Pervasive Computing Middleware

Gregor Schiele, Marcus Handte and Christian Becker

1 Introduction

Pervasive computing envisions applications that provide intuitive, seamless and distraction-free task support for their users. To do this, the applications combine and leverage the distinct functionality of a number of devices. Many of these devices are invisibly integrated into the environment. The devices are equipped with various sensors that enable them to perceive the state of the physical world. By means of wireless communication, the devices can share their perceptions and they can combine them to accurate and expressive models of their surroundings. The resulting models enable applications to reason about past, present and future states of their context and empower them to behave according to the expectations of the user. This ensures that they provide high-quality task support while putting only little cognitive load on users as they require only minimal manual input. To provide a truly seamless and distraction-free user experience, the applications can be executed

Gregor Schiele
Universität Mannheim, Germany, e-mail: gregor.schiele@uni-mannheim.de

Marcus Handte
Fraunhofer IAIS and Universität Bonn, Germany, e-mail: handte@cs.uni-bonn.de

Christian Becker
Universität Mannheim, Germany, e-mail: christian.becker@uni-mannheim.de

in a broad spectrum of vastly different environments. Thereby, the require only little manual configuration since they can autonomously adapt and optimize their execution depending on the capabilities of the environment. In many cases, changes to the environment are compensated with little impact on the support provided by the application. Failures are handled transparently and the capabilities of newly available devices are integrated on-the-fly.

Given this or similarly ambitious visions, pervasive applications are very attractive from a user's perspective. In essence, they simply promise to offer more sophisticated and more reliable task support for everyone, everywhere. From an application developers perspective, on the other hand, they are close to a nightmare come true: unprecedented device heterogeneity, unreliable wireless communication and uncertainty in sensor readings, unforeseeable execution environments that span the complete spectrum from static to highly dynamic, changing user requirements, fuzzy user preferences and the list goes on and on. As a result, even simple applications that process a small number of user inputs can often exhibit enormously large spaces of possible execution states and these spaces need to be considered by the application developer. Thus, without further precautions, the development of pervasive applications is a non-trivial, time-consuming and error-prone task. It is exactly at this point, where pervasive computing middleware can come to the rescue.

Over the last years, middleware for pervasive computing has come a long way. Given the diversity of pervasive applications and the harsh conditions of their execution environments, the design of a comprehensive and adequate middleware for pervasive applications is certainly a rewarding but also a challenging goal. The pursuit of this goal has lead to a multitude of different concepts which resulted in an even greater number of supportive mechanisms. It is due to this variety of concepts and mechanisms that the anatomy of existing middleware differs widely. As a conse-

quence, it is necessary to take a closer look at the influencing factors in middleware design in order to enable a meaningful in-depth discussion and comparison.

2 Design Considerations

In this section we discuss the three main influencing factors for the design of pervasive computing middleware: the organizational model of the middleware, the level of abstractions provided by the middleware, and the tasks supported by the middleware.

2.1 Organizational Model

The first key influential factor of middleware design is the organizational model of the execution environment. Given a certain organization, the middleware developer may introduce assumptions to improve or simplify middleware mechanisms and application development. In addition, the organizational model also introduces quantitative thresholds for important variables such as the maximum and typical number of devices or the frequency and types of fluctuations that can be expected. At the present time, there are two predominant organizational models, as shown in Figure 1. In the past, middleware developers primarily focused on either one:

- *Smart environment:* A large fraction of pervasive computing middleware systems such as (Garlan, Siewiorek, Smailagic, and Steenkiste, 2002), (Román and Campbell, 2000), (Paluska, Pham, Saif, Chau, and Ward, 2008), (Ponnekanti, Johanson, Kiciman, and Fox, 2003) are geared towards the organizational model of a smart environment. Thereby, a smart environment can be defined as a spatially limited area, e.g. a meeting room or an apartment, equipped with various sensors

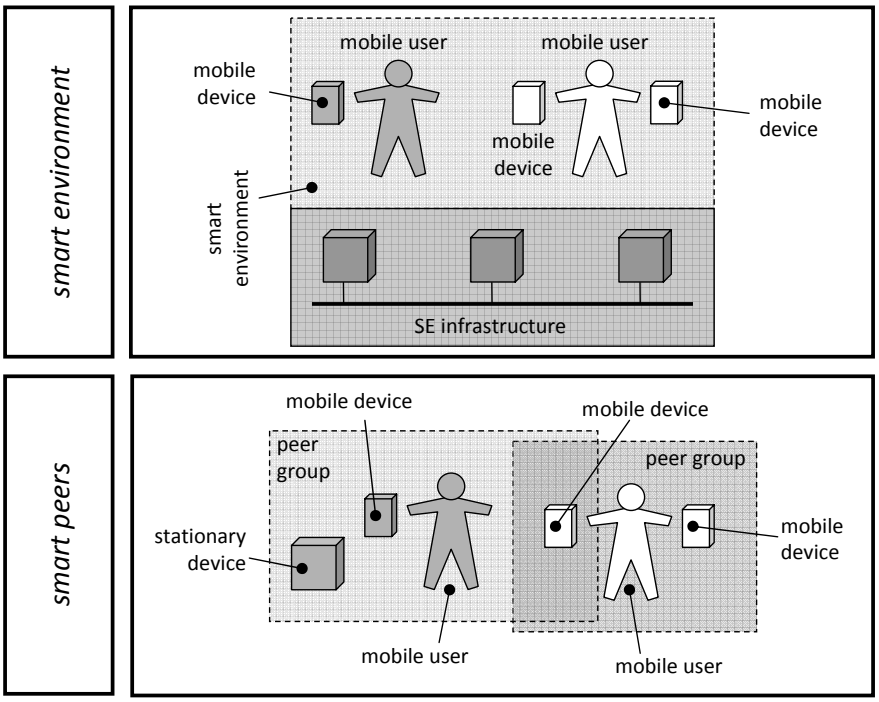

Fig. 1 Predominant organizational models

and actuators. As a result, many devices in smart environments are stationary. However, as users might be using mobile devices such as laptops or personal digital assistants (PDAs) as well, some devices must be dynamically integrated. Typically, the integration is performed depending on the physical location of the device which can be determined automatically by the available sensors or it can be manually configured through some user interface.

Due to the fact smart environments usually contain a number of stationary devices, middleware for smart environments relies typically on an stationary and rather powerful coordinating computer to provide basic services. Since the services are often an essential building block of the applications, the presence of the coordinating computer is required at all times and thus, its availability must be high. Clearly, demanding the availability of a powerful and highly available com-

puter may be problematic due to technical and economic constraints. However, if it can be guaranteed, a coordinating computer can significantly ease middleware and application development, since it provides a central point of management that can enforce policies, mediate communication, distribute information, etc.

- *Smart peers:* In addition to middleware for smart environments, there are also a good amount of middleware systems such as (Becker, Schiele, Gubbels, and Rothermel, 2003), (Becker, Handte, Schiele, and Rothermel, 2004), (Edwards, W.Newman, Sedivy, Smith, Balfanz, and Smetters, 2002), (Grimm, Davis, Lemar, MacBeth, Swanson, Anderson, Bershad, Borriello, Gribble, and Wetherall, 2004) that are based on the idea of dynamically formed collections of smart peers. In contrast to smart environments that view a pervasive system as a set of computers located in a certain area, smart peers postulate a people-centric as opposed to location-centric perspective on pervasive applications. The key idea is to understand a pervasive system as the collection of computers that surrounds a person independent from the physical location.

  As a result, the devices in such as system cannot rely on the continuous presence of any of the other devices. This prohibits centralized coordination on a powerful and highly available system. Instead, the smart peers must make use of suitable mechanisms to coordinate in a decentralized manner. In order to get the necessary locality required to keep the dynamic set of devices manageable, the middleware systems typically limit the size on the basis of spatial proximity. While this may seem more flexible at a first glance, the smart peer model is not without issues. Specifically, the management mechanisms tend to be more complicated and if not properly designed, they can also be less efficient.

2.2 Provided Level of Abstraction

Another influential factor of the middleware design is the provided level of abstraction. Given that pervasive computing middleware may provide a great variety of services, the level of abstraction can also differ widely, in general. However, from a high-level point of view, it is possible to identify three main levels and in most pervasive computing middleware, the services can be classified as providing either one:

- *Full transparency:* From an application developer's perspective, the ultimate middleware system should provide full transparency with respect to the addressed challenges. If full transparency can be provided, the application developer can simply ignore all related issues since they are handled by the middleware. Yet, providing transparency requires a thorough understanding of the problem domain and the application domain since the middleware designer must be able to identify a *completely* generic solution that provides satisfying results for *all* possible application scenarios. As a consequence, middleware services that provide full transparency are usually restricted to a particular type of application and to a very specific type of problem.
- *Configurable automation:* Given that full automation often imposes severe restrictions on the supported scenario, a frequently used trade off is configurable automation. Instead of defining a middleware layer that abstracts from all details, the middleware designer creates an abstraction that automates the task with the help of some configuration of the application developer. As a consequence, the application developer can forget about the intrinsic problems after deciding upon the basic configuration. In cases where the number of possible solutions is reasonable small, configurable automation can greatly simplify application develop-

ment. Yet, as the number of possible solutions grows, the configuration becomes more and more complicated until it outweighs the benefits of automation.

- *Simplified exposure:* In cases where configurable automation can no longer achieve satisfying results, pervasive computing middleware can strive for simplified exposure. So instead of offering mechanisms to automate a certain task, the middleware can restrict itself to the automated gathering of information needed to fulfill the task. Given that many tasks are not well-understood or have a huge solution space, this level of abstraction is sometimes the only reasonable option. Clearly, this might seem to be a very low level of abstraction. However, it can still be very valuable for two reasons. On the one hand, the gathering of the necessary information can be a complex task in itself. On the other hand, centralized gathering of information can often be more efficient, especially, in cases where multiple applications require the same type of information.

2.3 Supported Tasks

Besides from the organizational model and the level of abstractions, the most influential factor in pervasive computing middleware design is certainly the task supported by the system. Due to the great variety of challenges, the set of tasks is equally large. Yet, several tasks must also be handled by conventional systems and in many cases, the solutions applied by pervasive computing middleware follow traditional patterns. To avoid the repetition of main stream concepts, the in-depth discussion of pervasive computing middleware in this chapter is restricted to the following three main services that can be found in a majority of the existing system.

- *Spontaneous Interaction:* Independent from the underlying organizational model, pervasive applications need to interact spontaneously with an ever-changing set

of heterogeneous devices. To do this, applications not only need to communicate with each other but they also need to detect and monitor the available set of devices. As a consequence, even the most basic middleware systems provide functionality to enable this kind of spontaneous remote interaction. The details of the support, however, depend on the organizational model and the level of abstraction can vary accordingly.

- *Context Management:* Besides from interaction, a key difference between traditional and pervasive computing applications is that pervasive applications need to take the state of the physical world in to account. In addition to writing appropriate application logic, this bears many complex challenges. First and foremost, the relevant observations need to be made. Depending on the phenomena, this may require coordinated measurements of multiple distributed sensors and the fusion of several readings. During this process, the application has to deal with varying sensor availability and uncertainties in measurements. Moreover, the application might have to deal with various types of sensors resulting in different data representations, etc. As a result, server middleware systems aim at hiding these complexities by providing integrated context management services.
- *Application Adaptation:* The dynamics and heterogeneity of pervasive systems as well as the fact that pervasive applications are usually distributed raises an inherent need for adaptation. In order to function properly despite their highly dynamic execution environment, applications need to adapt to the overall system properties, the available capabilities that can be leveraged at a certain point in time and the context of their users. To simplify application adaptation, many middleware systems provide services that hide several aspects or that provide at least configurable automation. To do this, they introduce additional abstractions, e.g. to specify dependencies on the execution environment or to define optimization goals, and they provide supportive mechanisms.

It is noteworthy that only the most basic systems cover solely one of the services described above. In fact, most systems have building blocks for each of them. While some systems have provided basic support for all of these tasks right from the early stages of their development (e.g. (Garlan, Siewiorek, Smailagic, and Steenkiste, 2002), (Román and Campbell, 2000)), other systems have been extended with additional services over time (e.g. (Ponnekanti, Johanson, Kiciman, and Fox, 2003), (Becker, Handte, Schiele, and Rothermel, 2004)). Yet, due to differences in the underlying organizational model and the specifics of their application scenario, the resulting middleware solutions explore vastly different abstractions. The following sections, provide a more detailed discussion of the three middleware services described above and they relate the internal structure as well as the provided level of abstraction to the organizational model and the application scenario.

3 Spontaneous Interaction

Pervasive computing applications are typically distributed over a number of participating devices, e.g. a wall-mounted display for graphical output, a mobile device for user input, and a fixed infrastructure server for data storage. To support this, a suitable distribution platform is needed. We distinguish between three different requirements that such a platform or middleware must fulfill:

- *Ubiquitous communication and interaction:* First, the middleware should support ubiquitous and configuration-free interaction with other devices. This includes basic communication abilities, i.e., the ability to reach other devices and to communicate with them interoperably. In addition a suitable interaction abstraction is required to access remote devices. As an example, a service-based approach can be used.

- *Integration of heterogeneous devices:* Second, the middleware should allow to integrate a wide range of different devices into the system. This includes not only devices based on different hardware and software platforms – the classical goal of interoperability – but also devices with very different resources. As an example, one device might offer a lot of CPU power and memory but no graphical output. Another device might have very scarce computational and memory resources but a large graphical display attached to it.
- *Dynamic mediation:* Third, the middleware should provide a dynamic mediation service that allows to select suitable interaction partners at runtime. This service can be realized in different ways, e.g. by letting devices interact directly with each other or by using an additional system component to mediate between devices.

In the following we discuss each of these issues in more detail and analyze how different existing middleware systems for pervasive computing address them.

3.1 Ubiquitous Communication and Interaction

Enabling devices to cooperate with each other requires interoperability between these devices. Interoperability is a multilayer issue that needs to be guaranteed starting at the lowest layers of the networking stack up to the application layer. Here, a common understanding of the semantics of shared functionalities is required. To achieve this, different syntactic or semantic descriptions have been proposed. Middleware has to provide the necessary mechanisms to achieve interoperability on these different layers, or at least has to support applications in this task.

3.1.1 Interoperability

In general there are three main possibilities to solve the issue of interoperability as shown in Figure 2.

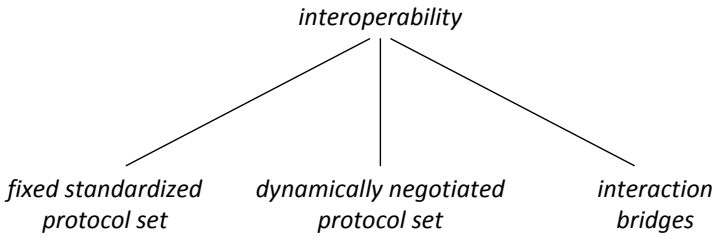

Fig. 2 Possibilities for interoperability

The first possibility is to standardize a fixed common set of protocols, technologies and data formats to interact with other devices. As long as a device uses this standard set of functionality, it will be able to communicate and interact with everybody else in the system. This approach is used by most classical middleware systems, e.g., CORBA (Object Management Group, 2008), Java RMI (Sun Microsystems, 2006) or DCOM (Eddon and Eddon, 1998). One prominent example for a pervasive computing middleware relying on a fixed set of standard protocols is UPnP (UPnP Forum, 2008). It defines a set of common communication protocols, e.g., HTTP and SOAP, to allow interaction between devices. Messages are encoded with XML. The exact format of each message must be defined between the communication partners before the interaction can take place. Clearly, the main challenge of this approach is the standardization process itself. Once a common standard is established, interaction becomes possible efficiently. However, it is difficult to change the standard protocols without updating or exchanging all devices. This is specifically problematic in pervasive computing systems, since (1) many devices are embed-

ded into everyday items and thus cannot be updated easily, and (2) devices have a relatively long life time if they are embedded into long living objects, like furniture.

The second, much more flexible possibility to achieve interoperability is to allow the middleware to negotiate the protocols used to interact with others at runtime. This additionally allows to switch between different communication technologies at runtime. As users and their devices move around in a pervasive computing system, the available communication technologies may vary over time. As an example, a wireless local area network (WLAN) might be available to users in a building but become unavailable when the user leaves it. By switching to another technology, the middleware is able to offer the maximum achievable communication performance. One middleware allowing such so-called *vertical handoffs* is the BASE middleware (Becker, Schiele, Gubbels, and Rothermel, 2003). Based on specific communication requirements that can be selected by an application, BASE negotiates the used protocols and technology with the communication partner dynamically. If during an interaction a technology becomes unavailable, it reselects another one automatically and continues the interaction. Another example for a middleware in this class is Jini (Waldo, 1999). Jini is a Java-based middleware that enables the utilization of distributed services in dynamic computer networks. It relies on mobile code to distribute service-specific proxies to clients. These proxies can include proprietary protocol stacks to access the remote service. This allows to use different protocols for accessing different services, and thus enables already deployed devices to interact with new devices that use newer protocols. However, Jini does not allow to switch between different protocols for the same service dynamically. A more sophisticated variant of this approach are dynamically reconfigurable middleware systems (e.g. (Ledoux, 1999), (Román, Kon, and Campbell, 1999a), (Becker and Geihs, 2000), (Blair, Coulson, Robin, and Papathomas, 2000), (Blair, Coulson, Andersen, Blair, Clarke, Costa, Duran-Limon, Fitzpatrick, Johnston, Moreira, Parla-

vantzas, and Saikoski, 2001), (Román, Kon, and Campbell, 2001)). These middleware systems are able to adapt their behavior at runtime to different requirements and protocols, e.g., by selecting which wireless communication technique should be used or how marshalling is done. To do so, these systems offer a reflection interface to allow the application to determine and change the current middleware behavior dynamically.

A third possible approach is to introduce interaction bridges into the system that map between different approaches. This approach is very well known on lower layers of the networking stack, e.g., to interconnect different networking technologies. In addition, it can be used on higher layers, too. As an example, the Unified Object Bus (Román and Campbell, 2001) used in Gaia provides interoperability between different component systems by defining a common interface to control component life-cycles.

3.1.2 Communication Abstractions

Another important issue when discussing ubiquitous interoperable interaction is the used communication abstraction. Conventional middleware offers various communication abstractions to develop distributed applications. These abstractions range from network-oriented message passing primitives and publish-subscribe abstractions (Eugster, Felber, Guerraoui, and Kermarrec, 2003) over associative memory paradigms such as tuple spaces (Carriero and Gelernter, 1986) to remote procedure calls (Birrell and Nelson, 1984) or their object-oriented counterpart of a remote method invocation (RMI). Since the latter extends the concept of a local procedure call - or a method invocation respectively - to distributed application development, it is commonly supported by mainstream middleware like CORBA (Object Man-

agement Group, 2008), Java RMI (Sun Microsystems, 2006), and Microsoft's .NET Remoting (Chappell, 2006).

Many pervasive computing middleware systems (e.g. (Román and Campbell, 2000), (Becker, Schiele, Gubbels, and Rothermel, 2003), (Waldo, 1999), (UPnP Forum, 2008)) have adopted RMI as their primary communication abstraction. The main advantage of RMI is that it is well known and generally well understood by developers. However, since a remote call requires to previously bind to a specific communication partner, RMI leads to rather tightly coupled communication partners. This may be problematic in highly dynamic environments with frequent disconnections between communication partners. Without further precautions, such a disconnection results in a communication failure that must be handled programmatically by the application developer. A possible solution is to store requests and responses in persistent queues as done, e.g., by the Rover Toolkit (Anthony D. Joseph, 1997). This way the communicating devices can be disconnected temporarily at any point in time and can continue their interaction once they are connected again. Yet, the application of such queued remote procedure calls is only suitable in cases where very high latencies can be tolerated and requires that the communication partners meet eventually to finish their interaction. If this is not the case, the binding between the communication partners must be dissolved and a new binding must be determined. Note that in case of a stateful interaction that has a shared interaction state at both partners, it is necessary to restore this state at the new communication partner before the interaction can continue. This can, e.g. be done using message logs (Handte, Schiele, Urbanski, and Becker, 2005b) and checkpointing (Handte, Schiele, Urbanski, and Becker, 2005b) (Grimm, Davis, Lemar, MacBeth, Swanson, Anderson, Bershad, Borriello, Gribble, and Wetherall, 2004).

To avoid such problems, communication abstractions with loosely coupled communication partners can be used. Here communication partners do not communicate

directly but are decoupled by a intermediate communication proxy, e.g., a message queue or tuple space. To initiate an interaction, a device sends a message to the proxy, which stores the message. If another device is interested in the message it can access the proxy and retrieve the message. This approach is e.g. used in iRos (Ponnekanti, Johanson, Kiciman, and Fox, 2003). Applications send event messages to a so-called *event heap*. The event heap stores events and allows other applications to retrieve them at a later time. Events are automatically removed from the event heap after a predefined time. Since there is no explicit binding between communication partners, the event sender does not need to know if its events are consumed by anyone or simply dropped. In addition, the recipient of the events can change during runtime without the sender knowing about it. Similarly to iRos, the one.world system (Grimm, Davis, Lemar, MacBeth, Swanson, Anderson, Bershad, Borriello, Gribble, and Wetherall, 2004) is based on loose coupling. It uses tuple spaces instead of an event heap. Each interaction results in an event data item being stored in the tuple space. An interested recipient can retrieve the data item, consuming it in the process, and can place an appropriate answer in the tuple space.

3.2 Integration of Heterogeneous Devices

To integrate heterogeneous devices into the system, the middleware must be able to be executed on each of them. This requires the middleware to be adapted to different hardware and software platforms, e.g. by introducing a small system abstraction layer that encapsulates the specific platform properties. On top of this abstraction layer, the system is realized in a platform independent way. Another possibility is to develop the system using a platform independent execution environment like e.g. the Java virtual machine. This way the middleware itself is platform independent and all platform dependent system parts are pushed down in the underlying execution envi-

ronment. Clearly this works only for target environments for which the used execution environment is available. The challenge of developing cross-platform systems that can be executed on different heterogeneous devices is not specific to pervasive computing. It exists for all cross-platform applications. Therefore we will not go into further detail on this issue here. The reader is referred to (Bishop and Horspool, 2006) for a more elaborate discussion.

In pervasive computing the middleware faces an additional challenge when trying to integrate different devices. The used devices may differ widely between the capabilities and resources they offer. A stationary server may have a lot of resources with respect to CPU power, memory, network bandwidth and energy. However, it may not offer any user input or output capabilities. On the other side, a mobile device might have very scarce resources but offer such user interface capabilities. The middleware must take this into account to be usable for all these devices. It should operate with few resources if necessary but utilize abundant resources if possible. On a mobile and thus energy-poor device it should offer energy saving mechanisms to achieve suitable device life times.

There are different approaches to create system software with minimal resource requirements that still maintains enough flexibility to make use of additional resources. The two extremes are (1) building multiple, yet compatible systems for different classes of devices, and (2) building modular systems with a minimal yet extensible core.

As an example for the first approach, the Object Management Group defined minimumCORBA (Object Management Group, 2002), a subset of CORBA intended for resource-poor devices. Another example is PalmORB (Román, Singhai, Carvalho, Hess, and Campbell, 1999b), a minimal middleware for personal digital assistants that only offers client-side functionality. Thus, the mobile device cannot offer services itself but only acts as a client for remote services.

The second approach is to provide a modular and extensible system architecture. One example for this is the Universally Interoperable Core (UIC) (Román, Kon, and Campbell, 2001). It provides a minimal reflective middleware that is configurable to integrate various resources. UIC can be used in static and dynamic variants. The static version allow to tailor the middleware to a given device but does not allow to change the middleware configuration at runtime. This is only possible in the dynamic variant. BASE (Becker, Schiele, Gubbels, and Rothermel, 2003) is another micro-broker-based middleware system for pervasive computing. It allows its minimal core to be extended by adding so-called plugins, each one abstracting a certain communication protocol or technology.

The Gaia project (Román and Campbell, 2000) combines both aforementioned approaches. For extremely resource-limited devices, they provide a specialized middleware called LegORB (Román, Mickunas, Kon, and Campbell, 2000). On other devices, they employ dynamicTAO (Román, Kon, and Campbell, 1999a), which relies on a small but dynamically extensible core.

3.3 Dynamic Mediation

Applications in pervasive computing are faced with unforeseeable and often highly dynamic execution environments. The communication partners that will be available at runtime are therefore often unknown at development time and may even change dynamically. To cope with that a suitable mediation service must be available that allows to decide dynamically with which devices to communicate and cooperate. Due to the dynamics of the execution environments, this mediation must be performed continuously. Mediation can be discussed for tightly or loosely coupled communication partners.

As described before, loosely coupled communication partners do not communicate directly but via an intermediate communication proxy. In this case, mediation is rather simple as it is implicitly done by the proxy storing the data. A message is simply delivered to every device that accesses the proxy and reads the stored data. It is the recipient device's responsibility to filter the messages that are interesting to it. Thus, mediation is done on the receiver side in this scenario.

Tightly coupled communication partners require a much more sophisticated communication mediation. In such systems, the partners are first bound to each other at runtime. After that they exchange data directly. If one of the partners becomes unavailable, the communication binding breaks and must be renewed with different partners. Typical examples for this approach are service-based systems. Therefore, the process of searching for suitable communication partners is typically referred to as *service discovery*. Existing service discovery approaches can be classified into *peer-based* (or server-less) and *mediator-based* (or server-based) discovery approaches.

3.3.1 Peer-based Discovery

In peer-based approaches (e.g., (UPnP Forum, 2008), (Nidd, 2001)), all nodes participate in the discovery process directly (see Figure 3). To find a service, a client broadcasts a discovery request to the whole or part of the network. Nodes offering a suitable service answer the request. Alternatively, service providers broadcast service announcements periodically, notifying others about their services. The provider can announce only its own services (UPnP Forum, 2008) or all services it knows of (Helal, Desai, Verma, and Choonhwa, 2003). Recipients of these announcements store them in a local cache. The cache can then be used to answer discovery requests of local (UPnP Forum, 2008) or both local and remote clients (Helal, Desai,

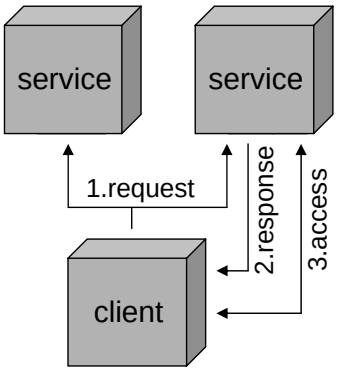

Fig. 3 Peers-based discovery

Verma, and Choonhwa, 2003), thereby enhancing the system's efficiency. The main advantage of peer-based approaches is their simplicity and flexibility. This makes them specifically suited for highly dynamic environments. On the downside, to detect new services continuously, devices have to send requests and/or announcements regularly, largely limiting the scalability of peer-based approaches due to the resulting communication overhead and resulting energy consumption.

A service discovery system that specifically tries to reduce this energy consumption is DEAPspace (Nidd, 2001). It uses synchronized time windows to broadcast service announcements. Between these windows, devices can be deactivated to save energy. Service descriptions are replicated on all devices and announced collectively. To lessen communication overhead, only one device sends its announcement in each cycle, preferably one with currently unknown services or much energy. Still, all service descriptions must be broadcast regularly, even if no client is interested in them. In addition, to enable new devices to join in, devices have to stay active for more than 50% of their time to ensure overlapping time windows.

3.3.2 Mediator-based Discovery

In contrast to peer-based discovery systems, mediator-based service discovery approaches (e.g., (Sun Microsystems, 2001), (Hodes, Czerwinski, Zhao, Joseph, and Katz, 2002), (uddi.org, 2004)) delegate service discovery to a number of special devices – the mediators – that manage a possibly distributed service registry on behalf of all devices (see Figure 4). The mediators can either be independent from each other (Sun Microsystems, 2001), can coordinate their entries with each other (uddi.org, 2004), or can form a hierarchy of mediators (Adjie-Winoto, Schwartz, Balakrishnan, and Lilley, 1999). To publish services, providers register their ser-

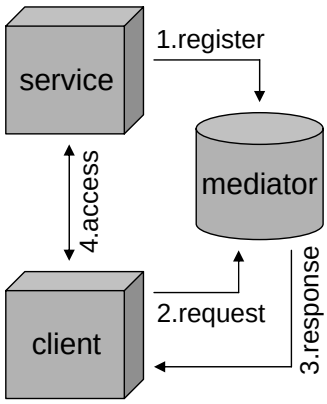

Fig. 4 Mediator-based Discovery

vices at the registry. Clients query it to detect services. As an optimization, clients can register their required services at the mediators to provide them with up-to-date information about new services via updates. Therefore, no broadcast is needed to detect or announce services, resulting in less communication overhead than with peer-based approaches. This makes mediator-based approaches especially suited to create highly scalable systems (e.g. (Hodes, Czerwinski, Zhao, Joseph, and Katz, 2002), (uddi.org, 2004)). Their main drawback is that without a mediator in the

vicinity, no discovery is possible. This is specifically problematic in highly dynamic environments without fixed infrastructure devices, because we cannot assume continuous connectivity to a specific device that could act as mediator.

SLP (Hodes, Czerwinski, Zhao, Joseph, and Katz, 2002) handles missing mediators by allowing nodes to switch to peer-based discovery if they do not find a mediator. Still, mediators are predefined. As they cannot sleep, battery-operated mediators will run out of energy quickly, forcing all other nodes to switch back to peer-based discovery. In addition, there may be more than one mediator in the vicinity, e.g. if two people come together with their devices.

Gaia (Román and Campbell, 2000) uses a heartbeat concept for service discovery. As long as a service is available in a given environment, it sends periodic heartbeat messages to the mediator, the so-called *PresenceService*. As a result, the PresenceService sends enter and leave messages on a service presence channel that interested clients can listen into.

An example for an energy-efficient mediator-based discovery system is SANDMAN (Schiele, Becker, and Rothermel, 2004). Similar to DEAPspace, SANDMAN saves energy by deactivating devices. However, it does so by grouping devices with similar movement patterns into dynamic clusters. Each cluster selects a mediator that takes over service discovery for its cluster. The mediator stays active and answers discovery requests from clients. All other devices power themselves down until they are needed. This approach is more complicated than DEAPspace but scales to larger systems and allows longer deactivation times.

4 Context Management

As applications adapt to changes in the environment, the relevant parameters of the environment need to be modeled and changes have to be propagated to interested

or affected applications in an appropriate manner. Context management denotes the task of context modeling, acquisition and provisioning where context relates to relevant environmental information of an entity. We will first take a look at context itself and established definitions. Thereafter we will discuss the three major responsibilities of context management, namely the acquisition and fusion, modeling and distribution, provisioning and access.

After more than a decade of research in context management a number of definitions exist. We do not want to present a survey but discuss two classical definitions.

Schilit, Adams and Want identify three important aspects of context (Schilit, Adams, and Want, 1994). These are

- where you are,
- who you are with, and
- what resources are nearby.

Thus, they classify context-aware systems according to their adaptation to the location of use, the collection of nearby people resembling the social context, and the available network and computing infrastructure. Also, the change in these three sets over time plays a crucial role to the adaptation of a context-aware application. A context-aware application must be capable of examine the available context and adapt accordingly.

Dey provided one of the first approaches that integrated a framework for adaptive applications, the context toolkit (Dey and Abowd, 2000), along with software engineering aspects. A number of applications has been investigated.

Dey defines context as any information that can be used to characterize the situation of an entity (Dey, 2001). Entities can be considered to be persons, places, or any object that is considered relevant to the interaction between a user and an application, including the user and applications themselves. This definition specifically addresses applications that are user centered.

Dey further defines a system to be context-aware if it uses context to provide relevant information and/or services to the user. Relevancy depends on the user's task. A further classification of applications leads to three classes of applications:

- Context-aware presentation: the application changes its appearance depending on context information. Examples are navigation systems that change the level of detail depending on the speed or multi-modal application that can select different output media, e.g., audio or video, based on context information, e.g., environmental noise.
- Automatic execution of a service: in this class, applications are triggered depending on context information. Examples are reminder services that can notify users based on configured patterns, e.g., during a shopping tour, or automatic execution of services in an automated home.
- Tagging of context to information for later retrieval: this class of applications allows to augment the daily environment of user with virtual artifacts. Messages that are linked to places or objects can become visible to users due to a distinct context, e.g., their identity, their activity, their role.

So far we have seen on a rather abstract view how applications can utilize context in order to change their behavior based on context. A yet open question is the management of context. Figure 5 shows a simple and generic model of context management.

The physical world is situated at the bottom of the model. Sensors – which may also be users entering data about our environment into an information system – feed the context model. The context model thus contains a more or less comprehensive abstraction of a part of our physical environment. This allows applications to reason about context and take actions. The interfaces between sensors and the context model as well as between context model and applications can be synchronous (pull)

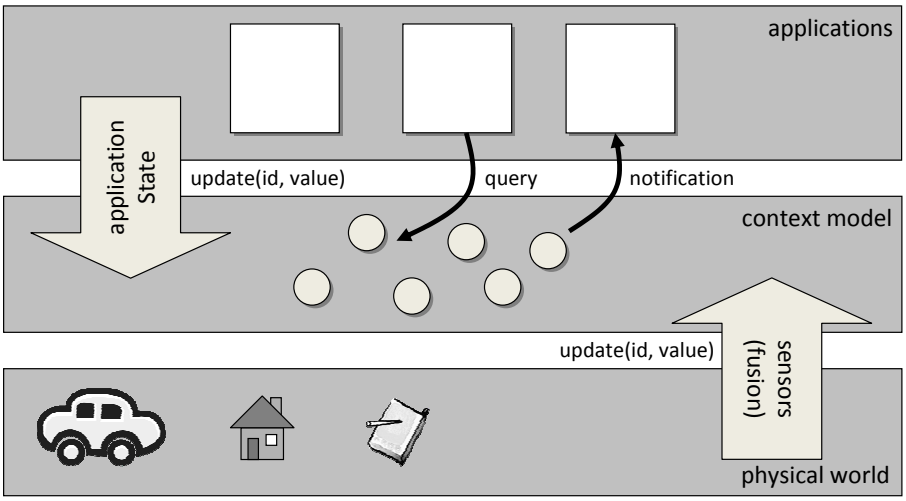

Fig. 5 Generic Model of Context Management

or asynchronous (push). Finally, the applications also can provide context information to the model, e.g., user preferences or tagging of context information.

We will briefly look at some relevant aspects of context management in the next subsections.

4.1 Acquisition and Fusion

Since context to a large degree is related to the physical world, we need sensors to capture the state. This immediately leads to two major problems that have to be handled:

- Accuracy: sensors measure and by doing so, they exhibit a deviation with respect to the actual value of the measured entity. Sensors typically specify the mean and maximum error when reading data. A GPS sensor refers to a sphere in space which depends on the number of satellites that can be seen and may be improved by some additional signal. Temperature readers only work with a distinct degree.

As a consequence, one can hardly handle context data as facts with discrete values. Intervals, possibly enriched with a distribution function of the values, can reflect this fact.

- Freshness: once a value is read, it ages. Depending on the characteristics of the entity which is measured, sensor readings show a rather long period where their value can be considered ”up to date”. Once we know how values in the physical world can change, e.g., the maximum speed of an object, we can estimate the maximum deviation over time. This can help us to reduce the rate in which sensors are read. Basically, this is a transformation from freshness into accuracy once we know the rate of change of the monitored entity.

Although we can only gather information about our environment with a distinct accuracy, there are methods to gain a more accurate view. First, combinations of sensors can be used. Using different technology can help to reduce errors. Even the same sensor technology can help to gain more accurate sensor readings, e.g., when visuals are taken from different perspectives.

In contrast to redundant or multiple readings from the same or similar sensor technology one can use additional information in order to increase the accuracy or reduce the error probability. If, for example, an indoor positioning system returns two rooms as possible position additional information can help to narrow down the position. Additional information could be login information at a computer or key stroke activity of an instant messenger. Combining different sensor information in order to reduce errors or deduce other context information is called *sensor fusion*. Higher context information based on low level sensors is a typical task of sensor fusion. As an example consider a meeting room that allows readings of light condition, temperature, ambient noise, etc, e.g., via the ContextCube (Bauer, Becker, Hähner, and Schiele, 2003). A situation with closed blinds, artificial light (can be determined by the wave length), and one person talking can be reasoned to be a

presentation. Rules for sensor fusion and reasoning require adequate modeling of sensor information and the relation between different sensors. We will talk briefly about context modeling in the next section.

4.2 Modeling and Distribution

Context acquisition can be a costly task. Besides simple and cheap sensors, context can also be gathered by expensive methods, e.g., surveying. In order to mitigate the resulting costs, sharing the costs among a number of applications can be helpful. However, if a single institution cannot handle the costs sharing context information across institutions can pave the way. A popular example is map data. Although it is quite expensive to gather, the scale of its applications from classical maps to modern navigation systems allows achievable costs for the individual user. A prerequisite for sharing is standardization. This can be done by standardized interfaces: the internal representation of the data is decoupled by a a set of functions from the using applications. This eases using data being distributed across a number of services. However, this still requires that the retrieved information can be interpreted across a number of applications. This leads to the second prerequisite: standardized data. Applications need to interpret the data – obviously they need type information. There is a plethora of different context modeling schemes which can only covered by a brief discussion here.

Simple approaches use name value pairs. Clearly, this requires that every application knows the names of the context concepts and interprets the associated data the same. Name value pairs do not allow to relate concepts to each other. Thus, specialization of existing concepts is not possible as well as modeling hierarchies, e.g., inclusion of locations. Object-oriented modeling techniques allow to model specialization of concepts and thus extension of existing hierarchies. Additional relations

between the concepts cannot be handled and the modeling is restricted to classes. Ontologies offer additional means for modeling and allow reasoning over context. This is, however, a complex field on its own. The interested reader is referred to (Strang and Linnhoff-Popien, 2004).

Other aspects of distribution is the placement of data in a set of distributed services. This can help to balance requests from services and allow scalability. This requires directory services that forward the requests to the servers hosting the information. Clearly, such directories should not result in performance bottlenecks or single points of failure themselves.

4.3 Provisioning and Access

As discussed above, the interface to a context model – realized as a context service – is crucial in order to provide suitable abstractions for applications. Dey as well as Bauer, Becker and Rothermel identify three classes of context that are relevant for context access:

- identity: as in regular databases, context information can be identified by a unique key. This allows to query for context information in a direct way.
- location: since context information is related to our physical environment, many queries will relate to objects and their spatial relation, i.e., the nearest objects (nearest neighbor queries), objects residing in a distinct spatial area (range queries), and queries for an objects position (position queries).
- time: past context can be stored and retrieved later allowing for, e.g., life-logging applications. Also, some context, e.g., weather or traffic jams, can be predicted. Thus, a context model should support queries for the past, current context and the future.

Not every context model has to support all these queries. Also, the kind of queries can differ. Applications can poll context by issuing queries to the context model. This is similar to classical databases. Since many applications react on context changes, it can ease the development of applications when they register only for a notification that is issued, when the context changes.

5 Application Adaptation

In general, the distraction-free and intuitive task support envisioned by pervasive computing cannot be achieved by a single device alone. As Mark Weiser pointed out already (Weiser, 1991), the real power of pervasive computing emerges from the interaction of all of them. Given that only very simple tasks can be supported by one device, thorough task support requires some form of distributed coordination to integrate the unique capabilities of multiple devices.

In pervasive computing environments, the integrating coordination of devices is specifically challenging for several reasons. First and foremost, due to the embedding of devices into everyday objects, it is pretty unlikely that different environments will exhibit similar sets of devices. Secondly, due to mobility, failures and continuous evolution, the set of devices that is available in a certain environment is changing over time and many of the changes cannot be predicted reliably. Finally, even if the set of devices is known in advance and does not change, different users may require or prefer the utilization of different devices for the same task in the same environment.

As a consequence, the coordination of the devices cannot be done statically in advance. In order to support diverse settings and to provide distraction-free task support for different users, the coordination must be done on the basis of the target environment and user. Furthermore, since the environment may change at any

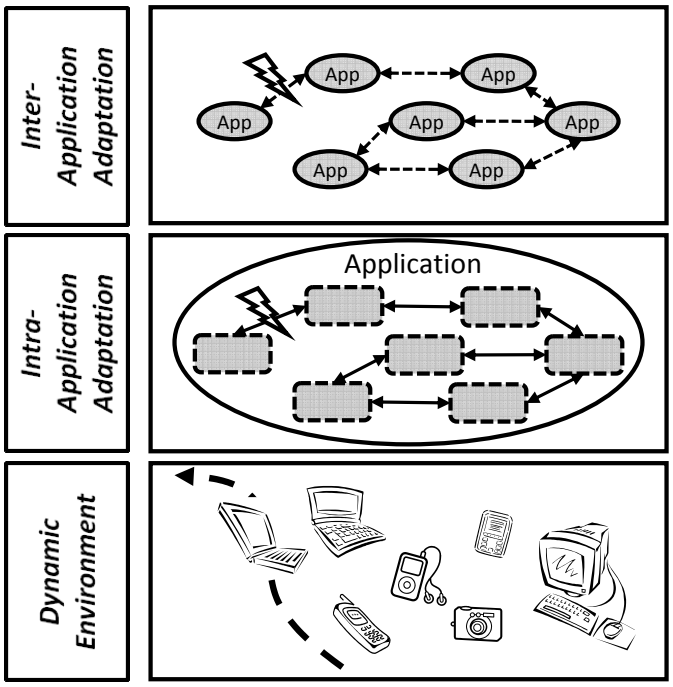

Fig. 6 Application Adaptation

given point in time, the coordination must be able to adapt to many changes. Without appropriate coordination support at the middleware layer, coordination must be handled by the application layer. Besides from complicating the task of the application developer, this approach has the a number of significant drawbacks. For instance, if some parts of the coordination logic cannot or shall not be handled by the application, e.g. to support customization for different users, it is hard to provide an application-overarching customization interface. Similarly, if multiple applications are executed at the same time, their coordination logic might interact in unforeseeable and undesirable ways.

In the past, researchers have proposed two alternative coordination approaches which are depicted in Figure 6. Both have been supported by corresponding middleware systems and in principle, they are complementary. However, they have

not been integrated in a single unifying system so far. The first approach, inter-application adaptation, coordinates the execution of several non-distributed applications (Garlan, Siewiorek, Smailagic, and Steenkiste, 2002), (Ponnekanti, Johanson, Kiciman, and Fox, 2003). The second approach, intra-application adaptation, strives for coordination of a single application that is distributed across several devices (Coen, Phillips, Warshawsky, Weisman, Peters, and Finin, 1999), (Becker, Handte, Schiele, and Rothermel, 2004), (Roman, Hess, Cerqueira, Ranganathan, Campbell, and Nahrstedt, 2002), (Paluska, Pham, Saif, Chau, and Ward, 2008). The following sections provide a detailed look at each of the two approaches and discusses exemplary middleware services to support the arising tasks.

5.1 Inter-Application Adaptation

The goal of inter-application adaptation is to provide thorough task support by coordinating the execution of multiple applications across a set of devices. Usually, the applications themselves are not distributed and they are not aware of each other. This enables the utilization of arbitrary compositions, however, this does not necessarily induce that the applications do not interact with each other at all. They may interact either through some form of mediated communication or they may rely on classical mechanisms such as files.

A charming benefit of inter-application adaptation is that it enables the reuse of traditional applications in pervasive computing environments with minor or even no modifications. Considering the amount and the value of available applications this argument is quite convincing. In addition, inter-application adaptation can shield the application developer completely from all aspects of distribution since the coordination does not require application support. Last but not least, as applications do not

interact with each other directly, intra-application adaptation facilitates robustness and extensibility.

To provide support for inter-application adaptation, middleware can take care of multiple tasks. First, the middleware can determine the set of applications that shall be executed. Upon execution, it can supply the applications with user-specific session state. Secondly, the middleware can detect changes and adapt the set of applications accordingly. If one application shall be replaced with another one, it can provide transparent transcoding services for user data. Finally, the middleware can also provide services to facilitate the interaction of applications.

Clearly, not all existing middleware systems that target inter-application adaptation provide all services. The following briefly describes how two prominent middleware systems, namely Aura (Garlan, Siewiorek, Smailagic, and Steenkiste, 2002) and iRos (Ponnekanti, Johanson, Kiciman, and Fox, 2003), support inter-application adaptation. To understand their mechanisms it is noteworthy to point out that both middleware systems are targeted at smart environments and thus, they assume that some configuration can be done manually and that it is possible to implement centralized services on a device with guaranteed high availability.

To manage application compositions automatically, Aura introduces a task abstraction. As indicated by the name, a task actually represents a user task including the relevant data. Examples for tasks are the preparation of a certain document or presentation. Tasks can be specified by the user through some graphical user interface. When a user wants to start or continue a specified task, Aura automatically maps the task to a certain set of applications on a set of the currently available devices. This mapping is done automatically by the PRISM component of Aura. To do this, the PRISM component relies on a so-called environment manager which is a centralized component that maintains a repository of the available applications. In order to supply the applications with the user data and in order to deal with discon-

nections, Aura relies on the advanced capabilities of the Coda distributed file system (Satyanarayanan, 2002). Thereby, Aura can also adapt the fidelity of the data using Odyssey (Noble and Satyanarayanan, 1999) in order to optimize the execution despite fluctuating network quality. However, interaction between applications that are supporting the same task is not supported by any of the services of Aura.

While iRos does not support an automated mapping between user tasks and applications, it provides services that are specifically targeted at interaction support. To prevent a tight coupling of applications, iRos introduces a so-called event heap (Johanson and Fox, 2002). The event-heap allows applications to post and receive notifications. The notifications can be generated and processed by applications that have been specifically developed with the event heap in mind or by scripts that extend existing applications that are not aware of iRos. A presentation application, for example, may generate an event whenever the slide changes. Some other application may then use the corresponding notifications to dim the light in the room as long as the presentation is running. In addition to pure event distribution, the event heap also stores events for a limited amount of time. This reduces timing constraints between applications and it can be used to recover from application failures. For example, by processing the recent events stored in the event heap, a script may be able to restore the internal state of an application after a failure. Just like Aura, iRos also takes care of data management but it uses a slightly different approach. Instead of reusing an existing distributed filesystem, iRos introduces a so-called data heap. Besides from persistent data storage, the data heap can also provide automated type conversions on the basis of the Paths system (Kiciman and Fox, 2000). Thus, instead of selecting an application on the basis of the data type of the input files, the user can simply select an application that provides the desired feature set.

Despite its undeniable benefits, inter-application adaptation is not without issues. On the one hand, mediated communication or indirect interaction by sharing

session data can ensure that individual applications are not tightly coupled. On the other hand, however, it only supports weak forms of coordination between applications well. In cases where stronger forms are required, e.g. if a certain action should take place shortly after a certain event occurred, additional middleware support can greatly simplify application development. Similarly, if a single application must be distributed to fulfill its task, e.g. because it requires resources that cannot be provided by a single device, the application does not benefit from inter-application adaptation. Mitigating this is the primary goal of intra-application adaptation.

5.2 Intra-Application Adaptation

The goal of inter-application adaptation is to provide thorough task support by coordinating the execution of a distributed application across a set of devices. The level of integration that can be achieved by this approach exceeds the achievable level of multiple independent applications. Providing thorough task support without coordinating the actions of multiple devices is often simply not possible. For example, a collaboration application that runs on several mobile devices might need to share data in a tightly controlled way.

Unlike inter-application adaptation which can benefit on an unlimited degree of compositional flexibility, a distributed application usually requires the availability of certain functionality. For example, in order to print a document, a word processing application requires a printer. In addition to minimum requirements an application may also want to achieve certain optimization goals that may also depend on user preferences, e.g. use the nearest or the cheapest printer. As a consequence, the flexibility of the composition of functionalities is usually limited by application requirements and optimization criteria.

Depending on the complexity of the application, the configuration and runtime adaptation of the composition can be quite complicated. As a result, a primary goal of many pervasive computing middleware systems that support intra-application adaptation is the automation or at least the simplification of these aspects. A noteworthy exception is the approach taken by Speakeasy (Edwards, W.Newman, Sedivy, Smith, Balfanz, and Smetters, 2002). Instead of automating the composition, Speakeasy proposes the manual composition of a distributed application from components that can be recombined without restrictions. Thus, a user can create compositions that have not been foreseen at development time. However, manual composition is hardly viable for applications that consist of a larger number of components or for environments that exhibit a higher degree of dynamics.

The system-supported configuration and runtime adaptation of a composition requires a model of the application that captures the individual building blocks, their dependencies on each other and the dependencies on the environment. The models that are applied in practice can be classified depending on the supported degree of flexibility:

At the lowest level, the model contains a static set of building blocks with static dependencies. As a consequence, a valid configuration can be formed on a set of devices that fulfills the requirements and runs the specified building blocks of the application. This approach is taken by the MetaGlue (Coen, Phillips, Warshawsky, Weisman, Peters, and Finin, 1999) agent platform, for example. Although, the approach results in simple applications that must only deal with a fixed and know set of code, the lack of flexibility can be a significant short-coming in practical scenarios. One reason for this is that the approach requires devices to run an application-defined building block. In many scenarios this requires that the devices are capable and willing to install additional code or it simply restricts the set of possible devices to those that are equipped with the code already.

In order to mitigate this, it is possible to refrain from restricting the building block to a specific implementation. In order to ensure the correctness of the application despite the unknown implementation, it is necessary to model the required building blocks using a detailed functional and non-functional description. As this description must be provided by the application developer, this approach increases the modeling effort. However, in return, it becomes possible to use alternative implementations of the same functionality without risking undesirable effects on the application behavior. In addition to broadening the number of target devices, this indirect specification also facilitates the evolution of individual building blocks. Yet, having a static set of building blocks may be limiting in cases where one building block can be replaced by a combination of building blocks. The same holds true in cases where the number of building blocks cannot be specified in advance, e.g. because it depends on the environment.

To support such scenarios, it is possible to introduce flexible dependencies into the application model and several middleware systems follow this approach. The exact implementation, however, differs depending on the middleware. The Gaia system (Román and Campbell, 2000), for example, relies on an application model that allows the utilization of a varying number of components to display or perceive some information. This can be used to adapt the output of an application to the number of users or to the number of available screens. The PCOM component system (Becker, Handte, Schiele, and Rothermel, 2004) attaches dependencies to each component and requires that all recursively occurring dependencies are satisfied by another component. As a result, applications in PCOM are essentially trees of components that are composed along their dependencies. The system proposed in (Paluska, Pham, Saif, Chau, and Ward, 2008) also relies on recursion. However, instead of attaching dependencies to components, the authors introduce an additional level of indirection called Goals (Saif, Pham, Paluska, Waterman, Terman,

and Ward, 2003). Goals represent a specific user goal that can be decomposed automatically into a set of simpler sub Goals. After the decomposition has succeeded, they are mapped to components using Techniques. A Technique essentially resembles a script that defines how the components need to be connected and thus, it is possible to support more complex composition structures than in PCOM. Although, a flexible dependency specification is appealing at a first glance, it is noteworthy that the resulting configuration problems are often quite complex which may limit the scalability of such approaches.

On the basis of the application model, pervasive computing middleware can then provide services to automate the configuration and runtime adaptation. The canonical approach is to construct a configuration that satisfies the requirements defined by the application model. The complexity of this depends heavly on the underlying problem and on the completeness of construction procedure. If the construction systematically enumerates all solutions, it can often exhibit an exponential runtime (Paluska, Pham, Saif, Chau, and Ward, 2008), (Handte, Becker, and Rothermel, 2005a). To mitigate this, it is possible to tradeoff completeness for increased performance (Becker, Handte, Schiele, and Rothermel, 2004). Alternatively, if the environment is mostly static, the configuration can be computed once and reused multiple times (Ranganathan, Chetan, Al-Muhtadi, Campbell, and Mickunas, 2005).

During the construction, additional optimization goals can be introduced in the form of some utility function (Paluska, Pham, Saif, Chau, and Ward, 2008). The concrete implementation of the utility function may be specified by the user or the application developer. In the case of runtime adaptation, the utility function may encompass cost components (Handte, Becker, Schiele, Herrmann, and Rothermel, 2007). Usually, the cost components model the effort for changing the running configuration which is dependent on the application model and the adaptation mechanisms. During the construction of the configuration, the utility value is usually

optimized heuristically as a complete optimization would lead to considerable performance penalties.

Middleware systems that are based on the organizational model of a smart environment can assume the permanent presence of a powerful computer that may be used to compute configurations. As a consequence, these systems usually perform the configuration centralized (Paluska, Pham, Saif, Chau, and Ward, 2008). In peer-based systems, the availability of a single powerful device cannot be guaranteed. As a result, the computer system that is used for configuration is either selected dynamically (Schuhmann, Herrmann, and Rothermel, 2008) or the configuration process is distributed among the peers (Handte, Becker, and Rothermel, 2005a).

6 Conclusion

The development of pervasive applications that provide seamless, intuitive and distraction-free task support is a challenging task. Pervasive computing middleware can simplify application development by providing a set of supportive services. The internal structure of the services is heavily influenced by the overall organization of the pervasive system and the targeted level of support. Three main services that can be provided by pervasive computing middleware are support for spontaneous interaction, support for context management and support for application adaptation.

As basis for the cooperation of heterogeneous devices found in pervasive systems, middleware support for spontaneous interaction alleviates the handling of low-level communication issues. Beyond communication, support for context management ensures that application developers do not have to deal with the intrinsics of gathering information from a multitude of unreliable sensors. Finally, support for application adaptation simplifies the task of coordinating a changing set of devices to provide a seamless and distraction-free user experience.

Although these services can greatly simplify many important tasks of the developer, the development of intuitive task support remains a significant challenge. As new scenarios and devices continue to emerge, application developers have to invent and explore novel ways of supporting user tasks with the available technology. Undoubtedly, this will result in additional challenges that may at some point spawn the development of further middleware systems and services.

References

Adjie-Winoto W, Schwartz E, Balakrishnan H, Lilley J (1999) The design and implementation of an intentional naming system. In: Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP99), ACM

Anthony D Joseph MFK Joshua A Tauber (1997) Mobile computing with the rover toolkit. IEEE Transactions on Computers 46(3):337–352

Bauer M, Becker C, Hähner J, Schiele G (2003) ContextCube - providing context information ubiquitously. In: Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003)

Becker C, Geihs K (2000) Generic QoS-support for CORBA. In: Proceedings of 5th IEEE Symposium on Computers and Communications (ISCC'2000)

Becker C, Schiele G, Gubbels H, Rothermel K (2003) Base – a micro-broker-based middleware for pervasive computing. In: Proceedings of the IEEE international conference on Pervasive Computing and Communications (PerCom), URL citeseer.nj.nec.com/550575.html

Becker C, Handte M, Schiele G, Rothermel K (2004) Pcom - a component system for adaptive pervasive computing applications. In: 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 04)

Birrell AD, Nelson BJ (1984) Implementing remote procedure calls. ACM Trans Comput Syst 2(1):39–59, DOI http://doi.acm.org/10.1145/2080.357392

Bishop J, Horspool N (2006) Cross-platform development: Software that lasts. In: SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop, IEEE Computer Society, Washington, DC, USA, pp 119–122, DOI http://dx.doi.org/10.1109/SEW.2006.14

Blair G, Coulson G, Andersen A, Blair L, Clarke M, Costa F, Duran-Limon H, Fitzpatrick T, Johnston L, Moreira R, Parlavantzas N, Saikoski K (2001) The design and implementation of Open ORB version 2. IEEE Distributed Systems Online Journal 2(6)

Blair GS, Coulson G, Robin P, Papathomas M (2000) An architecture for next generation middleware. In: Proceedings of Middleware 2000

Carriero N, Gelernter D (1986) The s/net's linda kernel. ACM Trans Comput Syst 4(2):110–129, DOI http://doi.acm.org/10.1145/214419.214420

Chappell D (2006) Understanding .NET (2nd Edition). Addison-Wesley Professional

Coen M, Phillips B, Warshawsky N, Weisman L, Peters S, Finin P (1999) Meeting the computational needs of intelligent environments: The metaglue system. In: 1st International Workshop on Managing Interactions in Smart Environments (MANSE'99), pp 201–212

Dey AK (2001) Understanding and using context. Personal Ubiquitous Comput 5(1):4–7, DOI http://dx.doi.org/10.1007/s007790170019

Dey AK, Abowd GD (2000) The context toolkit: Aiding the development of context-aware applications. In: Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing

Eddon G, Eddon H (1998) Inside Distributed Com. Microsoft Press

Edwards KW, WNewman M, Sedivy JZ, Smith TF, Balfanz D, Smetters DK (2002) Using speakeasy for ad hoc peer-to-peer collaboration. In: 2002 ACM Conference on Computer Supported Cooperative Work, pp 256–265

Eugster PT, Felber PA, Guerraoui R, Kermarrec AM (2003) The many faces of publish/subscribe. ACM Comput Surv 35(2):114–131, DOI http://doi.acm.org/10.1145/857076.857078

Garlan D, Siewiorek D, Smailagic A, Steenkiste P (2002) Towards distraction-free pervasive computing. IEEE Pervasive Computing 1(2):22–31

Grimm R, Davis J, Lemar E, MacBeth A, Swanson S, Anderson T, Bershad B, Borriello G, Gribble S, Wetherall D (2004) System support for pervasive applications. ACM Transactions on Computer Systems 22(4):421–486

Handte M, Becker C, Rothermel K (2005a) Peer-based automatic configuration of pervasive applications. Journal on Pervasive Computing and Communications pp 251–264

Handte M, Schiele G, Urbanski S, Becker C (2005b) Adaptation support for stateful components in PCOM. In: Proceedings of Pervasive 2005, Workshop on Software Architectures for Self-Organization: Beyond Ad-Hoc Networking

Handte M, Becker C, Schiele G, Herrmann K, Rothermel K (2007) Automatic reactive adaptation of pervasive applications. In: IEEE International Conference on Pervasive Services (ICPS '07)

Helal S, Desai N, Verma V, Choonhwa L (2003) Konark - a service discovery and delivery protocol for ad-hoc networks. In: Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC 2003), vol 3, pp 2107–2113

Hodes TD, Czerwinski SE, Zhao BY, Joseph AD, Katz RH (2002) An architecture for secure wide-area service discovery. Wirel Netw 8(2/3):213–230, DOI http://dx.doi.org/10.1023/A:1013772027164

Johanson B, Fox A (2002) The event heap: A coordination infrastructure for interactive workspaces. In: 4th IEEE Workshop on Mobile Computing Systems and Applications, pp 83–93

Kiciman E, Fox A (2000) Using dynamic mediation to integrate cots entities in a ubiquitous computing environment. In: 2nd International Symposium on Handheld and Ubiquitous Computing, pp 211–226

Ledoux T (1999) OpenCorba: A reflective open broker. In: Proceedings of the 2nd International Conference on Reflection (Reflection'99), pp 197–214

Nidd M (2001) Service discovery in DEAPspace. IEEE Personal Communications 8(4):39–45

Noble B, Satyanarayanan M (1999) Experience with adaptive mobile applications in odyssey. Mobile Networks and Applications 4(4):245–254

Object Management Group (2002) Minimum CORBA specification, revision 1.0

Object Management Group (2008) Common object request broker architecture (corba/iiop), revision 3.1. online publication, http://www.omg.org/spec/CORBA/3.1/

Paluska J, Pham H, Saif U, Chau G, Ward S (2008) Structured decomposition of adaptive applications. In: 6th Annual IEEE International Conference on Pervasive Computing and Communications

Ponnekanti SR, Johanson B, Kiciman E, Fox A (2003) Portability, extensibility and robustness in iros. In: Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PERCOM 2003)

Ranganathan A, Chetan S, Al-Muhtadi J, Campbell R, Mickunas D (2005) Olympus: A high-level programming model for pervasive computing environments. In: 3rd IEEE International Conference on Pervasive Computing and Communications, pp 7–16

Román M, Campbell RH (2000) GAIA: Enabling active spaces. In: Proceedings of the 9th ACM SIGOPS European Workshop

Román M, Campbell RH (2001) Unified object bus: Providing support for dynamic management of heterogeneous components. Technical Report UIUCDCS-R-2001-2222 UILU-ENG-2001-1729, Universiy of Illinois at Urbana-Champaign

Román M, Kon F, Campbell RH (1999a) Design and implementation of runtime reflection in communication middleware: The dynamictao case. In: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshops, Workshop on Electronic Commerce and Web-Based Applications, pp 122–127

Román M, Singhai A, Carvalho D, Hess C, Campbell R (1999b) Integrating PDAs into distributed systems: 2K and PalmORB. In: Proceedings of the International Symposium on Handheld and Ubiquitous Computing (HUC'99)

Román M, Mickunas D, Kon F, Campbell RH (2000) Legorb and ubiquitous corba. In: Proceedings of the IFIP/ACM Middleware'2000 Workshop on Reflective Middleware

Román M, Kon F, Campbell RH (2001) Reflective middleware: From your desk to your hand. IEEE Distributed Systems Online Journal, Special Issue on Reflective Middleware

Roman M, Hess C, Cerqueira R, Ranganathan A, Campbell R, Nahrstedt K (2002) Gaia: A middleware infrastructure to enable active spaces. IEEE Pervasive Computing 1(4):74–83

Saif U, Pham H, Paluska J, Waterman J, Terman C, Ward S (2003) A case for goal-oriented programming semantics. In: System Support for Ubiquitous Computing Workshop, 5th Annual Conference on Ubiquitous Computing

Satyanarayanan M (2002) The evolution of coda. ACM Transactions on Computer Systems 20(2):85–124

Schiele G, Becker C, Rothermel K (2004) Energy-efficient cluster-based service discovery. In: 11th ACM SIGOPS European Workshop, pp 20–22

Schilit B, Adams N, Want R (1994) Context-aware computing applications. In: Proceedings of the Workshop on Mobile Computing Systems and Applications, pp 85–90

Schuhmann S, Herrmann K, Rothermel K (2008) A framework for adapting the distribution of automatic application configuration. In: 2008 ACM International Conference on Pervasive Services (ICPS '08), pp 85–124

Strang T, Linnhoff-Popien C (2004) A context modeling survey. In: First International Workshop on Advanced Context Modeling, Reasoning And Management (UbiComp 2004)

Sun Microsystems (2001) Jini technology core platform specification, version 1.2. online publication

Sun Microsystems (2006) Jdk6 remote method invocation (rmi) - related apis and developer guides. online publication, http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html

uddiorg (2004) UDDI spec technical committee draft, version 3.0.2. online publication, http://uddi.org/pubs/uddi_v3.htm

UPnP Forum (2008) Universal plug and play device architecture, version 1.0, document revision date 24 april 2008. online publication, http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0-20080424.pdf

Waldo J (1999) The jini architecture for network-centric computing. Communications of the ACM 42(7):76–82

Weiser M (1991) The computer for the 21st century. Scientific American 265(3):66–75