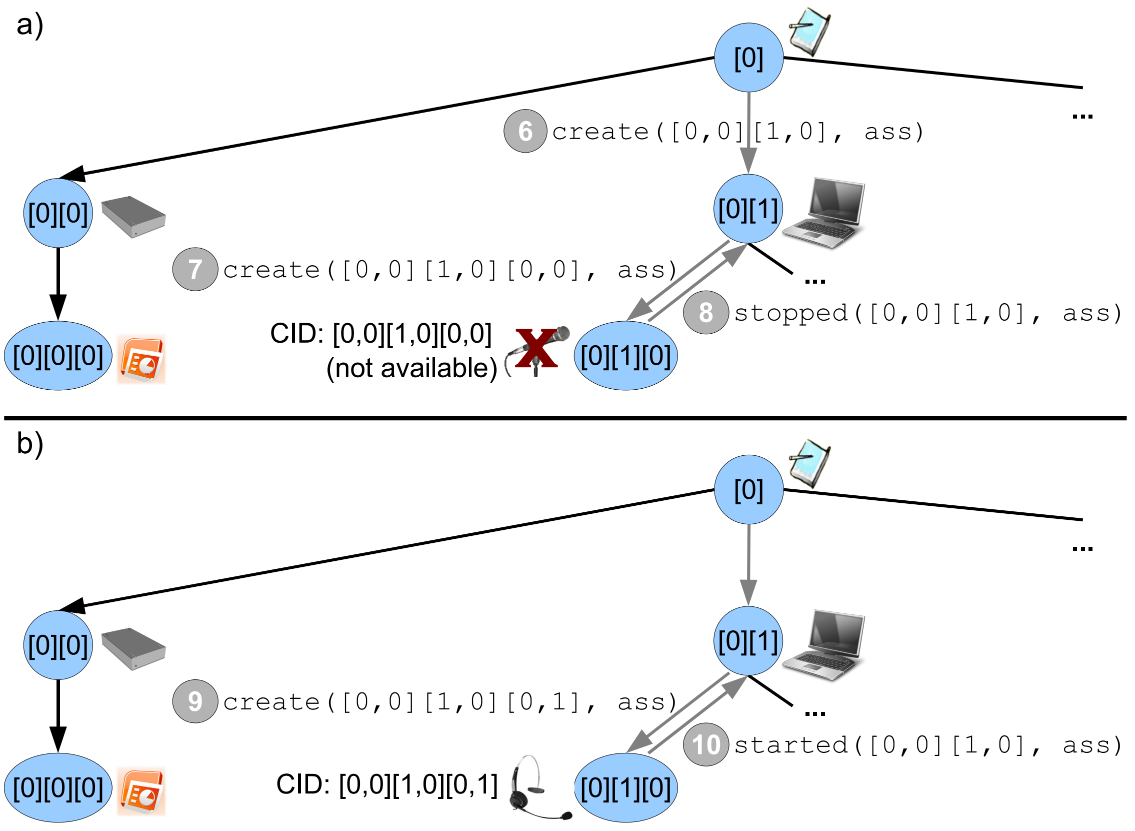

a)
[0]
6 create([0,0][1,0], ass)
...
[0][0]
[0][1]
7 create([0,0][1,0][0,0], ass)
...
8 stopped([0,0][1,0], ass)
[0][0][0]
CID: [0,0][1,0][0,0]
(not available)
[0][1][0]
b)
[0]
...
[0][0]
[0][1]
9 create([0,0][1,0][0,1], ass)
...
10 started([0,0][1,0], ass)
[0][0][0]
CID: [0,0][1,0][0,1]
[0][1][0]

Figure 4.13.: a) Call of `stopped` method of DBT due to unavailable component, b) Subsequent successful selection of alternative component

instances with IIDs [0][1] and [0][1][0]. However, the microphone component (CoID [0,0][1,0][0,0]) which is initially chosen for instance [0][1][0] is currently not plugged in and, thus, cannot be used by the application. So, the algorithm calls the `stopped` method to notify its parent instance [0][1] that another component has to be found for this dependency. Next in step 9, shown in Figure 4.13b, DBT tries to instantiate the alternative headset component with CoID [0,0][1,0][0,1]. As this component is available, the respective `started` method is called (step 10).

Here, the call of the `stopped` method increased the runtime of the algorithm, but as an alternative component was available, the configuration of this contract was successful at the second attempt. However, it may also be possible that none of the alternative components is currently available (lines 4ff. in Listing 4.5). In this situation, the `backtrack` (line 10) method is called to resolve this resource conflict by re-configuring (or adapting) a previously configured contract. Therefore, the algorithm determines the contract instance `toStop` that is stopped (line 8) and chooses an alternative component `toStart` (line 9) beforehand. After a successful backtrack, the algorithm re-calls the `create` method (line 11) to re-instantiate the component which could previously not be started because of the resource conflict.

The actual `backtrack` method is shown in Listing 4.6 and consists of the execution of the following four helper methods:

- `stopSubtreeInstances(toStop)`: As the adaptation of a contract instance `toStop` also requires the adaptation of all of its child contracts (to remain valid links between the chosen components), the complete subtree instances are stopped in a first step within this helper method. To minimize the number of subsequent adaptations, we will discuss an *intelligent backtracking* process in Section 4.2.3 which takes the size of the subtree into consideration when choosing a contract for adaptation.
- `stopInstance(toStop)`: After having stopped all instances of the subtree from instance `toStop` on, the instance itself is stopped to make the component available for the contract where the backtracking process was started.
- `startInstance(toStart)`: The component `toStart` – which was selected within the `stopped` method (cf. Listing 4.5) and represents an alternative component to the component whose instance was previously stopped within `stopInstance` – is instantiated. This guarantees that the functionality previously provided by the instance `toStop` is still supplied.
- `startSubtreeInstances(toStart)`: Finally, the subtree instances that have been stopped within the helper method `stopSubtreeInstances` need to be re-instantiated. However, as their parent component has been changed, the links between the components as well as the CoIDs of the subtree components need to be adapted.

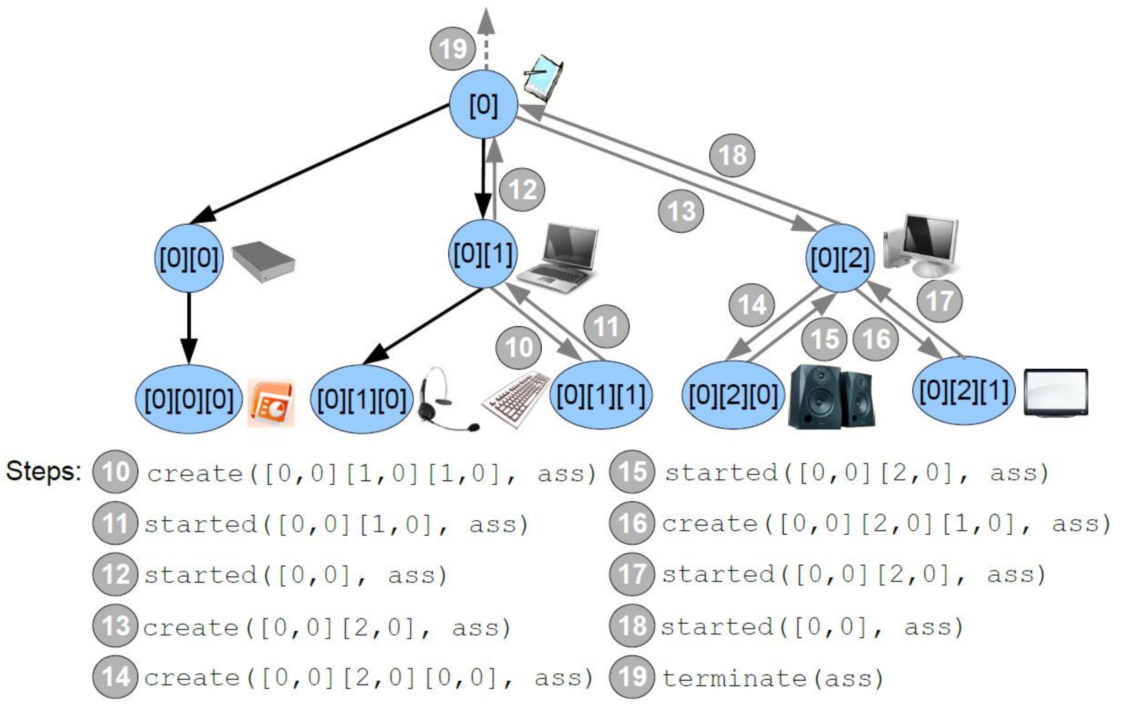
An example for the `backtrack` process is given in Section 4.2.3 where the intelligent backtracking is described.

```
backtrack(Instance toStop, Component toStart) {
    stopSubtreeInstances(toStop);
    stopInstance(toStop);
    startInstance(toStart);
    startSubtreeInstances(toStart);
    return;
}
```

Listing 4.6: `backtrack` method of DBT

In the last step of the algorithm, the `terminate(ass)` method is executed. Here, two possibilities exist:

- The provided assembly `ass` is `null`. This indicates that a valid configuration could not be found due to missing resources. In this case, the user is notified about the configuration failure.
- The provided assembly `ass` represents the found complete application configuration. Then, the devices are notified about the successful configuration by transmitting the found assembly to them, and creating the bindings between the determined component instances. After the bindings are established, the application is successfully configured and can be used.



```
10 create([0,0][1,0][1,0], ass)
11 started([0,0][1,0], ass)
12 started([0,0], ass)
13 create([0,0][2,0], ass)
14 create([0,0][2,0][0,0], ass)
15 started([0,0][2,0], ass)
16 create([0,0][2,0][1,0], ass)
17 started([0,0][2,0], ass)
18 started([0,0], ass)
19 terminate(ass)
```

Figure 4.14.: Termination of exemplary application configuration with DBT

```
terminate(Assembly ass) {
    if (ass != null) { // start application
        notifyDevices(ass);
        createInstanceBindings();
    }
    else // notify user of failure
        System.out.println(”Configuration not successful − missing resources!”);
}
```

Listing 4.7: Final `terminate` method of DBT

To finish the configuration of the exemplary application, Figure 4.14 shows the final steps of the configuration process. Here, we suggest that each of the requested components was available and, thus, no more `stopped` and `backtrack` calls were needed. The algorithm continues with the instance with IID [0][1][1], where we suppose that the keyboard component (CoID [0,0][1,0][1,1]) is available and is chosen for instantiation (step 10). Then, the `started` method for the instances with IIDs [0][1][1] and [0][1] is called (steps 11 and 12). Finally, the `create` and `started` methods for the right subtree are called (steps 13 to 18), and the algorithm successfully terminates by calling the `terminate` method in step 19.

4.2.2. Proactive Backtracking Avoidance

In case a contract for a certain dependency is multi-optional, the selection of a component has to be made cautiously in order to avoid conflict situations right from

Index	Component ID	Available Resources	Container ID
0	C_0	α_0	c_i
1	C_1	α_1 $(< \alpha_0)$	c_j
2	C_2	α_1	c_k $(> c_j)$
...	...	...	...
$n-1$	C_{n-1}	α_m $(< \alpha_{m-1})$	c_l

Table 4.2.: Ordered list for specific multi-optional contract with n options

the start and reduce the number of situations in which backtracking is necessary. For this purpose, Direct Backtracking contains a proactive mechanism which carefully selects the component option to be instantiated in order to avoid backtracking.

Within multi-optional contracts, options are ordered in a list according to their resource availability: The component with most available resources has highest priority. If there exist multiple options with the same number of available resources, they are additionally ordered according to their container ID, i.e., a component on a container with a lower ID has higher configuration priority. Table 4.2 shows the structure of the list which is descendingly ordered by the globally available amounts α_i of the resources and, as a secondary criterion, by the ascending container IDs c_i.

```
determOption() {
Contract ctc = getCurrentContract();
boolean looping = true;
int i = 0; // Index of selected component in list
while (looping) {
    comp = option[i];
    while (comp.consumed >= comp.getContainer().getAvailableResources()) {
        i++;
        if (i > list.length)
            performBacktracking(); // no suitable component found
        else
            comp = option[i];
    }
    if (freeResources >= comp.consumed)
        looping = false;
    else {
        i++;
        if (i > list.length)
            performBacktracking(); // no suitable component found
    }
}
return comp;
```

Listing 4.8: Pseudo-code of DBT's `determOption` helper method

Listing 4.8 is a pseudo-code presentation of Direct Backtracking's Proactive Backtracking Avoidance, which decides in favor of a specific component option and tries to proactively avoid backtrackings. As one can see, Direct Backtracking performs the following steps on the ordered list of options in the given order:

1. Initially, DBT selects the first component option in the list to be instantiated, i.e., the highest-prior component with most available resources.
2. If the currently selected component consumes more than a defined threshold of the currently unassigned amount of a resource R, the algorithm scans the ordered list of options (`option[]`) for alternatives. Then, the component with highest priority is selected. This reduces the conflict potential for the contracts which have not yet been configured.
3. For the currently selected component, DBT verifies that there are enough global resources remaining (variable `freeResources`) to fulfill all missing contracts in theory after the initialization of the selected component *comp*. This means that there have to exist at least i free components among **all** devices with sufficient resources to fulfill the i dependencies of the application that have not been resolved yet. Otherwise, *comp* is not instantiated at this moment, as this would yield a future inevitable backtrack process. In this case, the algorithm selects the next lower-prior option in the ordered list and continues with step 2. If no further option is remaining, the intelligent backtracking process, as described in the following, is performed by calling `performBacktracking()`.

As an example, consider the situation depicted in Figure 4.15. The figure represents a part of an application that deals with visual output components. The component instance with IID [0][0] represents the required Graphical User Interface (GUI) for the user with which he or she can control the output components for displaying the presentation slides (IID [0][0][0]) and the additional video support (IID [0][0][1]). As it can be seen in the table within Figure 4.15, there are four different output components available in the environment: a laptop display (device ID 0), a standard PC monitor (device ID 1), and two video projectors (device IDs 2 and 3). Furthermore, let us consider that additional restrictions such as a minimum size of the display component or a minimum resolution restrict the fulfillment of specific contracts, leading to the possible options for specific dependencies as depicted in the figure.

In this situation, Direct Backtracking would initially priorize the component options as described above. Since there are two video projectors, but only one PC monitor available, the use of a video projector as component for the instance with IID [0][0] is priorized over the use of the PC monitor. Thus, DBT creates the video projector instance for [0][0] and continues with instance [0][0][0]. For this instance, there are one laptop display and one video projector available. Thus, DBT chooses according to the container IDs: As the container ID of the laptop is lower than the container ID of the PC, DBT selects the laptop display as instance for [0][0][0]. In the following, DBT uses the still available PC monitor as component instance [0][0][0] and does not need to perform any backtrackings here.

Now, regard Figure 4.17 which illustrates the effects of the related *Synchronous Backtracking (SBT)* algorithm in the same situation. SBT only uses the container

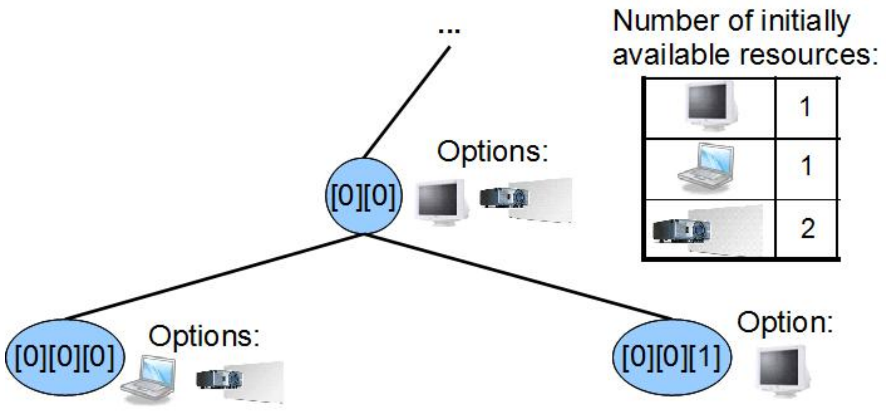


Figure 4.15.: Proactive backtracking avoidance: Initial situation

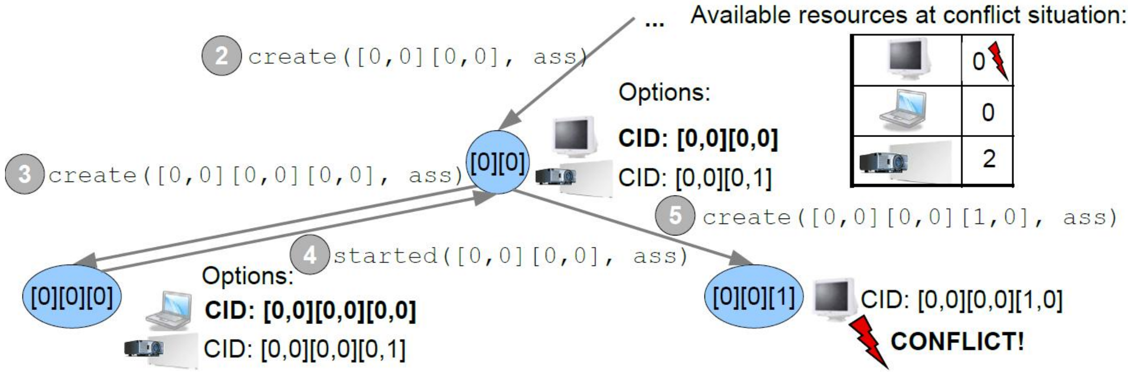

...
Available resources at conflict situation:
0
0
2
2 create([0,0][0,0], ass)
Options:
CID: [0,0][0,0]
CID: [0,0][0,1]
3 create([0,0][0,0][0,0], ass)
[0][0]
5 create([0,0][0,0][1,0], ass)
4 started([0,0][0,0], ass)
[0][0][0]
Options:
CID: [0,0][0,0][0,0]
CID: [0,0][0,0][0,1]
[0][0][1]
CID: [0,0][0,0][1,0]
CONFLICT!

Figure 4.16.: Proactive backtracking avoidance: Proceeding of DBT

IDs and does not priorize over additional criteria. As the container ID of the PC is lower than the one of the devices where the video projectors are connected to, the PC monitor is selected as component for the instance with IID [0][0]. Regarding instance [0][0][0], SBT simply chooses the laptop. However, SBT is not able to instantiate the PC monitor for instance [0][0][1] now, since this component has already been used within instance [0][0]. As no alternative component option is available at the moment, this represents a conflict situation where SBT needs to perform backtracking: It adapts the next higher- prior instance, which is (due to the depth-first approach) [0][0][0]. Thus, the laptop display component is stopped, and one of the video projectors is instantiated. Then, it retries to instantiate the PC monitor. However, since the PC monitor is still not yet available, another backtracking is required: SBT adapts instance [0][0] by stopping the PC monitor and starting the video projector. Then, the PC monitor can be used as instance [0][0][1]. In summary, SBT required two adaptation processes to start the three needed visual output components, leading to significantly increased latencies compared to DBT which did not need any adaptation at all. An evaluational comparison of SBT and DBT is given in Section 4.5.2.

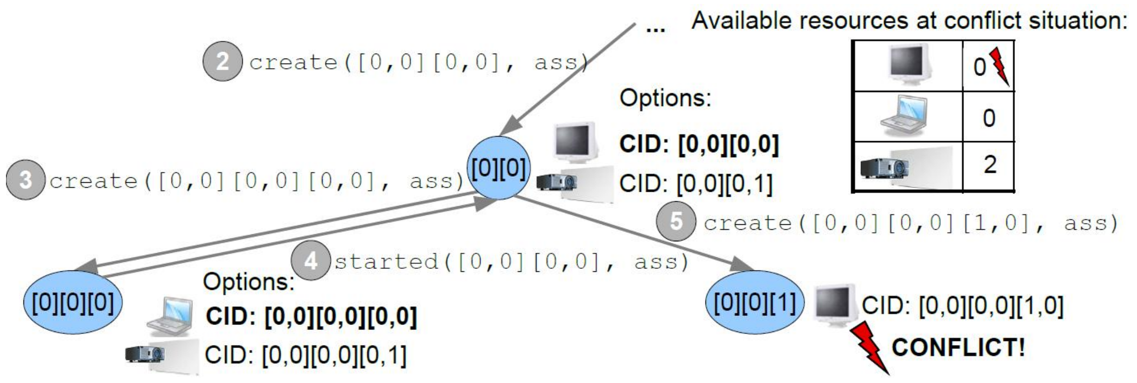

...
Available resources at conflict situation:
0
0
2
2 create([0,0][0,0], ass)
Options:
CID: [0,0][0,0]
CID: [0,0][0,1]
[0][0]
3 create([0,0][0,0][0,0], ass)
5 create([0,0][0,0][1,0], ass)
4 started([0,0][0,0], ass)
[0][0][0]
Options:
CID: [0,0][0,0][0,0]
CID: [0,0][0,0][0,1]
[0][0][1]
CID: [0,0][0,0][1,0]
CONFLICT!

Figure 4.17.: Proceeding of SBT requires backtracking

4.2.3. Intelligent Backtracking

Direct Backtracking is able to avoid most backtracking processes by proactive backtracking avoidance, which was described above. However, there may still be situations where none of the possible components for a contract Ctr_0 are available and, thus, backtracking has to be initiated. This means that another contract must be found whose adaptation yields the availability of the required resource. If there are several multi-optional contracts that have already been instantiated within the current configuration process, there is more than just one candidate for an adaptation. In such a case, DBT performs intelligent backtracking by carefully selecting a contract whose components can be adapted with little overhead.

The general approach of DBT's intelligent backtracking mechanism is as follows:

1. First, determine all components of Ctr_0 which could not be instantiated because of a missing resource R, and all devices D which host a component that consumes a specific amount of R.
2. Then, determine the set S of contracts on the devices found in step 1 that have been configured before by DBT and consume an amount of R on D. Steps 1 and 2 are performed to reduce the number of possible backtracking goals.
3. Subsequently, order the set S of contracts found in step 2 in descending order according to the amount of R they consume. Due to this specific order, we aspire a considerable deallocation of resources when backtracking is performed in step 4.
4. Adapt the first contract in S, i.e., perform backtracking.
5. Re-try to configure Ctr_0. If the configuration now is possible, intelligent backtracking was successful and DBT continues by configuring the next contract of the application. Otherwise, remove the first contract from S, and re-execute step 4.
6. If step 4 could not be performed for *any* contract in the set, DBT terminates unsuccessfully and notifies the user of scarce resources.

We will describe this approach more formally now. First, let us assume that every possible component Cmp_i ($i \in \{1, ..., m\}$) of the relevant contract Ctr_0 could not

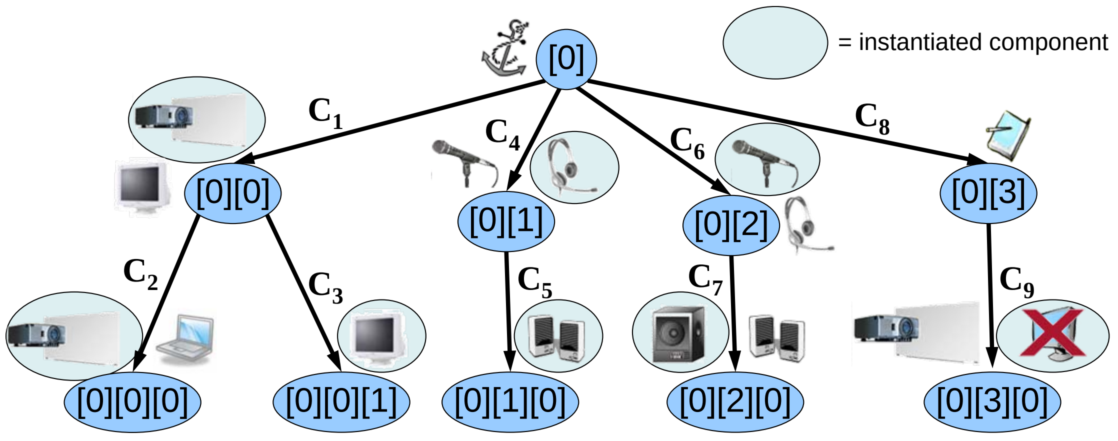


Figure 4.18.: Initial situation for backtracking

be instantiated due to a shortage of a specific resource R. Thus, Cmp_i form a set S_1 of m components, i.e.,

$$S_1 = \{Cmp_1, Cmp_2, ..., Cmp_m\} \tag{4.8}$$

If Ctr_0 is *not* multi-optional, then $m = 1$, i.e., S_1 includes only one component. Now, DBT determines the set S_2 of n devices

$$S_2 = \{D_1, D_2, ..., D_n\} \tag{4.9}$$

which host at least one component Cmp_k that is included in S_1. More formal, this condition can be described by the following equation:

$$\exists k \in \{1, ..., m\}, l \in \{1, ..., n\} : (Cmp_k \in S_1 \wedge Cmp_k \in_h D_l) \Rightarrow D_l \in S_2, \tag{4.10}$$

whereas $Cmp_k \in_h D_l$ denotes that device D_l hosts component Cmp_k.

Subsequently, DBT determines a third set S_3 of p multi-optional contracts

$$S_3 = \{Ctr_1, Ctr_2, ..., Ctr_p\} \tag{4.11}$$

This set contains only those multi-optional contracts Ctr_k ($k \in \{1, ..., p\}$) for which the following three conditions hold:

1. Ctr_k is of higher priority than Ctr_0. This means that Ctr_k has already been configured by DBT due to its depth-first approach.
2. The currently selected component Cmp_k of Ctr_k is resident on a device D_k which is included in S_2, i.e., $Cmp_k \in S_1$.
3. There exists an alternative component on another device D_y that can be instantiated *now* due to sufficient free amount of R on D_y.

The contracts in S_3 are ordered descendingly according to the amount of resource R that is consumed by the instantiated component. This means that the instantiated component which consumes the largest amount of R is at the beginning of the list, because its termination would cause a considerable deallocation of resources. This

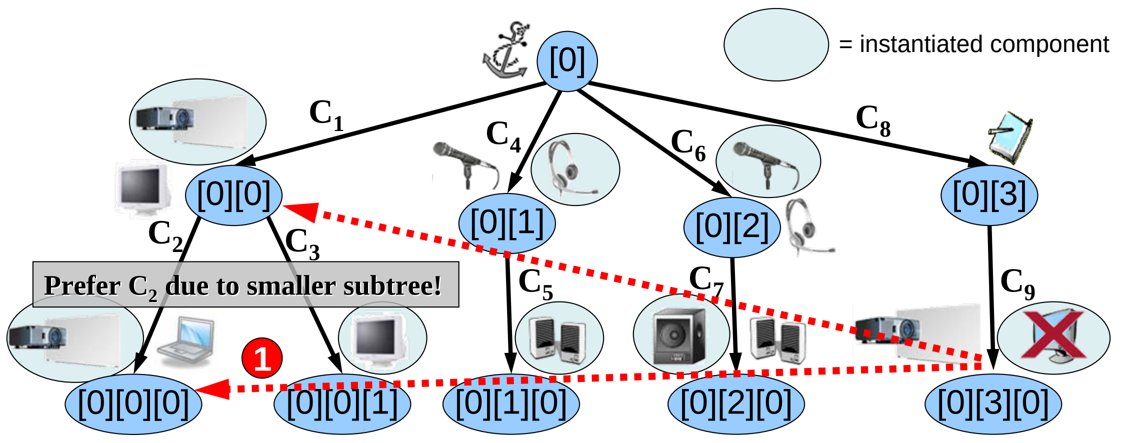


Figure 4.19.: Intelligent backtracking of DBT

helps to decrease the number of needless adaptations which would have to be revised later. If only one suitable component exists, the backtracking target is obviously found and the adaptation process can be initiated.

In case of more than one suitable backtracking targets that consume an identical amount of R, an additional selection criterion is necessary for weighting them according to their suitability for adaptation. Since adaptation is simpler for contracts with small adherent subtrees (as the subtree also has to be adapted), DBT selects the component C that has least descendants (number of all successors down to the leaves) and, hence, is closest to the bottom of the tree. Thus, contracts with little adaptation overhead are preferred. In case of multiple contracts with a subtree of the same size, the algorithm randomly selects one of these contracts and adapts this contract.

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, this indicates that there are not sufficient resources in the environment. Thus, the algorithm terminates unsuccessfully within the `stopped` function and notifies the user of this failure.

To clarify how DBT proceeds in intelligent backtracking, consider the exemplary application shown in Figure 4.18. Here, all contracts have successfully been configured by selecting the highlighted components. However, the PC monitor used as component instance [0][3][0] for contract C_9 suddenly fails, leaving this contract unconfigured. Besides the PC monitor, the introduced set S_1 includes a video projector as alternative component. However, as the two video projectors are currently used within other contracts (C_1 and C_2), the configuration has to be adapted, i.e., *backtracking* has to be performed.

In this situation, Direct Backtracking first determines the set S_3 of multi-optional contracts with higher priority which have currently instantiated a component that is able to resolve the dependencies required by C_9. As the instances [0][0] for C_1 and [0][0][0] for C_2 are currently represented by the two video projectors (which form set S_2), these two contracts are possible backtracking target for Direct Backtracking, as it can be seen in Figure 4.19. Since instance [0][0][0] does not have any child

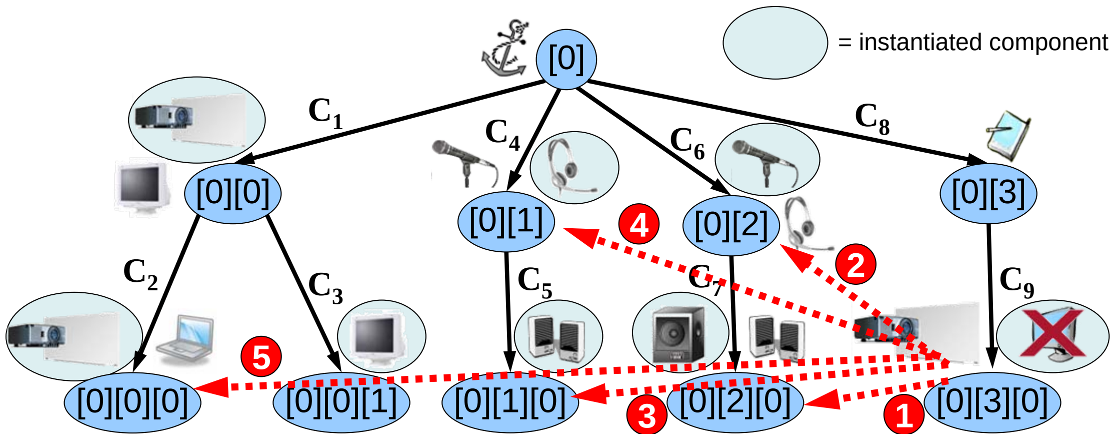


Figure 4.20.: Standard backtracking of SBT

components at all, DBT selects C_2 as backtracking target, stops the use of the instantiated video projector, and subsequently starts to bind the laptop display component to dependency C_2. Then, DBT returns to instance [0][3][0], where the video projector is available now. In the following, the adaptation is completed, and the application is available to the user again.

Now, regard how Synchronous Backtracking (SBT) acts in the same situation, as depicted in Figure 4.20. SBT does not incorporate the cause of backtracking into its considerations, so it simply adapts the nearest multi-optional contract according to the depth-first approach (i.e., the multi-optional contract with highest ID) to resolve the resource conflict. Therefore in this situation, SBT initially adapts the multi-optional contract C_7 by stopping the currently instantiated loudspeaker component and instantiating the alternative desktop speakers. Obviously, this does not help to resolve the resource conflict with the visual output components, so SBT has to perform another backtracking. This time, contract C_6 is adapted, which is however also not resolving this conflict. In the following, SBT unsuccessfully tries to resolve the resource conflict by adapting contract C_4, until the subsequent adaptation of C_2 finally represents the solution to the resource conflict problem. Summarizing, SBT needs four adaptations, which yields significantly increased adaptation latencies compared to the single adaptation that DBT has to take. Section 4.5.2 presents more extensive evaluations which compare DBT to SBT.

4.3. A Framework for Adapting the Degree of Decentralization

In heterogeneous environments, approaches alternative to the decentralized one may exploit the computation resources more efficiently. Therefore, concepts to support algorithms with various degrees of decentralization in their calculations, as well as a mechanism to automatically switch between the provided approaches is needed. In approaches different to the decentralized one, specific devices need to obtain information about the available resources and services of remote devices. If this information is not obtained before runtime, it increases the configuration latencies, since the configuration device(s) cannot start the configuration process before. To increase efficiency of non-decentralized configuration, we suggest the introduction of a *pre-configuration process* that is performed in time periods prior to configuration calculations and reduces the configuration latencies effectively [SHR08a].

Thus in this section, we first discuss the requirements for a framework to enable the automatic adaptation of configuration algorithms in Section 4.3.1. Then, we introduce a clustering framework (Section 4.3.2) to identify different device types and establish unique clusters. Following in Section 4.3.3, we introduce a strategy which selects the node weights in a resource-aware manner. Then, we discuss the initial cluster formation as well as the cluster maintenance by using an adapted version of the DMAC algorithm in Section 4.3.4. Next in Section 4.3.5, we present the VC concept which enables the local emulation of remote devices and is used by the centralized and hybrid configuration schemes. This enables the exclusion of the weak devices from configurations. Finally, we present a simple algorithm to automatically switch between different configuration approaches in Section 4.3.6.

4.3.1. Requirements

At time the research for this thesis started, the system software PCOM only supported static distributed configuration of pervasive applications, because the focus lay on peer-to-peer based homogeneous Ad Hoc scenarios in previous research efforts. In order to dynamically support heterogeneous environments, a framework is needed to enable the adaptation of the degree of decentralization and the automatic selection of the most suitable configuration algorithm in a specific environment. According to our system model from Section 2.1 and the problem analysis discussed in Section 2.2, we pose the following requirements to such a framework:

Information retrieval: Typical clustering schemes require specific information to guarantee their operability, like the ID, the computation resources, or the remaining battery capacity of the devices. The respective information for specific clustering strategies is deduced in Section 4.3.2. The framework needs to have automatic access on this information to enable the strategy-specific creation of the cluster topology.

Resource efficiency: The framework has to support powerful as well as weak devices. This means that the framework has to be runnable on weak devices with restricted functionalities by posing minimal requirements on the devices' capabilities, i.e., by only relying on the Java 2 ME Connected Limited Device Configuration Profile (CLDC) profile[2]. However, the framework also needs to exploit the increased functionality of powerful devices, e.g., class loaders to support the dynamic and automatic transfer of remote classes. Generally, the implemented clustering strategies

[2] CLDC represents the J2ME configuration with minimum requirements. CLDC does not provide floating point operations. Devices only need to allocate 160 kB of RAM for the Java Virtual Machine, guaranteeing the support of a broad spectrum of devices.

of the framework have to consider the restricted capabilities of the weak mobile devices, e.g., slow mobile CPUs.

Scalability: Besides small applications which are typically used in Ad Hoc environments, larger-sized applications in the magnitude of those discussed in Section 2.1.2 also have to be supported. For this purpose, the space and communication overhead of the framework have to be as low as possible.

Adaptivity: The framework has to be able to react on dynamic changes in the network topology and the availability of specific devices. Therefore, the framework has to be designed in a way that it supports cluster changes by adapting the cluster structure and automatically providing the relevant configuration information to newly arriving configuration devices.

Re-use of previously developed applications: The framework needs to be seamlessly integrated into PCOM in order to enable the re- use of applications which have been developed before.

Transparency: The distribution of the application configuration among the present devices has to be performed automatically and without user interaction. Alike, the cluster formation and maintenance also need to be performed independently from the user. Therefore, the framework needs to be transparent to the user.

4.3.2. Clustering Framework

For centralized and hybrid configuration approaches, we establish a clustering framework to identify devices that perform the actual configuration and act as cluster heads (i.e., coordinators) for the configuration process. As basic clustering algorithm, we rely on the DMAC algorithm introduced in Section 4.1.2 where devices communicate via the mentioned `CH` and `join` messages. DMAC establishes clusters where each cluster member has at least one cluster head as neighbor and affiliates with the cluster head that has highest weight.

To achieve applicability of DMAC in our context, we need to provide information that is relevant for this algorithm, which is the node IDs, the cluster weights, and the roles of each node. While a node achieves its own role directly by the algorithm, the remaining required knowledge is obtained as following:

- **Node ID:** A node obtains its unique ID at node initialization by relying on the MAC address of its wireless communication interface.
- **Cluster weight:** Basagni provides a generalized algorithm for establishing clusters based on the nodes' weights. However, he does *not* provide clustering strategies which assign specific weights to the nodes. As the cluster weights are critical for selecting the role of a node within a cluster, they should be

chosen by taking into account the resources, since only the powerful devices should become cluster heads. Therefore, we introduce a *benchmarking process* to determine the cluster weights of nodes in a resource-aware manner. This process is described in more detail in Section 4.3.3.

- **Weight and role of neighbors:** We use BASE's Lease mechanism [BSGR03] which defines that each node periodically has to broadcast a heartbeat message to communicate this node is still alive. Therefore, the heartbeat message is extended so that it additionally transmits the cluster weight and the role of a node (as a simple boolean variable which distinguishes between cluster member and cluster head) to all other nodes.

However, as we focus on environments where all devices are directly connected to each other (cf. Sections 2.1.1, 4.1.2), i.e., all nodes are neighbors and the network's diameter D is 1, relying on pure DMAC would always lead to exactly *one* cluster head, which is the node with highest cluster weight among all nodes. This may be the fitting scheme for weakly heterogeneous environments where exactly one powerful device is available, but needs to be adapted for strongly heterogeneous environments where *multiple* powerful devices are available. To achieve multiple cluster heads with high computation power in a one-hop environment, we introduce a benchmarking process to determine the cluster weights taking into account the devices' resources, in combination with a threshold for the weights that differentiates betweeen cluster heads and cluster members: Devices with weights above the threshold declare themselves as cluster heads, while the other devices become simple cluster members and have to be *mapped* to one of the cluster heads. Mapping means assigning a weak device to exactly one strong device which then is responsible for configuring this weak device's components.

We discuss the initial weight selection, the cluster formation, and the cluster maintenance in weakly heterogeneous environments in the following paragraphs. Furthermore, we introduce a modified version of DMAC which enables multiple clusters with balanced configuration load among the cluster heads within a one hop environment in Section 4.4. This scheme represents the base for the hybrid configuration.

4.3.3. Resource-Aware Weight Selection

In Section 2.2.2, we have stated that one of our main goals is to the minimize configuration latencies by efficiently exploiting the scenario heterogeneity. To achieve this, we aspire to select only those nodes as cluster heads which have high computation power. In particular, the configuration devices have to be especially performant in calculating application configurations. Therefore, the cluster weight of a device has to be assigned according to the computation power of this device. After having performed this benchmark, a device v_i can decide its role in the cluster by comparing its weight w_i to a pre-defined *threshold weight* w_{th}: If $w_i \geq w_{th}$, then v_i becomes a cluster head; otherwise, it becomes an ordinary cluster member.

Therefore, we introduce a benchmarking process which is performed when the system software is started on a device to determine its computational power: Given an

Device Type	**CPU Speed**	**RAM**	**Time in** $1/t_{min}$	**Weight**
Desktop PC	8x 4.2 GHz	8 GB	1.0	1.0
Desktop PC	4x 3.0 GHz	6 GB	1.37	0.730
Desktop PC	2x 2.9 GHz	4 GB	1.588	0.630
Laptop	2x 2.4 GHz	2 GB	1.736	0.576
Laptop	2x 2.0 GHz	2 GB	1.901	0.526
Netbook	1.6 GHz	1 GB	4.684	0.213
Smart Phone	667 MHz	256 MB	7.705	0.130
Smart Phone	520 MHz	64 MB	8.405	0.119
PDA	400 MHz	128 MB	9.401	0.106
Mobile Phone	166 MHz	32 MB	16.721	0.060

Table 4.3.: Benchmark Results

application tree with a number of dependencies that have to be resolved and a set of resources which can resolve the dependencies, the device calculates a valid configuration using the centralized greedy heuristic introduced by Handte et al. [HHS+07]. To make this benchmark more realistic, three different abstract applications of different sizes (7, 15, and 31 components) which are typical for the scenarios we rely on (cf. Section 2.1.2) have to be configured within this benchmark. Furthermore, the applications are constructed in a way that, with the available resources, the greedy algorithm has to perform some re-configurations and, thus, backtracking processes within the configuration process.

Benchmark processes have been performed with different types of devices that are typical for heterogeneous environments, and the overall configuration times have been measured. Table 4.3 gives an overview of the benchmarks. In this table, the overall latencies are given relative to the latency t_{min} of the fastest configuration device, in our case a fast desktop PC with octacore CPU and 8 GB of RAM. To achieve that devices with lower benchmark configuration latencies get higher weights, the weight w_i of a device v_i with benchmark latency t_i are calculated as follows:

$$w_i = (\frac{t_{min}}{t_i})^{\alpha}, \tag{4.12}$$

where α is a freely selectable scaling factor. For the values given in Table 4.3.3, the scaling factor was set to $\alpha = 1$. From the results shown in the table, it is obvious that the weight selection criterion given in Equation 4.12 in combination with $\alpha = 1$ is suited to reliably distribute the cluster weights, since powerful devices such as desktop PCs or laptop get weights close to 1, while weak mobile devices such as netbook or smart phones are assigned weights that are lower than 0.3. As threshold weight, we suggest to use $w_{th} = 0.5$. In all of our measurements, this yielded cluster heads which were significantly more powerful than the cluster members [SHR08a].

4.3.4. Cluster Formation and Maintenance in Weakly Heterogeneous Environments

As mentioned in Section 4.3.2, we rely on the DMAC algorithm to establish and maintain clusters in dynamic environments. In the initial situation where the cluster is not yet established, a node v has two possibilities: if it determines that none of its neighbored nodes has a higher cluster weight, it assigns itself the cluster head role and broadcasts a `CH(v)` to all of its neighbors. Otherwise, it determines its neighbor u with highest cluster weight and sends a `join(u, v)` to join the cluster of u. Proceeding like this, a unique cluster structure with exactly one cluster head is established in a weakly heterogeneous one-hop environment with one strong and several weak devices. The cluster head maintains a complete list which contains all devices that are mapped as cluster members to itself, while the cluster members only need to maintain the ID of their cluster head in order to transmit changes in their resource conditions to their cluster head.

Figure 4.21a gives an exemplary initial cluster establishment in a corresponding environment where all devices are in direct communication range. Let us assume all devices have already performed the benchmarking process discussed in Section 4.3.3 and assigned themselves their cluster weights as given in the figure. Figure 4.21b shows the situation after having finished the DMAC algorithm, as presented in Section 4.1.2: v_3 as the only node with a weight above the threshold weight 0.5 is assigned the cluster head role, and all other nodes have joined v_3's cluster. Each cluster member stores the device ID of its cluster head, while the cluster head stores the IDs of its mapped cluster members. When a user wants to start an application now, the selector decides in favor of centralized configuration, which we described in detail in Section 4.2.

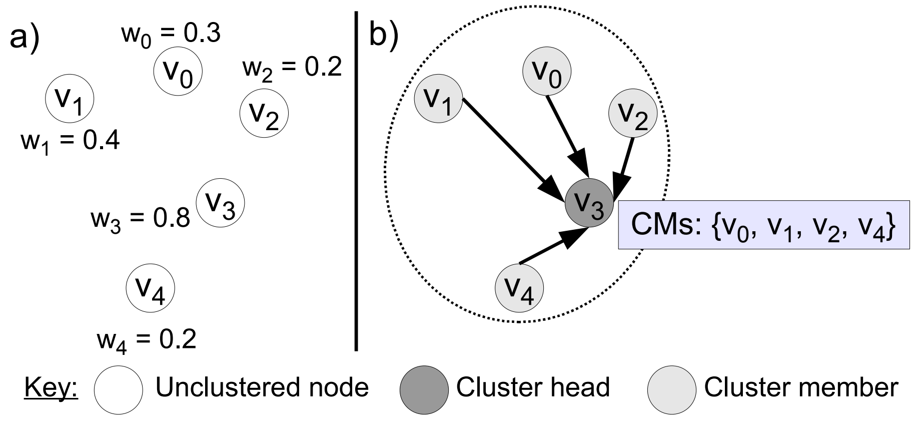


Figure 4.21.: Cluster Creation in weakly heterogeneous environment: a) Initial situation, b) Cluster establishment with DMAC

Regarding cluster maintenance, one has to distinguish between four different cases that may happen: the disappearence or appearence of a weak device, and the disappearence or appearence of a powerful device. The situation of (dis-)appearing weak devices is shown in Figure 4.22, representing the cluster structure established in Figure 4.21b. In this situation, consider the case that device v_1 disappears, e.g., because this device runs out of energy. Then, no more heartbeat messages from this device are transmitted, and the cluster head updates its cluster structure by removing v_1 from the list of its mapped devices. In the same situation, it may happen that a new mobile device, v_5, appears in transmission range, since its owner enters the room. With the first received heartbeat message from this device, the cluster head v_3 notices v_5 has a weight lower than the threshold and sends a `CH(`v_3`)` message. Then, v_5 joins the cluster of v_3 by sending a `join(`v_5`,` v_3`)` message which includes its resource information. Subsequently, v_3 adds v_5 to its list of mapped cluster members.

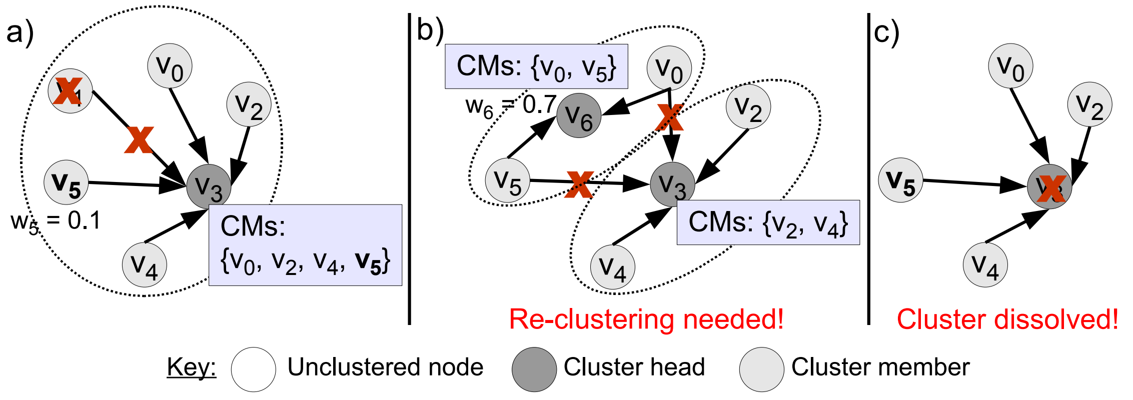


Figure 4.22.: Cluster Maintenance in weakly heterogeneous environment: a) Appearence and disappearence of weak devices, c) Appearence of a new powerful device, d) Disappearence of the last powerful device

In the situation described above, it may happen that a new powerful device v_5 with a weight above the chosen threshold appears (Figure 4.22b). Then, the cluster has to be split up into two clusters[3]. To keep a balanced configuration load among the two powerful devices, some of the cluster members from v_3 have to be re-mapped to the cluster of v_5. Details concerning these re-clustering processes will be given in Section 4.4 where we describe the hybrid configuration which is applied in such a situation.

Returning to the situation shown in Figure 4.22a, it may also happen that the powerful device v_3 disappears, for example because of a device failure. In this situation, illustrated in Figure 4.22c, there are no more powerful devices available in the environment, and the cluster structure is dissolved, leading to a pure Ad Hoc environment.

The *Direct Backtracking (DBT)* algorithm introduced in Section 4.2 was specifically developed for weakly heterogeneous scenarios, as it represents an efficient scheme for centralized configuration and adaptation of distributed applications in such scenarios. DBT relies on the clustering framework presented in Section 4.3.2 to proactively obtain the relevant resource information of the other devices, which

[3]In Section 4.3.6, we will present a selection mechanism to automatically decide in favor of a specific configuration algorithm based on the cluster structure.

is needed to perform centralized configuration completely locally, i.e., without involving any other devices.

4.3.5. Virtual Containers

In Section 4.3.3, a scheme to distinguish between strong and weak devices was introduced. Following in Section 4.3.4, we presented an approach to cluster the environment where the strong devices represent the cluster heads and the weak devices represent the cluster members. The cluster heads perform configurations and represent the coordinators for the adjacent devices, i.e., the cluster members. This means that the cluster head's resources are used for configuration, while the logic for validating a configuration is distributed among the devices involved in a configuration process.

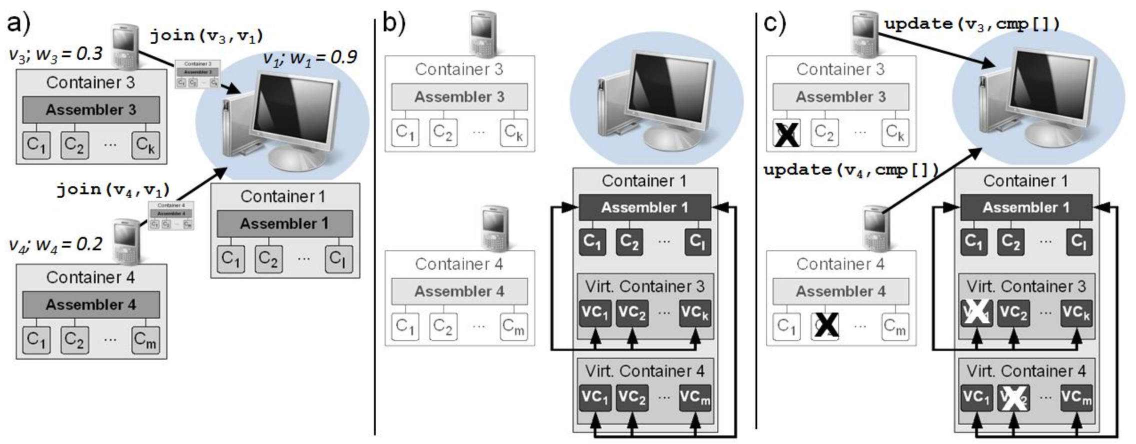


Figure 4.23.: Creation and update of Virtual Containers

This presumes that the cluster heads proactively acquire knowledge of the currently available components and resources on the mapped devices. For this purpose, we introduce the *Virtual Container (VC)* concept. Within our system software, a component container is responsible for hosting components and resources [Han09]. Correspondingly, a VC represents the emulation of another device – a mapped cluster member – at a cluster head. The VC contains the cluster member's information that is relevant for configuration processes, i.e., its resources and its components. Thus, in the broader sense, a VC represents a *proxy* for another device. Therefore, when a cluster member v maps itself to a cluster head u by sending a `join(v, u)` message, v additionally communicates its resource information to the cluster head. It is vital that this transmission happens at time of the cluster creation and, thus, *prior* to the configuration, to implement the pre-configuration process discussed in Section 4.1.3. If the creation of the Virtual Containers would be performed at configuration time, then the cluster head would not yet have local access to the remote components. Thus, the cluster head would need to manually request the resource information from the respective devices, like in the previously developed decentralized [HBR05] or centralized [HHSB07] configuration algorithm, causing additional configuration latencies and significantly reduced efficiency.

As an example for proactive creation of Virtual Containers, consider the situation in Figure 4.23a where we suppose that device v_1 has already sent a `CH(`v_1`)` message, as it is the device with highest cluster weight ($w_1 = 0.9$) in its neighborhood. Then, the two devices v_3 and v_4 join v_1's cluster and transmit their resource information. In the following, v_1 builds up two *Virtual Containers* as proxies for the remote devices, including the *Virtual Components* (abbreviated by VC in the figure) of the mapped devices that enable v_1's configuration assembler (the component which is responsible for computing configurations and adaptations) to configure these remote components by accessing their local proxies. In the situation which is shown in Figure 4.23b, the containers on the mapped devices are inactive, because the cluster head is responsible for the configurations now. However, as it may be possible that the cluster head device fails and, thus, the cluster members possibly are responsible for performing configurations again, v_3 and v_4 still conserve their containers as well as the assemblers in an inactive state.

In the future, it may be possible that the availability of mapped devices' components changes, e.g., because a component fails or is used by another application. To keep a consistent state of the remote components at the corresponding cluster head, a mapped device v sends an `update(v, cmp[])` message to the cluster head, containing its ID and a vector `cmp` which contains the availabilities of its components. Then, the cluster head updates the respective Virtual Container by marking these Virtual Components as currently available or unavailable. Such a situation is shown in Figure 4.23c where the mapped device v_3's component C_1 and the mapped device v_4's component C_2 become unavailable.

With the introduction of Virtual Containers, a device can locally calculate configurations for components that are hosted by other devices. Because of this, a cluster head does not need to perform remote procedure calls to obtain the current resource situation of other devices, thus completely avoiding communication overhead during the configuration process, which would significantly increase the latencies that are noticeable by the application users, especially if wireless communication technology with low bandwidth or high failure rates is used. Therefore, Virtual Containers enable a strong decoupling of the configuration processes from the real devices.

However, it has to be mentioned that Virtual Containers also introduce a disadvantage, since they are updated in case of changing resource conditions even if there are no configuration processes. This may increase the communication overhead unnecessarily. As the class sizes of the Virtual Containers are rather small with 8.8 kB per instance (cf. Section 6.5.2), we consider this to be a rather minor drawback.

The information which is needed to create a Virtual Container is sent from a cluster member to its cluster head using Mobile Code [CPV97]. Mobile Code enables the transmission of code segments from one runtime environment to another. It has to be considered that Mobile Code normally is subject to security risks, as the transmission of threats such as viruses or worms is potentially possible. However, as we assume cooperative user behaviours (cf. Section 2.1.1), the use of Mobile Code without the necessity of using digital signatures and, thus, the management of certificates is enabled. The only requirement is the existence of a class loader, which is even the case if a device supports the minimum Java CLDC profile specification.

4.3.6. Efficient Support of Adaptable Configuration Algorithms

To adapt the distribution of automatic application configuration, the need for a mechanism to allow the automatic selection of an assembler suited for application configuration in a specific environment arises. Therefore, we introduce the so-called *Selector* abstraction that enables the automatic selection of arbitrary assemblers on any container. Whenever a user wants to start an application on his or her device, a selection algorithm is started on this device to ask the provided selector about the way the configuration has to be taken. The selection strategy we provide so far supports the three different configuration approaches presented in this thesis: a decentralized scheme, a centralized scheme, and a hybrid scheme[4].

```
selection_algorithm(int strong) {
    if (strong == 0) {
        broadcastApplicationInformation();
        Container[] allDevices = (Container[])deviceRegistry.getAllDevices();
        foreach (Container c in allDevices) {
            c.startDecentralizedConfig();
        }
    }
    else if (strong == 1) {
        Container strongDevice = (Container)deviceRegistry.getStrongDevices();
        Container[] weakDevices = (Container[])deviceRegistry.getWeakDevices();
        foreach (Container c in weakDevices) {
            c.unicastApplicationInformation(strongDevice);
        }
        strongDevice.startCentralizedConfig();
    }
    else {
        Container[] strongDevices = (Container[])deviceRegistry.getStrongDevices();
        Container[] weakDevices = (Container[])deviceRegistry.getWeakDevices();
        foreach (Container c in weakDevices) {
            c.unicastApplicationInformation(c.getClusterHead());
        }
        foreach (Container c in strongDevices) {
            c.startHybridConfig();
        }
    }
}
```

Listing 4.9: Provided selection strategy

Listing 4.9 shows the implementation of the selection strategy in pseudo code: Based on the number of currently available strong devices (`strong`), the selector decides which of the provided configuration algorithms is chosen for the configuration: In case of an Ad Hoc scenario with no strong devices, the decentralized scheme is

[4]This scheme is discussed in the next section.

chosen and started on all devices. If exactly one strong device is available, the selector chooses the centralized approach and initiates the application configuration on this strong device (`strongDevice`). In case of multiple strong devices, configurations are calculated in a hybrid way involving all strong devices (`strongDevices`).

As the current environmental condition is provided by BASE's device registry and BASE is running on every device, the selectors on each device always have the identical view on the current scenario situation, i.e., each device knows which powerful and weak devices are currently available. Thus, when a user starts an application on his or her device, the system software running on this device looks up the currently available devices and decides to which other devices it has to send the information about a pending configuration and the corresponding application structure, i.e., to *all* present devices (Ad Hoc environment), to the single powerful device (weakly heterogenous environment), or to the group of powerful devices[5] (strongly heterogeneous environment). Figures 4.24a to c show three exemplary scenarios where a user starts an application in different scenarios.

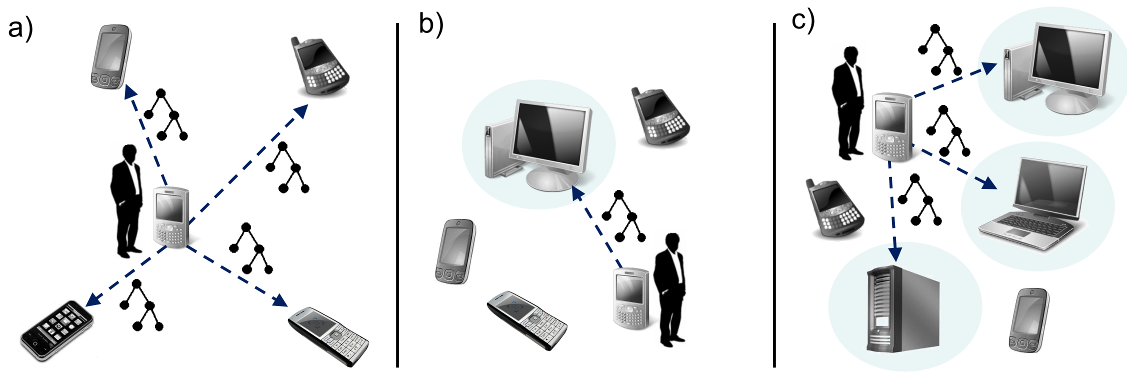


Figure 4.24.: Execution of selector algorithm in a) homogeneous Ad Hoc environment, b) weakly heterogeneous environment, c) strongly heterogeneous environment

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

4.4. Hybrid Application Configuration

Both the decentralized and centralized approaches have advantages, but also drawbacks that prevent an efficient configuration in *all* possible pervasive environments. A hybrid approach is able to combine the best properties of these two approaches to minimize the configuration latency [SHR10]. For this purpose, only the strong

[5]For sending a message to a specific group of devices, *multicast* is usually used. However, in one-hop environments, sending multicast messages simply degrades to sending multiple unicast messages

devices actively calculate application configurations. We call them *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. As the weak devices stay passive during the actual configuration calculations, 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 for each PD should be calculated by exactly one AD to minimize the interdepencies between the ADs. 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[6].

4.4.1. Initial Resource-Aware Cluster Formation

The following scheme establishes multiple stable clusters in heterogeneous environments with several strong devices. These devices automatically become the cluster heads (and, hence, the ADs) if a resource-aware clustering strategy (e.g., like in [SHR08a]) 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. The main goal of the scheme is to reduce the configuration 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 cluster indices $j \in \{0, \ldots, n-1\}$. Initially, each AD assigns itself a CID i according to its device ID, i.e. the AD with lowest device ID (of all ADs) assigns itself CID $i = 0$, and the AD with highest device ID gets CID $i = m-1$. The same holds for the PDs that assign themselves CIDs j according to their device ID. This means we rely on *global knowledge* among all devices. As we focus on single-hop environments like conference rooms and a relatively low degree of dynamics (cf. Sections 2.1 and 5.7.2) where devices typically stay within an environment for some time, we consider to maintain global knowledge as a feasible solution.

There is an overhead for each AD 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 each 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 testablishes 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.

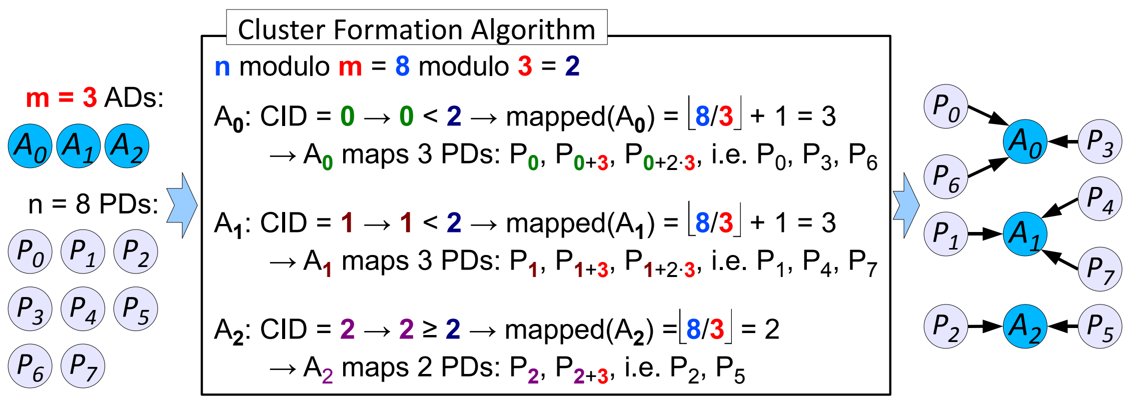
[6]Actually, this distinction between ADs and PDs was also taken in the centralized approach discussed in Section 4.2. However, as centralized configuration relies on exactly *one* configuration device, we did not introduce these notions there for simplification reasons.

This leads to the so-called *Balancing Condition* that has to be fulfilled at each Active Device:

$$mapped(A_i) = \begin{cases} \left\lfloor \frac{n}{m} \right\rfloor + 1, & i < n \text{ modulo } m \\ \left\lfloor \frac{n}{m} \right\rfloor, & i \geq n \text{ modulo } m \end{cases}, \tag{4.13}$$

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.


Cluster Formation Algorithm

n modulo m = 8 modulo 3 = 2

A_0: CID = 0 → 0 < 2 → mapped(A_0) = $\lfloor 8/3 \rfloor$ + 1 = 3
→ A_0 maps 3 PDs: P_0, P_{0+3}, $P_{0+2\cdot3}$, i.e. P_0, P_3, P_6

A_1: CID = 1 → 1 < 2 → mapped(A_1) = $\lfloor 8/3 \rfloor$ + 1 = 3
→ A_1 maps 3 PDs: P_1, P_{1+3}, $P_{1+2\cdot3}$, i.e. P_1, P_4, P_7

A_2: CID = 2 → 2 ≥ 2 → mapped(A_2) = $\lfloor 8/3 \rfloor$ = 2
→ A_2 maps 2 PDs: P_2, P_{2+3}, i.e. P_2, P_5

Figure 4.25.: Initial mapping in scenario with three ADs (A_0 to A_2) and eight PDs (P_0 to P_7)

For clarification, let us consider an exemplary scenario depicted in Figure 4.25, consisting of $m = 3$ ADs A_0 to A_2 and $n = 8$ unmapped PDs P_0 to P_7. Furthermore in this scenario, n modulo m resolves to 8 modulo $3 = 2$. The CID of A_0 is $i = 0 < 2$, so A_0 maps $\left\lfloor \frac{8}{3} \right\rfloor + 1 = 3$ PDs, i.e., one additional PD. The same holds for A_1, as $i = 1 < 2$. However, A_2 with CID $i = 2 \geq 2$ maps only $\left\lfloor \frac{8}{3} \right\rfloor = 2$ PDs. 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 on the right hand side of Figure 4.25.

4.4.2. 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. Generally, cluster maintenance has to deal with newly appearing devices, and with disappearing devices. More specifically, re-clustering comprises four different cases:

- The appearance of new PDs
- The appearance of new ADs
- The disappearance of PDs
- The disappearance of ADs

In the following, we will describe how our scheme performs in case of each of these situations.

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 $0, \dots, m-1$ 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.

Appearance of new PDs: If a new PD appears, the AD with cluster index $i = n$ modulo m maps this device. Proceeding like this, the round robin distribution of the PDs to the ADs is continued. After this mapping, each device increments the number n of PDs.

As an example, consider Figure 4.26 which shows at the left hand side the cluster established in the example above. In this situation, a group of four people with mobile devices (PDs P_8 to P_{11}) enters the environment. By broadcasting heartbeat messages including their cluster weight, the users' mobile devices announce themselves as new PDs. Then, each cluster head updates the number of Passive Devices to $n = 12$ and re-calculates the number of PDs it has to map. In this situation, each cluster head calculates the number of mapped PDs as $mapped(A_i) = \lfloor \frac{12}{3} \rfloor = 4$. Proceeding in the round robin manner, A_0 maps the new PD P_9, A_1 maps P_{10}, and A_2 maps P_8 and P_{11}, leading to the updated clusters as shown on the right hand side of Figure 4.26.

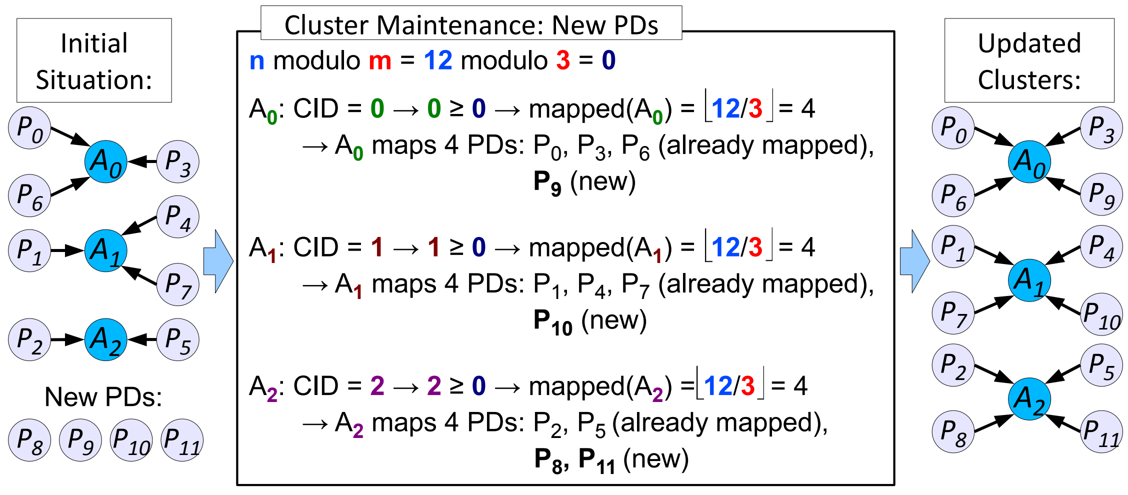

Initial Situation:
New PDs:
Cluster Maintenance: New PDs
n modulo m = 12 modulo 3 = 0
A0: CID = 0 → 0 ≥ 0 → mapped(A0) = ⌊12/3⌋ = 4
→ A0 maps 4 PDs: P0, P3, P6 (already mapped), P9 (new)
A1: CID = 1 → 1 ≥ 0 → mapped(A1) = ⌊12/3⌋ = 4
→ A1 maps 4 PDs: P1, P4, P7 (already mapped), P10 (new)
A2: CID = 2 → 2 ≥ 0 → mapped(A2) = ⌊12/3⌋ = 4
→ A2 maps 4 PDs: P2, P5 (already mapped), P8, P11 (new)
Updated Clusters:
P0
P3
P6
P9
P1
P4
P7
P10
P2
P5
P8
P11
A0
A1
A2

Figure 4.26.: Cluster Maintenance: Integration of appearing Passive Devices

Appearance of new ADs: Listing 4.10 shows the algorithm that is performed on a newly appearing Active Device A_x with CID x when y new ADs appear coevally: A_x needs to map $\lfloor \frac{n}{m} \rfloor$ devices to itself[7] so that the Balancing Condition is still fulfilled for all PDs. The re-mappings are executed in the following way: Initially in lines 2 to 4 of Listing 4.10, A_x initializes the variables `mapped` (which denotes the number of currently mapped own PDs), `remap` (which denotes the total number of PDs that have been mapped by newly appearing ADs with lower CIDs), and `remappings(`A_x`)` (which denotes the number of PDs to be mapped by A_x). Then, A_x initiates $\lfloor \frac{n}{m} \rfloor$ re-mappings of PDs from other ADs (lines 5 to 10): Initially, A_x re-maps 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

$$i = (n-1)\ modulo\ (m-y) \tag{4.14}$$

due to the round robin scheme (line 6). A_x 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 A_x. A_x awaits the resource information from the PD and creates the respective Virtual Container (line 8), increments the variable `mapped`, and repeats the re-mapping process $\lfloor \frac{n}{m} \rfloor$ times. As the ADs whose PDs are re-mapped by A_x are chosen in a round robin manner, the Balancing Condition is still fulfilled on all ADs after these re-mappings.

```
 1 request_remappings(){
 2     mapped := 0;
 3     remap := $\sum_{i<x}$ remappings($A_i$);
 4     remappings($A_x$) := floor(n/m);
 5     **while** (mapped < remappings($A_x$)) {
 6         remapId = (n−1−remap) modulo (m−y);
 7         send_remap_request_to_AD(remapId);
 8         create_virtual_container();
 9         mapped++;
10     }
11 }
```

Listing 4.10: Reclustering process executed by a newly appearing AD A_x (CID x) when y new ADs appear at the same time

If multiple ADs appear at almost the same time (i.e., $y > 1$), the problem of race conditions during the mapping process may arise and potentially lead to inconsistent mappings, e.g., two new ADs may map the same device. To analyze 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. To avoid inconsistencies, every new AD broadcasts a *notification message* including its own CID, notifying the other ADs that it will start the mapping process now. Every other new AD receiving a notification message answers with a notification message, so the new devices know about the CIDs of

[7]Here, A_x is already included in the number m of ADs

other new devices. All other ADs answer with a simple `OK` status message without body, representing their agreement to the new AD's mapping process.

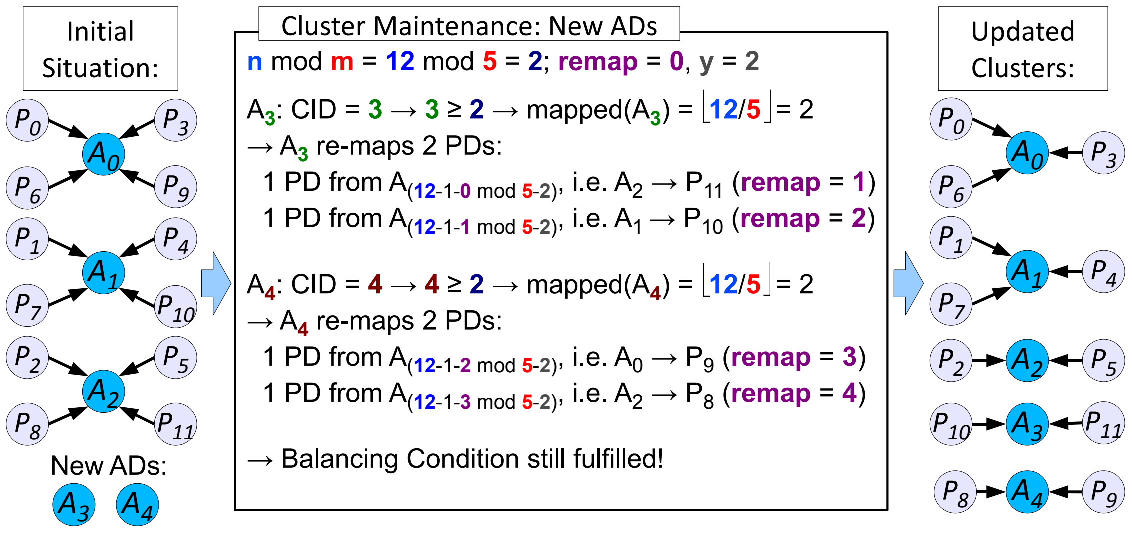
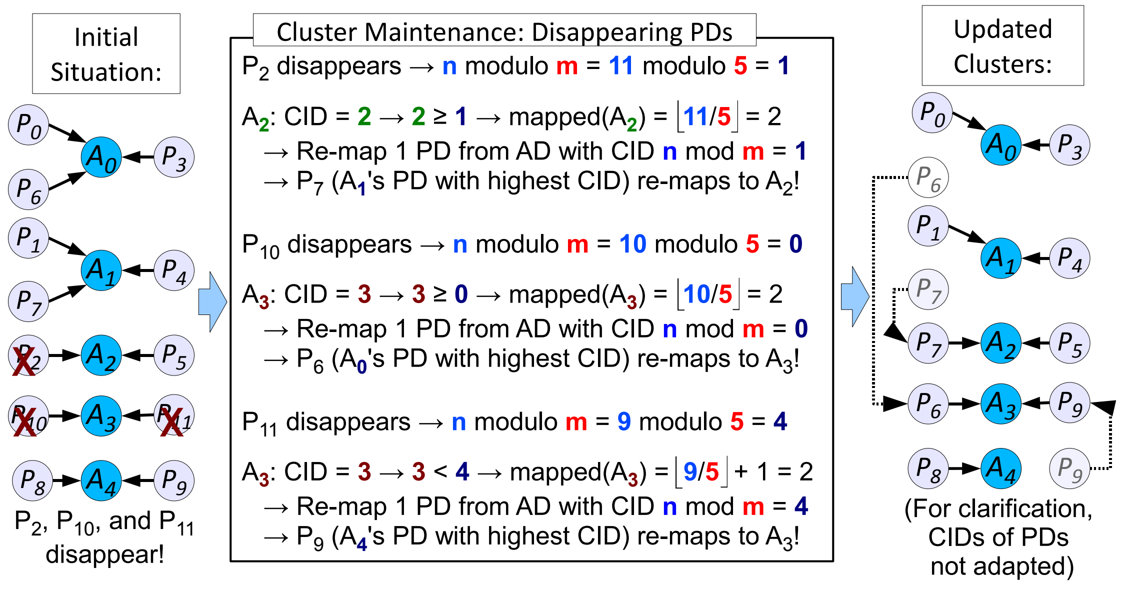

Initial Situation:
P0 P3 A0 P6 P9 P1 P4 A1 P7 P10 P2 P5 A2 P8 P11
New ADs: A3 A4
Cluster Maintenance: New ADs
n mod m = 12 mod 5 = 2; remap = 0, y = 2
A3: CID = 3 → 3 ≥ 2 → mapped(A3) = ⌊12/5⌋ = 2
→ A3 re-maps 2 PDs:
1 PD from A(12-1-0 mod 5-2), i.e. A2 → P11 (remap = 1)
1 PD from A(12-1-1 mod 5-2), i.e. A1 → P10 (remap = 2)
A4: CID = 4 → 4 ≥ 2 → mapped(A4) = ⌊12/5⌋ = 2
→ A4 re-maps 2 PDs:
1 PD from A(12-1-2 mod 5-2), i.e. A0 → P9 (remap = 3)
1 PD from A(12-1-3 mod 5-2), i.e. A2 → P8 (remap = 4)
→ Balancing Condition still fulfilled!
Updated Clusters:
P0 A0 P3 P6 P1 A1 P4 P7 P2 A2 P5 P10 A3 P11 P8 A4 P9

Figure 4.27.: Cluster Maintenance: New Active Devices A_3 and A_4 re-map PDs from the other ADs

As an example, consider Figure 4.27 where $y = 2$ new Active Devices appear at almost the same time and assign themselves the CIDs 3 and 4. After AD 3 has sent its notification message, AD 4 answers with a notification message, while the other ADs respond with an `OK` message. Then, the new ADs start their mapping processes. As the environment now covers $n = 12$ PDs and $m = 5$ ADs (whereas 2 of them are new), n *modulo* m calculates to 12 *modulo* $5 = 2$. Since the CID of A_3 as well as A_4 is higher than 2, both devices only map $\lfloor \frac{n}{m} \rfloor = \lfloor \frac{12}{5} \rfloor = 2$ devices, according to Equation 4.13. According to line 6 from Listing 4.10, A_3 finds out it needs to remap one PD from device *(12-1-0) modulo (5-2) = 11 modulo 3* = 2, and one PD from device *(12-1-1) modulo 3 = 10 modulo 3* = 1. Thus, it sends re-mapping requests to these devices. A_2 requests P_{11} (as its mapped PD with highest CID) to re-map to A_3, and A_1 requests P_{10} to re-map to A_3. After P_{11} and P_{10} have sent their resource information to A_3, the respective Virtual Containers are created by A_3 and enable the integration of the resources from P_{11} and P_{10} into local configurations at A_3.

A_4 proceeds in the same way, as it can be seen in the central box of Figure 4.27. Since A_4 can simply calculate that A_3 (as new AD with lower CID) re-maps two PDs, it sets `remap` to 2 and calculates the CIDs of the ADs from which it request to re-map PDs: As *(12-1-2) modulo (5-2) = 9 modulo 3* = 0 and *(12-1-3) modulo (5-2) = 8 modulo 3* = 2, A_4 requests one PD from A_0 and one PD from A_2 to be re-mapped to itself. Then, A_0 and A_2 notify their mapped PDs with highest CIDs (i.e., P_9 for A_0, and P_8 for A_2) to re-map to A_4. Finally, P_9 and P_8 send their resource information to A_4, the respective Virtual Containers are created, and the re-mapping process is completed. The emerging updated cluster structure is shown at the right hand side of Figure 4.27.

Disappearance of PDs: 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 for itself. 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 process 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 adapts the configuration by selecting alternative components for them, if available.



Cluster Maintenance: Disappearing PDs

P_2 disappears $\rightarrow$ n modulo m = 11 modulo 5 = 1

A_2: CID = 2 $\rightarrow 2 \geq 1 \rightarrow$ mapped(A_2) = $\lfloor 11/5 \rfloor = 2$
$\rightarrow$ Re-map 1 PD from AD with CID n mod m = 1
$\rightarrow$ P_7 (A_1's PD with highest CID) re-maps to A_2!

P_{10} disappears $\rightarrow$ n modulo m = 10 modulo 5 = 0

A_3: CID = 3 $\rightarrow 3 \geq 0 \rightarrow$ mapped(A_3) = $\lfloor 10/5 \rfloor = 2$
$\rightarrow$ Re-map 1 PD from AD with CID n mod m = 0
$\rightarrow$ P_6 (A_0's PD with highest CID) re-maps to A_3!

P_{11} disappears $\rightarrow$ n modulo m = 9 modulo 5 = 4

A_3: CID = 3 $\rightarrow 3 < 4 \rightarrow$ mapped(A_3) = $\lfloor 9/5 \rfloor + 1 = 2$
$\rightarrow$ Re-map 1 PD from AD with CID n mod m = 4
$\rightarrow$ P_9 (A_4's PD with highest CID) re-maps to A_3!

Figure 4.28.: Cluster Maintenance: Passive Devices P_2, P_{10} and P_{11} disappear in this order and induce re-mappings for ADs A_2 and A_3

As an example, consider the situation depicted at the left hand side of Figure 4.28, where three Passive Devices – P_2, P_{10}, and P_{11} – disappear in this order. When P_2 disappears, A_2 (as the Active Device to which P_2 was mapped to) finds out that it needs to have two devices mapped (cf. Equation 4.13), but only has one device mapped. Thus, it send a re-mapping request to the AD with CID n modulo m = 1. In the following, A_1 requests its mapped PD with highest CID (i.e., P_7) to send its resource information to A_2, which then creates the corresponding Virtual Container.

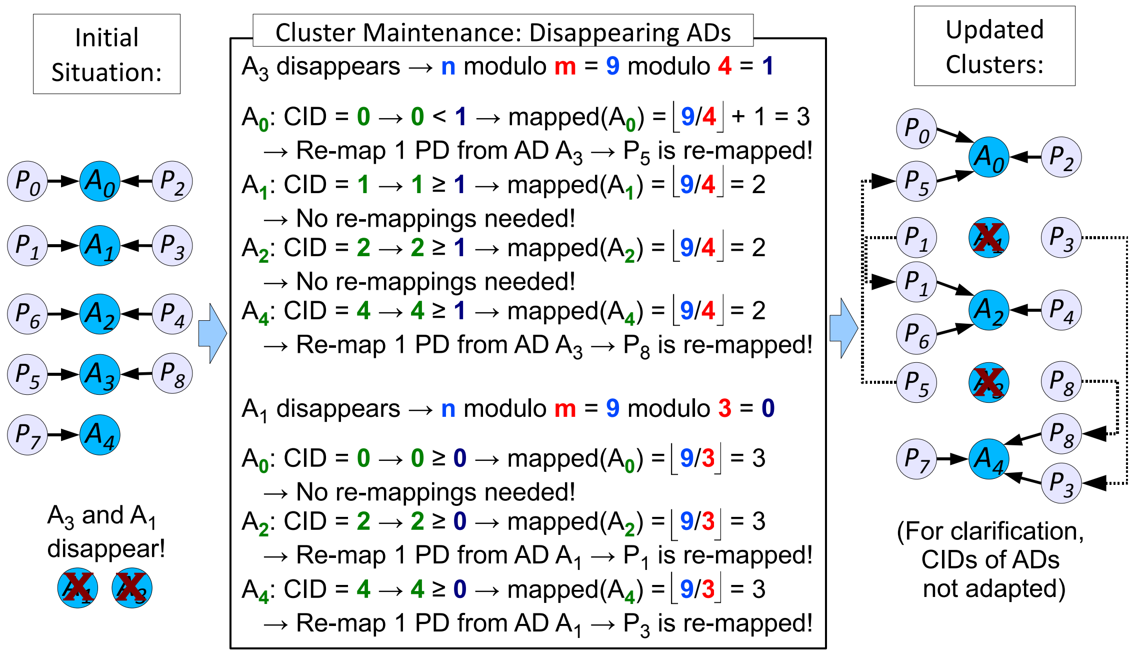
Now, consider the case that P_{10} which was mapped to A_3 disappears. A_3 determines it needs to have $\left\lfloor \frac{10}{5} \right\rfloor = 2$ devices mapped, and requests to re-map one PD from the AD with CID 10 modulo 5 = 0. After having received the resource information of P_6, A_3 creates the corresponding Virtual Container, and the clusters are balanced again.

When P_{11} – which was also mapped to A_3 – disappears, A_3 needs to re-map again one PD from another AD, and determines the AD with CID 9 modulo 5 = 4 as the respective device. Finally after A_3 has re-mapped P_9 from A_4, the cluster structure as shown at the right hand side of Figure 4.28 is established. As one can see, the Balancing Condition is obviously fulfilled again due to the taken re-mappings. It has to be mentioned that in this example, the CIDs of those PDs with a higher CID than the disappearing PDs (i.e., the PDs with CID higher than 2) are *not* adapted in this example for the sake of clearness. Actually, the CIDs of the remaining PDs would need to be reduced accordingly when PDs with lower indices disappear, e.g., if P_2 disappears, the PDs with CID 3 up to 11 would reduce their CID by 1.

Disappearance of ADs: 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, re-mapping 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 fulfill 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 device 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 our evaluations in Section 4.5.3. 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.

Figure 4.29 shows a scenario where at first, AD A_3 disappears, and then, A_1 disappears. When A_3 disappears, the PDs of A_3 are unmapped. Moreover, the number of ADs is reduced to $m = 4$ and, thus, $n \ modulo \ m = 1$. This induces that A_0 has to have three PDs mapped, and A_1, A_2, and A_4 need to have two PDs mapped. As each AD can simply calculate how many PDs every other AD needs to have mapped (since the numbers n and m and the CIDs of all devices are known to each device), every AD knows that A_0 and A_4 re-map the two PDs P_5 and P_8

(which were previously mapped to A_3). Since the CID of A_0 is lower than the CID of A_4, A_0 re-maps P_5 (as the unmapped PD with lower CID), and A_4 re-maps P_8. Now, every PD is mapped to one AD again, and the Balancing Condition remains fulfilled. After A_1 has disappeared, m is reduced to 3, and P_1 and P_3 are unmapped now. Same as above, every AD can simply calculate how many PDs it needs to have mapped. In the situation given, all ADs need to have 3 PDs mapped, so A_2 and A_4 have to take care for P_1 and P_3. Regarding their CIDs, A_2 re-maps P_1 and A_4 re-maps P_3. This leads to the situation shown at the right hand side of Figure 4.29.



A_3 disappears → n modulo m = 9 modulo 4 = 1

A_0: CID = 0 → 0 < 1 → mapped(A_0) = $\lfloor 9/4 \rfloor$ + 1 = 3
→ Re-map 1 PD from AD A_3 → P_5 is re-mapped!
A_1: CID = 1 → 1 ≥ 1 → mapped(A_1) = $\lfloor 9/4 \rfloor$ = 2
→ No re-mappings needed!
A_2: CID = 2 → 2 ≥ 1 → mapped(A_2) = $\lfloor 9/4 \rfloor$ = 2
→ No re-mappings needed!
A_4: CID = 4 → 4 ≥ 1 → mapped(A_4) = $\lfloor 9/4 \rfloor$ = 2
→ Re-map 1 PD from AD A_3 → P_8 is re-mapped!

A_1 disappears → n modulo m = 9 modulo 3 = 0

A_0: CID = 0 → 0 ≥ 0 → mapped(A_0) = $\lfloor 9/3 \rfloor$ = 3
→ No re-mappings needed!
A_2: CID = 2 → 2 ≥ 0 → mapped(A_2) = $\lfloor 9/3 \rfloor$ = 3
→ Re-map 1 PD from AD A_1 → P_1 is re-mapped!
A_4: CID = 4 → 4 ≥ 0 → mapped(A_4) = $\lfloor 9/3 \rfloor$ = 3
→ Re-map 1 PD from AD A_1 → P_3 is re-mapped!

Figure 4.29.: Cluster Maintenance: Active Devices A_3 and A_1 disappear in this order and induce re-mappings at the other Active Devices

If in the depicted situation, two of the three remaining ADs A_0, A_2 and A_4 would disappear, all PDs would be mapped to the remaining AD. This AD would manage n Virtual Containers then and calculate upcoming application configurations in a centralized fashion using Direct Backtracking which was presented in Section 4.2. In case the single remaining AD would also disappear, the cluster structure would be dissolved, and completely decentralized configuration would be performed.

4.4.3. Hybrid Configuration Algorithm

Our new approach calculates configurations in a decentralized manner involving the subset of ADs, while the configuration of each PD's components is performed locally on the AD it was mapped to, i.e., in a centralized way. Therefore, the created VCs are used (cf. Section 4.3.5). This leads to faster configuration calculations compared to the decentralized configuration, as we will show in the evaluations. Moreover, the resource-weak PDs are not involved in these calculations. This helps to conserve the (usually limited) energy resources of the PDs and ensures a configuration which is aware of the devices' computation resources. In summary, the decentralized and centralized parts of the configuration resolve to the aspired *hybrid* configuration approach. We suppose that the resource conditions are stable during configuration.

An adapted version of Yokoo's Asynchronous Backtracking (ABT, [YDIK98]) algorithm which was introduced by Handte et al. [HBR05] is used for the cooperative configuration between the ADs. ABT is a decentralized algorithm which 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 that is solely based on look-back techniques is used. This means that the scheme tries to improve the value assignment by learning from previous assignments that failed. Thus, Asynchronous Backtracking tries to avoid failures that have been made in the past to some extent. ABT needs total priority ordered variables [Han09]. Moreover, unidirectional communication links between variables sharing a constraint have to be established from the variables with higher priority to the variables with lower priority. In the following, ABT is started in parallel: Each agent assigns some value to its variable so that it does not conflict with its known constraints. According to the depth-first search approach, the priorities are given from left to right and from top to bottom of the tree. This means that the top-left instance has highest priority, while the bottom-right instance gets the lowest priority. While agents with variables with higher priority send their current assignment to linked agents with variables with lower priority on every variable change, agents with lower priority have to evaluate the constraints that they share with agents with higher priority. If an agent needs to change an assignment, it sends its new assignment to all linked agents. However, when an agent detects that it cannot assign a value to its variable without conflicting with a constraint, it creates a so-called *nogood*, containing the set of assignments from higher prior agents that caused the conflict. Then, this nogood is sent to the agent that created this assignment, which then checks whether the nogood is still valid by comparing the value assignments of variables that it has already recorded with those contained in the nogood: The nogood is only valid, if all the recorded variable assignments and the variable assignments contained in the nogood match. As decentralized configuration is not in the focus of this thesis, we refer to Handte [Han09] for further details concerning configuration of an application using Asynchronous Backtracking.

For the local configuration of the PDs' components on the ADs, we use the Direct Backtracking algorithm introduced in Section 4.2 to reduce the number of required backtracking processes and minimize the overhead of the remaining backtrackings to guarantee efficient centralized configuration processes. However, regarding the choice of the backtracking goals as described in Section 4.2.3, the scheme used within hybrid configuration differs from the completely centralized approach by introducing an additional selection criterion which priorizes the components to be chosen in a more fine-grained way that additionally considers the locality of the components: If several components are available that fulfill the requirements given by a dependency, the algorithm at first decides in favor of a component which is locally available on an Active Device A, since this does not produce any communication overhead at all. If there is no local component available on A, the algorithm tries to choose a component which is available on a Passive Device that has been mapped to A, as this does not produce communication overhead during the configuration calculation phase (due to the established VC), but only at the subsequent initialization of the component bindings.

In case there is neither a local component nor a component on a mapped device via emulated VC available, the algorithm decides in favor of a component available on a remote device which is not mapped to A. This represents the worst case for resolving an application dependency with the hybrid scheme, as it produces communication overhead both at the configuration calculation phase *and* at the initialization of the component bindings. As a secondary criterion for selecting an application component (in case the locality criterion does not yield a unique decision in favor of a specific component), the available resource amount and the resource consumption of a specific component are evaluated, as described for the centralized configuration algorithm (cf. Section 4.2.2). Thus, the locality of backtracking goals represents the highest selection priority and is more important in the hybrid scheme than the amount of resources a component consumes, as remote communication significantly increases configuration latencies and introduces waiting times at the involved Active Devices within configuration calculations. Furthermore, it also reduces the achievable degree of parallelism.

To illustrate how decentralized and centralized configuration are combined by the hybrid approach, we now present an exemplary hybrid configuration on the basis of the exemplary distributed presentation application introduced in Section 2.1.2. Therefore, consider Figure 4.30 which presents the devices and the established cluster structure using the scheme discussed previously in this section. The regarded scenario consists of two strong devices, a desktop PC (Active Device A_0) and a laptop (A_1). Concerning locally available components, the PC provides a microphone, a keyboard and a connected video projector, while the laptop supplies a headset and a mouse. Regarding the available weak devices, a netbook (Passive Device P_0, providing access to loudspeakers) and a tablet device (P_2, providing access to a TV monitor) have been mapped to the PC, and a smart phone (P_1, providing the presentation source file and a touch display as GUI) has been mapped to the laptop. This leads to the illustrated two clusters and the fulfillment of the Balancing Condition for every AD.

For the actual hybrid configuration, consider Figure 4.31 which presents the structure of the distributed presentation application and the proceeding of the involved decentralized (Asynchronous Backtracking, [HBR05]) and centralized (Direct Backtracking, [SHR08b]) configuration algorithms. When the user wants to start the application on his or her smart phone P_1, the application anchor is initialized and the application information is sent to A_1 as the Active Device to which P_1 is mapped to. The anchor instance has three child components, whereas one of these child components (CoID [0,0][2,0]) is locally available on A_0, and two of these child components (CoIDs [0,0][0,0] and [0,0][1,0]) are resident on devices for which A_1 is responsible. Thus, A_1 tries to configure the left and the middle subtree of the application anchor, and A_1 requests A_0 to configure the right subtree, as the requirements of component instance [0,0][2,0] cannot be fulfilled by any component for which A_1 is responsible.

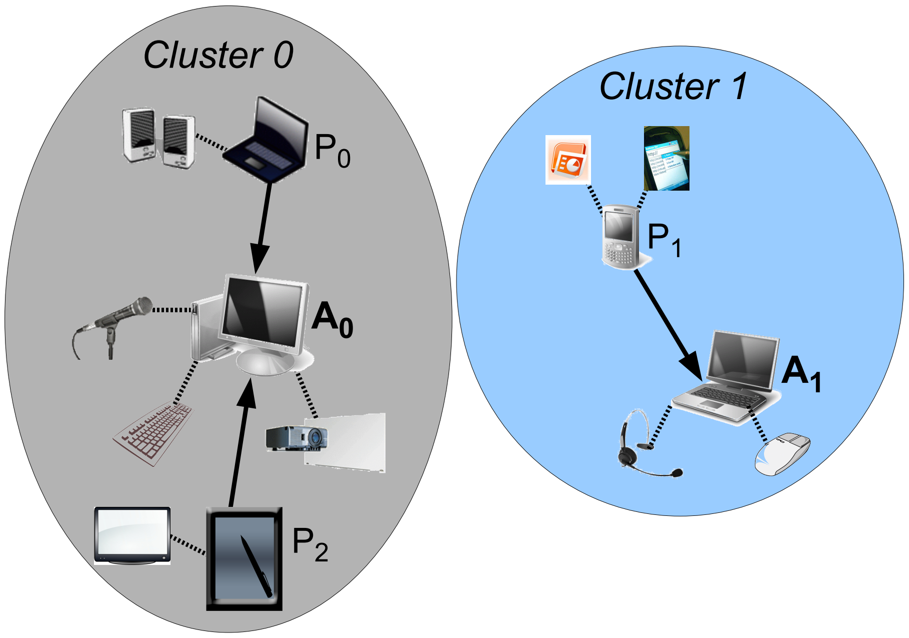


Figure 4.30.: Cluster Structure for exemplary Hybrid Configuration

The two contracts of the left subtree are not multi-optional, so A_1 simply selects the smart phone as source file input device (CoID [0,0][0,0]) and the presentation file on the smart phone (CoID [0,0][0,0][0,0]) as actual source file. In the middle subtree, A_1 can decide between the microphone available at A_0 (CoID [0,0][1,0][0,0]) and the headset which is locally available (CoID [0,0][1,0][0,1]). According to the locality selection criterion introduced above, A_1 decides in favor of the headset to avoid remote communication both at the configuration calculation stage and the component initialization phase. For the haptic input component, A_1 can choose amongst the locally available mouse (CoID [0,0][1,0][1,0)), the keyboard at the remote device A_0 (CoID [0,0][1,0][1,1]), and the touch display (CoID [0,0][1,0][1,2]) of the smart phone which is mapped to A_1. Again, A_1 decides in favor of the local component and selects the mouse.

For the contract with component ID [0,0][2,0] of the right subtree, configured by A_0, the loudspeakers (CoID [0,0][2,0][0,0]) connected to the netbook represent the only possible component. Thus, A_0 selects these loudspeakers which are locally available within a Virtual Container. Concerning the optical output component, A_0 prefers the locally connected video projector (CoID [0,0][2,0][1,0]) over the TV (CoID [0,0][2,0][1,1]) to avoid communication overhead at the subsequent component binding phase, as the TV is connected to the tablet PC that is mapped to A_0 and, hence, only available via VC.

The components which are configured by A_0 are independent from the configuration results that are provided by A_1, as the respective contracts deal with different types of resources. Thus, nogoods do not have to be created and, subsequently, no distributed adaptation processes are required. Instead, the configuration results are made permanent by instiantiating the respective components and establishing the bindings between parent and child components of a contract.

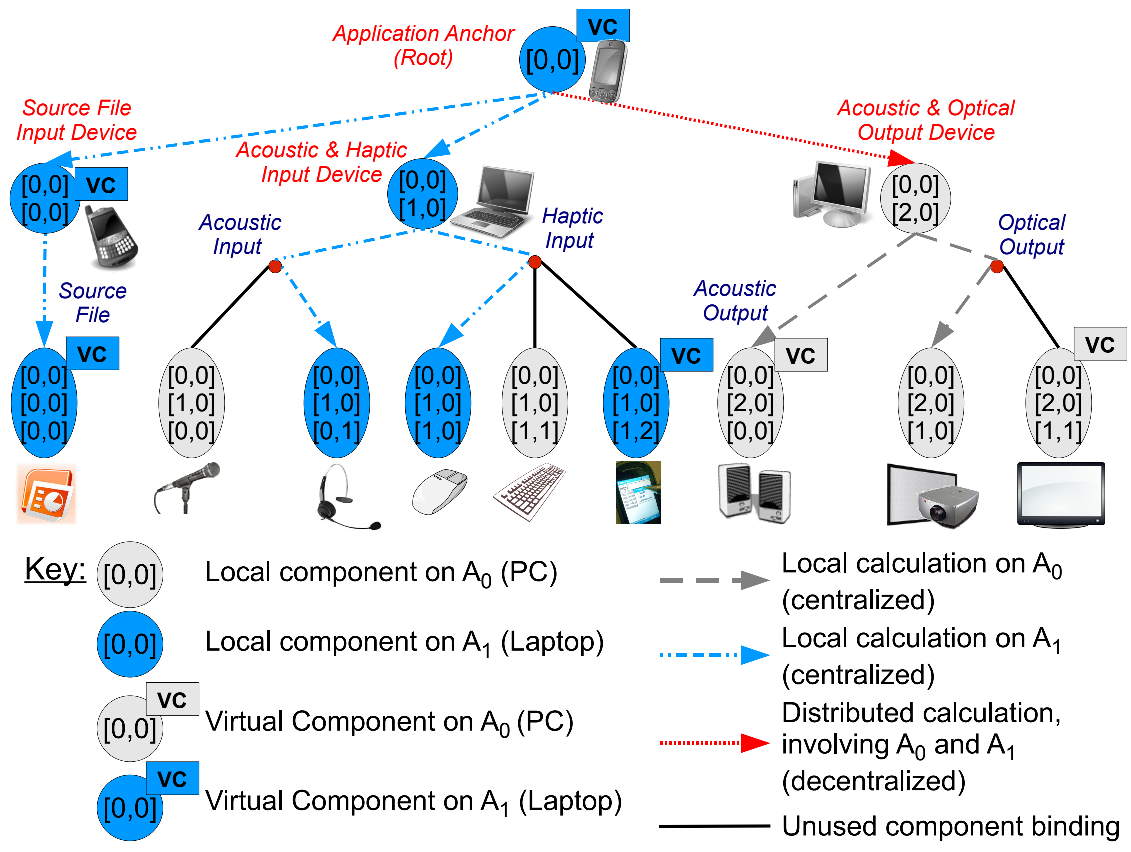

Application Anchor (Root)
VC
[0,0]
Source File Input Device
[0,0] [0,0]
VC
Acoustic & Haptic Input Device
[0,0] [1,0]
Acoustic & Optical Output Device
[0,0] [2,0]
Source File
Acoustic Input
Haptic Input
Acoustic Output
Optical Output
[0,0] [0,0] [0,0]
[0,0] [1,0] [0,0]
[0,0] [1,0] [0,1]
[0,0] [1,0] [1,0]
[0,0] [1,0] [1,1]
[0,0] [1,0] [1,2]
[0,0] [2,0] [0,0]
[0,0] [2,0] [1,0]
[0,0] [2,0] [1,1]
Key:
Local component on A_0 (PC)
Local component on A_1 (Laptop)
Virtual Component on A_0 (PC)
Virtual Component on A_1 (Laptop)
Local calculation on A_0 (centralized)
Local calculation on A_1 (centralized)
Distributed calculation, involving A_0 and A_1 (decentralized)
Unused component binding

Figure 4.31.: Proceeding of Hybrid Configuration Algorithm

In summary, it can be seen from Figure 4.31 that only one remote communication between A_0 and A_1 (for the configuration of the acoustic and optical output device instance with CoID [0,0][2,0]) was needed during the configuration calculation phase, and two additional communications are required at the component binding phase because of components which are resident on mapped Passive Devices and, thus, only available through the provided Virtual Containers at the configuration phase. The respective bindings have to be established between the instances with CoIDs [0,0] (resident on the smart phone P_1) and [0,0][1,0] (resident on the laptop A_1), and between the instances with CoID [0,0][2,0] (resident on the PC A_0) and CoID [0,0][2,0][0,0] (resident on the netbook P_2).

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

Comparing hybrid configuration to pure decentralized and centralized approaches in this example would yield the following differences:

- Decentralized configuration would provide a higher degree of parallelism, as the left and the middle subtree could be calculated independently by the Smart

Phone and the laptop. However, it would involve additional communication overhead between the weak and strong devices, and the weak devices may probably slow down the configuration calculation phase due to their reduced computation power.

- Centralized configuration is performed *completely locally* on the single configuration device, but at the same time does not perform any kind of parallel calculations. Furthermore, the complete configuration results would have to be distributed to all other devices after the configuration, thus increasing the latencies for the component binding stage of the configuration.

A more fine-grained comparison is given in the Evaluation Section 4.5.3.

4.4.4. Exemplary Hybrid Configuration Process

To clarify the proceeding of our hybrid scheme, we present a complete configuration process here, covering the initial clustering, the actual hybrid configuration, and the distribution of the configuration results. Resuming the introductory scenario from Section 2.1.2, Figure 4.33 shows an exemplary environment where a distributed presentation application is executed in a strongly heterogeneous environment. 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 or her mobile phone (P_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 for instance the conference organizer's smart phone (P_3). 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 auditorium's stationary PC (A_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 (A_1) or even their mobile phone (P_1). Thus, the application tree to be configured could look quite similar like the one displayed in Figure 4.10.

Initially as first step, each device performs the benchmarking process to discover its own resource power for configuration processes. In the examplary scenarios, this yields clustering weights which are above the threshold of 0.5 for the desktop PC (becoming Active Device A_0 as it has lowest ID among the devices with high cluster weights), the laptop (A_1) and the server (A_2). As the four Smart Phones assign themselves cluster weights below 0.5, they become the Passive Devices P_0 to P_3, assigned according to their device IDs, as shown in Figure 4.32.

In step 2, the cluster structure is established using the round robin scheme discussed in Section 4.4.1. This yields the desktop PC (A_0) as cluster head for the PDs P_0 and P_3, the laptop (A_1) as cluster head for P_1, and the server (A_2) as cluster head for P_2. Then, 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.

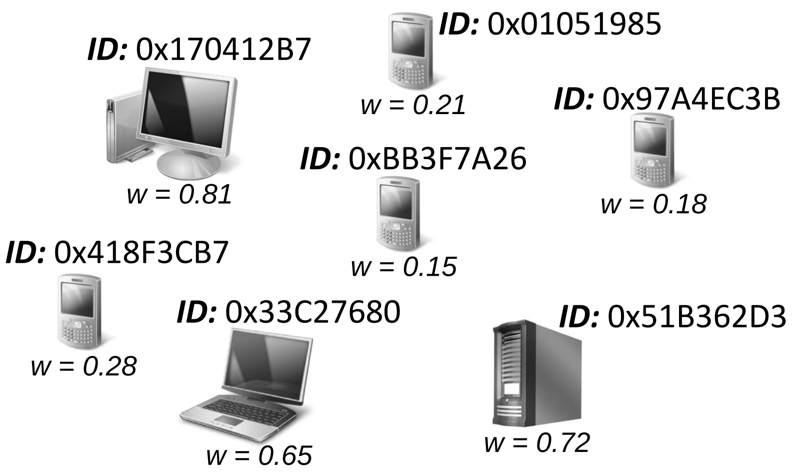


Figure 4.32.: Heterogeneous scenario for hybrid application configuration in initial situation

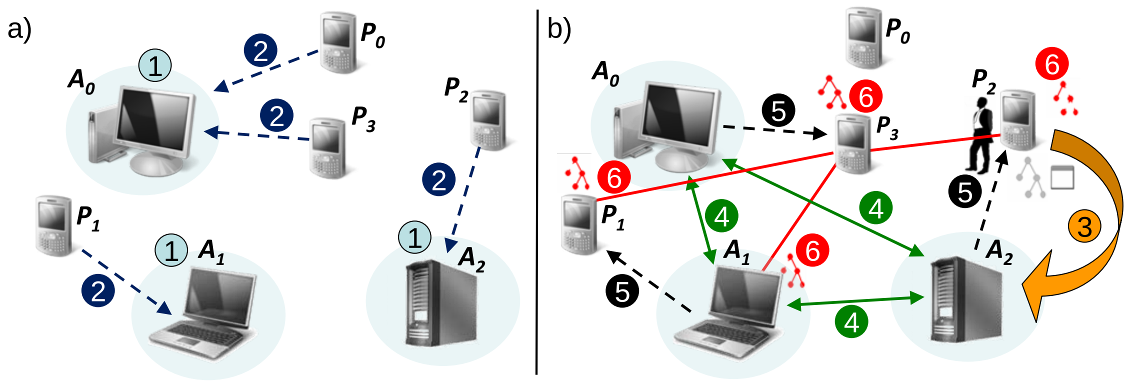


Figure 4.33.: Hybrid application configuration example

Now, regard the situation when a user wants to start an application on his or her mobile device, represented by P_2, as it is shown within step 3 in Figure 4.33b. Subsequently, the information about the application start is transmitted to A_2, as the responsible cluster head for P_2.

In the following, A_2 initiates the hybrid configuration of the application, which is shown in step 4: At first, A_2 verifies which of the dependencies can be resolved by components of its local container and the Virtual Container that represents its mapped device P_2. For the remaining unresolved functionalities, A_2 requests ADs A_0 and A_1 to resolve these dependencies: While A_0 provides the presentation file available on the conference organizer's smart phone (P_3), A_1 supplies the corresponding information about the listener's available display components for showing the slides (using the laptop's (A_1) and the smart phone's (P_1) displays) and the auditorium's multimedia components which are connected to the laptop (A_1).

Subsequently, the complete configuration is constructed by A_2 from the partial information provided by ADs A_0 and A_1. After successful configuration, the cluster heads inform their mapped passive devices whose components are used in the configuration, i.e., P_1 (display as visual output component), P_2 (display as GUI), and

P_3 (presentation file as input component), about their component configurations (step 5).

Finally in step 6, the determined components are initialized, the bindings between the components – as negotiated within step 5 – are established, and the application is successfully executed.

4.5. Evaluation

In the following evaluations, two significant results are shown:

1. In weakly heterogeneous environments, a centralized configuration approach in combination with a proactive loading of the relevant resource information from other devices reduces the latencies drastically. If the centralized configuration is calculated on the resource-richest device, latencies drop by almost 40 % on average compared to decentralized configuration.
2. In strongly heterogeneous environments, a hybrid scheme further reduces the configuration latencies by around 35 % both compared to centralized and to decentralized configuration. This is because configurations are calculated cooperatively and parallel by the resource-rich devices only, without involving the resource-poor devices.

4.5.1. Evaluation Setup

We evaluated our approaches using applications with a binary tree structure and varying tree heights between 2 and 6, yielding application sizes between 7 and 127 components. The evaluations of applications with more than 31 components were conducted to show scalability of the approaches. Unless otherwise stated, each value given in a graph represents the average of 50 measurements, respectively. Standard deviations did not exceed 15 %. We performed three different kinds of evaluation:

1. Prototype Implementation: We implemented the concepts in a prototype of the component system PCOM [BHSR04], which is described in more details in Section 6.3. We performed measurements on devices which are typically used in Pervasive Computing environments: Laptops (Pentium M 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 (11 MBit/s) as standard wireless communication technology and rely on the device and service registry provided by the communication middleware BASE [BSGR03], as described in Section 6.2 in detail. In scenarios with more than 12 devices, we used simulation or emulation.
2. Simulation: We used a discrete-event simulator of PCOM (cf. Section 6.4) to compare our algorithm Direct Backtracking (DBT) with Synchronous Backtracking (SBT) as a standard backtracking algorithm. This simulator enables fast and easy comparisons of different configuration algorithms at large scales concerning the pure computation time of the algorithm.

3. Emulation: The PCOM simulator cannot simulate the arising communication latencies of distributed algorithms. Since we wanted to perform our evaluations on the middleware systems PCOM and BASE to obtain realistic results, we performed the configuration latency measurements in scenarios with more than 12 devices on the network emulation cluster NET [HR02, GHR09] available at our institute.

Below, we will describe in more detail the evaluation methods and the parameters we used for the specific measurements in weakly and strongly heterogeneous environments.

The scenarios where we took our evaluations can generally be distinguished by the heterogeneity of the involved devices. In accordance with the different configuration approaches which have been developed (decentralized, centralized, and hybrid), we rely on three different kinds of Pervasive Computing environments for our evaluations:

- *Homogeneous MANET scenario* (denoted S_1 below): This scenario typically emerges spontaneously. There is no supporting infrastructure, the computational resources on the devices are quite low and almost homogeneous. In such environments, the configuration needs to be performed in a decentralized manner, as infrastructure support cannot be guaranteed. Common use cases of this scenario are meetings of businessmen or students carrying mobile devices like Smart Phones or PDAs. In this work, we mainly focus on heterogeneous scenarios. However, to show the general applicability of our approach, evaluations in homogeneous Ad Hoc scenarios have also been taken. We prove that our concepts are efficiently realizable in the complete spectrum of Pervasive Computing environments. Configuration in a homogeneous MANET scenario is evaluated in combination with our new PAC approach in Section 5.7.
- *Weakly heterogeneous scenarios* (S_2): Besides the resource-weak mobile devices, exactly one additional resource-rich infrastructure device is available in weakly heterogeneous scenarios. This device has significantly increased computational power. A typical realistic scenario is an office where a desktop PC or a laptop is present besides the user's mobile devices. A reasonable approach to reduce configuration latencies here is to perform centralized configuration on the single resource-rich infrastructure device.
- *Strongly heterogeneous scenarios* (S_3): In this scenario, multiple resource-rich devices as well as various mobile devices are present, which may be the case in an auditorium during a presentation where users bring their resource-rich laptops, but also their rather resource-weak Smart Phones or netbooks. In such scenarios, utilizing a hybrid configuration scheme, which combines the advantages of the centralized and decentralized approaches, increases the configuration efficiency. Hence, it improves the performance of the configuration, as we will show in Section 4.5.3. Strongly heterogeneous scenarios are also evaluated in combination with the PAC concept.

Obviously, it is possible that the kind of scenario changes during the execution of an application. Such dynamic scenarios are evaluated in combination with the determination of suitable dynamic parameters for the PAC concept in Section 5.7.

4.5.2. Centralized Configuration in Weakly Heterogeneous Scenarios

Initially, we focused on weakly heterogeneous environments, consisting of exactly one powerful device and several weak devices. In these scenarios, we first compare the new centralized Direct Backtracking algorithm to its closest competitor to show the efficiency of our algorithm. Then, we evaluate Direct Backtracking in comparison to the decentralized algorithm which was previously used in our system.

Experimental Setup

We first evaluate the centralized DBT algorithm in the PCOM Simulator and compare our algorithm to Synchronous Backtracking (SBT, [YDIK98]), which is a depth-first search algorithm that does neither perform backtracking avoidance, nor does it provide intelligent backtracking mechanisms. As mentioned in Section 3.1, advantageous algorithms like Synchronous Backjumping, Dependency-Directed Backtracking or Dynamic Backtraking either rely on a stack with monotonously growing space consumption, or change the order of the variables, which is not an option here.

After having shown that DBT significantly outperforms SBT in typical Pervasive Computing scenarios, we compare DBT to the distributed assembler that is based on Asynchronous Backtracking (ABT, [Bak05]) and was developed in an earlier work by Handte et al. [HBR05].

For our evaluations, we use the following variable system input parameters to distinguish between different scenarios:

- **Application size:** For these evaluations, we use applications with abstract components which form binary trees. By selecting different depths for the application trees, we adapt the size of the applications to analyze the behavior of the algorithms in various realistic scenarios.
- **Number of multi-optional contracts:** Multi-optional contracts have a large impact on the efficiency of specific configuration algorithms, as they represent points where the algorithm can choose among several options. Thus, when we compare DBT to SBT, we evaluate different numbers of multi-optional contracts to compare the performance of the algorithms in scenarios with difference degrees of resource availability.
- **Number of involved devices:** While the efficiency of the actual centralized configuration calculation is rather depending on the computation power of the single configuration device, the number of totally available devices has a strong impact on decentralized configuration approaches, as all devices have to communicate to each other there. Furthermore, the number of available devices also influences the time it takes to gather the resource information of the remote devices as well as distributing the results after the configuration to the other devices.

Furthermore, we evaluate the algorithms according to the following metrics:

- **Configuration latencies:** The most important metric in our evaluatiosn to identify the efficiency of an algorithm is the configuration latency, which represents the time span between the start of the configuration process and the provision of the calculated application composition. Thus, we present results of extensive measurements where we compare our centralized approach with related centralized and decentralized approaches in weakly heterogeneous environments.
- **Space and memory overhead of Direct Backtracking:** We compare DBT and SBT according to the space consumption of their Java classes on the device's hard disk, and according to the consumed Random Access Memory (RAM) while the algorithms are running.
- **Class loading latencies of the clustering framework:** As the developed pre-configuration process requires a proactive loading of the needed configuration classes, we measure the latencies that arise by this process. Please notice that these times do not affect the configuration calculation latencies, since these tasks are supposed to be performed *prior* to the configuration.

Moreover, we determine the **break-even points** when we compare DBT to SBT. There, we show that there exist only few scenarios where SBT outperforms DBT. However, by determining the break-even points in scenarios with different application sizes and fractions of multi-optional contracts, we will discover that DBT performs better than SBT in the vast majority of the evaluated scenarios.

Latency Comparison of Direct Backtracking and Synchronous Backtracking

The main goal in the development of an advanced centralized configuration algorithm was the reduction of the configuration latency that is noticeable for the user. Figure 4.34 shows DBT's latency relative to the respective SBT performance with three different application sizes (15, 31, and – to evaluate scalability of the approaches – 63 components). Values above 1.0 indicate a DBT performance that is worse than that of SBT, while values below 1.0 indicate better performance.

On average, there is an immense improvement when using DBT, which becomes particularly large when many contracts are multi-optional: SBT needs to perform many backtracking processes there, while DBT avoids most backtracking processes due to its proactive backtracking avoidance. Furthermore, it performs the remaining adaptations much smoother due to its intelligent backtracking, and avoids thrashing effects by considering the cause of a backtrack. DBT's performance gain even increases with increasing application size, as adaptation processes cover more components then. For instance, DBT induces just about 4.2 % of the latency of SBT if an application consists of 63 contracts and 14 of these contracts are multi-optional.

Nevertheless, it must be mentioned that for small fractions of multi-optional contracts, SBT performs better than DBT by up to 20 %. This is because of DBT's additional checks for avoiding conflict situations and the process of storing the back-

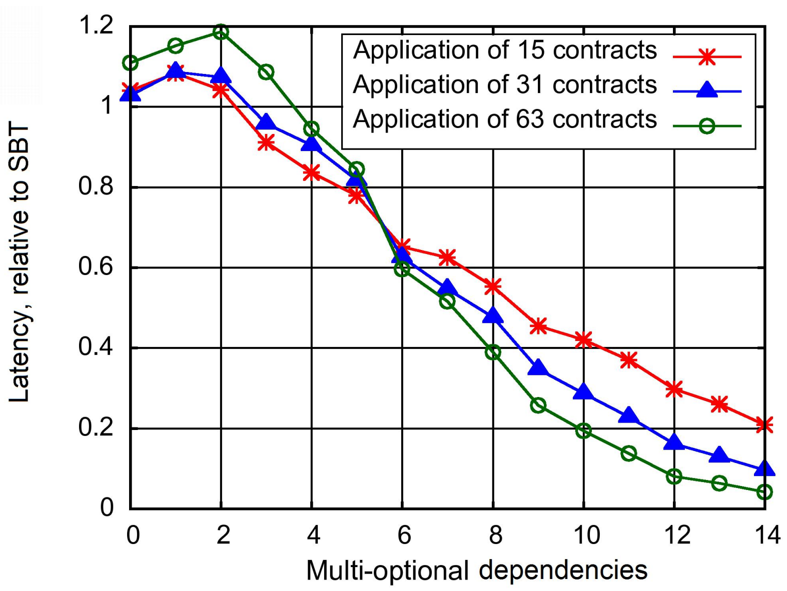


Figure 4.34.: Latency of *centralized configuration* with DBT, relative to SBT reference ($k \in \{15, 31, 63\}$)

tracking causes. Since absolute latencies were very small in those cases (in the range of few milliseconds), we consider this overhead as neglectable.

Subsequently, we determine the *break-even points*, which represent the fractions of multi-optional contracts where DBT starts to outperform SBT when an application of a specific size is used. Therefore, we perform measurements with various tree heights (i.e., application sizes) and numbers of multi-optional contracts. Figure 4.35 shows the relative performance of DBT and SBT depending on these two parameters. The figure shows that with an increasing tree height, the break-even points exponentially decrease in dependence of the fraction of multi-optional contracts. This means that for huge applications, even if just a small amount of contracts is multi-optional, DBT is the better choice regarding configuration latency. Thus, DBT is particularly helpful in reducing the latencies when extensive calculations have to be performed.

Besides the configuration latencies, we evaluated the following issues:

- **Communication overhead:** Compared to the other centralized backtracking algorithms presented in Section 3.1, no additional communication overhead arises during runtime of our algorithm, as the calculations are solely performed on a single device.
- **Success quota:** The scenarios have been created in a way that at least one valid configuration exists for each scenario. In every single simulation run, both SBT and DBT terminated successfully with a valid application configuration.
- **Memory overhead:** Compared to SBT, DBT needs to store additional information about arising conflicts, especially the contract chosen for adaptation and the contract where the algorithm has to continue after an adaptation has been performed. We measured the average random access memory consump-

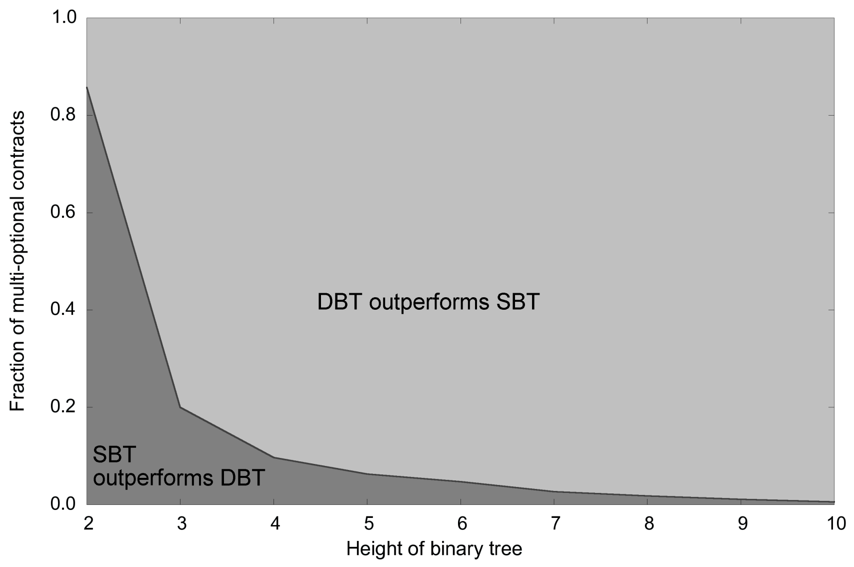


Figure 4.35.: Break-even points, depicted by borderline

tion of the algorithm on a common desktop PC and compared it to SBT. While the memory consumption of SBT was almost independent from the application size and the number of conflicts (the standard deviation was below 2 % in all runs), the overhead of DBT increased with the application size and the number of multi-optional contracts, but remained within acceptable limits. The average memory overhead of DBT varies between 8.0 % for small applications with tree height 2, and 27.6 % for very large applications with tree height 10. The maximum overhead of DBT compared to SBT in a single simulation run was 38.5 %, the absolute amount of required memory was 19.2 Megabytes. The results of the memory evaluation are shown in Figure 4.36.

- **Source code size:** While our Synchronous Backtracking implementation needs 10.8 kB of disk space, Direct Backtracking consumes about 96.9 kB, mainly because of the code overhead for proactive backtracking avoidance and intelligent backtracking.

Regarding that the algorithm is optimized for the use on resource-rich devices because of its centralized nature, the additional code overhead as well as the memory overhead do not prevent the use of DBT.

Latency Comparison of Direct Backtracking and Asynchronous Backtracking

In the previous section, we have shown that DBT significantly outperforms SBT due to its advanced backtracking mechanisms. Now, we focus on the efficient exploitation of the computation power of an available resource-rich device in a weakly heterogeneous environment now.

Therefore, we perform measurements on our prototype and compare centralized configuration that uses DBT (cf. Section 4.2.1) and VCs (cf. Section 4.3.5) to the decentralized approach [HBR05] based on Asynchronous Backtracking (ABT)

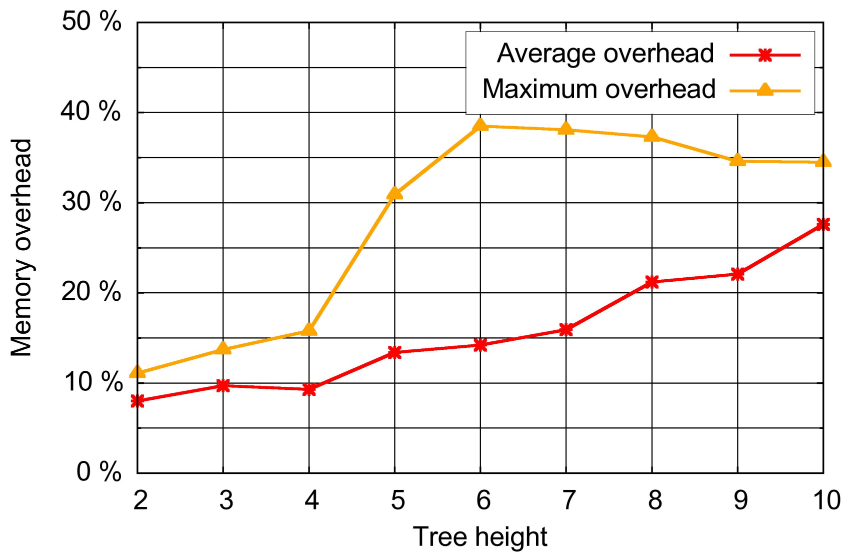


Figure 4.36.: Memory overhead of Direct Backtracking

[YDIK98] in two environments: a homogeneous environment consisting only of resource-weak devices, and a weakly heterogeneous environment that consists of one resource-rich device and up to six resource-poor devices.

As the realized configuration process uses the presented *Virtual Container* concept, the required classes have to be loaded via mobile code, which causes additional latencies. This has to happen initially, and also after environmental changes, e.g. the presence of new devices, have taken place. These latencies are presented in Figure 4.37. The figure states that in the heterogeneous environment, the class loading is performed by 33 % faster in case of two devices, and up to 55 % if five devices are involved compared to the homogeneous environment (where the classes were loaded on one of the resource-weak devices). This is because the loading of remote classes is performed significantly faster on a resource-rich device.

When the cluster head changes, a handover mechanism (implemented within the so-called *Handover Mobile Code Accessor*, cf. Section 6.3.2) immediately transmits the previous cluster head's state to the new cluster head. Because of this, the new cluster head does not need to request the configuration classes from the involved devices. Figure 4.37 displays that when the cluster head changes for a number of resource-weak devices, the use of this handover mechanism reduces class loading latencies up to 33 % in a pure Ad Hoc environment with five involved devices, and around 31 % in the corresponding heterogeneous scenario. Thus, the use of such a handover mechanism can significantly speed up class loading processes, particularly when the number of involved devices rises.

Figure 4.38 shows the measured overall configuration latencies in the same scenarios. In every single measurement, DBT calculates a valid configuration much faster than ABT. On average, DBT outperforms ABT by around 38.6 %. This is because DBT calculates configurations completely local on the cluster head, as the required resource information is proactively obtained when a cluster member is mapped and

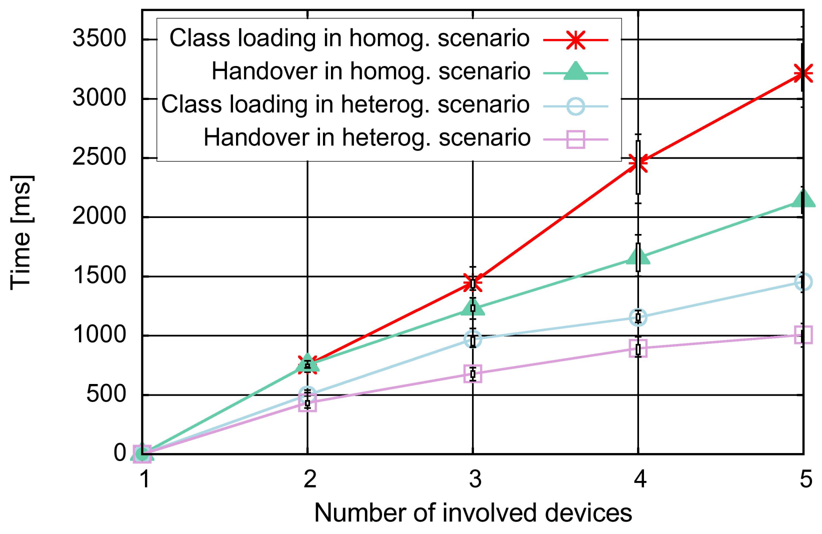


Figure 4.37.: Class loading latencies ($k = 15$)

stored in the Virtual Containers. Hence, the resource-rich cluster head does not have to wait for I/O operations.

Thus, centralized DBT configuration should be preferred over decentralized ABT configuration whenever exactly one resource-rich device is available, as this leads to considerable performance gains.

4.5.3. Hybrid Configuration in Strongly Heterogeneous Scenarios

In scenarios with several strong devices, relying on a hybrid configuration that combines the advantages of centralized and decentralized configuration as discussed in Section 4.4 is highly reasonable, as we will show in the following evaluations. To measure the performance of the hybrid approach in strongly heterogeneous scenarios, we compare its overall latencies with those of the decentralized and centralized approaches. We perform evaluations on our prototype as well as an emulation testbed in heterogeneous scenarios with two different application sizes and different numbers of devices, and 50 % resource-rich devices in each scenario.

Experimental Setup

For evaluations based on our prototypical implementation, we use six laptops and six Smart Phones with the specifications as described in Section 4.5.1. The laptops are assigned cluster heads (ADs), while the Smart Phones become cluster members (PDs) and are equally distributed among the cluster heads by the clustering scheme introduced in Section 4.4. In all scenarios, we use the IEEE 802.11b Ad Hoc mode in combination with broadcast messages between the devices. The configuration process is initiated by invoking the application anchor on one of the smart phones. Apart from the evaluations where we rely on our prototype implementation, we also

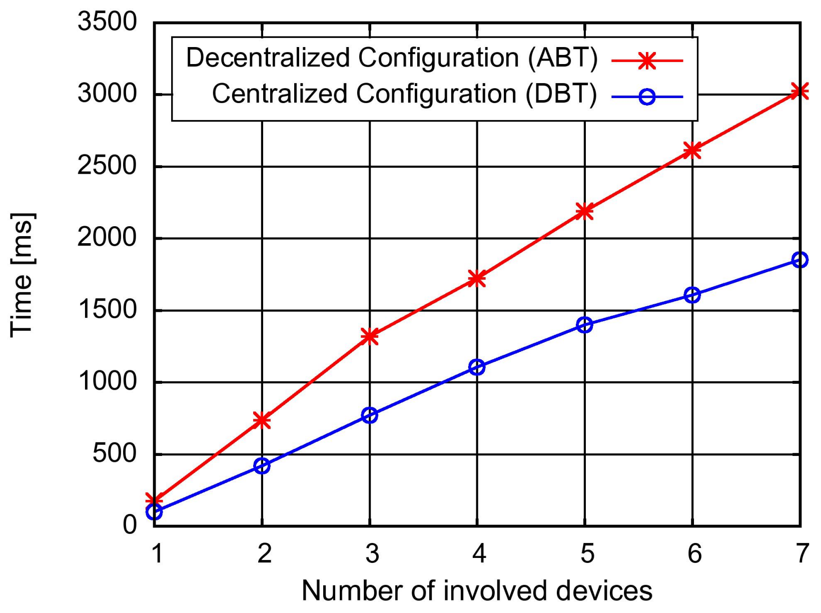


Figure 4.38.: Configuration latencies of centralized DBT configuration and decentralized ABT configuration in weakly heterogeneous environments ($k = 15$)

perform extensive experiments using the Network Emulation Testbed (NET, [HR02]) to evaluate the scalability of our approach in larger scenarios with up to 85 devices. In these evaluations, we rely on the same wireless network setup. To find a suitable value for the parameter T_1 for gathering the *unmapped* and *remapping* messages (cf. Section 4.4.2), we perform 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 is 0.57 s. Furthermore, the gathering process never takes longer than 0.83 s, even in large scenarios. As a precaution, we initialize 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 are below 15 % in all cases and below 10% in 90% of all measurements.

We use the PCOM [BHSR04] system for our evaluations. We measure the configuration latencies in a scenario with a binary tree of depth 4, i.e. $k_1 = 31$, which represents a typical application size according to the conducted survey we presented in Section 2.1.2. Additionally, we perform scalability measurements using a larger binary tree of depth 6, i.e., the application consists of $k_2 = 127$ components. In the evaluations, the laptops get an increased number of resources compared to the Smart Phones (factor 2 to 5, randomly chosen per laptop) to consider that they are usually equipped with more resources. We evaluate the hybrid scheme in comparison to the totally decentralized and centralized approaches to show the advantage over these standard approaches. We measure the message overhead and the latencies that arise at the various stages – preconfiguration, configuration, result distribution, component bindings – of the configuration process. Moreover, we evaluate an adaptation process, where only 50 % of the components need to be adapted.

Communication Overhead Measurements

Figure 4.39 shows the message overhead at the various stages of the configuration at the large scale scenario ($k_2 = 127$). In these graphs, "Hybrid-x" denotes measurements for the hybrid approach with x ADs (laptops), where $2 \leq x \leq 6$. The remaining devices (smart phones) represent the PDs.

In the preconfiguration process (Figure 4.39a), 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. The decentralized scheme does not use preconfiguration and, thus, does not produce any overhead here.

Figure 4.39b shows the message overhead required for the configuration calculations. 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 use a fixed-size application, the overhead is 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 4.39c) and the component binding overhead (Figure 4.39d) converge for the same reason.

As the devices piggyback the configuration results during the decentralized configuration process to increase efficiency of this approach, no further messages are needed for distributing the results, as it can be seen in Figure 4.39c. Compared to the centralized approach, the piggybacking increases the overhead during the configuration process by 403 kB, but reduces 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.

The overhead for establishing the component bindings (Figure 4.39d) 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 4.39e shows the total message overhead for one configuration process as the sum of all previously described overheads. The decentralized approach scales best,

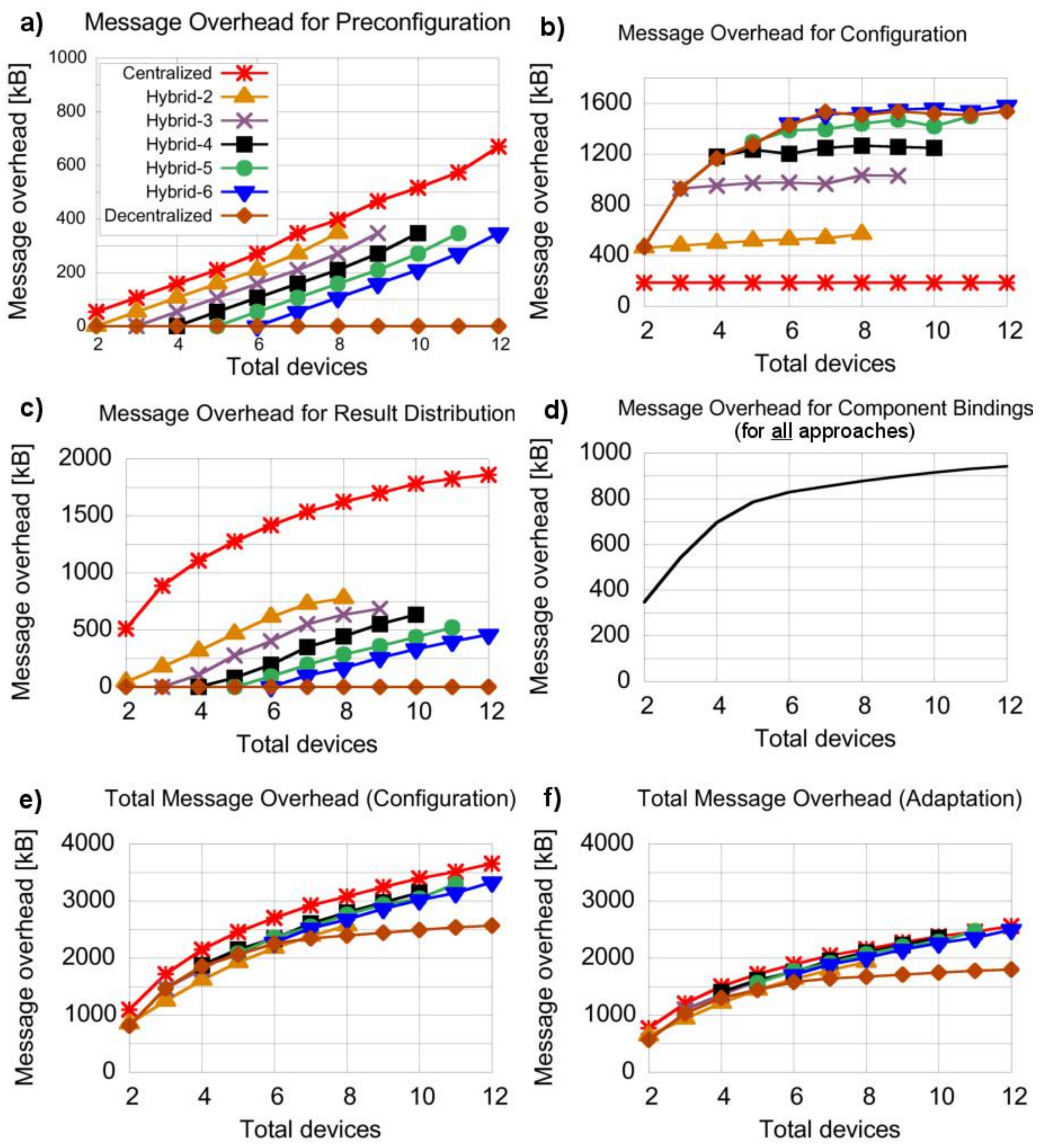


Figure 4.39.: Communication overhead at the different stages of one configuration process ($k_2 = 127$)

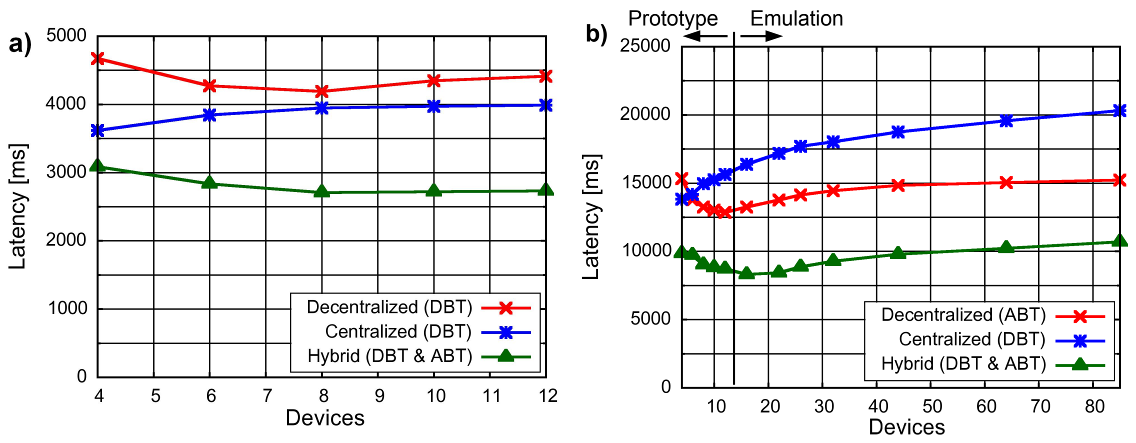


Figure 4.40.: Overall configuration latencies: a) $k_1 = 31$ components, b) $k_2 = 127$ components

as the configuration result are already piggybacked at the configuration process. Its total message overhead converges with a rising number of involved devices because of the almost constant overhead for configuration and no further distribution overhead (cf. Figures 4.39b and 4.39c). 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 of only 50 % of all application components, the total message overhead is shown in Figure 4.39f. Compared to configuration, the overheads for the centralized and decentralized schemes are reduced by around 30 %, as only parts of the application need to be re-calculated and distributed. The message overhead of the hybrid scheme decreases by 25 % only, as the remapping messages need to be sent, too. Thus, the hybrid and centralized schemes produce a comparable message overhead, while the decentralized schemes' overhead is around 22 % lower.

Configuration Latency Measurements

We compare the overall latencies of the three investigated approaches with the two mentioned application sizes ($k_1 = 31$ components, $k_2 = 127$ components) and with differing device numbers and 50 % resource-rich devices in each scenario. Figure 4.40 shows the total latencies. The evaluations on our prototype (Figure 4.40a) were performed with 4 to 12 devices, and the emulations in the large- scale scenario with $k_2 = 127$ with up to 85 devices, where each laptop holds two 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 4.40b 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 % ($k_1 = 31$) and by 44.1 % ($k_2 = 127$), respectively. These results point out the hybrid approach's scalability concerning the arising configuration latencies, as latency reduction still holds with large applications and many involved devices.

For clarification, Figure 4.41 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 4.41a. 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 4.5.3). 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 4.41e and 4.41f, as they are performed once *prior* to the configuration. However, the re-clustering latency is included in the overall adaptation latency shown in Figure 4.41f.

Regarding the latency for the configuration calculation itself (Figure 4.41b), the centralized approach performs best, as the resource-richest device locally calculates the configuration, without having to wait for partial results from remote devices. 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 approach at the configuration stage (cf. Figure 4.39b). In the hybrid approach, only the resource-rich devices perform the calculation, but message exchanges between them still take time. Thus, the latencies of hybrid configuration are slightly above the centralized scheme's latencies.

Figure 4.41c shows the latencies 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 4.39c). 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 4.41d) 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 depicted in Figures 4.39c and 4.41c, 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, respectively.

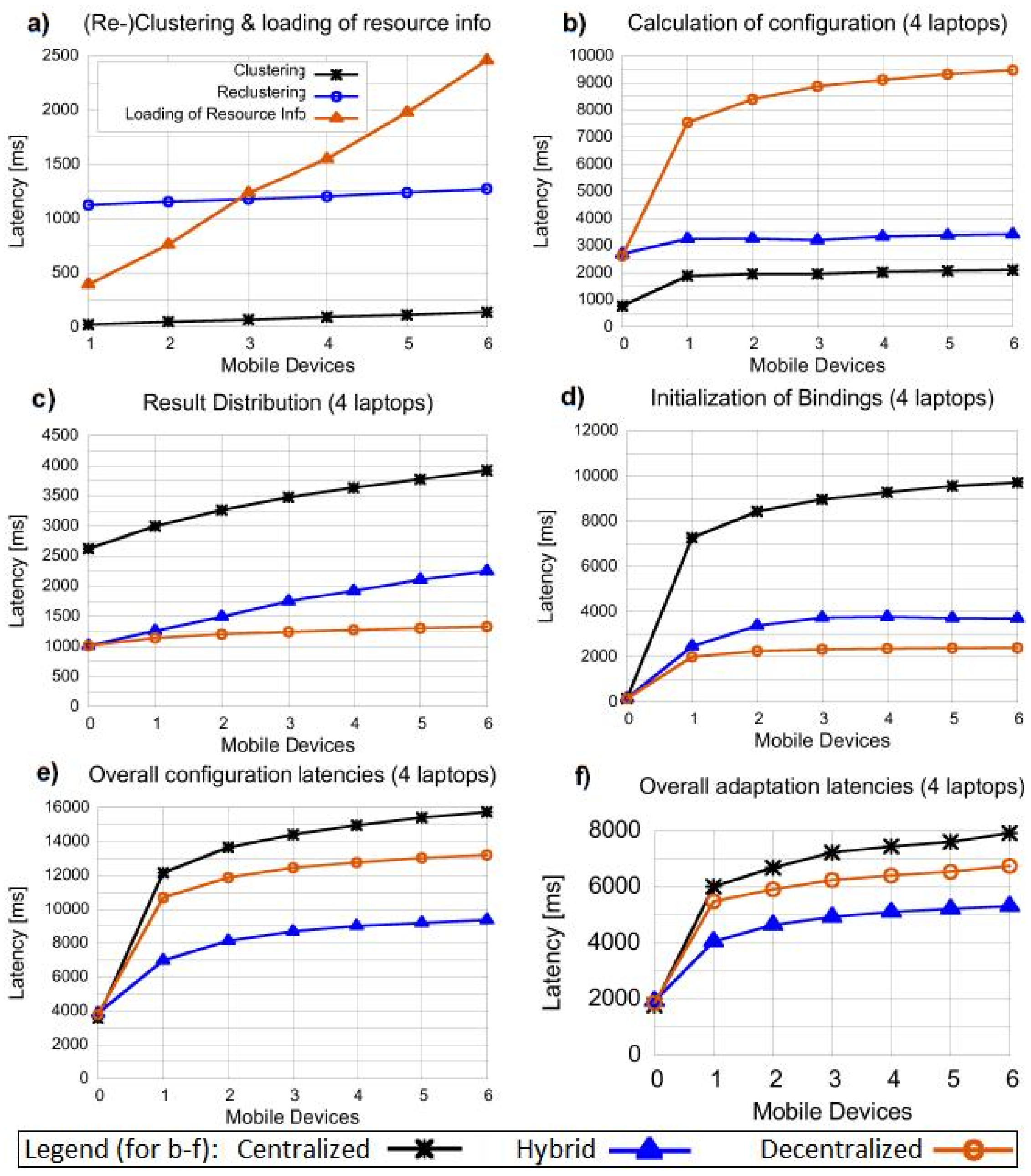


Figure 4.41.: Latency comparison at the different stages of the configuration

Figure 4.41e shows the total latencies as sum of the latencies from Figures 4.41b to 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 4.41f), 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 which only arises at the hybrid approach (cf. Figure 4.41a).

4.6. Summary and Discussion

In this chapter, we have presented our approach to efficiently exploit the heterogeneity of resource-rich Pervasive Computing environments. To achieve this, we have developed configuration approaches which fundamentally differ from the decentralized one designed for homogeneous Ad Hoc scenarios.

First of all, we have discussed our design rationale: Starting from a completely decentralized approach developed within an earlier work for Ad Hoc scenarios [HBR05], we have sketched the way towards an efficient support of the device heterogeneity which is typical in infrastructure-based environments. In such scenarios, a hybrid approach where configuration calculations are only conducted on the subset of powerful devices has shown to be more efficient than decentralized configuration, as the weak mobile devices such as Smart Phones or PDAs (as possible performance bottlenecks) are excluded from the configuration calculations. To distinguish between strong and weak devices, we suggest the use of a clustering scheme. Based on this scheme, the introduction of a pre-configuration process where configuration-relevant information is proactively transmitted to the subset of strong configuration devices increases the efficiency of the configuration.

For the flexible support of dynamically changing environments, a framework was introduced [SHR08a] which enables, based on the monitoring functionalities provided by the communication middleware BASE [BSGR03], the support of various configuration schemes and the automatic selection of a fitting scheme when an application configuration is outstanding in a specific environment. The framework provides generic interfaces to enable the simple inclusion of various clustering schemes, supports the efficient exploitation of the scenario heterogeneity via a resource-aware cluster generation, and causes only low overheads in terms of space and communication for transmitting clustering messages. Furthermore, our framework adapts the clustering schemes transparent to the application user and allows the re-use of previously developed applications. The framework additionally implies a completely new concept called *Virtual Containers* (VCs). A VC represents the emulation of a remote device for local configuration processes without requiring communications between the devices. This concept enables the proactive loading of weak devices' resource information at the strong devices and is capable to drop the configuration latencies, as parts of the configuration work are performed before an actual configuration process is initiated. The efficient support of exchangeable configuration algorithms is established by a simple selection strategy which is, however, easily extendable to support additional configuration schemes in the future. Most of the commonly known clustering schemes try to equally distribute the load among *all* nodes or aim at extending the overall network lifetime and, thus, frequently merge or split clusters, which leads to rather low cluster stability. In contrast, our new scheme balances the load only among the subset of strong infrastructure devices to minimize the (re-)configuration latencies. It performs re-clustering processes only when they are needed to obtain the balanced configuration load among the strong devices.

Decentralized configuration is usually considered to be the right choice for homogeneous MANET scenarios where the availability of any device cannot be guaranteed. However, when an additional strong device is available, e.g., a desktop PC in an office environment, decentralized algorithms calculate configurations in an inefficient way, as they distribute the configuration load among all devices. A strong device, though, has much higher computation power than the weak mobile devices. Thus, performing the configuration calculations locally on the strong device increases the efficiency and reduces the communication overhead during the configuration phase. To efficiently exploit this increased computation power of a single device in configuration calculations, we have presented Direct Backtracking (DBT, [SHR08b]), an advanced centralized configuration algorithm which features two mechanisms to render configuration processes more efficiently than standard backtracking algorithms from the research area of distributed artificial intelligence: the proactive backtracking avoidance of DBT helps to reduce the number of needed adaptations within a configuration by carefully selecting the components to be instantiated based on their resource consumption and the currently available resources. DBT's intelligent backtracking performs unavoidable adaptation processes with lower overhead than its closest competitor, Synchronous Backtracking, by considering the cause of a backtracking process and choosing the backtracking goal according to the expected adaptation overhead.

Direct Backtracking relies on the clustering framework presented in Section 4.3.2 and the established Virtual Containers, which enable completely local calculations without the need of any remote communication during the configuration phase. After a valid configuration has been found by Direct Backtracking, the configuration results are distributed to all devices whose components are part of the configuration.

Compared to its most related competitors like Synchronous Backtracking [BM04] or Dynamic Backtracking [Gin93], DBT does not rely on a large set of no-goods, considers the cause of adaptation processes, cautiously chooses the components to be instantiated to avoid adaptations, and does not need to change the value ordering of its variables, giving it an advantage over related approaches.

Centralized configuration using DBT performs fine in weakly heterogeneous environments where exactly one strong device is available. However in *strongly* heterogeneous environments, it does not distribute the configuration load on several strong devices and, thus, cannot be applied in parallel. For such scenarios, we have presented an advanced hybrid configuration scheme [SHR10] which combines the advantages of decentralized and centralized configuration: The configuration is calculated only on the subset of strong devices in a decentralized manner, whereas the strong devices gain access to the resources of the weak devices via Virtual Containers and, thus, perform centralized configuration for the weak devices. Therefore, the hybrid scheme relies on the previously developed decentralized [HBR05] and centralized [SHR08b] schemes. To uniquely identify which strong device is responsible for the configuration calculation of a weak device, we have presented an advanced clustering scheme which relies on the clustering framework discussed in Section 4.3.2, but establishes *several* clusters around the strong devices (which represent the cluster heads) and maps each weak device to exactly one strong device. This ensures a strict task sharing among the strong devices within configuration calculations. Furthermore, the new clustering scheme balances the configuration load among the strong devices by mapping similar numbers of weak devices. Because of this, bottlenecks for the configuration are avoided and the degree of parallelism within configuration calculations is maximized.

With the introduced centralized and hybrid configuration approaches and the developed framework to automatically select the configuration algorithm that is most appropriate for a specific environment, our solution reduces configuration latencies compared to standard decentralized configuration by more than 35 % in average, as our evaluation results have shown. Moreover, our system exceeds the related projects dealing with service composition and application configuration concerning the flexibility and the support of a broad range of possible Pervasive Computing scenarios by far: While many projects such as Gaia [RHC$^+$02] or BEACH [Tan01] provide system-level composition for resource-rich environments, they cannot be applied to Ad Hoc scenarios, as they rely on an existing infrastructure. Some projects for infrastructure-based environments like iRoom [JFW02] or MEDUSA [DGIR11] even rely on manual composition by users or application programmers. Manual composition is also supplied by projects such as P2PComp [FHMO04] or Speakeasy [ENS$^+$02], which are however at least applicable in pure Ad Hoc scenarios, but are not capable to efficiently exploit the device heterogeneity. Automatic configuration in Ad Hoc scenarios without efficient heterogeneity support is provided by systems such as Aura [SG02] or a previous version of our system PCOM [BHSR04]. In summary, it can be seen that only our system provides such an efficient support of various typical Pervasive Computing environments.

5 Partial Application Configurations

In this chapter, we introduce a completely new approach that integrates the results from configuration processes that have taken place before to reduce the number of contracts which actually need to be configured [SHRB13]. Thus, the configuration latency is significantly reduced in *all* of the scenarios and algorithms introduced in Section 4, as we will show in our evaluations. In Section 5.1, we give a motivation for the use of Partial Application Configurations (PACs). Following in Section 5.2, we discuss the challenges that come along with the partial application configuration concept and have to be addressed by the aspired solution. After discussing the structure of the PACs in Section 5.3, we suggest a solution which is based on the introduction of utility values for each partial application configuration (Section 5.4). As the partial application configurations are stored in a cache of limited size, the maintenance of the cache entries is addressed in Section 5.5. Following in Section 5.6, we illustrate the integration of the PAC concept into the existing configuration approaches. After presenting the results of our evaluation measurements in Section 5.7, we summarize this chapter in Section 5.8 and discuss the developed approach.

5.1. Motivation

In many typical Pervasive Computing environments, there is often a fixed set of applications, devices and components which are frequently used, e.g., a presentation application in combination with an auditorium's multimedia system, covering video projectors, microphones and the stationary speaker system. In such scenarios, the same set of components is used in subsequent configuration calculations. Thus, the involved devices undergo a quite similar configuration process whenever an application is launched. However, starting the composition from scratch every time not only consumes a lot of time, but also increases communication overhead and energy needs of the involved devices. Using pre-cached component sets in configuration processes helps in solving these problems. As they represent pre-configured application parts, 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. If all components included in a cached PAC are currently available, this PAC can be integrated into the current configuration. As the cache size is typically limited due to practicability issues, only those PACs with the largest expected utility for future configuration processes should be cached.

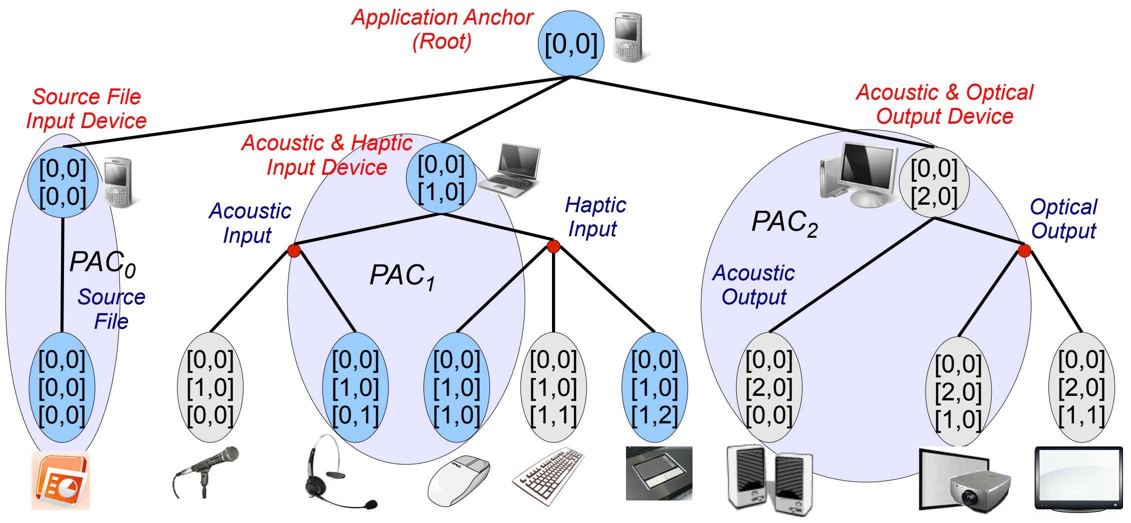


Figure 5.1.: Configuration using Partial Application Configurations (PACs)

As an exemplary scenario where the PAC concept is suited to reduce configuration latencies, consider Figure 5.1 which shows the distributed presentation application introduced before, and three PACs which have been determined and cached for their future re-use:

- When a speaker gives a talk, he or she may have the presentation file stored on his or her Smart Phone. If the speaker gives the same talk (e.g., the presentation of a new product) several times for changing audiences, the configuration always comprises the use of the Smart Phone and the presentation file maintained on the Smart Phone's internal storage. Because of this, PAC_0 which covers these two components is created and cached.
- The input components are typically provided using the speaker's laptop or a dedicated presentation device. Thus, PAC_1 – which covers the speaker's laptop (CoID [0,0][1,0]), the microphone (CoID [0,0][1,0][0,1]) and the mouse (CoID [0,0][1,0][1,0]) that are connected to the laptop – is frequently used and represents another suitable candidate for caching.
- Concerning the components for acoustic and optical output, PAC_2 – covering a speaker system (CoID [0,0][2,0][0,0]) and a video projector (CoID [0,0][2,0][1,0]), which are connected to a control PC (CoID [0,0][2,0]) – represents a third possible PAC for caching.

Figure 5.2 shows the interaction diagram of a configuration process which uses the PAC concept. Please note that in this figure, PACs are used in combination with the hybrid configuration approach. We expect that strongly heterogeneous environments offer the highest potential for the use of PACs because of the available resource-rich, typically stationary infrastructure devices. However, the PAC concept itself is independent from specific configuration algorithms and, thus, can also be used by configuration algorithms other than the hybrid one, i.e., by centralized or decentralized algorithms (cf. Chapter 4).

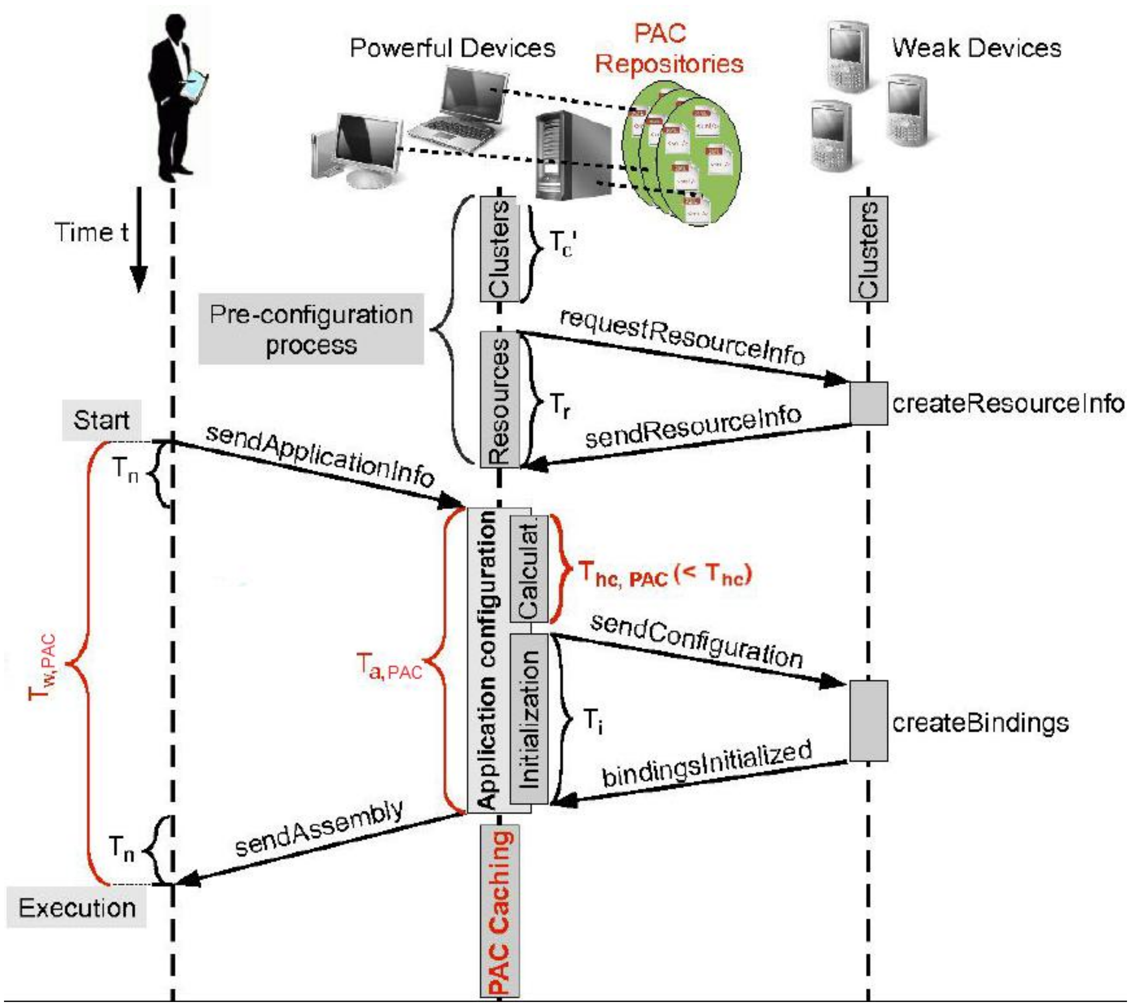


Figure 5.2.: Interaction diagram of hybrid configuration with pre-configuration process and use of PACs

As major difference compared to the configuration without PACs (as illustrated in Figure 5.2), each configuration device has access to a so-called *PAC Repository*, which contains the Partial Application Configurations that have been used in previous configuration processes and cached locally in this repository. Configuration with PACs should also use the pre-configuration process introduced in Section 4.3.2 to maintain the up-to-date resource information of the mapped Passive Devices on the Active Devices. Furthermore, the pre-configuration process is used to update the validity of a PAC: A PAC is supposed to be valid and, thus, usable within a configuration process only if all components which are covered by the PAC are currently available, meaning the device where they are resident is up and the component is not used by any other currently executed application. More formally, the validity $V(P)$ of a PAC P consisting of n components $C_1, \ldots, C_n$ can be determined by

$$V(P) = V(C_1) \wedge V(C_2) \wedge \cdots \wedge V(C_n), \tag{5.1}$$

where $V(C_i)$ represents the validity of a component C_i ($i \in 1, \ldots, n$). In the configuration phase, the applied configuration algorithm tries to configure as many components as possible via the cached PACs to reduce the number of contracts which actually need to be configured, i.e., by matching the available components' offers of functionality with a contract's requirements. As the simple integration of pre-cached components is much faster than the standard configuration (we will show more details on that issue in our evaluations in Section 5.7), the configuration latency $T_{hc,PAC}$ is reduced compared to the latency T_{hc} of hybrid configuration which does not rely on a PAC framework. As the initialization phase remains unchanged, the application configuration latency $T_{a,PAC}$ becomes the sum of the configuration calculation latency $T_{hc,PAC}$ and the component initialization latency T_I. In consequence, the total latency

$$T_{w,PAC} = 2 \cdot T_n + T_{a,PAC} = 2 \cdot T_n + T_{hc,PAC} + T_I, \tag{5.2}$$

where T_n represents the network latency, is also reduced compared to configuration without PAC use.

After a successful configuration process, the configuration devices update their PAC Repository by including new PACs and, in case of exceeded cache space, remove selected PACs from the repository. We will present more details on the utility of specific PACs in Section 5.4, and introduce cache replacement strategies in Section 5.5.

5.2. Challenges

The goal of the PAC approach is to reduce the configuration load on the involved devices by providing a cached set of partial configurations which have been used in previous configurations. To achieve this, the system needs to be extended by several services to solve the challenges which arise.

First of all, a discussion about the structure of the PACs is needed, as the number of possible PACs may be considerable in resource- rich environments with many different devices and components. Hence, limiting the number of valid PACs to be used in application configurations significantly helps to reduce storage and computation overhead for the involved devices. An issue that arises is the cache space that has to be provided to enable the storage of these configurations. As a cache of unlimited space does not exist, the size of the cache needs to be restricted. A question that arises is about this cache size: On the one hand, the cache space should be minimal to reduce the requirements on the involved devices, since some mobile devices still have to face close restrictions on their internal disk space. On the other hand, a large-sized cache is able to store an enormous number of configurations and usually reduces the cache miss rate drastically, which leads to a much higher PAC use rate.

The restriction of the cache size introduces another issue that becomes relevant when the cache space is exceeded: Which of the pre-cached partial configurations should be maintained, and which should be removed from the cache to provide space for currently more relevant PACs? To solve this challenge, *cache replacement strategies* are typically provided that decide which cache entries are removed when the cache space is exceeded. Typical replacement strategies [PB03] consider influence factors like the usage recency (e.g., LRU), the usage frequency (e.g., LFU), the time when an entry was stored (e.g., FIFO), or a (possibly weighted) combination of several factors for evaluating cache entries and determining a unique order for the required replacements.

When a user wants to start a distributed application and, thus, a configuration process is initiated, the configuration algorithm needs to access the cached partial configuration and integrate them into the assembly. Therefore, the existing configuration algorithms have to be adapted to enable the automatic use of the cached components. Moreover, it has to be ensured that only those PACs are used within the configuration whose components are currently available.

We will present our solutions to these challenges in the following sections.

5.3. Structure of Partial Application Configurations

We construct PACs in a **bottom-up** approach, beginning with the application tree leaves (which represent basic functionalities available in a specific environment) and combining them to higher-level functionalities. Consequently, PACs consist of completely dissolved dependencies which are specific for an environment and may be used by various applications. Because of this, PACs are *environment-based*.

This model partitions the application tree based on the type of the devices hosting the involved components, i.e., based on the Pervasive Computing scenario present at the time of configuration. The partitions that are hosted by infrastructure devices are added to the partial solution of the application and are reused when the application starts in the future on the same device and after proper validation that the respective components are available at configuration time.

In practice, the environment-based partial solution model describes a subtree of the whole application tree that is completely resolved. This subtree always includes a subset of the application tree leaves. This particular model is suitable if the same initial resources are always available and offered by the environment. A typical scenario where such PACs may be extremely useful are heterogeneous environments like auditoriums where some infrastructure devices are assumed to be present at all times. These devices are typically used by many applications to utilize their services, like a *Smart Board* for displaying presentation slides. Hence, the respective configuration processes are quite similar whenever these applications are started.

The part of the application tree that represents components which are hosted by these devices is added to the partial solution after the first configuration process is completed. This partial solution is reused whenever the application is started again.

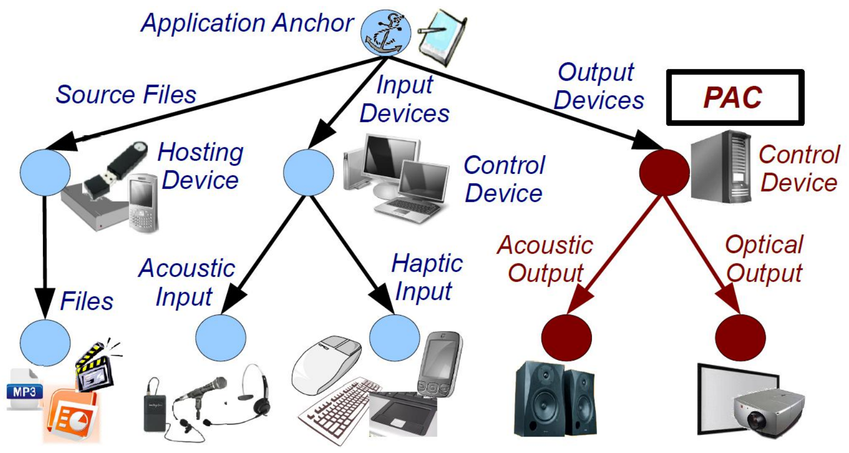


Figure 5.3.: Use of a PAC in the distributed presentation application

As a concrete example, consider Figure 5.3, showing again the distributed presentation application as it could be represented in an auditorium where a research conference takes place. In the application tree, there are three components of the right subtree highlighted as PAC. This partial configuration represents this environment's output devices: the room's speaker system as acoustic output, the video projector in combination with a screen as visual output, and the infrastructure PC as control device for these output components.

An alternative way of structuring PACs is to model them in a **top-down** approach, where the application root – or, as we call it, the *anchor* of the application – and a subset of the application tree components which are connected to each other are part of the PAC. Moreover, a **hybrid** approach which includes the application anchor, one or more of the leaves, and a subset of the components in between may be possible. However, both the top-down and the hybrid approaches may face one huge drawback due to partially unresolved dependencies. As we proceed in a depth-first approach in our configuration processes, the configuration algorithm would first load the anchor-based PACs, and then try to resolve the remaining (i.e., the lower-level) dependencies of the respective subtrees. However, when deciding to use a specific PAC for an application execution, it is not yet known if the lower-level components that represent the basis for the higher-level functionalities of the PAC are currently available, leading to a higher number of adaptations that have to be performed. This problem becomes particularly severe when the respective PAC covers a large number of components, as a larger part of the application needs to be re-configured then. But especially large PACs promise large reductions at the configuration time, as they significantly reduce the number of components which have to be configured in the traditional way.

The severity of this drawback could be decreased when introducing speculative calculations into the application configuration that reduce the number of adaptations that have to be taken during a configuration. However, speculative calculations and proactive PAC creation are beyond the scope of this thesis and left as future research directions, as we state in our outlook (cf. Sections 7.2.2 and 7.2.4).

5.4. PAC Utility Value

The utility value $u_p(t)$ of a PAC p at a specific time t is introduced for expressing the expected gain when a PAC is used within a configuration process at time t. The two most important factors for defining the value of a PAC for automatic configuration processes are the *frequency* of use of the PAC within configuration processes, and the expected *latency reduction* when a PAC is used. Both of these factors are depending on the size and the involved devices within a PAC: On the one hand, small PACs that cover just few components usually tend to involve only a low number of different devices. As all components of a PAC need to be available at the same time to make this PAC thus *usable* for configuration, small PACs whose components are spread among only few devices are more often usable compared to large PACs which involve a larger number of different devices. On the other hand, the usage of large PACs leads to a much higher expected configuration latency reduction, as they cover a larger subset of the application. However, large PACs need more space in the cache than smaller PACs. Thus, neither providing only large PACs, nor only small PACs is obviously the preferable choice. Therefore, we use a utility function which was initially suggested for web caching [LCK+01] and also used for location-based routing [DR08]. This function defines the utility, $u_p(t)$, of a PAC p at time t in our context. The utility value considers the recency and the frequency of a PAC's usage when it is updated, yielding a mix of Least Recently Used (LRU) and Least Frequently Used (LFU) as replacement strategy. Each device which is involved in the configuration process updates the utility of each cached PAC at time t_c of a configuration process. The utility $u_p(t) \in [0,1]$ of a specific PAC p at the current configuration's time $t_c = t' + x$ (where t' = time of the previous configuration, x = time span between the previous and the current configuration) is calculated as follows:

$$u_p(t'+x) = \begin{cases} u_p(t') \cdot f(x) + f(x), \text{ if PAC } p \text{ was usable} \\ u_p(t') \cdot f(x), \text{ if PAC } p \text{ was unusable} \end{cases} \tag{5.3}$$

When $f(x)$ is a monotonously decreasing function, the utility value of a PAC is increased if it was usable at configuration time t_c (i.e., all respective components were available), and reduced if it was not usable. As proposed by related approaches [BKMN02, DR08], we use f(x) $= 0.5^{\lambda \cdot x}$ with parameter $\lambda \in [0,1]$ that defines the influence of reference recency and frequency, leading to the above mentioned Least Recently/Frequently Used (LRFU) policy. With a used radix of 0.5 in $f(x)$, switching to simple LFU ($\lambda = 0$) and LRU ($\lambda = 1$) strategies is easily possible by just adjusting λ. The question of how to choose λ is discussed later in the evaluation section 5.7.3. As we rely on a communication middleware that provides a device registry [BSGR03], all configuration devices have a global view on the currently available devices. Hence, the current utility values of the cached PACs are identical at each configuration device. With this utility value definition, every past reference contributes to the current utility of p, but just the utility value at the last reference time needs to be stored. Thus, the summary of the reference history of a PAC is stored efficiently with constant space overhead.

With this scheme, we aim at maximizing efficiency of the cache usage by storing only those PACs which are supposed to be often usable in the near future due to their historical usage. Regarding the initial utility values for PACs when they are cached, Dürr et al. [DR08] suggest an initial utility of $a = 0$ for new cache entries. However, this leads to the fact that new PACs are initially stored at the end of the cache table. Hence, there is the risk for new PACs to be removed from the cache quickly after their creation, considering the typically high degree of dynamics in Pervasive Computing environments. Contrary to this, giving new PACs the maximum possible initial utility value initially puts them at the beginning of the cache table. This could lead to the problem of a polluted cache: if PACs are no longer usable shortly after their creation, they would still stay in the cache for a long period due to their high initial value. Thus, a reasonable compromise is to give new PACs the average value of all PACs $p_1, \cdots, p_n$ from the current cache table as an initial utility value, i.e.

$$u_p(t' + x) = \frac{\sum_{i=1}^{n} u_{p_i}(t' + x)}{n}. \tag{5.4}$$

Accordingly, new PACs are placed around the middle of the cache table. If the cache is empty, all PACs stored after the first configuration process get an initial utility value of 0.5.

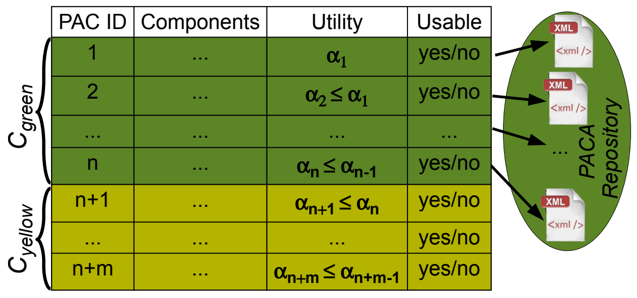
Besides the PAC utility value determined by LRFU, alternative well-known approaches to identify which PACs are to be removed when the cache space is exceeded are the following:

- **First In First Out (FIFO)**: If the cache size is exceeded, the PACs which are in the cache for the longest time are removed first, as they are supposed not to be valid anymore because of the dynamic environment.

- **Remove Smallest First (RSF)**: If the cache size is exceeded, the smallest PACs (in terms of covered components) are replaced first, since these PACs only cover a low portion of the overall application and, thus, do not help significantly to reduce the configuration problem.

- **Remove Largest First (RLF)**: If the cache size is exceeded, the largest PACs are replaced first, as they are composed of many different components. Thus, it is very likely that they are not usable at the moment because one or more components are currently not available.

We show by evaluation in Section 5.7.3 that LRFU with the determined parameters significantly outperforms these approaches concerning the arising configuration latencies when PACs are used.



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

Figure 5.4.: Cache structure of C with C_{green}, the PAC Repository, and C_{yellow}

5.5. PAC Cache Maintenance

According to the introduced PAC utility values, each active configuration algorithm can easily determine which PACs should be cached on which devices for their further use in configuration processes. As the available cache space is limited, it has to be used as efficient as possible.

The structure information of each PAC is stored in the PAC Cache C of size $|C|$. Figure 5.4 shows that C is divided into two areas, C_{green} and C_{yellow}, that hold the respective PAC information described in the following. C_{yellow} can be regarded as a "waiting area", allowing PACs to increase their utility values due to recent and frequent usage. 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 validity $V(P)$, as stated in Equation 5.1 as the conjunction of the availabilities of the involved components. Only currently available PACs are usable in a configuration.

According to the cache division in C_{green} and C_{yellow}, we distinguish between two different types of PACs that are relevant for PAC configuration and are stored in the respective fraction of the cache[1]:

- **Green PACs** represent the PACs with currently highest utility value among all PACs. The information that is relevant for configuration processes – the PAC components, the involved devices and the utility values – is stored in the cache table. Furthermore, the complete information about the involved components and devices is stored in the component middleware's specific XML format in the so-called *PAC Repository*. The XML file includes information about the involved components – the functionality provided by them and the hosting device – as well as their interdepencies within the application tree, i.e., the PAC structure. This complete XML representation is called the *PAC Assembly* and enables the integration of the PAC into the complete *application assembly* within a configuration process.
- **Yellow PACs** have been used in configuration processes in the past, but not as frequently or recently as green PACs, leading to a lower utility value.

[1] A similar concept is used for efficient geographic routing by Dürr and Rothermel [DR08]

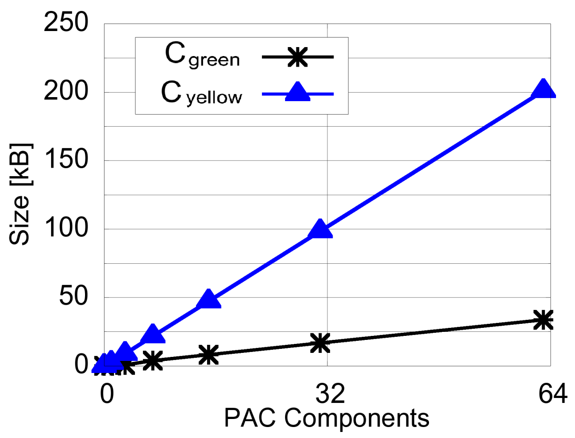


Figure 5.5.: Comparison of PAC space overhead in C_{green} (entry in cache table & XML file) and C_{yellow} (only cache table entry)

Yellow PACs comprise only the cache table entries. Thus, the information is stored in a more compact form to reduce their space consumption. Thus, a yellow PAC consumes only around 1/6 of the space that the corresponding green PAC consumes, as shown in Figure 5.5. However, a yellow PAC is not directly usable in configuration processes, as the configuration algorithms rely on uncompressed XML assemblies. But as its utility value is recorded, it may become green in the future. Then, the XML assembly is automatically constructed from the information maintained in the yellow PAC. Relying on yellow PACs helps to reduce cache miss rates in most Pervasive Computing scenarios, 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. Hence, they are not relevant for application configuration and not considered anymore in the following.

C_{green} is of limited size $|C_{green}| < |C|$. $|C_{green}|$ defines how many green PAC can be stored in the cache. Correspondingly, the maximum allowed size for C_{yellow} is $|C_{yellow}| = |C| - |C_{green}|$. A main issue in the evaluations is to find the optimal partitioning of $|C|$ between $|C_{green}|$ and $|C_{yellow}|$ in specific environments to maximize cache efficiency. As utilities of PACs may change over time because of dynamic pervasive environments or device failures, yellow PACs may obtain larger utility values than green PACs. This means a *replacement condition* is required which defines when a PAC in C_{yellow} replaces a PAC in C_{green}. The replaced PAC is moved from C_{green} to C_{yellow}. Similar to Dürr and Rothermel [DR08], we define the Least-Green PAC, p_{lg}, as the PAC with the minimum utility value in C_{green} at a specific time t:

$$\forall p \in C_{green} : u_{p_{lg}}(t) \leq u_p(t) \tag{5.5}$$

Then, PAC p' as the PAC with highest utility value within C_{yellow} at configuration time t_c replaces p_{lg}, if

$$u_{p'}(t_c) > u_{p_{lg}}(t_c) \tag{5.6}$$

Subsequently, the PAC Assembly is created and added for the (now green) PAC p', while the assembly of p_{lg} is removed. These replacements are continuously accomplished until the replacement condition from Equation 5.6 is no longer fulfilled.

One problem that may arise are oscillating PACs, switching their status frequently from green to yellow and vice versa. However, this would mean that the corresponding devices are *either* at the border of the configuration environment and often lose contact to the other devices (which makes it likely that they are not interested in the application execution and, thus, seldomly involved in the configuration), *or* they consecutively appear and disappear at a frequency similar to the application execution period, which is a rather unlikely user behavior according to conducted user mobility studies. We refer to Section 5.7.2 and [SHRB13] for further details concerning this issue.

5.6. Configuration involving PACs

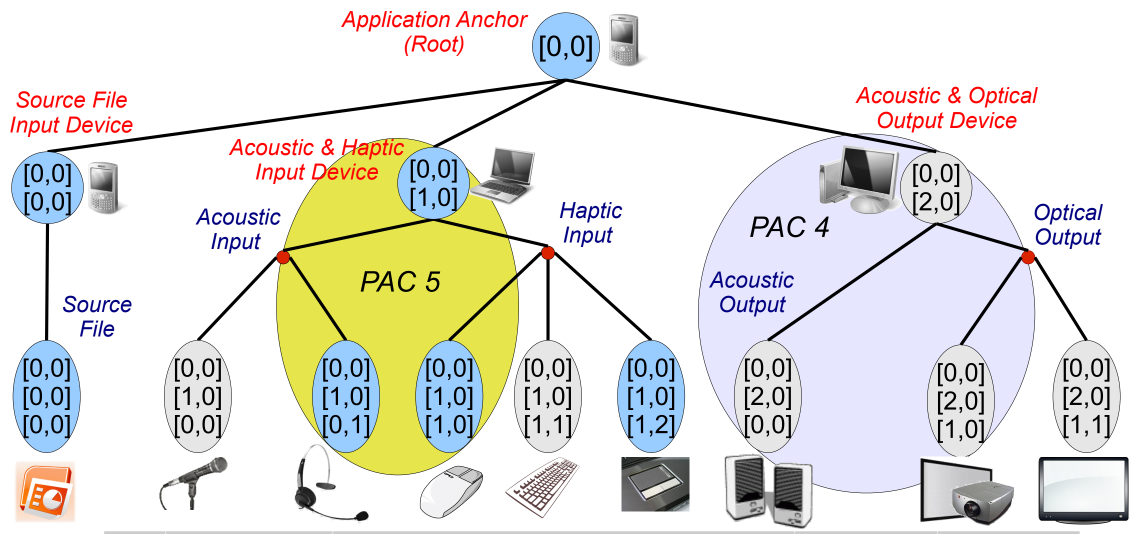
The PAC concept was developed without focusing on specific configuration approaches. It can rather be seen as an *extension* which provides additional functionality to the previously developed algorithms. Thus, the changes to the configuration approaches are of minor nature: When the configuration algorithm tries to configure a contract `ctc`, it first scans the list of green PACs in the PAC Repository for currently usable PACs which fulfill the constraints given by `ctc`. Furthermore, the algorithm tries to avoid remote communications at the subsequent component binding phase by selecting the PAC with the highest degree of locality, i.e., the PAC where the highest number `maxLocalComps` of components is locally available at a configuration device (meaning, the device hosts the components). For each PAC, the number of locally available components is obtained via the function `getLocalComps()`, which counts the number of local components. If this criterion does not lead to a definite PAC, the highest number of components (variable `maxVcComps`), which are accessed via the Virtual Containers established locally at a configuration device (function `getVcComps()`), is chosen as a secondary selection criterion. In case of continuing ambiguity, the PAC with highest utility value (i.e., with lowest rank within the PAC Repository) is chosen.

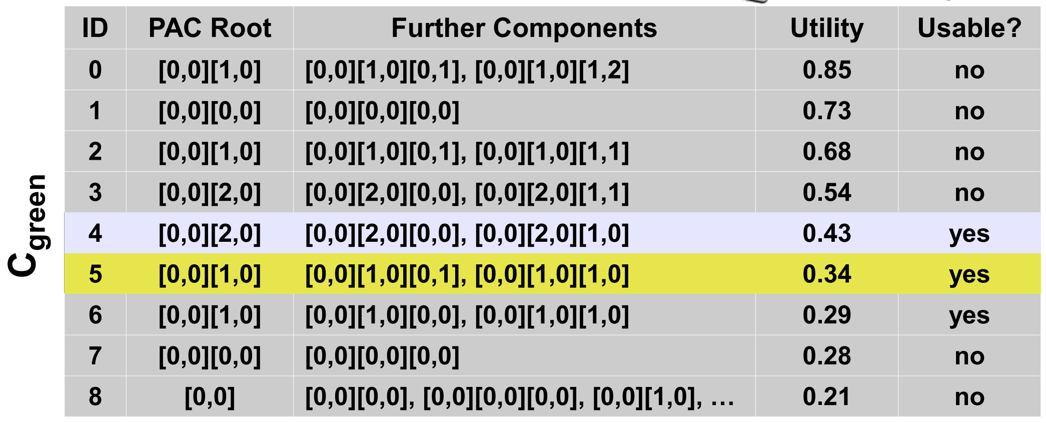
The integration of PACs into configurations is exemplarily shown in Listing 5.1 for the centralized Direct Backtracking (cf. Section 4.2.1) configuration approach. The code that integrates PACs covers only 14 lines of the pseudo-code (lines 4 to 17) (plus the additional (simple) functions to determine the number of local components and VC components) to find a usable PAC that matches the requirements of the application. The remaining code represents the standard approach of Direct Backtracking to proactively avoid backtracking processes, as presented in Section 4.2.2.

```
determOption() {
    Contract ctc = getCurrentContract();
    boolean continue = true;
    Pac[] greenPacs = pacRepository.getGreenPacs();
    int maxLocalComps = −1, maxVcComps = −1, maxValue = −1;
    for (int p = 0; p < greenPacs.length; p++) {
        if (greenPacs[p].isUsable() && greenPacs[p].getRootComp().matches(ctc)) {
            if ((greenPacs[p].getLocalComps() > maxLocalComps)
                || (greenPacs[p].getLocalComps() == localComps
                    && greenPacs[p].getVcComps() > maxVcComps)
                || (greenPacs[p].getLocalComps() == localComps
                    && greenPacs[p].getVcComps() == vcComps
                    && greenPacs[p].getValue() > maxValue)) {
                comp = greenPacs[p].getRootComp();
                markCoveredComponentsAsConfigured();
                continue = false;
    }}}
    int i = 0; // Index of selected component in list
    while (continue) { // standard proceeding as before
        comp = option[i];
        while (comp.consumed >= comp.getContainer().getAvailableResources()) {
            i++;
            if (i > list.length)
                performBacktracking(); // no suitable component found
            else
                comp = option[i];
        }
        if (freeResources >= comp.consumedResources)
            continue = false;
        else {
            i++;
            if (i > list.length)
                performBacktracking(); // no suitable component found
    }}
    return comp;
}
```

Listing 5.1: Integration of PACs in Direct Backtracking

To clarify how an application configuration that uses cached PACs is executed, regard again the distributed presentation application introduced in Section 2.1.2, which is shown in Figure 5.6. When the user starts the application, the configuration algorithm tries to configure the application with the use of the nine green PACs that were cached before and are shown in the table in Figure 5.6. According to the depth-first search manner which our algorithms follow (cf. Section 2.1), the configuration starts at the application root (CoID [0,0]). There is one pre-stored PAC (ID




C_{green}

ID	PAC Root	Further Components	Utility	Usable?
0	[0,0][1,0]	[0,0][1,0][0,1], [0,0][1,0][1,2]	0.85	no
1	[0,0][0,0]	[0,0][0,0][0,0]	0.73	no
2	[0,0][1,0]	[0,0][1,0][0,1], [0,0][1,0][1,1]	0.68	no
3	[0,0][2,0]	[0,0][2,0][0,0], [0,0][2,0][1,1]	0.54	no
4	[0,0][2,0]	[0,0][2,0][0,0], [0,0][2,0][1,0]	0.43	yes
5	[0,0][1,0]	[0,0][1,0][0,1], [0,0][1,0][1,0]	0.34	yes
6	[0,0][1,0]	[0,0][1,0][0,0], [0,0][1,0][1,0]	0.29	yes
7	[0,0][0,0]	[0,0][0,0][0,0]	0.28	no
8	[0,0]	[0,0][0,0], [0,0][0,0][0,0], [0,0][1,0], …	0.21	no

Figure 5.6.: Exemplary configuration process involving the PAC Repository

8) in the table, which represents a previously used complete configuration. However, this PAC is currently not usable because of components that are not available at the moment (e.g., the external hard disk which was used to access the presentation source file). Thus, the algorithm continues on the next level of the tree and tries to configure the instance with CoID [0,0][0,0]. For this component, it finds PACs 1 and 7 in the table. However, both of them are not usable, as the corresponding devices are no longer available. So, the algorithm configures this part of the application without PAC usage by determining suitable components, e.g., the current speaker's USB stick that holds his or her presentation file. This is performed using the centralized [SHR08b], hybrid [SHR10], or decentralized [HBR05] configuration approach, according to the current environmental conditions. Then, the algorithm continues with the next sub-tree – the acoustic and haptic input devices, beginning with the component with CoID [0,0][1,0]. C_{green} currently holds PAC 5 which uses a headset (CoID [0,0][1,0][0,1]) as input device for the user's voice, and the mouse (CoID [0,0][1,0][1,0]) of the presentation laptop (CoID [0,0][1,0]) to allow the speaker to switch between the slides. As all corresponding components are locally available (which is not the case for the alternative PAC 6, as the microphone is connected to another device), the algorithm uses PAC 5 and does not need to configure these

components in the standard way. For the output devices, the algorithm finds the cached PAC 4 which uses the stationary speaker system (CoID [0,0][2,0][0,0]) and the video projector (CoID [0,0][2,0][1,0]) that are both connected to the multimedia system's control PC (CoID [0,0][2,0]) and currently available. Subsequently, the application configuration is finished, and the presentation application is executed. Consider that only the components in the left sub-tree needed to be configured in the standard way, while the rest of the application was resolved just by importing the PACs stored in C_{green}, which takes much less time than standard configuration.

5.7. Evaluation

In the following evaluations, we show that the re-use of previous configurations' results decreases the configuration latencies in *all* of the regarded homogeneous and heterogeneous environments. We determine reasonable static values for the most relevant cache parameters – the overall cache size $|C|$, the fractions of $|C|$ which are provided for C_{green} and C_{yellow}, and the factor λ which determines the influence of the recency and the frequency for the LRFU cache replacement strategy. Moreover, we evaluate parameter values which automatically adapt to dynamically changing environments. We show in the following performance evaluations that latencies are reduced by up to 31 % when static parameters are used, and even by up to 66 % when adaptive parameters are used.

5.7.1. Evaluation Setup

We evaluate the PAC approach using applications with a binary tree structure and varying tree heights between $h_1 = 2$ and $h_2 = 4$, yielding application sizes between $k_1 = 7$ and $k_2 = 31$ components. We measure the time it takes to configure a *single* application. Unless otherwise stated, the values given in the graphs represent the average of 50 measurements. Two different kinds of evaluation are performed:

1. Prototype Implementation: The PAC concepts were implemented in the prototype of PCOM [BHSR04] (cf. Section 6) for scenarios with up to 12 devices. Measurements are performed using Laptops (Core 2 Duo, $2 \cdot 2.0$ GHz) as powerful devices, Smart Phones (PXA 270 CPU, 528 MHz) as weak devices, and IEEE 802.11b (max. 11 MBit/s) as standard wireless communication technology.

2. Emulation: We perform the configuration latency measurements in scenarios with more than 12 devices on the network emulation cluster NET [HR02, GHR09], as the inclusion into this emulation cluster allows to obtain realistic results by executing our system software PCOM's code in a distributed large-scale environment.

In a first step, we investigate three different static scenarios, correspondingly to the environments introduced in Section 2.1.1: (1) a homogeneous Ad Hoc environment (denoted by S_1), consisting only of weak mobile devices; (2) a weakly heterogeneous

Environment & Reference	Scen.	α_{st}	$x_{m,st}$	α_{ict}	$x_{m,ict}$
ACM SIGCOMM 2001 [BVBR02]	Infra	1.78	1.23	-	-
UCSD Campus, San Diego [MV03]	Infra	1.58	0.33	1.33	0.52
Corporate Environment [BC03]	Infra	2.36	0.55	-	-
Dartmouth College Campus [HKA04]	Infra	1.44	0.46	1.33	0.94
MIT Laboratory, Boston [EP06]	Infra	-	-	1.4	0.35
MIT Laboratory, Boston [EP06]	Ad Hoc	-	-	1.21	0.43
Microsoft Cambridge Res. [CHC+05]	Ad Hoc	1.5	0.65	1.6	0.94
IEEE Infocom 2005 [HCS+05]	Ad Hoc	2.1	1.29	1.4	0.71
Average values		**1.79**	**0.845**	**1.38**	**0.674**

Table 5.1.: Overview of mobility studies' parameters

environment with one powerful infrastructure device (S_2); (3) a strongly heterogeneous environment (S_3), consisting of 50 % powerful devices and 50 % weak devices. Then, we evaluate the PAC concept in dynamically changing environments. According to the typical application sizes we determined in Table 2.1 of Section 2.1.2, we use applications consisting of 7 (S_1), 15 (S_2), and 31 (S_3) components for these evaluations. Then, we switch to dynamic cache parameters in Section 5.7.4 to allow the adaptation of the configuration process to changing environments.

5.7.2. Mobility of Users

Several studies analyzed the mobility of users in typical infrastructure-based as well as Ad Hoc pervasive scenarios, such as conference environments [BVBR02, HCS+05], University campuses [MV03, HKA04] or research laboratories [EP06, CHC+05]. In this respect, we do not focus on specific mobility models, but rather on the probability distributions for the (un-)availability of devices. The *Session Time (ST)* determines the average time a user's device is connected to a network, the so-called *connection session length* [BVBR02]. It represents the interval until the session is terminated, e.g., because the user is mobile. Furthermore, the *Inter-Contact Time (ICT)* specifies the average time span a user device is disconnected between two sessions, i.e. the time interval over which devices are not in contact [KLBV07]. Thus, the time can be divided into alternating ST and ICT slots. Karagiannis et al. [KLBV07] found that the ICT of users in four studies [MV03, EP06, CHC+05, HCS+05] can closely be approximated by the *General Pareto Distribution (GPD)* for times up to several hours. The GPD represents a specific power law distribution function: Let X be a random variable with Pareto distribution $Pr(x_m, \alpha)$ with the *scale parameter* $x_m > 0$, and the *shape parameter* $\alpha > 0$. Then, the probability that X is greater than some number x is given by

$$Pr(X > x) = \begin{cases} (\frac{x_m}{x})^\alpha, & \text{for } x \geq x_m \\ 1, & \text{for } x < x_m \end{cases} \tag{5.7}$$

We determined the GPD parameters for the Session Times $(x_{m,st}, \alpha_{st})$ and the Inter-Contact Times $(x_{m,ict}, \alpha_{ict})$ by the graphs given in the corresponding papers.

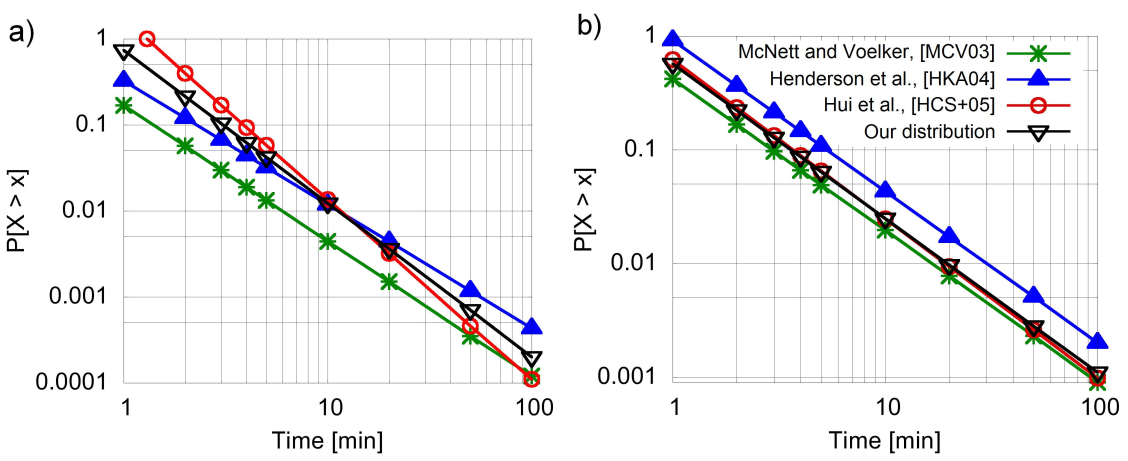


Figure 5.7.: Distribution of a) ST and b) ICT in different studies (in log-log scale)

Table 5.1 gives a summary of our investigations. It can be seen that there is no significant general difference between the GPD parameters in infrastructure-based and in Ad Hoc scenarios. Thus, we chose the average parameters given in the table for all of our evaluations.

Figures 5.7a and b show the ST (a) and ICT (b) distribution functions of three studies that focus on scenarios which are very similar to ours [MV03, HKA04, HCS$^+$05] and the one used in our evaluations. The power law nature of the distribution function becomes obvious as the graphs represent straight lines in the log-log scale figures. It can be seen the distribution we have chosen for our measurements is close to these studies, particularly to the one conducted by Hui et al. [HCS$^+$05] which was gained in a conference environment. This indicates that the simulated user mobility in the following evaluations is based on realistic distributions in typical Pervasive Computing environments.

5.7.3. Evaluation based on Constant Resource Availability

First of all, we are looking for reasonable values for the fixed cache size $|C|$. For this purpose, suitable overall cache sizes for the different environments where the resource conditions are supposed to do not change over time are investigated. Regarding the space consumption of the PACs, as discussed in Section 5.5 and illustrated in Figure 5.5, a green PAC for the larger application with $k_2 = 31$ components consumes around 100 kB, while a yellow PAC consumes around 18 kB. Accordingly, the PACs for the smaller application with $k_1 = 7$ components consume only 23 kB (green PAC) or 4 kB (yellow PAC). In the following, we determine the size $|C|$ which has to be reserved *per single application* with $k_2 = 31$ components.

The application user obviously wants to have the application configured as fast as possible, which is the case when the cache miss rate is close to zero and a pre-stored configuration is just loaded from disk. 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 Ad Hoc environments. Thus, a tradeoff between small cache

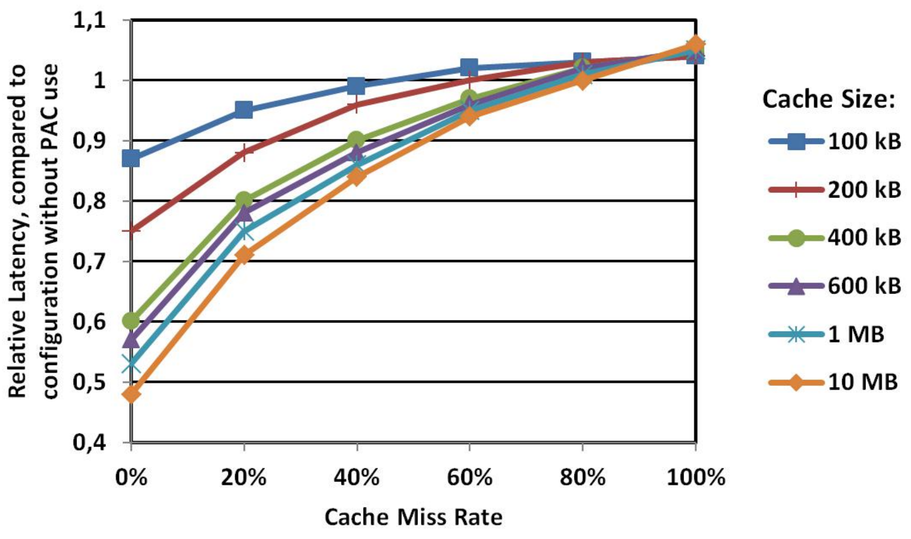


Figure 5.8.: Correlation between PAC Cache Miss Rate and configuration latency at different cache sizes between 100 kB and 10 MB

size at the one hand and low cache miss rate at the other hand is needed. Figure 5.8 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, when different cache sizes between 100 kB and 10 MB per application are used. A cache with 10 MB capacity can be seen as an unbounded cache, as all PACs that are possible for the investigated application can be stored simultaneously in that cache. For all cache sizes, the configuration latency is lowest when no cache misses appear at all, and rises monotonously with increasing cache miss rates. From the figure, it can also be seen that when the cache miss rate exceeds a specific value, then the latency with PAC usage becomes higher than the latency of the standard configuration, due to the contract matchings that have to be performed for the cached PACs. With a rising cache size, the expected latency at specific cache miss rates decreases monotonously, as the number of PACs that can be stored concurrently becomes higher and, thus, the percentage of the application that needs to be configured in the traditional way is reduced. When increasing the cache from 100 kB up to 400 kB, the latency significantly drops. From these results, it becomes obvious that cache sizes of 100 kB and 200 kB are too restrictive for a reasonable use of the PAC approach. The figure also shows that when increasing the cache size to values above 400 kB, the latency reduction is minor compared to the increased space overhead for the involved devices. For example, when comparing the results of 400 kB and 1 MB cache size measurements, the factor between the space overheads is 2.5, while the latency reduction at the same time is only around 4 %. From these results, we conclude that choosing 400 kB as value for the cache size $|C|$ per application with $k_2 = 31$ components is a reasonable choice, as the latency overhead compared to the best possible latency – when choosing a cache size of 10 MB – represents only 7 %. Thus, we rely on a 400 kB cache size limit in the following. With this cache size, the latency with PAC usage exceeds the latency of the standard configuration when the cache miss rate becomes higher than 71 %. In case of cache miss rates close to 1 (i.e., none of the application contracts are covered by a PAC), the PAC approach's configuration latency is around 5 % higher than standard configuration without PAC use.

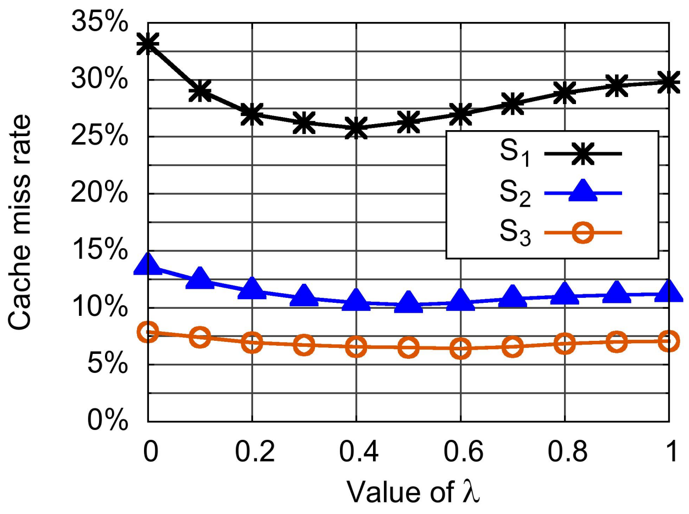


Figure 5.9.: Determination of optimal static λ values for S_1, S_2 and S_3

Next, we determine the optimal static values for λ for the LRFU strategy (cf. Section 5.4) in the different scenarios, i.e. the values where the cache miss rate becomes minimal. In these initial evaluations, we set $|C_{yellow}| = 0$. In Figure 5.9, 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, e.g., due to periodically repeating activities like working days. In this case, it needs to be considered *how often* PACs have been used before, leading to a higher utility only for these PACs. Thus, both the recency and the frequency of a PAC's availability need to be taken into account to maximize the usefulness of the PAC approach and minimize the cache miss rate. The optimal λ changes from 0.4 in the Ad Hoc scenario 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 the heterogeneous scenarios S_2 and S_3. Moreover, the general influence of λ to the variance of the resulting cache miss rate increases with the dynamics of the environment. 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.

Subsequently, we determine the optimal split factor f which determines the relative amount of $|C|$ that is spent for C_{yellow} and, thus, the partitioning of the cache into the areas for green and yellow PACs. Therefore, we perform several simulation runs with differing fractions of cache space for C_{yellow}. The results are shown in Figure 5.10. It can be seen that reserving an increasing portion of the cache size for

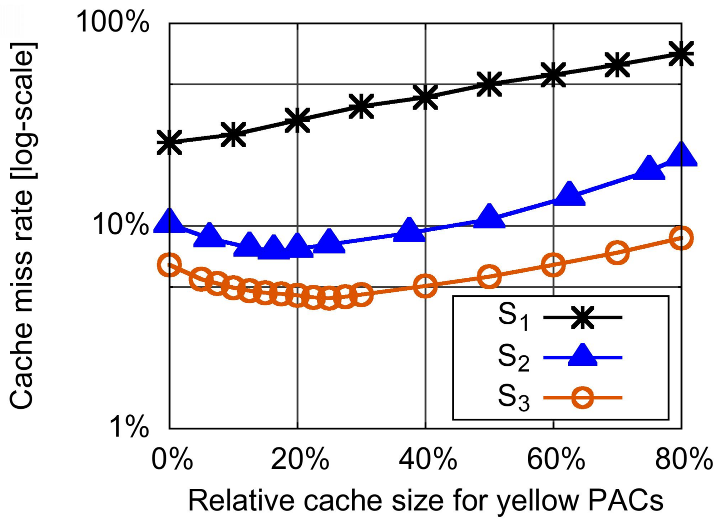


Figure 5.10.: Determination of optimal size for C_{yellow} when LRFU-0.5 and $|C| = 400\ kB$ is used

the yellow PACs monotonously increases the cache miss rate in the homogeneous scenario S_1. This is because S_1 represents a highly dynamic scenario, as it involves only mobile devices. Thus, the cache content changes very rapidly, which yields more frequent replacements in the cache. However, when $|C_{green}|$ is reduced in favor of $|C_{yellow}|$, the overall number of green PACs that can be stored and used in the configuration becomes the limiting factor in this highly dynamic scenario. In the heterogeneous environment (S_2 and S_3), the degree of dynamics is lower due to additonal infrastructure devices which are supposed to be continuously available. Thus, the cache is filled with some PACs whose components are resident only on infrastructure devices. These PACs durably reserve a specific fraction of C_{green} and form the basis for a large coverage of the application components. Moreover, the cache content changes much less between two subsequent configuration processes, and the overall cache size for C_{green} is not that crucial as in scenario S_1. So, shifting some space from C_{green} to C_{yellow}, particularly for PACs involving components on the mobile devices, leads to a lower cache miss rate, as the cache holds more information about yellow PACs, which may possibly change to green PACs after some time. In Figure 5.10, you can also see that the cache miss rate increases again in S_2 and S_3 when a specific fraction for C_{yellow} is exceeded, as the reduced size of C_{green} then starts to become the limiting factor. This optimal fraction for C_{yellow}'s space increases from 16.3 % in scenario S_2 to 25.1 % in scenario S_3, as the degree of dynamics in S_3 is lower because it features more resource-rich devices. Introducing yellow PACs in the scenarios S_2 and S_3 with these optimal fractions of the overall cache size reduces the cache miss rate from 10.2 % to 7.6 % (i.e., by more than a quarter) in S_2, and from 6.4 % to 4.4 % (i.e., by almost a third) in S_3.

In the following, we compare the LRFU strategy (with $\lambda = 0.5$ and optimal cache size fractions for C_{yellow}) with the standard replacement strategies First In First Out (FIFO), Remove Smallest First (RSF), and Remove Largest First (RLF), as described in Section 5.4, to show its outstanding performance.

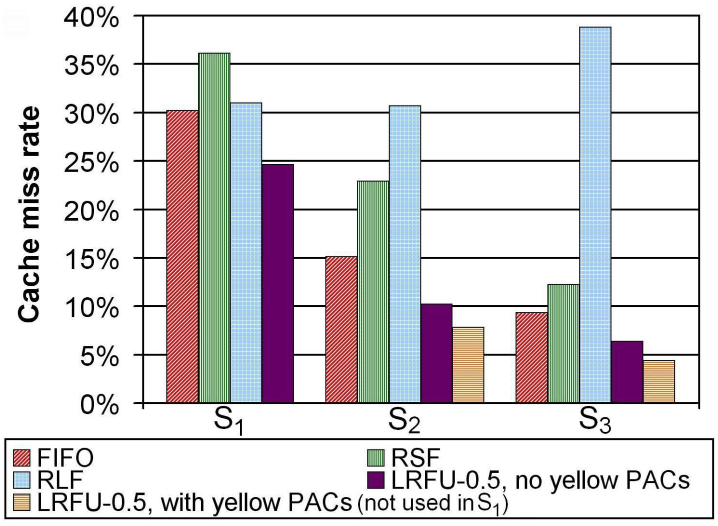


Figure 5.11.: Comparison of different cache replacement strategies

Figure 5.11 illustrates that even LRFU-0.5 without usage of C_{yellow} outperforms FIFO, RSF and RLF in all scenarios. FIFO performs better than RSF and RLF, as it keeps all cached PACs for a certain time in the cache, but worse than LRFU, as FIFO does not consider the usability of the involved components. RSF faces the problem of only storing large PACs, which are potentially not usable for a large amount of time, as they involve much more devices and components. RLF only keeps small PACs in the cache. Because of this, it usually does not cover the complete application and leaves some components uncovered. This problem becomes even more severe in the heterogeneous environments S_2 and S_3, as the application size increases, leading to very bad performance there. Moreover, introducing yellow PACs (with the optimal fractions determined in Figure 5.10) further drops the cache miss rate in the heterogeneous environments. Thus, using LRFU with $\lambda = 0.5$ and yellow PACs[2] leads to much less cache misses than standard strategies like FIFO.

Figure 5.12 shows the cache miss rate (depicted in z-axis) with variable overall cache sizes (x-axis) and split factors f (y-axis) in the three evaluated scenarios. It can be seen that the cache miss rate is comparatively high in all scenarios in case of low 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 resource-richer the environment gets, the smaller the achieved cache miss rates become. For example, if the cache miss rate should not exceed 10 %, then you can infer from Figures 5.12a to c that the previously determined static cache size of 400 kB is sufficient to stay below this limit: With $|C| = 400\ kB$, the cache miss rates are 9.2 % (S_1), 8.6 % (S_2), and 7.3 % (S_3). Regarding the choice of f, the cache miss rates become minimal if we choose values of 0 % (S_1), 16.3 % (S_2) and 25.1 % (S_3) for f. The corresponding data points X_1 $(400\ kB, 0\ \%, 9.2\ \%)$, X_2 $(400\ kB, 16.3\ \%, 8.6\ \%)$ and X_3 $(400\ kB, 25.1\ \%, 7.3\ \%)$ are drawn in Figures 5.12a to c.

[2] As determined before, yellow PACs should only be used in heterogeneous environments

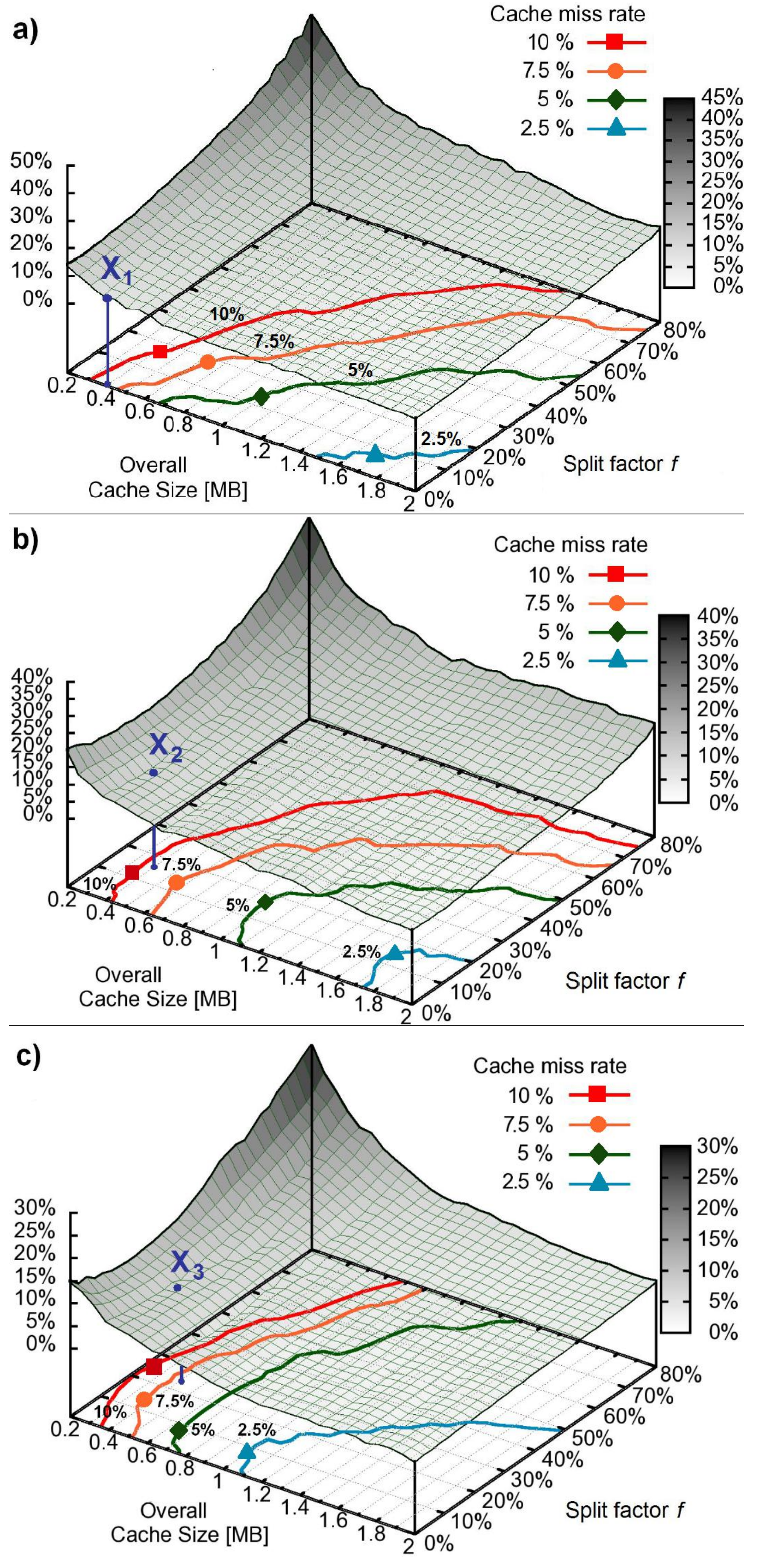


Figure 5.12.: Distribution of Cache Miss Rate in a) S_1, b) S_2, c) S_3

5.7.4. Evaluation based on Dynamically Changing Resource Availability

Since static parameters obviously perform suboptimal in dynamic environments, we now focus on gaining adaptive parameters for f and λ which automatically adjust themselves to changing resource conditions. For this purpose, we simulate the user mobility based on a Pareto distribution with the average parameters gained from Table 5.1. Regarding $|C|$, we decided to retain the static value of 400 kB, since this value provided good performance in all scenarios and an adaptive $|C|$ would require to allocate and de-allocate memory every time $|C|$ is changed, yielding significant overhead especially on the weak mobile devices. From our experiences with the developed configuration approaches [SHR10], we found that the ratio s of powerful devices (i.e., $s = \frac{\#\ strong\ devices}{\#\ all\ devices}$) is the most important factor for the choice of a specific configuration scheme in an environment. Thus, we determine the adaptive parameters solely dependent on this ratio s.

At first, we focus on determining an adaptive split factor f. Therefore, we evaluate the arising cache miss rates with various values of s (from 0 % to 40 %) and determine the optimal f value for each scenario. To find if there are interdependencies between λ and f, we perform these measurements with three different values of λ: 0.25, 0.5, and 0.75. The results of these measurements are shown in Figure 5.13. 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 only little interdependency on each other. 2.) The optimal f values (i.e., the values where the cache miss rate becomes minimal) rise with increasing values of s. This is because an increasing number of powerful devices leads to a lower degree of dynamic, yielding less frequent replacements in the cache. Thus, the cache size $|C_{green}|$ for the green PACs is not that crucial as in higher dynamic scenarios. So, shifting some space from C_{green} to C_{yellow} leads to a lower cache miss rate, as the cache holds more information about yellow PACs, which may possibly become green PACs after some time. Figure 5.13 also shows that a polynomial approximation by the Least Mean Squares Method (LMS) method is very accurate, so we rely on this approximation function in the following.

Figure 5.14 shows the optimal adaptive λ values (i.e., the values where the cache miss rate becomes minimal) which we gained by evaluating the same scenarios with the determined LMS-approximated adaptive f values and differing λ values between 0.0 and 1.0. The optimal adaptive λ becomes slightly larger with a rising number of strong devices, since the degree of dynamic 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 around 0.6 if we have many strong devices in the environment ($s \approx 40$ %).

Finally, we perform evaluation runs over a time span of 3600 s where we periodically configure an application with a period time of 50 s, i.e., we perform 72 configurations in total. The availability of the devices is dynamically changing, according to the GPD discussed in Section 5.7.2, and the fitting configuration scheme ac-

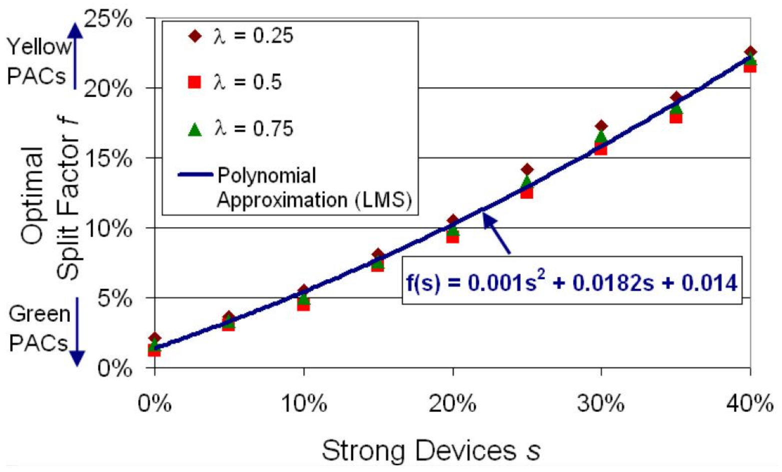


Figure 5.13.: Determination of optimal adaptive split factor $f(s)$

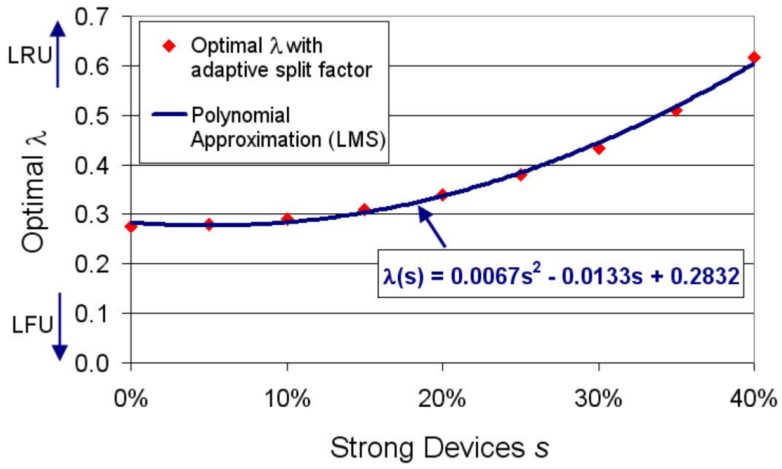


Figure 5.14.: Determination of optimal adaptive $\lambda(s)$

cording to the selection strategy presented in Section 4.3.6 is chosen. We compare configuration without PAC use to PAC configuration with $|C| = 400\ kB$ and static and adaptive cache parameters (f, λ), and to the optimal case (PAC use with unbounded cache, i.e., every used PAC is cached forever and does not need to be replaced). Figure 5.15a shows the latency overhead compared to the optimum with different values for s. PAC configuration with adaptive parameters is very close to the optimum with only around 9 % overhead, particularly if many powerful devices are available. PAC use with static parameters leads to an increased overhead of around 31 % compared to the optimum, since the parameters do not adapt to the dynamic scenarios. Configuration without PACs performs worst with an average overhead of 66 % and can only compete with PAC configuration which uses static parameters in configuration runs where no powerful devices are available, but shows very poor performance with an increasing number of powerful devices. Figure 5.15b shows the overall average latencies. This figure illustrates that PAC configuration with adaptive parameters is only around 150 ms slower than the optimal case of PAC use with unbounded cache, despite the 400 kB cache size restriction. PAC

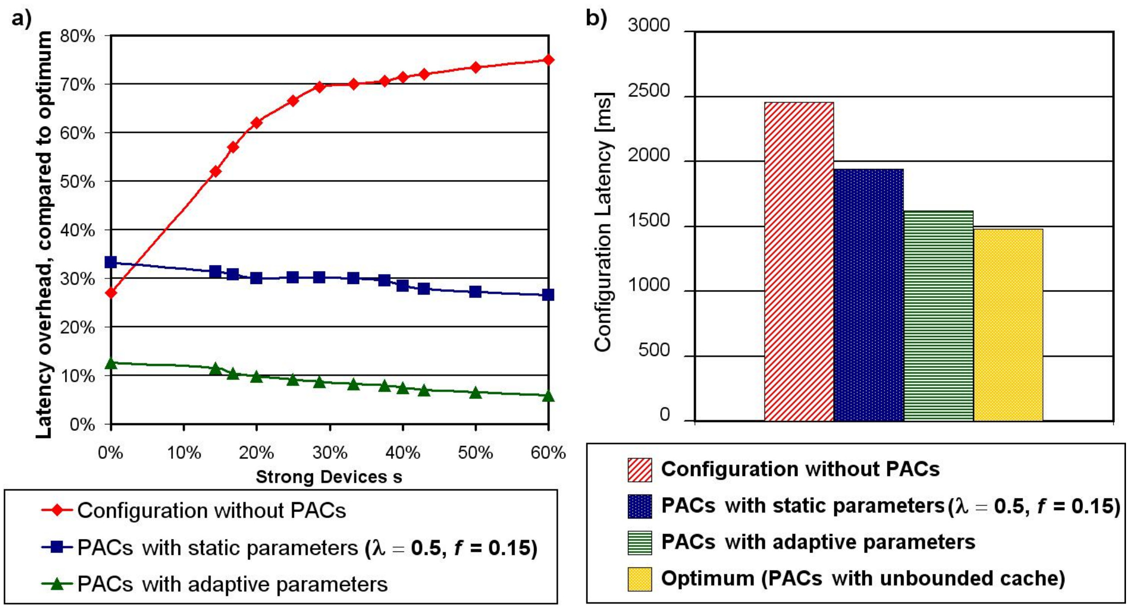


Figure 5.15.: a) Configuration Latencies with static and adaptive cache parameters, depending on s, b) Comparison of average configuration latencies

configuration with static parameters is around 450 ms worse than the optimum, 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.

Thus, in scenarios where fast configuration processes are vital, the use of PACs with adaptive parameters is strongly recommended. If timing constraints are lowered for specific scenarios, PAC use with static parameters or even no use of PACs may be the fitting choice.

5.8. Summary and Discussion

In this chapter, we have introduced the concept of integrating the results from previous configurations into future configuration processes. This is achieved by caching parts of a valid configuration for their future re-use. We call a set of components which have been used within a previous application configuration in combination and form a subset of the application tree a *Partial Application Configuration (PAC)*.

We have addressed several issues that have to be regarded while integrating partial configurations into application compositions. In order to enable the storage of PACs, a cache has to be provided. This cache needs to be of limited size to obtain applicability of the approach in realistic pervasive scenarios. The determination of a suitable cache size which is neither too small, nor too large has shown to be a main issue for the performance and the applicability of the PAC approach.

Another important issue that arised was how to proceed when the cache space is exceeded. Therefore, we have suggested to use a replacement strategy which introduces a utility value for each cached PAC that is based on the recency and frequency of the PAC's availability. The PAC utility values are updated whenever a configuration process is taken. When the cache space is exceeded, only the PACs with the lowest utility values are replaced. Moreover, to increase the efficiency of the cache space usage, we have decided to distinguish between two different types of PACs in the cache: While *green PACs* represent the PACs with currently highest utility value and are stored in a complete format which can simply be integrated into configuration assemblies, *yellow PACs* represent partial configurations with lower utility values that store the information about the PAC components in a much more compact format. Because of this, the number of storable PACs is significantly increased. However, yellow PACs cannot be directly integrated into configuration assemblies. But as their utility values are recorded, they may become green PACs in the future when their utility value increases. In this case, the complete format of this PAC is automatically created, which is possible as the yellow PACs contain all required information. Then, the formerly green PAC with lowest utility value is replaced. The provision of a reasonable number of green PACs in combination with additional yellow PACs guarantees to always have a large set of highly usable PACs in the cache, yielding low cache miss rates, as our evaluations have shown.

To integrate PACs into configuration processes, the configuration approaches presented in Chapter 4 needed to be extended so that they first check if a usable PAC exists that fulfills the dependencies given by an application contract. In this case, the PAC is simply loaded from cache, and no calculations need to be taken for the respective application parts. Otherwise, the contract dependency needs to be resolved in the standard way using contract matching. The corresponding extensions of the configuration assemblers are of minor nature, so the simple integration of the PAC concept into additional assemblers is ensured.

In our evaluations, we have determined suitable parameters of the PAC approach for the overall cache size, the distribution of the cache into green and yellow PACs, and the influence of the recency and frequency for the replacement strategy. In these measurements, we have shown that a replacement strategy which equally weighs the recency and frequency of the PAC availability provides the lowest average cache miss rates. This strategy also yields significantly lower cache miss rates than standard replacement strategies like FIFO or pure LRU or LFU. Following, we have determined an overall cache size of 400 kB as a good tradeoff that provides a low cache miss rate and, hence, a low configuration latency, while at the same time, keeps the disk space requirements on the involved devices low. The fraction f of cache space which should be reserved for yellow PACs has shown to monotonously rise with an increasing fraction of resource-rich devices. After having determined static values for the PAC parameters (overall cache size, fraction of cache for yellow PACs, recency/frequency parameter of the replacement strategy), we focused on providing dynamic parameters which automatically adapt themselves to changing environments. Therefore, we have presented approximation functions that solely depend on the relative amount of resource-rich devices as input parameter, since this parameter has shown to have the highest influence on the performance of the PAC approach in our measurements. With dynamic parameters, the configuration latencies could further be reduced compared to the use of static parameters.

In summary, we have presented a solution for the effective caching and use of PACs that are highly valuable for future configuration. Through this, the number of components that actually need to be configured is reduced, leading to configuration processes which are around 35 % faster on average than configuration without PAC use. This leads to less user distraction and, in consequence, to more seamless configurations and adaptations. So far, none of the related projects features a similar mechanism. Thus, the use of PACs represents a completely novel approach in the research area of automated composition of distributed applications. Correspondingly, the PAC concept represents a large step towards realizing the vision of Pervasive Computing in configuring adaptive distributed applications.

6 Prototype

6.1. System Architecture

To realize the concepts and mechanisms developed in this work and discussed in the previous Chapters 4 and 5, we extended the systems BASE [BSGR03] and PCOM [BHSR04], which represent communication and component middleware systems for Pervasive Computing developed at the University of Stuttgart. The extended system architecture with the newly integrated elements (highlighted by dark black boxes) is shown in Figure 6.1.

The foundation of the BASE micro-broker is the device capability layer, i.e., the device *platform*. On top of this platform, the *plug-in manager* is resident, enabling flexible and extensible automatic configuration of the supported protocols and services. The BASE *Invocation Broker* delegates method calls to the corresponding services on the mobile devices. Registries maintain the currently available *devices* and *services*. While our new *Event Service* realizes the automatic update of mapped devices' resource conditions at the cluster heads, the *Mobile Code Service* is responsible for the actual transmission of the resource information classes. These two new components of BASE are described in more details in Section 6.2.

The component system PCOM provides a runtime environment for the design of distributed applications with automated runtime configuration and adaptation. Within PCOM, devices are represented by *containers*. Besides the standard containers, *Virtual Containers* are additionally provided and allow the remote configuration of application components. The access and retrieval of remote devices' configuration information happens via mobile code, using specific *Mobile Code Accessors*.

So-called *assemblers* with different degrees of decentralization represent the configuration algorithms. While some of the assemblers are complete algorithms, others represent greedy heuristics. Handte et al. [HBR05] have developed a greedy distributed and a greedy centralized assembler, as well as a complete distributed assembler based on Asynchronous Backtracking. We provide a new centralized Mobile Code Assembler based on our Direct Backtracking algorithm for weakly hetero-

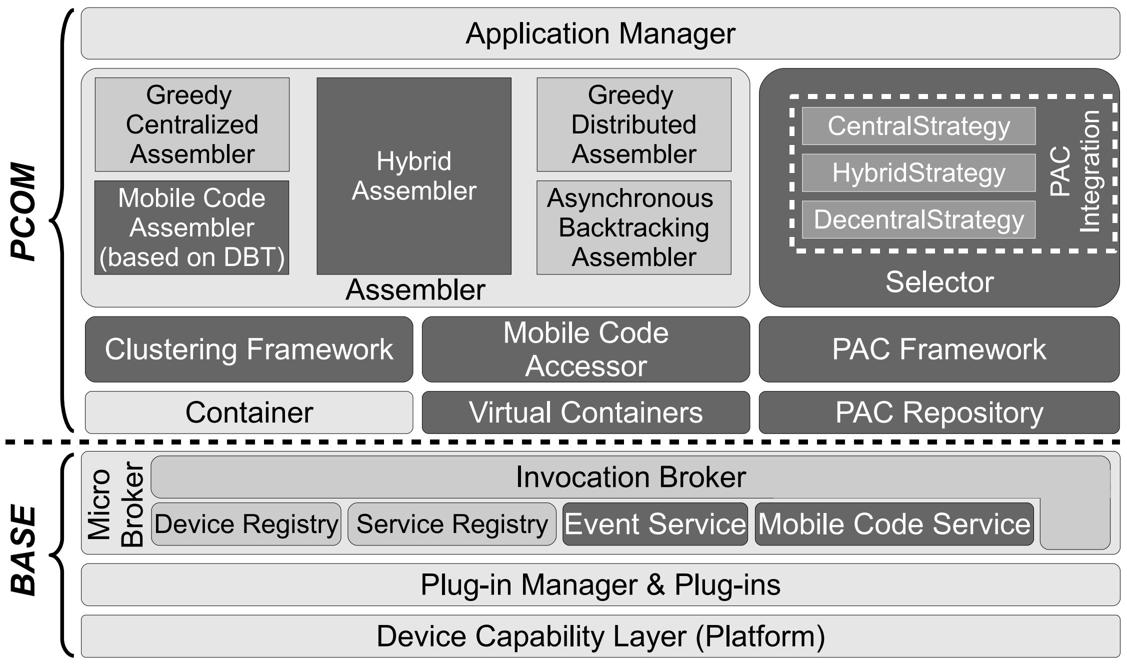


Figure 6.1.: Extended System Architecture of BASE and PCOM: New system elements are represented by the dark boxes

geneous environments, and a hybrid assembler tailored for strongly heterogeneous environments.

On top of the assemblers, the *application manager* performs the lifecycle management of applications and allows to start, stop, and configure the distributed applications with the aid of the containers and assemblers. The actual selection of a specific assembler in a distinct environment is regulated by *selectors*, which can additionally decide whether Partial Application Configurations are used or not. In case of PAC use, the *PAC Framework* automatically controls the access and update of the PACs that are cached in the *PAC Repository*.

Thus, the BASE extensions are of rather minor nature, while PCOM's architectural changes are wide-ranging to enable efficient heterogeneity support. In the following, we will discuss the main components of BASE (Section 6.2) and PCOM (Section 6.3) in more details.

6.2. BASE Communication Middleware

In this section, we first briefly present the basic functionality of the communication middleware BASE, as presented by Becker et al. [BSGR03]. Then, we discuss the extensions we made to BASE to enable the automated and proactive loading of remote devices' resource information and to create the Virtual Containers (cf. Section 4.3.5).

6.2.1. Basic Functionality

As middleware system which handles all aspects of communication, we rely on the *BASE Microbroker* [BSGR03] to support automatic configuration and adaptation of communication protocols at runtime. This enables a more stable and flexible communication platform. BASE provides distribution-independent access to the offered services and decouples the application from the underlying communication protocols.

The main component of BASE is the *Invocation Broker* which delegates method calls to the corresponding services on the mobile devices. Furthermore, BASE manages the devices in range through the *Device Registry* and the services available on a device through the *Service Registry*. A *Lease* mechanism – relying on periodic heartbeat message exchange – guarantees the up-to-dateness of the registries. The automatic configuration of the supported protocols is possible by the *Plugin Manager*, as the protocols are outsourced in plugins which are loaded and configured at runtime. The plugin concept of BASE allows the microbroker to support a wide range of end-user devices that reaches from simple microcontrollers and mobile phones up to full servers. The only requirement is the presence of a JVM that supports the Connected Limited Device Configuration Profile (CLDC) [Sun03] profile.

6.2.2. Extensions

The proactive loading of resource information from cluster members to their cluster head requires the communication middleware to provide two additional services: Firstly, a mechanism is needed to automatically transfer configuration classes between different devices. And secondly, as the cluster heads always needs to have up-to-date information of their cluster members' resource conditions, a mechanism to notify devices about resource changes at specific remote devices is required. As one can see in Figure 6.1, the *Event Service* and the *Mobile Code Service* were realized to provide these two functionalities.

Event Service

The basic idea of the Virtual Container concept presented in Section 4.3.5 is the creation of one artificial container at a strong device per remote device that is mapped to this strong device. This Virtual Container emulates the remote container. As availability of resources on remote containers may change over time, the Virtual Containers have to be updated. This happens via the new **Event Service** which implements a distributed event service that supports undoable events and is seamlessly integrated into the BASE event framework. It provides the `IListener` interface and is therefore pluggable to nearly all components. In detail, this service works as follows: Whenever a cluster member joins a cluster head, this cluster head registers at the cluster member for changes in its resource information. In case of changed resource conditions, e.g., due to additional applications which allocate or deallocate resources, the cluster member fires a *Resource Change Event* to notify its cluster head (as its listener) about the updated resource conditions. The listener then updates the resources in its Virtual Container correspondingly. Obviously, when a cluster head notices that one of its cluster members leaves the cluster since it does not receive any heartbeat messages for a specific period of time, it deregisters from this cluster member, as the respective resource changes are no longer relevant for the cluster head's configuration.

Mobile Code Service

In order to actually update the state of a remote container, the configuration logic has to be transfered to the current cluster head. This is possible using the new **Mobile Code Service** of BASE, which allows the loading of class files from a remote system. The Mobile Code Service provides standard proxy and skeleton implementations for remote communications and is capable of transfering general classes, which also enables the distribution of specific classes such as BASE plug-ins, for instance.

There exist multiple ways to obtain remote code. Carzaniga et al. [CPV97] present three paradigms besides the classical client-server model where code is only executed on a server: Remote Evaluation, Code On Demand, and Mobile Agents. In our case, Code On Demand is the preferrable choice, as it represents the scenario that a process A on device D_A possesses the required resources for configuration and loads the executable code of process B on another device D_B to execute it locally. Furthermore, this code may effectively be constrained to solely executing uncritical actions if Code On Demand is used, avoiding the necessity of using security mechanisms like digital signatures and, thus, the management of certificates.

6.3. PCOM Component System

While BASE provides basic communication features that are required by distributed applications, the actual configuration of the applications as well as the management of the devices and their resources requires an independent system which builds upon BASE's functionality. We use the component system PCOM for this purpose. Here, we first discuss the initial functionality of PCOM at the beginning of the works for this thesis, as presented by Becker et al. [BHSR04] and Handte et al. [HBR05, HHSB07, HHS$^+$07]. Then, we present our extensions to PCOM which enable the use of the concepts we introduced in the Chapters 4 and 5.

6.3.1. Basic Functionality

Originally, PCOM was designed for the use in Mobile Ad Hoc Networks without additional infrastructure. Thus, PCOM is a system that does not rely on any central instance. This has serious impacts on PCOM's inital system architecture.

PCOM provides a runtime environment for the *components* with contractually specified interfaces for the design of distributed applications that are automatically configured and adapted during runtime without user interaction. Cooperative user behavior is assumed, i.e., all users are trustworthy. A demonstration of PCOM was shown at the IEEE PerCom 2006 conference [HUB+06].

Container

The components which may be used by applications are executed within a PCOM *Container* that provides a specified interface to the PCOM middleware and basic services for the components. A component features *factories* which are representatives for locally installed components and support a negotiation phase that recursively determines the non-functional parameters of a component without starting it. This enables a resource-conserving configuration of the components. In order to support contracts (i.e., resource dependencies between components), PCOM provides *allocators* that encapsulate the access to exclusive resources (e.g., a single display) and, thus, allow transparent access to these resources. *Allocators* are used both during configuration and execution process of an application. Such as factories, they have to be registered on the corresponding container which then can determine a valid configuration for the distributed application, and execute the application.

Assemblers

The automatic configuration of components is performed by so-called *assemblers* which enable access to components prior to their instantiation and, thus, decouple the configuration processes from the lifecycle management of the components. At the time this thesis started, PCOM supplied a complete, distributed assembler based on the *Asynchronous Backtracking (ABT)* algorithm by Yokoo et al. [YDIK98], and a distributed (*Greedy Distributed Assembler (GDA)*) as well as centralized greedy heuristic (*Generalized Clustering Algorithm (GCA)*) [HBR05]. These assemblers are optimized for the use on resource-poor mobile devices.

Application Manager

Lifecycle management of applications is performed by the *Application Manager* which was introduced in 2007 in the system [HHSB07]. The application manager allows to start, stop, and configure the distributed application with the aid of the containers and the assemblers. With the help of the application manager, a user can also request that a certain anchor should be started or a running anchor should be stopped.

6.3.2. Extensions

Clustering Framework

The Clustering Framework is a main component of the new PCOM system architecture. Figure 6.2 shows the UML class diagram of the clustering framework. The

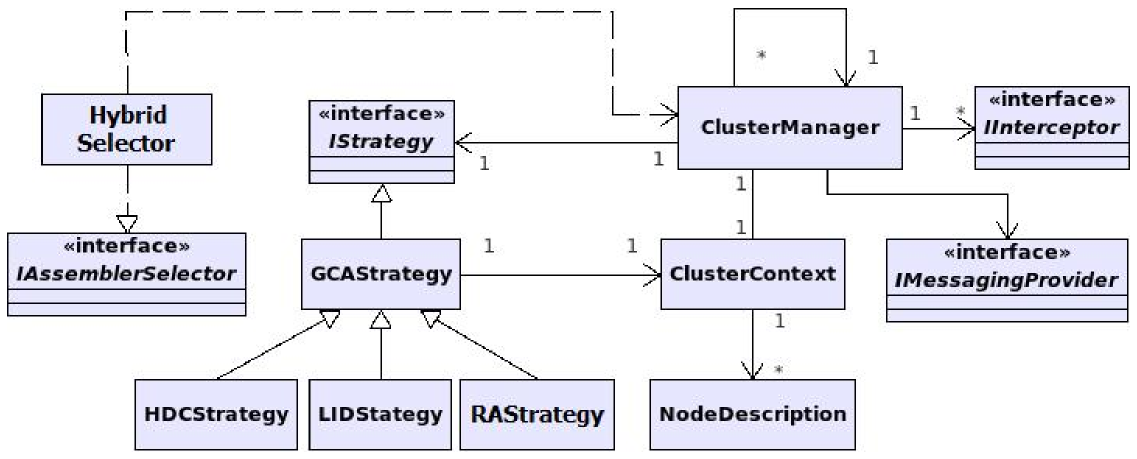


Figure 6.2.: Implementation of Clustering Framework

central component of this framework is a service within the *Cluster Manager* which provides methods for requesting the current cluster head and the cluster members. It delegates occurring events (e.g., neighborhood changes) to the corresponding clustering strategies, which have to implement the interface *IStrategy*. Regarding the clustering strategies, we have implemented the completely distributed version of the Generalized Clustering Algorithm of Basagni (GCA, [BCFJ97]) as basic clustering strategy. In this algorithm, adjacent nodes initially exchange their node weights. To assign the clustering weights to the present nodes, we used the well-known Strategy design pattern [HW03] and implemented standard strategies like *Highest Degree Clustering* (HDCStrategy, [GTCT95]) or *Lowest ID Clustering* (LIDStrategy, [LG06]), and the developed Resource-Aware strategy (RAStrategy) that is based on the introduced benchmarking process and determines a threshold to distinguish between cluster heads and cluster members, as discussed in Section 4.3.3. In each of these schemes, the nodes with highest weight become cluster heads. Obviously, the implementation of additional clustering strategies is easily possible.

Via the `IStrategy` interface, the clustering strategies get access on the *Cluster Manager* that provides services to send messages to neighbored nodes (via the interface `IMessagingProvider`), access the current state of the cluster (via the interface `IInterceptor`) or the selection of a specific assembler (via the interface `IAssemblerSelector`). As the cluster strategies may widely differ in their cluster formation and cluster maintenance phases, the *Cluster Context* class provides information about the internal cluster state and the current cluster nodes, whose properties are stored within the *Node Description* class.

Mobile Code Framework

In order to allow the loading of remote classes via Mobile Code, a framework to access the required classes was realized. The main part of this *Mobile Code Accessor Framework* are various so-called *Accessors* that allow different access methods.

Those parts of the original *Allocators* and *Factories* which are relevant for configuration processes were outsourced into new serializable configuration classes. This enables their transmission to the cluster head which then can locally calculate a valid configuration. As resources typically are limited, changes in the state of remote resources have to be transmitted to the cluster head. This happens by the previously described *BASE Event Service* that represents a distributed event listener. Every configurator has to report changes in its state by events. To enable access to context objects on remote containers for the cluster head, we introduced the *Virtual Container (VC)* concept described in Section 4.3.5, where every configuration class is administrated within its own VC. The actual configuration is performed on the cluster heads by accessing their own container and the established Virtual Containers.

Following, we focus on the realized access methods. Besides the already existing **Remote Accessor** which simply delegates requests by remote invocation to other containers and does not rely on Mobile Code, three new *accessors* were developed and represent different strategies for obtaining the required configuration logic:

- **Eager Mobile Code Accessor:** This accessor loads configuration classes from cluster members in advance, as soon as they are in communication range. Thus, it implements the pre-configuration process presented in Section 4.1.3. In case a cluster head changes, the old head unloads all classes to deallocate memory, as they are no longer needed. This yields the advantage that the system performs proactive class loading to reduce the latency in a possibly following application configuration. In highly dynamic scenarios or if the cluster head frequently changes, the Eager Mobile Code Accessor permanently transfers remote configuration classes, leading to increased communication overhead. This drawback can be reduced by choosing a suitable cluster strategy or the use of the *Handover Mobile Code Accessor* (see below).

- **Lazy Mobile Code Accessor:** Using a lazy strategy, the required classes are not loaded until the time they are really needed. This complements with eager class loading and implicates that the cluster head's resources are conserved. Furthermore, the traffic load of the network is reduced. A disadvantage of this method is the increased configuration latency for the user, as the class loading happens during the user interaction period.

- **Handover Mobile Code Accessor:** Frequent cluster head changes can significantly burden the network in case of eager class loading. To reduce this network load, we have implemented an additional *Handover Mobile Code Accessor* which performs eager class loading. In case the cluster head changes, this accessor immediately transmits the previous cluster head's state to the new cluster head. Consequently, the new cluster head does not need to inquire the configuration classes from the involved devices. This helps to decrease the configuration latency, as our evaluations have shown.

The Mobile Code framework and the implemented access strategies are depicted in Figure 6.3.

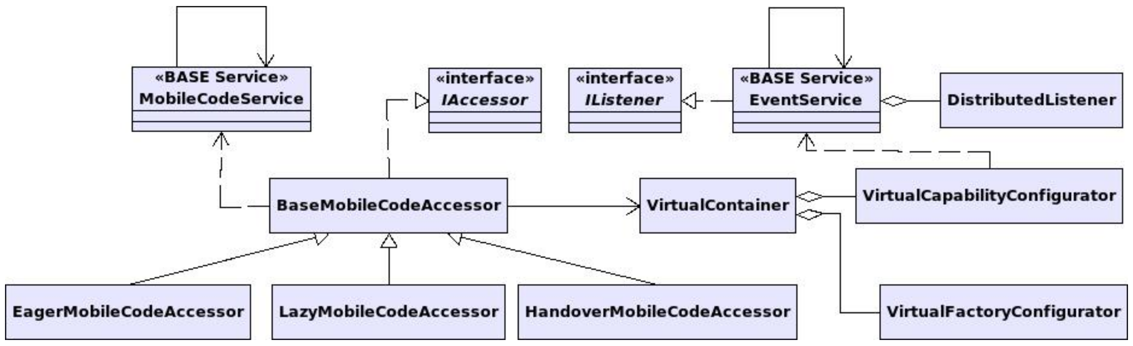


Figure 6.3.: Implementation of Mobile Code Framework

Mobile Code Assembler

The *Mobile Code Assembler* (MCA) represents a part of the Virtual Container concept discussed in Section 4.3.5. It is an assembler that uses mobile code to get the needed configuration logic. A component is configured if it finds all required resources or additional components. The participating configuration classes contain a `configure()` method that is recursively called for every element of the tree and, thus, follow the depth-first search approach of Direct Backtracking (cf. Section 4.2.1). Each class of the application tree configures itself and then calls the `configure()` method of its child components. Finally, the obtained configuration is serialized into a particular object which then can be transmitted to other assemblers. A demonstration of the PCOM system which relies on centralized MCA configuration on a laptop and involved smart phones [Sch09] was shown at the review meeting of the European Union-funded project ALLOW which deals with adaptation issues in flow-based Pervasive Computing environments [HRKD08].

Hybrid Assembler

The hybrid assembler is the central component of the hybrid application configuration that is described in Section 4.4. The hybrid approach's architecture is shown in Figure 6.4. This assembler is based on the Mobile Code Assembler as described above, since it uses the same mechanisms for the configuration: the active devices create Virtual Containers for every mapped passive device which store the current resource information of the remote devices. The access of the hybrid assembler on the containers is abstracted via the accessors described in the previous paragraph. The hybrid assembler is only running on the resource-rich devices and equally distributes the load among the configuration devices by relying on the resource-aware clustering strategy introduced in Sections 4.4.1 and 4.4.2, thus combining the VC concept with the ability of a parallel and distributed configuration process.

To clarify the hybrid approach's proceeding, we will describe the interaction of the distributed components within a hybrid configuration process now. Let us suppose that the user starts a distributed application on his or her mobile device which is represented in Figure 6.4 as `Container_2`. First, the application manager on this device calls the selector which decides to use the hybrid assembler, as several resource-rich

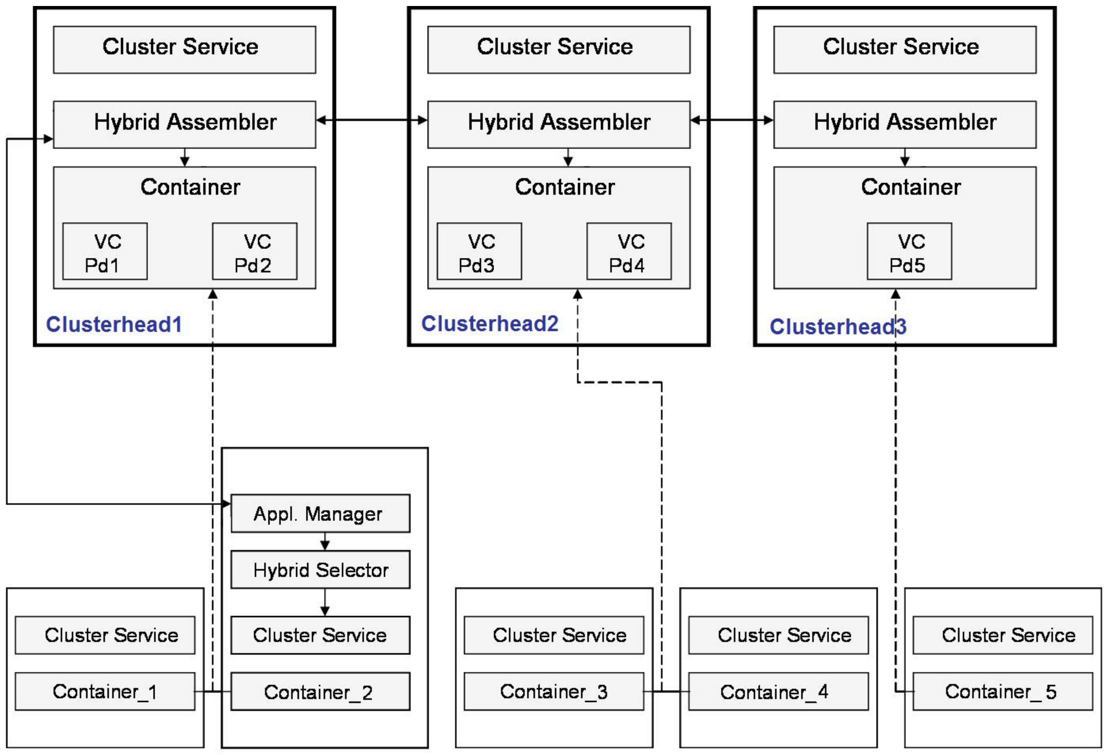


Figure 6.4.: Implementation of Hybrid Assembler

devices are available – and start the application configuration on cluster head 1, to which the user's device is mapped. Within the configuration process, the hybrid assemblers on the cluster heads access the local container of the respective resource-rich device as well as the Virtual Containers of the mapped devices. To maximize the configuration efficiency, the Virtual Containers are only accessed if the local container could not resolve the resource requirements of a dependency. If both the local and the Virtual Containers cannot resolve a resource dependency, the hybrid assembler requests the other cluster heads to resolve this resource conflict and waits for the configuration results from the other devices. If the configuration process was successful, the hybrid assembler supplies the complete assembly (which maintains the references to the determined resources) to the device where the application was started and, after the component bindings have been established, executes the application.

PAC Framework

The *PAC Framework* contains the logic to create and update the cached set of PACs. It contains the PAC replacement strategy presented in Section 5.5 to update the utilities of PACs and replace specific PACs if the cache space is exceeded. Furthermore, the PAC Framework enables the automatic distribution of stored PACs to newly arriving devices. If a cluster head realizes a new cluster member has joined its cluster, the *PAC Framework* transmits the cached PAC set as a serialized object to this device.

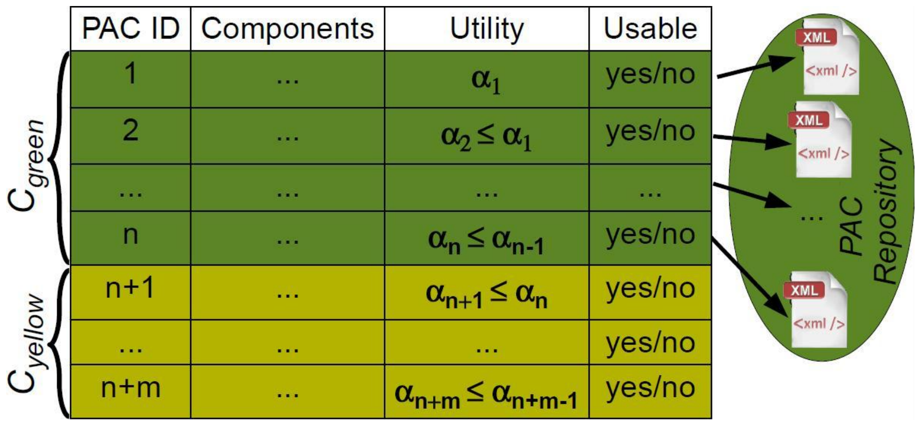
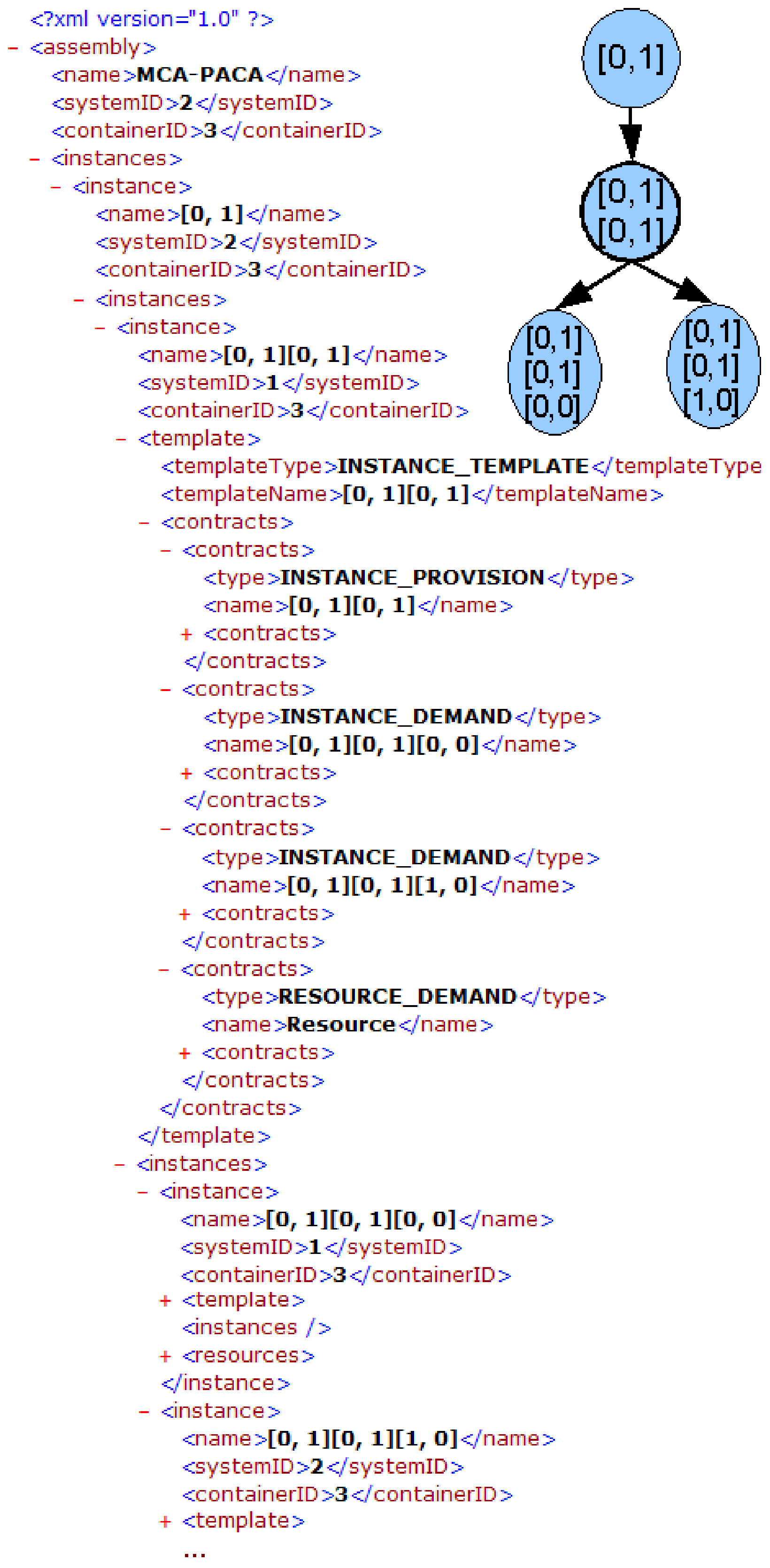


Figure 6.5.: Cache structure with Lookup Table where green and yellow PACs are stored, and the PAC Repository holding the corresponding XML-based assemblies of the cached PACs

In order to enable the use of PACs, the PAC Framework creates, accesses and updates the *PAC Lookup Table* including entries for the green and yellow PACs, and the *PAC Repository* which represents the storage location for the XML representations of the green PACs within the cache. The layout of the Lookup Table and the additional PAC Repository are shown in Figure 6.5.

The Lookup Table is arranged as a heap, with complexity of O(log$|C|$) for update operations. The elements within C are re-sorted after each configuration process using heap sort (with worst case complexity O($|C| \cdot log|C|$) for sorting the complete heap). By using the utility function $f(x)$ introduced in Section 5.4, the utility value of PAC p decreases monotonically between two consecutive references to p. This induces that two PACs which are not referenced do not change the order defined by their utility values. Therefore, one only needs to compare the utility of a yellow PAC p_y to the utility of the least-green PAC p_{lg} when p_y is referenced, but not on each reference to other PACs between two consecutive references to p_y. This ensures an efficient implementation of the replacements.

As mentioned before, a Partial Application Configuration is represented by an XML file in the PAC Repository, including information about the involved components and the devices which host these components, as well as the interdependencies between the components (i.e., the service *provisions* to parent components and the service *demands* to child components), which are relevant to establish the component bindings for the application execution. Figure 6.6 shows an excerpt of an exemplary green PAC in complete XML format, which covers information about the component itself (CoID [0,1][0,1]), its parent component (CoID [0,1]), and its child components (CoIDs [0,1][0,1][0,0] and [0,1][0,1][1,0]). The XML file lists the `INSTANCE_PROVISION` of the component to its parent component, as well as the `INSTANCE_DEMAND`s which represent the component instances that provide the implementation of the component-specific functionality. A component instance may additionally require an arbitrary, but fixed number of resources, which is encoded by a `RESOURCE_DEMAND` within the XML file.



```
  <?xml version="1.0" ?>
- <assembly>
    <name>MCA-PACA</name>
    <systemID>2</systemID>
    <containerID>3</containerID>
  - <instances>
    - <instance>
        <name>[0, 1]</name>
        <systemID>2</systemID>
        <containerID>3</containerID>
      - <instances>
        - <instance>
            <name>[0, 1][0, 1]</name>
            <systemID>1</systemID>
            <containerID>3</containerID>
          - <template>
              <templateType>INSTANCE_TEMPLATE</templateType
              <templateName>[0, 1][0, 1]</templateName>
            - <contracts>
              - <contracts>
                  <type>INSTANCE_PROVISION</type>
                  <name>[0, 1][0, 1]</name>
                + <contracts>
                </contracts>
              - <contracts>
                  <type>INSTANCE_DEMAND</type>
                  <name>[0, 1][0, 1][0, 0]</name>
                + <contracts>
                </contracts>
              - <contracts>
                  <type>INSTANCE_DEMAND</type>
                  <name>[0, 1][0, 1][1, 0]</name>
                + <contracts>
                </contracts>
              - <contracts>
                  <type>RESOURCE_DEMAND</type>
                  <name>Resource</name>
                + <contracts>
                </contracts>
              </contracts>
            </template>
          - <instances>
            - <instance>
                <name>[0, 1][0, 1][0, 0]</name>
                <systemID>1</systemID>
                <containerID>3</containerID>
              + <template>
                <instances />
              + <resources>
              </instance>
            - <instance>
                <name>[0, 1][0, 1][1, 0]</name>
                <systemID>2</systemID>
                <containerID>3</containerID>
              + <template>
                ...
```

Figure 6.6.: Excerpt from XML representation for exemplary PAC component with CoID [0,1][0,1]

6.4. PCOM Simulator

With the extensions provided by this work, the PCOM component system represents a prototype that enables the automatic configuration of distributed applications in heterogeneous environments via different configuration algorithms. This prototype is well suited for evaluating small or mid-size applications in rather static scenarios. However, for the evaluation of large-scale applications and scenarios with dynamically changing device availabilites, we rather rely on a simulator which provides support for faster and much more flexible evaluations than with the prototype. Here, we briefly present the basic functionality of this simulator, and the extensions that were necessary to perform the mentioned measurements.

6.4.1. Basic Functionality

To simplify the comparative evaluation of centralized configuration algorithms in a larger scale, we rely on an event-discrete simulator of the PCOM component system which was initially presented by Handte et al. [HBR05]. The PCOM simulator abstracts from the underlying computer hardware and networking technology. It is event-discrete in a way that it delivers and processes all messages generated in one discrete time step within the next step. To do so, the simulator relies on a module which provides a light-weight implementation of the fundamental concepts of the component system, such as containers, components, or resources. Moreover, the simulator provides a scenario generator for the creation of different applications and scenarios. This enables a simple and quick abstract evaluation of different configuration algorithms in various environments, but still relies on the systems BASE and PCOM to guarantee realistic evaluation results. Further details about this simulator as well as its system architecture are given by Handte [Han09].

6.4.2. Extensions

We rely on the core functionality provided by the initial version of the simulator. To compare our newly developed Direct Backtracking algorithm [SHR08a] to its closest relative, the Synchronous Backtracking algorithm [YDIK98], we implemented both algorithms as additional assemblers to the simulator.

Moreover, for the evaluation of the PAC concept in dynamic environments, we extended the simulator by the user mobility model which we have discussed in Section 5.7.2 to simulate dynamic changes in the availability of mobile and stationary devices, and we implemented several standard replacement strategies such as FIFO or LFU to update the content of the PAC cache whenever the cache size is exceeded.

6.5. System Software Footprint

The extensions provided to the middleware systems BASE and PCOM produce additional code and communication overhead, which are briefly discussed now.

Message	Type	System	Size [byte]
Event	EventService Message	BASE	335
Subscribe	EventService Message	BASE	399
Unsubscribe	EventService Message	BASE	401
AcquireHead	Clustering (DMAC)	PCOM	292
Join	Clustering (DMAC)	PCOM	285
RequestInfo	Clustering (DMAC)	PCOM	335
ComponentConfig	Configuration Class	PCOM	8.767
ResourceConfig	Configuration Class	PCOM	5.123

Table 6.1.: Message sizes

6.5.1. Message Sizes

A significant requirement to a comprehensive solution for homogeneous as well as heterogeneous systems was the gentle use of the limited system resources on mobile devices.

Regarding communication overhead, the message sizes of the distinct clustering and event service messages and the configuration classes that have to be transmitted to create the Virtual Containers are specified in Table 6.5.1. Usually, the cluster head loads the configuration classes of all of its cluster members and registers at the *BASE Event Service* of the cluster members for two events that notify about changed total and free resources on this device, respectively. While the *subscription* to remote events takes 399 bytes in average, the *unsubscription* requires 401 bytes to be transmitted. A single actual *event* message (e.g., a device wants to become cluster head) of the BASE Event Service takes 335 bytes in average. As we used IEEE 802.11b radio technology in our evaluations, the maximum effective data rate is about 50 % of 11 Mbit/s. Since the required configuration classes (*Component-Config* and *ResourceConfig*) have a size of around 14 kB in sum, the actual process of transmitting this amount of data requires only few milliseconds and represents a small fraction compared to the hardware and software latencies arising during the class loading process, as you can see in our evaluations. Moreover, these classes only need to be transmitted when the cluster structure changes, which is rather seldom the case in real-world scenarios, as we have stated in Section 5.7.2. This retains applicability of the Mobile Code concept. Furthermore, regarding the message and class sizes of the *AcquireHead*[1], *Join*[2] and *RequestInfo*[3] messages and the small number of messages that need to be transmitted when the DMAC clustering scheme is used, the transmitting medium has enough capacity for the required communication between cluster head and cluster members. Correspondingly, the cluster head's CPU is the restrictive factor for application configuration.

[1]Request of a device to become a cluster head
[2]Device wants to join another cluster head as simple cluster member
[3]Request of a new device to obtain the resource information of cluster members

Framework component	System	Size [kb]
Event Service	BASE	12.9
Mobile Code Service	BASE	11.7
Clustering Framework (incl. Selector abstraction)	PCOM	151.1
Direct Backtracking Assembler	PCOM	96.9
Hybrid Assembler	PCOM	99.3
Mobile Code Accessor	PCOM	92.5
PAC Framework (incl. 400 kB cache repository)	PCOM	597.0
Virtual Container instance	PCOM	8.8

Table 6.2.: Sizes of new framework components' binaries

6.5.2. Class Sizes

Concerning the sizes of BASE's and PCOM's new components' classes, Table 6.5.2 gives an overview of the consumed disk space of the binary files. The new Event Service and Mobile Code Service of BASE produce only minor space overhead of less than 25 kB. Regarding PCOM's new components, the PAC Framework represent the component that consumes most disk space, due to the provided cache repository of 400 kB. Furthermore, the Clustering Framework also requires a larger amount of space than most of the other new components, as it provides the benchmarking process and the clustering logic to automatically generate clusters based on the provided strategies. The total space overhead of the Clustering Framework (which is required by all devices) is 151.1 kB. We consider this overhead to be acceptable, compared to the available space on common up-to-date mobile devices.

The new centralized and hybrid assemblers as well as the Mobile Code Accessor to get access to remote configuration classes require less than 100 kB disk space, respectively. In a Virtual Container instance, only the relevant resource information of the respective remote device is stored, thus each Virtual Container requires only 8.8 kB of disk space in average. In summary, the complete framework extensions consume around one Megabyte of disk space, ensuring the framework's applicability on nowadays' standard mobile and stationary devices.

6.6. Summary and Discussion

In this chapter, we have discussed the middleware systems we used for a prototypical implementation of our concepts and mechanisms, consisting of the communication middleware BASE [BSGR03] and the component system PCOM [BHSR04]. Moreover, we have discussed the changes in the system architecture of PCOM/BASE that realize the concepts which were developed within this thesis.

BASE was extended by a service to automatically retrieve the configuration classes of remote devices, which is required to realize the pre-configuration process discussed in Section 4.1.3. Moreover, another service was integrated into BASE to allow the automatic update of these remote configuration classes by registering at the respective remote devices for update messages.

The adaptations of PCOM are more extensive. First of all, we implemented the clustering framework discussed in Section 4.3.2 into PCOM to allow the arrangement of the devices into different groups, which is needed to uniquely distribute the configuration load among the present devices. To consider the heterogeneity of the devices, we implemented the resource-aware clustering strategy introduced in Section 4.3.3. For the automatic retrieval of remote configuration classes via Mobile Code as discussed in Section 4.3.5, PCOM was extended by an additional framework that provides several access methods which guarantee the efficiency of the proactive class loading in various scenarios. To combine the resource-aware centralized configuration approach with the distributed configuration that exploits the parallelism available through multiple devices, an implementation of the hybrid assembler that was presented in Section 4.4.3 was implemented. This advanced assembler relies on the clustering framework and the local emulation of remote devices via Virtual Containers and increases the configuration efficiency especially in highly heterogeneous environments. Last but not least, the PAC Framework enables the use of the Partial Application Configurations introduced in Chapter 5. The implemented PAC Framework is general in the way that it is independent from specific configuration approaches, but may optionally be used by any configuration algorithm. Additional disk space was reserved for the provision of a *PAC Repository*, maintaining cached PACs which have been used in previous configuration processes.

For highly-dynamic large scale scenarios that involve a huge number of mobile as well as stationary devices, the discussed prototype could not be used for evaluation measurements due to device restrictions at our institute. Therefore, we revert to an event-discrete simulator of the component middleware that enables the simple comparison of different configuration approaches concerning the number of discrete steps an algorithm has to take to configure a specific application. This simulator, which has initially been developed in a previous work [HBR05], was extended by additional configuration algorithms and the possibility to simulate the (dis-)availability of devices according to specific probability distributions.

At the beginning of the work for this thesis, the discussed system software mainly focused on peer-based homogeneous Ad Hoc scenarios, consisting only of resource-poor mobile devices. With the far-reaching adaptations taken on the system software, the prototype's functionality has been significantly extended to additionally provide efficient support of heterogeneous scenarios with present resource-rich as well as resource-poor devices. We relied on this prototypical implementation as well as the simulator for the evaluations presented in the Chapters 4 and 5 to show the efficiency of our approach in realistic homogeneous as well as heterogeneous environments with various degrees of decentralization.

7 Conclusion

7.1. Summary

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. While many projects exist which provide system support for a specific kind of environment, comprehensive solutions that automatically adapt to changing conditions in dynamic environments are lacking.

Therefore in this dissertation thesis, we have presented a novel approach which efficiently supports heterogeneous Pervasive Computing scenarios. Our solution is based on various configuration schemes, reaching from centralized via hybrid up to completely decentralized approaches. Our approach is capable of adapting the degree of decentralization according to the currently available devices and services in dynamic environments. Furthermore, we have suggested to utilize the results from previous configuration results by integrating Partial Application Configurations (PACs) that were cached after successful configurations for their future re-use.

For weakly heterogeneous environments with an additional resource-rich infrastructure device, we have presented *Direct Backtracking (DBT)*, a new centralized algorithm for the efficient configuration and adaptation of tree-based distributed applications. DBT avoids thrashing effects completely due to an intelligent backtracking mechanism, while memory and code overhead are of acceptable size. Furthermore, DBT prevents unnecessary adaptations in many situations as it employs a proactive backtracking avoidance mechanism. We have shown that DBT significantly outperforms *Synchronous Backtracking (SBT)*, its closest predecessor, in configuring applications with various sizes, particularly when the applications are larger and include many contracts where multiple alternative components provide the required functionality.

Further in this thesis, we have presented a mechanism that enables the efficient support of automatic application configuration both in heterogeneous infrastructure-based and in homogeneous Ad Hoc pervasive environments. This approach is based on the introduction of a framework that enables the formation of clusters and the election of a cluster head according to the resource-awareness of the devices. Furthermore, we have developed the new concept of *Virtual Containers* that supports the use of mobile code to proactively load configuration-specific classes to the cluster head and, hence, enables efficient centralized application configuration on resource-rich devices. Class loading in advance and handovers between changing cluster heads for further decrease of the configuration latency are provided by *accessors*. The actual configuration is performed by a new centralized assembler which relies on Direct Backtracking as configuration algorithm and integrates the obtained Virtual Containers to avoid any communication latencies during configuration processes. By evaluation measurements, we proved that this framework can reduce configuration latencies significantly, particularly in heterogeneous environments and when eager class loading is used. This enables the desired support of various environments.

Furthermore, we have 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. The hybrid 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 multiple active devices. The hybrid approach automatically adjusts its degree of decentralization to the available resources in the network. In the evaluations of the hybrid scheme, we proved that in strongly heterogeneous environments, this 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.

Moreover, we have introduced the concept of Partial Application Configurations (PACs) in automatic composition of distributed applications. PACs exploit the results of previous configuration processes by re-using cached parts of a valid configuration in current configuration processes. As none of the related projects features a similar mechanism, the PAC approach is completely novel in this research area. We presented concepts for the efficient caching of PACs that are highly valuable for future configurations. Through this, the use of PACs reduces the number of components that actually need to be configured. In our evaluations, we showed that, compared to our approaches that do not rely on PACs, configuration which uses PACs significantly decreases configuration latencies in homogeneous as well as heterogeneous environments. The achievable configuration latencies are very low with only around 1.5 seconds in average for typical Pervasive Computing environments. This leads to less user distraction and, in consequence, to more seamless configurations and adaptations.

In summary, the provision of several configuration schemes suited for different environments, an automatic adaptation of the degree of decentralization, and the integration of the results from previous configuration processes represent large steps towards realizing the vision of true pervasiveness in the domain of automatic composition of distributed applications.

7.2. Outlook

Despite the hybrid configuration approach in combination with the use of partial application configuration provides automated efficient application composition in heterogeneous pervasive scenarios, we identified additional research directions which go beyond the ideas presented in this thesis and represent possible future work. These research topics address two different goals:

- The user shall not even be aware of ongoing configuration processes. Therefore, *configuration latencies* have to be dropped to values significantly below one second in various thinkable scenarios. One approach to reduce configuration latencies is to provide a more sophisticated cluster creation scheme (Section 7.2.1) that considers various influence factors for the mapping of the weak to the powerful devices. Furthermore, proactive approaches which configure parts of the application prior to the actual application start by the user (Section 7.2.2) and automatically create PACs independent from ongoing configuration processes (Section 7.2.4) may be useful to achieve this goal.
- Valid configurations have to be found with a high probability even in strictly resource-constrained scenarios. Thus, the aspired goal is to achieve a permanently high *application availability*. Application-comprehensive conflict handling (Section 7.2.3) and a more flexible component model (Section 7.2.5) aim at maximizing this application availability to the user.

7.2.1. Application-specific Cluster Creation

The configuration schemes discussed in Chapter 4 perform configuration calculations according to a depth-first search in the application tree. To fulfill a contract which is closed between components that are resident on different devices, the established Partial Configurations have to be communicated among the corresponding devices. In case of the hybrid configuration approach introduced in Section 4.4, this communication overhead only arises in case a contract is closed among weak devices which are mapped to different powerful devices.

To minimize this communication overhead and, thus, maximize the configuration efficiency, the distribution of the application components among the present devices as well as the application structure have to be analyzed before the clusters are established. Then, an improved clustering scheme has to be developed which exploits the local dependencies within the application structure and reduces the number of communications within configuration processes. For instance, if two powerful devices P_0 and P_1 are calculating configurations using the hybrid configuration scheme, then a simple algorithm could determine all possible cluster structures with P_0 and P_1

as cluster heads and select the one which minimizes the number of communications between P_0 and P_1 within the configuration. An advanced scheme could use a more fine-grained clustering selection scheme that judges the possible cluster structures according to additional influence factors such as the achieved load balance among the cluster heads, which has significant impact on the achievable degree of parallelism in the calculations.

7.2.2. Speculative Calculations in Hybrid Configurations

Even in the efficient hybrid configuration scheme discussed in Section 4.4, there still exist periods where a cluster head has to wait for receiving partial results from remote cluster heads. Within these waiting periods, the cluster head's CPU is idle and could be used for speculative calculation of so far unconfigured application parts that are also not covered by available PACs to further drop configuration latencies. Therefore, the configuration algorithm has to be extended to determine waiting periods and identify probable results of the configurations that are expected from remote cluster heads. Based on these assumptions about the currently missing configuration parts, the cluster head then starts to proactively calculate further application parts. In order to determine which of the remotely calculated configuration parts are likely, the extended algorithm has to consider gained information about the application structure, the functionality of the components, and the mobility patterns of the devices to estimate the probability of specific compositions.

When the configuration results from the remote devices are finally communicated, the cluster head has to analyse if his speculatively calculated configuration results match the results from the communicated compositions and are valid concerning the functional and resource constraints of the environment. If this is the case, the speculative configuration results are adopted to the current composition and the configuration continues with the first unconfigured contract. Otherwise, the speculative calculations are refused and the respective contracts are configured in the standard way.

Another challenge that has to be considered is the simultaneous speculative calculation of unconfigured application parts on *several* cluster heads. Particular attention has to be paid if multiple speculative configuration results revert to the same resource, as they may exceed the current availability of this resource. In such cases, the hybrid algorithm has to determine which of those partial configurations provide the highest contribution concerning the reduction of the latencies. This requires an extended cost model to evaluate the use of specific partial configurations.

7.2.3. Application-comprehensive Conflict Handling

Until now, we only consider the configuration and adaptation of single applications. However, in many typical environments, multiple applications may run concurrently. If these applications share the same resources, conflicts among these applications are inevitable. The respective resources thus become *conflicting* for the applications.

These conflicts may lead to a reduced quality of the running applications, but also to the unavailability of complete applications.

As the present devices and services are known prior to the application execution, possible conflicts in future application executions are proactively recognizable if an application-comprehensive conflict handling is performed. Therefore, strategies and mechanisms to analyse application execution situations in dynamic scenarios are required. The goal of this conflict analysis is to determine several conflict classes. One possibility to classify the conflicts is to distinguish between those conflicts that are avoidable if cautious allocation of conflicting resources is performed, and those conflicts which cannot be avoided without stopping the execution of specific applications.

To solve the first class of conflicts, the configuration algorithms have to be adapted to integrate information about other applications into the decomposiition decisions. For the second class of conflicts, the conflicts have to be handled by adapting the conflicting applications to deallocate the corresponding conflicting resources. If the conflict cannot be solved at all, e.g., because of insufficient resources, this has to be detected by the system, and the user needs to be notified of this problem.

The main goal of such an application-comprehensive conflict handling is to increase the availability of the applications to the users. A recent work of Tuttlies et al. [MSS+10] also considers application-comprehensive conflict handling, but does not explicitly focus on heterogeneous scenarios.

7.2.4. Proactive PAC Creation

So far, the Partial Application Configurations are created immediately after they were used in a configuration for the first time. This induces that it takes some amount of time to establish a considerable PAC basis which we can rely on in the cache. An approach to fill the PAC cache in a faster way is to proactively create the PACs, i.e., *before* they were used for the first time. This decreases the cache miss rate and, thus, reduces the expected configuration latency, especially in early configuration runs.

Schemes for proactive PAC creation should consider information about past configuration processes in terms of application histories as well as group mobility information [Sch07]. Furthermore, information about conflicts among different applications, as described in the previous section, may be helpful to decide which combinations of PACs yield a low application-comprehensive conflict risk. Based on this information, the goal is to proactively create only those PACs which have a high practical relevance and low conflict probability due to their specific component composition.

7.2.5. Flexibilization of Component Model

For the research presented within this thesis, we rely on a rather stiff component model where a component *either* matches the requirements of a contract completely, *or* not at all. However, in practice, in may happen that a component fulfills a contract's requirements, but with a reduced quality of service. For example, a display that provides a specific size or resolution may not fit the required size or resolution of a contract, as its specifications are below the predetermined specifications. This may lead to the undesirable situation that an application cannot be started.

However, in situations where the provided quality of service is only marginally below the one given by the contract specifications, the application availability may significantly be improved if the corresponding components are also taken into account by the configuration algorithm when no perfectly matching component is found.

Therefore, a more flexible component model is required which classifies suitable components according to their fulfillment of the contract's specifications. Moreover, the configuration algorithms need to be adapted in a way that they also regard components that are not perfectly matching the requirements. Advanced approaches in the field of semantic matching [GYS07], machine learning [DC97, DDG+08], pattern recognition [JDM00], or ontology-based matching [WB99] can be used to automatically search, create and evaluate alternative configurations in a specific environment. Obviously, it has to be ensured that the found compositions still provide an adequate service to the users. For instance, choosing a medium-sized LCD monitor instead of a large flat TV may be appropriate to users to show a video file. However, using the small screen of their mobile phone is probably not sufficient. Through this flexibilization of the component model, the situations where a composition cannot be found in a specific scenario may drastically be reduced, leading to an increased application availability, with the cost of a marginally reduced quality of service.

Bibliography

[ACKM03] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications.* Book. Springer Verlag, October 2003.

[AK06] Motilal Agrawal and Kurt Konolige. Real-time localization in outdoor environments using stereo vision and inexpensive gps. In *Proceedings of the 18th International Conference on Pattern Recognition (ICPR '06)*, pages 1063–1068, Hongkong, China, August 2006. IEEE Computer Society Press.

[Amo01] Daniel Amor. Pervasive Computing: The Next Chapter on the Internet. *Pearson Information Technology (InformIT) Journal. Electronic version: http://www.informit.com/articles/article.aspx?p=165227*, pages 1–5, October 2001.

[AP00] Alan D. Amis and Ravi Prakash. Load-Balancing Clusters in Wireless Ad Hoc Networks. In *Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00)*, pages 25–32, Richardson, Texas, USA, March 2000. IEEE Computer Society Press.

[ASW$^+$99] Ken Arnold, Robert Scheifler, Jim Waldo, Bryan O'Sullivan, and Ann Wollrath. *Jini Specification.* Book. Addison-Wesley Longman Publishing, June 1999.

[BAG01] Rajkumar Buyya, David Abramson, and Jonathan Giddy. A Case for Economy Grid Architecture for Service Oriented Grid Computing. In *Proceedings of the 10th Heterogeneous Computing Workshop (HCW 2001) at the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS '01)*, pages 1–15, San Francisco, CA, USA, April 2001. IEEE Computer Society Press.

[Bak05] Andrew B. Baker. *Intelligent Backtracking on Constraint Satisfac-*

tion Problems: Experimental and Theoretical Results. PhD Thesis, University of Oregon, March 2005.

[Bas99] Stefano Basagni. Distributed Clustering for Ad Hoc Networks. In *Proceedings of the 4th International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN'99)*, pages 310–315, Fremantle, Australia, June 1999. IEEE Computer Society Press.

[BB02] Guruduth Banavar and Abraham Bernstein. Software infrastructure and design challenges for ubiquitous computing applications. *Communications of the ACM*, 45(12):92–96, December 2002.

[BBG$^+$00] Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy Sussman, and Deborra Zukowski. Challenges: an application model for pervasive computing. In *Proceedings of the 6th annual international conference on Mobile computing and networking (MobiCom '00)*, pages 266–274, Boston, MA, USA, August 2000. ACM Press.

[BBM$^+$97] Michael Baentsch, Lothar Baum, Georg Molter, Steffen Rothkugel, and Peter Sturm. Enhancing the Web's infrastructure: from caching to replication. *IEEE Internet Computing*, 1(2):18–27, March 1997.

[BC03] Magdalena Balazinska and Paul Castro. Characterizing Mobility and Network Usage in a Corporate Wireless Local-Area Network. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys '03)*, pages 303–316, San Francisco, CA, USA, May 2003. ACM Press.

[BCFJ97] Stefano Basagni, Imrich Chlamtac, Andras Farago, and Erik Jonsson. A Generalized Clustering Algorithm for Peer-to-Peer Networks. In *Proceedings of the 24th International Conference on Automata, Languages and Programming (ICALP 1997), Satellite Workshop on Algorithmic Aspects of Communication (AC)*, pages 1–15, Bologna, Italy, July 1997. Springer Verlag.

[BCLM99] Azer Bestavros, Mark Crovella, Jun Liu, and David Martin. Distributed Packet Rewriting and its Application to Scalable Server Architectures. Technical Report BUCS-TR-1999-001, Boston University, Computer Science Department, Boston, MA, USA, January 1999.

[Bec04] Christian Becker. *System Support for Context-Aware Computing.* Habilitation Thesis, Universität Stuttgart, Institut für Parallele und Verteilte Systeme (IPVS), November 2004.

[Bes94] Christian Bessière. Arc-Consistency and Arc-Consistency Again. *Elsevier Journal on Artificial Intelligence (AI)*, 65(1):179–190, January 1994.

[BFM$^+$06] Jeff Burke, Jonathan Friedman, Eitan Mendelowitz, Heemin Park, and Mani B. Srivastava. Embedding expression: Pervasive computing architecture for art and entertainment. *Elsevier Journal on Pervasive and Mobile Computing*, 2(1):1–36, February 2006.

[BHSR04] Christian Becker, Marcus Handte, Gregor Schiele, and Kurt Rothermel. PCOM - A Component System for Pervasive Computing. In *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom '04)*, pages 67–76, Orlando, FL, USA, March 2004. IEEE Computer Society Press.

[BK87] Amnon Barak and Yoram Kornatzky. Design principles of operating systems for large scale multicomputers. In *Experiences with Distributed Systems*, pages 104–123, Kaiserslautern, Germany, September 1987. Springer Berlin / Heidelberg.

[BKL01] Prithwish Basu, Naved Khan, and Thomas D. C. Little. A Mobility Based Metric for Clustering in Mobile Ad Hoc Networks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01), Workshop on Wireless Networks and Mobile Computing (WNMC 2001)*, pages 413–418, Phoenix, AZ, USA, April 2001. IEEE Computer Society Press.

[BKMN02] Hyokyung Bahn, Kern Koh, Sang Lyul Min, and Sam H. Noh. Efficient Replacement of Nonuniform Objects in Web Caches. *IEEE Computer Journal*, 35(6):65–73, June 2002.

[BL98] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Elsevier Journal on Future Generation Computer Systems - Special Issue on HPCN '97*, 13(4-5):361–372, March 1998.

[BM04] Ismel Brito and Pedro Meseguer. Synchronous, asynchronous and hybrid algorithms for DisCSP. In *Proceedings of the 5th International Workshop on Distributed Constraints Reasoning (DCR-04)*, pages 791–805, Toronto, Canada, September 2004. Springer Berlin / Heidelberg.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BPT96] Pravin Bhagwat, Charles Perkins, and Satish Tripathi. Network Layer Mobility: an Architecture and Survey. *IEEE Personal Communications Magazine*, 3(3):54–64, June 1996.

[BSGR03] Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel. BASE - A Micro-broker-based Middleware For Pervasive Computing. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom '03)*, pages 443–451, Fort Worth, TX, USA, March 2003. IEEE Computer Society Press.

[BVBR02] Anand Balachandran, Geoffrey M. Voelker, Paramvir Bahl, and P. Venkat Rangan. Characterizing User Behavior and Network Performance in a Public Wireless LAN. In *Proceedings of the ACM International Conference on Measurements and Modeling of Computer*

Systems (ACM SIGMETRICS 2002), pages 195–205, Marina del Rey, CA, USA, June 2002. ACM Press.

[CAMCM05] Shiva Chetan, Jalal Al-Muhtadi, Roy Campbell, and M. Dennis Mickunas. Mobile Gaia: A Middleware for Ad-hoc Pervasive Computing. In *Proceedings of the 2nd IEEE Consumer Communications & Networking Conference (CCNC 2005)*, pages 223–228, Las Vegas, NV, USA, January 2005. IEEE Computer Society Press.

[CBW03] Tracy Camp, Jeff Boleng, and Lucas Wilcox. Location Information Services in Mobile Ad Hoc Networks. In *Proceedings of the 38th IEEE International Conference on Communications (ICC '03)*, pages 3318–3324, Anchorage, AK, USA, May 2003. IEEE Computer Society.

[CCG$^+$07] Paolo Costa, Geoff Coulson, Richard Gold, Manish Lad, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, Thirunavukkarasu Sivaharan, Nirmal Weerasinghe, and Stefanos Zachariadis. The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario. In *Proceedings of the 5th IEEE International Conference on Pervasive Computing and Communications (PerCom '07)*, pages 69–78, White Plains, NY, USA, March 2007. IEEE Computer Society Press.

[CCS08] Javier Cámara, Carlos Canal, and Gwen Salaün. Multiple Concern Adaptation for Run-time Composition in Context-Aware Systems. *Elsevier Electronical Notes on Theoretical Computer Science*, 215:111–130, June 2008.

[CCY99] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. DNS Dispatching Algorithms with State Estimators for Scalable Web-Server Clusters. *Baltzer Science World Wide Web Journal*, 2(3):101–113, August 1999.

[CDT02] Mainak Chatterjee, Sajal K. Das, and Damla Turgut. WCA: A Weighted Clustering Algorithm for Mobile Ad Hoc Networks. *Springer Cluster Computing*, 5(2):193–204, April 2002.

[CHC$^+$05] Augustin Chaintreau, Pan Hui, Jon Crowcroft, Christophe Diot, Richard Gass, and James Scott. Pocket-Switched Networks: Real-world mobility and its consequences for opportunistic forwarding. Technical Report 617, University of Cambridge, Computer Laboratory, Cambridge, UK, February 2005.

[CICY99] Valeria Cardellini, Roma I, Michele Colajanni, and Philip S. Yu. Dynamic Load Balancing on Web-server Systems. *IEEE Internet Computing*, 3(3):28–39, May-June 1999.

[CJBM02] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris. Span: an energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. *ACM Wireless Networks Journal*, 8(5):481–494, September 2002.

[CLM$^+$08] Tiziana Catarci, Massimiliano de Leoni, Andrea Marrella, Massimo Mecella, Berardino Salvatore, Guido Vetere, Schahram Dustdar, Lukasz Juszczyk, Atif Manzoor, and Hong-Linh Truong. Pervasive Software Environments for Supporting Disaster Responses. *IEEE Internet Computing*, 12(1):26–37, January-February 2008.

[CP09] Antonio Coronato and Giuseppe De Pietro. Formal specification of dependable pervasive applications. In *Proceedings of the 4th IEEE Asia-Pacific Services Computing Conference (IEEE APSCC 2009)*, pages 358–365, Singapore, Singapore, December 2009. IEEE Computer Society Press.

[CPFJ04] Harry Chen, Filip Perich, Timothy W. Finin, and Anupam Joshi. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In *Proceedings of the 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous 2004)*, pages 258–267, Boston, MA, USA, August 2004. IEEE Computer Society Press.

[CPV97] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 22–32, Boston, MA, USA, May 1997. ACM Press.

[CWLG97] Ching-chuan Chiang, Hsiao-Kuang Wu, Winston Liu, and Mario Gerla. Routing in Clustered Multihop Mobile Wireless Networks with Fading Channel. In *Proceedings of the IEEE Singapore International Conference in Networks (SICON '97)*, pages 197–211, Singapore, Singapore, April 1997. IEEE Computer Society Press.

[DBGW10] Christophe Dumez, Mohamed Bakhouya, Jaafar Gaber, and Maxime Wack. Formal Specification and Verification of Service Composition using LOTOS. In *Proceedings of the 7th International Conference on Pervasive Services (ICPS 2010)*, pages 1–8, Berlin, Germany, July 2010. ACM Press.

[DC97] David M. Dutton and Gerard V. Conroy. A Review of Machine Learning. *The Knowledge Engineering Review Journal. Cambridge University Press*, 12(4):341–367, December 1997.

[DDG$^+$08] Thomas G. Dietterich, Pedro Domingos, Lise Getoor, Stephen Muggleton, and Prasad Tadepalli. Structured machine learning: the next ten years. *Springer Machine Learning*, 73(1):3–23, October 2008.

[Dec90] Rina Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Elsevier Journal on Artificial Intelligence (AI)*, 41(3):273–312, January 1990.

[DGIR11] Oleg Davidyuk, Nikolaos Georgantas, Valérie Issarny, and Jukka Riekki. MEDUSA: Middleware for End-User Composition of Ubiquitous Applications. *Handbook of Research on Ambient Intelligence and Smart Environments: Trends and Perspectives. Edited by IGI Global*, pages 197–219, May 2011.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[DHB02] Sajal K. Das, Daniel J. Harvey, and Rupak Biswas. MinEX: a latency-tolerant dynamic partitioner for grid computing applications. *Elsevier Journal on Future Generation Computer Systems*, 18(4):477–489, March 2002.

[Dow97] Troy B. Downing. *Java RMI: Remote Method Invocation*. Book. IDG Books Worldwide, Inc., Foster City, CA, USA, December 1997.

[DR07] Krista M. Dombroviak and Rajiv Ramnath. A Taxonomy of Mobile and Pervasive Applications. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC '07)*, pages 1609–1615, Seoul, Korea, March 2007. ACM Press.

[DR08] Frank Dürr and Kurt Rothermel. An Adaptive Overlay Network for World-wide Geographic Messaging. In *Proceedings of the 22nd IEEE International Conference on Advanced Information Networking and Applications (AINA '08)*, pages 875–882, GinoWan, Okinawa, Japan, March 2008. IEEE Computer Society Press.

[DSSK06] Anind K. Dey, Timothy Sohn, Sara Streng, and Justin Kodama. iCAP: Interactive prototyping of context-aware applications. In *Proceedings of the 4th International Conference on Pervasive Computing (Pervasive 2006)*, pages 254–271, Dublin, Ireland, May 2006. Springer Berlin / Heidelberg.

[ENS$^+$02] W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy, Trevor F. Smith, Dirk Balfanz, D. K. Smetters, H. Chi Wong, and Shahram Izadi. Using Speakeasy for Ad Hoc Peer-to-Peer Collaboration. In *Proceedings of the 2002 ACM Conference on Computer supported cooperative work (CSCW '02)*, pages 256–265, New Orleans, LA, USA, November 2002. ACM Press.

[EP06] Nathan Eagle and Alex Pentland. Reality Mining: Sensing Complex Social Systems. *Springer Personal and Ubiquitous Computing*, 10(4):255–268, 2006.

[Fai05] Jamal Faik. *A Model for Resource-Aware Load Balancing on Heterogeneous and Non-Dedicated Clusters*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 2005.

[FHMO04] Alois Ferscha, Manfred Hechinger, Rene Mayrhofer, and Roy Oberhauser. A Light-Weight Component Model for Peer-to-Peer Applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS) Workshops - Workshop 4: Mobile Distributed Computing (MDC)*, pages 520–527, Tokyo, Japan, March 2004. IEEE Computer Society Press.

[Fre78] Eugene C. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM (CACM)*, 21(11):956–966, November 1978.

[FS99] Jason Flinn and Mahadev Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 48–63, Charleston, SC, USA, December 1999. ACM Press.

[Gas77] John G. Gaschnig. A General Backtrack Algorithm That Eliminates Most Redundant Checks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI)*, page 457, Cambridge, MA, USA, August 1977. William Kaufmann.

[GBK99] Hans W. Gellersen, Michael Beigl, and Holger Krull. The MediaCup: Awareness Technology embedded in an Everyday Object. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC99*, pages 308–310, Karlsruhe, Germany, 1997 1999. Springer-Verlag.

[GHR09] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Efficient and Scalable Network Emulation using Adaptive Virtual Time. In *Proceedings of the 18th International Conference on Computer Communications and Networks (ICCCN 2009)*, pages 1–6, San Francisco, CA, USA, August 2009. IEEE Computer Society Press.

[Gin93] Matthew L. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, August 1993.

[Gon01] Li Gong. JXTA: a network programming environment. *IEEE Internet Computing Journal*, 5(3):88–95, May 2001.

[GPR$^+$98] Douglas Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. *Journal on Software – Practices and Experience. Special Issue on Multiprocessor Operating Systems*, 28(9):929–961, July 1998.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Book. Morgan Kaufmann, September 1992.

[Gri04] Robert Grimm. One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing Journal*, 3(3):22–30, 2004.

[GTCT95] Mario Gerla and Jack Tzu-Chieh Tsai. Multicluster, mobile, multimedia radio network. *Journal of Wireless Networks*, 1(3):255–265, 1995.

[GYS07] Fausto Giunchiglia, Mikalai Yatskevich, and Pavel Shvaiko. Semantic Matching: Algorithms and Implementation. *Journal on Data Semantics IX*, 9:1–38, September 2007.

[Ham02] Youssef Hamadi. Optimal Distributed Arc-Consistency. *Springer Journal on Constraints*, 7(3-4):367–385, July-October 2002.

[Han09] Marcus Handte. *System Support for Adaptive Pervasive Applications*. PhD thesis, University of Stuttgart, Institute of Parallel and Distributed Systems (IPVS), 2009.

[HBR05] Marcus Handte, Christian Becker, and Kurt Rothermel. Peer-based Automatic Configuration of Pervasive Applications. *Journal on Pervasive Computing and Communications*, 1(4):251–264, 2005.

[HCS$^+$05] Pan Hui, Augustin Chaintreau, James Scott, Richard Gass, Jon Crowcroft, and Christophe Diot. Pocket Switched Networks and Human Mobility in Conference Environments. In *Proceedings of the ACM SIGCOMM 2005 Workshop on Delay-Tolerant Networking (WDTN '05)*, pages 244–251, Philadelphia, PA, USA, August 2005. ACM Press.

[Hen05] Urs Hengartner. *Access Control to Information in Pervasive Computing Environments*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA, August 2005.

[HGKM98] Guerney D. H. Hunt, Germán S. Goldszmidt, Richard P. King, and Rajat Mukherjee. Network Dispatcher: a connection router for scalable Internet services. *Journal on Computer Networks and ISDN Systems*, 30(1-7):347–357, April 1998.

[HHS$^+$07] Marcus Handte, Klaus Herrmann, Gregor Schiele, Christian Becker, and Kurt Rothermel. Automatic Reactive Adaptation of Pervasive Applications. In *Proceedings of the IEEE International Conference on Pervasive Services (ICPS'07)*, pages 214–222, Istanbul, Turkey, July 2007. IEEE Computer Society Press.

[HHSB07] Marcus Handte, Klaus Herrmann, Gregor Schiele, and Christian Becker. Supporting Pluggable Configuration Algorithms in PCOM. In *Proceedings of the 5th IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW '07)*, pages 472–476, White Plains, NY, USA, March 2007. IEEE Computer Society Press.

[HKA04] Tristan Henderson, David Kotz, and Ilya Abyzov. The Changing Usage of a Mature Campus-wide Wireless Network. In *Proceedings of the 10th International Conference on Mobile Computing and Networking (MobiCom '04)*, pages 187–201, Philadelphia, PA, USA, September 2004. ACM Press.

[HMEZ$^+$05] Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura, and Erwin Jansen. The Gator Tech Smart House: A Programmable Pervasive Space. *IEEE Computer*, 38(3):50–60, March 2005.

[HR02] Daniel Herrscher and Kurt Rothermel. A Dynamic Network Scenario Emulation Tool. In *Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN 2002)*, pages 262–267, Miami, FL, USA, October 2002. IEEE Computer Society Press.

[HRKD08] Klaus Herrmann, Kurt Rothermel, Gerd Kortuem, and Naranker Dulay. Adaptable Pervasive Flows – An Emerging Technology for Pervasive Adaptation. In *Proceedings of the Workshop on Pervasive Adaptation at the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008)*, pages 108–113, Venice, Italy, October 2008. IEEE Computer Society Press.

[HSM$^+$12] Marcus Handte, Gregor Schiele, Verena Majuntke, Christian Becker, and Pedro J. Marrón. 3PC: System support for adaptive peer-to-peer pervasive computing. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(1):10:1–10:19, April 2012.

[HUB$^+$06] Marcus Handte, Stephan Urbanski, Christian Becker, Patrick Reinhardt, Michael Engel, and Matthew Smith. 3PC/MarNET Pervasive Presenter. In *Demonstrations Session at the IEEE International Conference on Pervasive Computing and Communications (PerCom '06)*, Pisa, Italy, March 2006.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: designing, building and deploying messaging solutions.* Book. Pearson Education, October 2003.

[HWS$^+$10] Marcus Handte, Stephan Wagner, Gregor Schiele, Christian Becker, and Pedro José Marrón. The BASE Plug-in Architecture – Composable Communication Support for Pervasive Systems. In *Proceedings of the 7th ACM International Conference on Pervasive Services (ICPS '10)*, pages 1–8, Berlin, Germany, July 2010.

[HY02] Katsutoshi Hirayama and Makoto Yokoo. Local Search for Distributed SAT with Complex Local Problems. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '02)*, pages 1199–1206, New York, NY, USA, July 2002. ACM Press.

[HY04] Peter Henderson and Jingtao Yang. Reusable Web Services. In *Proceedings of the 8th International Conference on Software Reuse: Methods, Techniques and Tools (ICSR 2004)*, pages 185–194, Madrid, Spain, July 2004. Springer Berlin / Heidelberg.

[HY05] Katsutoshi Hirayama and Makoto Yokoo. The Distributed Breakout Algorithms. *Journal on Artificial Intelligence*, 161(1-2):89–115, January 2005.

[IIS$^+$03] Masaki Ito, Akiko Iwaya, Masato Saito, Kenichi Nakanishi, Kenta Matsumiya, Jin Nakazawa, Nobuhiko Nishio, Kazunori Takashio, and Hideyuki Tokuda. Smart Furniture: Improvising Ubiquitous Hot-Spot Environment. In *International Conference on Distributed Computing Systems Workshops (ICDCSW 2003)*, pages 248–253, Providence, RI, USA, May 2003. IEEE Computer Society.

[JDM00] Anil K. Jain, Robert P. W. Duin, and Jianchang Mao. Statistical Pattern Recognition: A Review. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 22(1):4–37, January 2000.

[JFW02] Brad Johanson, Armando Fox, and Terry Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing, 1(2)*, 2002.

[JS03] Glenn Judd and Peter Steenkiste. Providing Contextual Information to Pervasive Computing Applications. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom '03)*, pages 133–142, Pittsburgh, PA, USA, March 2003. IEEE Computer Society Press.

[KBM+96] Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris MC: a multi computer OS. In *Proceedings of the 1996 Annual Technical Conference on USENIX (ATEC '96)*, page 16, Berkeley, CA, USA, January 1996. USENIX Association Berkeley, CA, USA.

[KBM+00] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. People, Places, Things: Web Presence for the Real World. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '00)*, pages 19–28, Monterey, CA, USA, December 2000. IEEE Computer Society Press.

[KDLS08] Priyantha Kumarawadu, Dan J. Dechene, Marco Luccini, and Allan Sauer. Algorithms for Node Clustering in Wireless Sensor Networks: A Survey. In *Proceedings of the 4th International Conference on Information and Automation for Sustainability (ICIAFS 2008)*, pages 295–300, Sri Lanka, December 2008. IEEE Computer Society Press.

[KLBV07] Thomas Karagiannis, Jean-Yves Le Boudec, and Milan Vojnović. Power Law and Exponential Decay of Inter Contact Times between Mobile Devices. In *Proceedings of the 13th International Conference on Mobile Computing and Networking (MobiCom '07)*, pages 183–194, Montréal, Canada, September 2007. ACM Press.

[KNK05] Sachin Kogekar, Sandeep Neema, and Xenofon Koutsoukos. Dynamic Software Reconfiguration in Sensor Networks. In *Proceedings of the International Systems Communications Conferences on Wireless Technologies/High Speed Networks/Multimedia Communications Systems/Sensor Networks (ICW'05, ICHSN'05, ICMCS'05, SENET'05)*, pages 413–420, Montreal, Canada, August 2005. IEEE Computer Society Press.

[KWSW07] Mirko Knoll, Arno Wacker, Gregor Schiele, and Torben Weis. Decentralized Bootstrapping in Pervasive Applications. In *Proceedings of the 5th Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW 2007)*, pages 589–592, White Plains, NY, USA, March 2007. IEEE Computer Society Press.

[LAS07] Steffen Lamparter, Anupriya Ankolekar, and Rudi Studer. Preference-based Selection of Highly Configurable Web Services. In *Proceedings* *of the 16th International World Wide Web Conference (WWW '07)*, pages 1013–1022, Banff, Canada, May 2007. ACM Press.

[LCK$^+$01] Donghee Lee, Jongmoo Choi, J. Kim, S. Noh, Sang Min, Yookun Cho, and Chong Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001.

[LG06] Chunhung R. Lin and Mario Gerla. Adaptive Clustering for Mobile Wireless Networks. *IEEE Journal on Selected Areas in Communications*, 15(7):1265–1275, September 2006.

[LL05] Yawei Li and Zhiling Lan. A Survey of Load Balancing in Grid Computing. In *Proceedings of the 1st International Symposium on Computational and Information Science (CIS '04)*, pages 280–285, Shanghai, China, December 2005. Springer Berlin / Heidelberg.

[LNH03] Choonhwa Lee, David Nordstedt, and Sumi Helal. Enabling Smart Spaces with OSGi. *IEEE Pervasive Computing*, 2(3):89–94, July-September 2003.

[Log13] Logitech®. Harmony® Remotes Website: http://www.logitech.com/en-us/harmony-remotes. online, 2013.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification - Second Edition.* Book. Addison Wesley Professional, February 1999.

[Mac77] Alan K. Mackworth. Consistency in Networks of Relations. *Elsevier Artificial Intelligence*, 8(1):99–118, February 1977.

[Maj10] Verena Majuntke. Context-based Coordination for Pervasive Computing Applications. In *Proceedings of the 8th International Conference on Pervasive Computing and Communications Workshops (PerComW '10)*, pages 853–854, Mannheim, Germany, March 2010. IEEE Computer Society Press.

[Mat03] Friedemann Mattern. From Smart Devices to Smart Everyday Objects. In *Proceedings of the 2nd International Smart Objects Conference (sOc '2003)*, pages 15–16, Grenoble, France, May 2003. CNRS/France Telecom.

[MF04] Laurence Melloul and Armando Fox. Reusable Functional Composition Patterns for Web Services. *Proceedings of the 2004 IEEE International Conference on Web Services (ICWS '04)*, pages 498–505, July 2004.

[MH86] Roger Mohr and Thomas C. Henderson. Arc and Path Consistency Revisited. *Elsevier Journal on Artificial Intelligence*, 28(2):225–233, 1986.

[MK01] Dave Marples and Peter Kriens. The Open Services Gateway Initiative: an introductory overview. *IEEE Communications Magazine*, 39(12):110–114, December 2001.

[Mon74] Ugo Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Elsevier Journal on Information Sciences*, 7:95–132, 1974.

[Mor93] Paul Morris. The Breakout Method for Escaping from Local Minima. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI '93)*, pages 40–45, Washington, DC, USA, July 1993. AAAI Press.

[MPJL92] Steven Minton, Andy Philips, Mark D. Johnston, and Philip Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems. *Elsevier Journal on Artificial Intelligence*, 58(1-3):161–205, December 1992.

[MSKC04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing Adaptive Software. *IEEE Computer*, 37(7):56–64, July 2004.

[MSS$^+$10] Verena Majuntke, Gregor Schiele, Kai Spohrer, Marcus Handte, and Christian Becker. A Coordination Framework for Pervasive Applications in Multi-User Environments. In *Proceedings of the 6th International Conference on Intelligent Environments (IE 2010)*, pages 178–184, Kuala Lumpur, Malaysia, July 2010. IEEE Computer Society Press.

[MV03] Marvin McNett and Geoffrey M. Voelker. Access and Mobility of Wireless PDA Users. *ACM SIGMOBILE Mobile Computing and Communications Review (MC2R) Journal*, 7(4), October 2003.

[ND98] Thang T. Nguyen and Yves Deville. A Distributed Arc-Consistency Algorithm. *Elsevier Journal on Science of Computer Programming*, 30(1-2):227–250, January 1998.

[NES08] Mark W. Newman, Ame Elliott, and Trevor F. Smith. Providing an Integrated User Experience of Networked Media, Devices, and Services through End-User Composition. In *Proceedings of the 6th International Conference on Pervasive Computing (Pervasive 2008)*, pages 213–227, Sydney, Australia, May 2008. Springer Berlin / Heidelberg.

[NGL$^+$09] Cuong P. Nguyen, Serge Garlatti, Simon Lau, Thomas Vantroys, and Benjamin Barbry. Pervasive Learning System based on a Scenario Model integrating Web Service Retrieval and Orchestration. *International Journal of Interactive Mobile Technologies (iJIM)*, 3(2):25–32, March 2009.

[NIE$^+$02] Mark W. Newman, Shahram Izadi, W. Keith Edwards, Jana Z. Sedivy, and Trevor F. Smith. User Interfaces when and where they are needed: An Infrastructure for Recombinant Computing. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST '02)*, pages 171–180, Paris, France, October 2002. ACM Press.

[NS78] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Commununications of the ACM*, 21(12):993–999, 1978.

[OD07] Olufisayo Omojokun and Prasun Dewan. Automatic Generation of Device User-Interfaces? In *Proceedings of the 5th IEEE International Conference on Pervasive Computing and Communications (PerCom 2007)*, pages 251–261, White Plains, NY, USA, March 2007. IEEE Computer Society Press.

[OGT$^+$99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[OIK03] Tomoyuki Ohta, Shinji Inoue, and Yoshiaki Kakuda. An Adaptive Multihop Clustering Scheme for Highly Mobile Ad Hoc Networks. In *Proceedings of the The 6th International Symposium on Autonomous Decentralized Systems (ISADS'03)*, pages 293–300, Pisa, Italy, April 2003. IEEE Computer Society Press.

[OPID06] Olufisayo Omojokun, S. Pierce, L. Isbell, and Prasun Dewan. Comparing End-User and Intelligent Remote Control Interface Generation. *Springer Journal on Personal and Ubiquitous Computing*, 10(2-3):136–143, January 2006.

[O'R05] Tim O'Reilly. What Is Web 2.0? Design Patterns and Business Models for the Next Generation of Software. Online source: http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html, September 2005.

[PA05] Cesare Pautasso and Gustavo Alonso. Flexible Binding for Reusable Composition of Web Services. In *Proceedings of the 4th Workshop on Software Composition (SC 2005)*, pages 151–166, Edinburgh, UK, April 2005. Springer Berlin / Heidelberg.

[PB03] Stefan Podlipnig and László Böszörményi. A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys*, 35(4):374–398, December 2003.

[Phi10] Philips®. Prestigo SRT8215 Universal Remote Control Website: http://www.philips.co.uk/c/remote-control/15881/cat. Online, 2010.

[PLF$^+$01] Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Proceeding of the 2001 International Conference on Ubiquitous Computing (Ubicomp 2001)*, pages 56–75, Atlanta, GA, USA, September 2001. Springer Berlin / Heidelberg.

[PPS$^+$08] Justin M. Paluska, Hubert Pham, Umar Saif, Grace Chau, Chris Terman, and Steve Ward. Structured Decomposition of Adaptive Applications. *Elsevier Journal on Pervasive and Mobile Computing*, 4(6):791–806, December 2008.

[PSGS04] Vahe Poladian, Joao Pedro Sousa, David Garlan, and Mary Shaw. Dynamic Configuration of Resource-Aware Services. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 604–613, Edinburgh, UK, May 2004. IEEE Computer Society Press.

[PWR$^+$09] Trevor Pering, Roy Want, Barbara Rosario, Shivani Sud, and Kent Lyons. Enabling Pervasive Collaboration with Platform Composition. In *Proceedings of the 7th International Conference on Pervasive Computing (Pervasive '09)*, pages 184–201, Nara, Japan, May 2009. Springer Berlin / Heidelberg.

[QCH08] Zhao Qiu, Mingrui Chen, and Jun Huang. Improvement of High Reliable Cluster Load Balancing Algorithm based on Multiple Regression. In *Proceedings of the IEEE International Conference on Service Operations and Logistics, and Informatics (IEEE SOLI 2008)*, pages 2979–2982, Beijing, China, October 2008. IEEE Computer Society Press.

[RCAM$^+$05] Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H. Campbell, and M. Dennis Mickunas. Olympus: A High-Level Programming Model for Pervasive Computing Environments. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communication (PerCom '05)*, pages 7–16, Kauai Island, HI, USA, March 2005. IEEE Computer Society Press.

[RHC$^+$02] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing*, 1(4), 2002.

[RSC02] Jung-hee Ryu, Sanghwa Song, and Dong-Ho Cho. New Clustering Schemes for Energy Conservation in Two-Tiered Mobile Ad Hoc Networks. *IEEE Transactions on Vehicular Technology*, 51(6):1661–1668, November 2002.

[Rud01] Larry Rudolph. Project Oxygen: Pervasive, Human-Centric Computing - An Initial Experience. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE '01)*, pages 1–12, Interlaken, Switzerland, June 2001. Springer Berlin / Heidelberg.

[Sai03] Umar Saif. A Case for Goal-oriented Programming Semantics. In *Proceedings of the 5th International Conference on Ubiquitous Computing (UbiComp 2003) Workshops*, pages 1–8, Seattle, WA, USA, October 2003. Springer Berlin / Heidelberg.

[Sat90] Mahadev Satyanarayanan. A Survey of Distributed File Systems. In *Annual Review of Computer Science*, volume 4, pages 73–104. Annual Reviews, June 1990.

[Sat01] Mahadev Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications Journal*, 8(4):10–17, 2001.

[SBK06] Ramesh Singh, Preeti Bhargava, and Samta Kain. State of the Art Smart Spaces: Application Models and Software Infrastructure. *ACM Ubiquity Magazine*, pages 2–9, September 2006.

[Sch03] Jochen Schiller. *Mobile Communications*. Book. Addison Wesley, 2003.

[Sch07] Gregor Schiele. *System Support for Spontaneous Pervasive Computing Environments*. PhD thesis, Universität Stuttgart, Institute of Parallel and Distributed Systems (IPVS), 2007.

[Sch09] Stephan Schuhmann. Pervasive Presenter. Demonstration at the First Review of the FET7 project ALLOW – Adaptable Pervasive Flows. Brussels, Belgium, April 2009.

[Sch10] Veit Schwartze. Adaptive User Interfaces for Smart Environments. In *Proceedings of the 7th International Conference on Pervasive Services (ICPS 2010), Doctoral Colloquium*, Berlin, Germany, July 2010. ACM Press.

[SDFGB10] Hong Sun, Vincenzo De Florio, Ning Gui, and Chris Blondia. The Missing Ones: Key Ingredients Towards Effective Ambient Assisted Living Systems. *Journal on Ambient Intelligence and Smart Environments*, 2(2):109–120, 2010.

[Ses98] Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. Book. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[SG02] Joao Pedro Sousa and David Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Proceedings of the 3rd Working International IEEE/IFIP Conference on Software Architecture (WICSA'02)*, pages 29–43, Deventer, The Netherlands, August 2002. Kluwer, B.V.

[SH87] Ashok Samal and Tom Henderson. Parallel Consistent Labeling Algorithms. *International Journal of Parallel Programming*, 16(5):341–364, October 1987.

[SHR08a] Stephan Schuhmann, Klaus Herrmann, and Kurt Rothermel. A Framework for Adapting the Distribution of Automatic Application Configuration. In *Proceedings of the 2008 ACM International Conference on Pervasive Services (ICPS '08)*, pages 163–172, Sorrento, Italy, July 2008. ACM Press.

[SHR08b] Stephan Schuhmann, Klaus Herrmann, and Kurt Rothermel. Direct Backtracking: An Advanced Adaptation Algorithm for Pervasive Applications. In *Proceedings of the 21st International Conference on Architecture of Computing Systems (ARCS 2008)*, pages 53–67, Dresden, Germany, February 2008. Springer Berlin / Heidelberg.

[SHR$^+$08c] Stephan Schuhmann, Klaus Herrmann, Kurt Rothermel, Jan Blumenthal, and Dirk Timmermann. Improved Weighted Centroid Localization in Smart Ubiquitous Environments. In *Proceedings of the 5th International Conference on Ubiquitous Intelligence and Computing (UIC-08)*, pages 20–34, Oslo, Norway, June 2008. Springer Berlin / Heidelberg.

[SHR10] Stephan Schuhmann, Klaus Herrmann, and Kurt Rothermel. Efficient Resource-Aware Hybrid Configuration of Distributed Pervasive Applications. In *Proceedings of the 8th International Conference on Pervasive Computing (Pervasive 2010)*, pages 373–390, Helsinki, Finland, May 2010. Springer Berlin / Heidelberg.

[SHRB13] Stephan Schuhmann, Klaus Herrmann, Kurt Rothermel, and Yazan Boshmaf. Adaptive Composition of Distributed Pervasive Applications in Heterogeneous Environments. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) – to appear*, 2013.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, WA, USA, August 1994. AAAI press.

[SM03] Debashis Saha and Amitava Mukherjee. Pervasive Computing: A Paradigm for the 21st Century. *IEEE Computer*, 36(3):25–31, March 2003.

[SMA08] Haidar Safa, Omar Mirza, and Hassan Artail. A Dynamic Energy Efficient Clustering Algorithm for MANETs. In *Proceedings of the 2008 IEEE International Conference on Wireless & Mobile Computing, Networking & Communication (WiMob '08)*, pages 51–56, Avignon, France, October 2008. IEEE Computer Society Press.

[SPG$^+$06] Joao P. Sousa, Vahe Poladian, David Garlan, Bradley Schmerl, and Mary Shaw. Task-based Adaptation for Ubiquitous Computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36(3):328–340, May 2006.

[SR07] Xiang Song and Umakishore Ramachandran. MobiGo: A Middleware for Seamless Mobility. In *Proceedings of the 13th IEEE International Conference on Real-Time Computing Systems and Applications (RCTSA '07)*, pages 249–256, Daegu, Korea, August 2007. IEEE Computer Society.

[SS77] R. M. Stallman and G. J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Journal on Artificial Intelligence*, 9:135–196, October 1977.

[SSJ02] Inderjeet Singh, Beth Stearns, and Mark Johnson. *Designing Enterprise Applications with the J2EE Platform.* Book. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, April 2002.

[Sun03] Sun Microsystems. Connected Limited Device Configuration (CLDC) Specification - Version 1.1, 2003.

[SV08] Stephan Schuhmann and Lars Völker. Combining Passive Autoconfiguration and Anomaly-based Intrusion Detection in Ad-hoc Networks. In *Proceedings of the 8th International Workshop on Applications and Services in Wireless Networks (ASWN '08)*, pages 87–95, Kassel, Germany, October 2008. IEEE Computer Society Press.

[SW98] Yi Shang and Benjamin W. Wah. A Discrete Lagrangian-Based Global-SearchMethod for Solving Satisfiability Problems. *Springer Journal of Global Optimization*, 12(1):61–99, January 1998.

[Tan01] Peter Tandler. Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices. In *Proceedings of the 3rd International Conference on Pervasive and Ubiquitous Computing (UbiComp '01)*, pages 96–115, Atlanta, GA, USA, October 2001. Springer Berlin / Heidelberg.

[THA04] Khai N. Truong, Elaine M. Huang, and Gregory D. Abowd. CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home. In *Proceedings of the 6th International Conference on Pervasive and Ubiquitous Computing (Ubicomp 2004)*, pages 143–160, Nottingham, UK, September 2004. Springer Berlin / Heidelberg.

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Book. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, October 2006.

[TSB07] Verena Tuttlies, Gregor Schiele, and Christian Becker. COMITY - Conflict Avoidance in Pervasive Computing Environments. In *Proceedings of the OTM Confederated International Workshops (OTM 2007 Workshops): On the Move to Meaningful Internet Systems 2007*, pages 763–772, Vilamoura, Portugal, November 2007. Springer Berlin / Heidelberg.

[TSB09] Verena Tuttlies, Gregor Schiele, and Christian Becker. End-User Configuration for Pervasive Computing Environments. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS '09)*, pages 487–493, Fukuoka, Japan, March 2009.

[TYZ$^+$11] Lei Tang, Zhiwen Yu, Xingshe Zhou, Hanbo Wang, and Christian Becker. Supporting Rapid Design and Evaluation of Pervasive Applications: Challenges and Solutions. *Journal on Personal and Ubiquitous Computing*, 15(13):253–269, March 2011.

[Var07] Upkar Varshney. Pervasive Healthcare and Wireless Health Monitoring. *ACM/Springer Journal on Mobile Networks and Applications (MONET)*, 12(2-3):113–127, June 2007.

[Vin97] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 35(2):46–55, February 1997.

[WB97] Mark Weiser and John Seely Brown. The Coming Age of Calm Technology. *Book Article. Beyond calculation: the next fifty years*, pages 75–85, 1997.

[WB99] Peter C. Weinstein and William P. Birmingham. Comparing Concepts in Differentiated Ontologies. In *Proceedings of the 12th Workshop on Knowledge Acquisition, Modeling and Management (KAW'99)*, pages 1–22, Banff, Canada, October 1999. Springer Berlin / Heidelberg.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American - Special Issue on Communications, Computers, and Networks*, pages 94–104, February 1991.

[Wei99] Mark Weiser. The Computer for the 21st Century. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, September 1999.

[WFG92] Roy Want, Veronica Falcao, and Jon Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, January 1992.

[WHKB06] Torben Weis, Marcus Handte, Mirko Knoll, and Christian Becker. Customizable Pervasive Applications. In *Proceedings of the 4th IEEE International Conference on Pervasive Computing and Communications (PerCom 2006)*, pages 239–244, Pisa, Italy, March 2006. IEEE Computer Society Press.

[WL99] Jie Wu and Hailan Li. On Calculating Connected Dominating Set for Efficient Routing in Ad Hoc Wireless Networks. In *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM '99)*, pages 7–14, Seattle, WA, USA, August 1999. ACM Press.

[WP05] Roy Want and Trevor Pering. System Challenges for Ubiquitous & Pervasive Computing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 9–14, St. Louis, MO, USA, May 2005. ACM Press.

[WRvK$^+$08] Christine T. Whitman, Charles Reid, James von Klemperer, Josh Radoff, and Anthony Roy. New Songdo City – The Making of a New Green City. In *Proceedings of the CTBUH 8th World Congress*, Dubai, March 2008.

[XHE01] Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for Ad Hoc routing. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom '01)*, pages 70–84, Rome, Italy, July 2001. ACM Press.

[YC05] Jane Y. Yu and Peter H. J. Chong. A Survey on Clustering in Schemes for Mobile Ad Hoc Networks. *IEEE Communications Surveys and Tutorials*, 7(1):32–48, March 2005.

[YDIK92] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS '92)*, pages 614–621, Yokohama, Japan, June 1992. IEEE Computer Society Press.

[YDIK98] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, September-October 1998.

[YH96] Makoto Yokoo and Katsutoshi Hirayama. Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems. In *Proceedings of the 2nd International Conference on Multiagent Systems (ICMAS 1996)*, pages 401–408, Kyoto, Japan, December 1996. AAAI Press.

[Yok94] Makoto Yokoo. Weak-commitment Search for Solving Constraint Satisfaction Problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI '94, vol. 1)*, pages 313–318, Seattle, WA, USA, August 1994. AAAI Press.

[YP02] Jian Yang and Mike. Papazoglou. Web Component: A Substrate for Web Service Reuse and Composition. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE 2002)*, pages 21–36, Toronto, Canada, May 2002. Springer Berlin / Heidelberg.

[ZLH03] Yongguang Zhang, Wenke Lee, and Yi-An Huang. Intrusion Detection Techniques for Mobile Wireless Networks. *Springer Journal on Wireless Networks*, 9(5):545–556, September 2003.

[ZM91] Ying Zhang and Alan K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing (IPDPS '91)*, pages 394–397, Dallas, TX, USA, December 1991. IEEE Computer Society Press.

A

List of Abbreviations

ABT	Asynchronous Backtracking (an asynchronous, decentralized and complete algorithm which enables the concurrent configuration of components; presented by Yokoo et al. [YDIK98])
AD	Active Device (a device which actively calculates configurations in the centralized and hybrid configuration schemes presented here)
CG	Cluster Gateway (cluster nodes which are part of at least two clusters, so they can access neighboring clusters and forward information between clusters)
CH	Cluster Head (an exclusive node within a cluster which usually serves as a local coordinator within its cluster; in this thesis, the cluster heads are responsible for performing the configuration for their cluster members' components)
CM	Cluster Member (an ordinary node which represents a non-cluster head node without any inter-cluster links; in this thesis, these nodes remain passive during configuration, but only provide information about their locally available components)
CID	Cluster Index (an index used within our hybrid configuration scheme to enable a unique distinction of the different nodes within the cluster)
CLDC	Connected Limited Device Configuration Profile (a Java profile which defines the minimal configuration within a Java 2 ME runtime environment)
CSP	Constraint Satisfaction Problem (a well-known NP-complete problem; configuring a distributed application can be mapped to this problem, as Handte et al. [HBR05] have shown)
CoID	Component Identifier (an identifier which enables the unique identification of a component chosen within a configuration

	process within the application tree structure; extended version of an IID with additional information about the selected component)
DBT	Direct Backtracking (an advanced configuration scheme for centralized application configuration which is presented within this thesis)
DCSP	Distributed Constraint Satisfaction Problem (the completely distributed version of a Constraint Satisfaction Problem (CSP))
DDB	Dependency-Directed Backtracking (a centralized algorithm suitable to solve the configuration problem which avoids the undesired thrashing effect by storing a set of nogoods)
DyBT	Dynamic Backtracking (an advanced centralized algorithm presented by Ginsberg [Gin93] which is suitable to solve the configuration problem and outclasses most other approaches by removing thrashing completely without excessive waste of memory; however, requires the re-ordering of the involved variables)
DMAC	Distributed Mobility Aware Clustering (a completely distributed version of the GCA scheme, as introduced by Basagni et al. [Bas99])
FIFO	First In First Out (a standard cache replacement strategy which replaces the entries that resides for the longest time in the cache first)
GCA	Generalized Clustering Algorithm (a general clustering algorithm introduced by Basagni et al. [BCFJ97] which represents the foundation of many other clustering schemes, e.g. HCC or LID)
GDA	Greedy Distributed Assembler (a distributed configuration assembler introduced by Handte el al. [HBR05] in the PCOM system which performs in a greedy manner when selecting application components)
GPD	General Pareto Distribution (a statistical disribution which follows a power law)
GPS	Global Positioning System (a satellite-based global navigation system that provides location and time information in all weather and is typically used for outdoor navigation)
GUI	Graphical User Interface (a kind of user interface that enables users to interact with electronic devices with images rather than text commands.)
HCC	Highest Connectivity Clustering Scheme (a standard clustering scheme presented by Gerla et al. [GTCT95] which chooses those devices as cluster heads which have the highest number of direct neighbors)

ICT	Inter-Contact Time (the average time span a user device is disconnected between two communication sessions, i.e. the time interval over which devices are not in contact)
IID	Instance Identifier (an identifier which enables the unique identification of each dependency within the application tree structure, independent from the acutally chosen concrete component)
JVM	Java Virtual Machine (the part of the Java runtime environment (JRE) which is responsible for executing the bytecode of Java programs)
LFU	Least Frequently Used (a standard cache replacement strategy which replaces the least frequently used entries first)
LID	Lowest ID Clustering Scheme (a standard clustering scheme presented by Lin et al. [LG06] which selects nodes with lowest IDs within their environment as cluster heads)
LMS	Least Mean Squares Method (a standard algorithm which relies on the Steepest Descent Method to approximate a solution for the Least Mean Squares problem)
LRFU	Least Recently/Frequently Used (a cache replacement strategy which combines the properties of LFU and LRU)
LRU	Least Recently Used (a standard cache replacement strategy which replaces the least recently used entries first)
MANET	Mobile Ad-hoc Network (a self-configuring, infrastructure-less network of mobile devices connected by wireless links)
OSGi	Open Services Gateway initiative (an initiative which specifies a hardware-independent dynamic software platform that eases the modularization and management of aplications and their services)
P2P	Peer-to-Peer (a communication paradigm where the communication partners are considered to have equal rights regarding sending and receiving messages)
PAC	Partial Application Configuration (one of the main concepts used in this thesis, where parts of a complete configuration are stored in a repository for their future re-use, in order to reduce the arising configuration latencies)
PD	Passive Device (a device which is not actively participating in a configuration, but only provides its resource information to the Active Device (AD) it is mapped to)
PDA	Personal Digital Assistant (a compact, wearable computer which is typically used for the personal calendar, address and task planning)

RLF	Remove Largest First (a cache replacement strategy which removes the largest entries first from the cache)
RSF	Remove Smallest First (a cache replacement strategy which removes the smallest entries first from the cache)
SBJ	Synchronous Backjumping (an advanced synchronous and complete algorithm suited for centralized application configuration which keeps track of the reasons that led to an adaptation and reduces the number of steps that have to be taken within an adaptation)
SBT	Synchronous Backtracking (a synchronous, complete algorithm suited for centralized application configuration which executes a depth-first search in the application tree and performs consistency checks to reduce the execution time compared to exhaustive search)
ST	Session Time (the average time of a user device's connection session length, i.e. the interval until the session is terminated)
VC	Virtual Container – a concept presented in this thesis to emulate the resources of remote devices, in order to reduce the communication and latency overhead in centralized and hybrid configuration schemes
WLAN	Wireless Local Area Network (a local wireless network, which is usually refered to as the 802.11 standard that has been developed by the IEEE)
w.l.o.g.	without loss of generality
XML	eXtensible Markup Language (a standard markup language to illustrate hierarchically structured data)