IEEE distributed systems ONLINE
Expert-authored articles and resources

IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922 © 2006 Published by the IEEE Computer Society
Vol. 7, No. 4; April 2006

Middleware: Context Management in Heterogeneous, Evolving Ubiquitous Environments

Richardo Couto Antunes de Rocha, *Pontifícia Universidade Católica do Rio de Janeiro*
Markus Endler, *Pontifícia Universidade Católica do Rio de Janeiro*

Design strategies and a flexible, adaptable middleware architecture for resource-limited, evolving systems.

Mobile computing environments are characterized by heterogeneity—systems consisting of different device types, operating systems, network interfaces, and communication protocols. Such heterogeneity calls for middleware that can adapt to different execution contexts, hide heterogeneity from applications, and transparently and dynamically switch between network and sensor technologies.

Additionally, middleware for context-aware systems must keep a *context model* (a model of their environment), taking into account several aspects of the environment. The more complex and heterogeneous an execution environment is, the more complicated its underlying context model. Moreover, because systems can evolve, context management must also support model evolution without restarting, reconfiguring, or redeploying applications and services.

We describe a context management middleware that can efficiently handle context despite the execution environment's heterogeneity and evolution. It uses context meta-information to improve a context-aware system's overall performance.

Heterogeneous environments

In pervasive environments, heterogeneity has hardware, software, and network aspects. *Hardware heterogeneity* refers to the presence of different computing devices, such as desktop computers, palmtops, and mobile phones. This type of heterogeneity demands middleware infrastructures that are deployable on servers, workstations, and portable mobile devices. *Software heterogeneity* means the environment is executing different operating systems and applications, requiring software interoperability and the adoption of context models that address specific application requirements. *Network heterogeneity* means that network interconnections among the system's devices don't

conform to a single architecture or technology. This heterogeneity demands adaptable, scalable middleware that supports seamless communication. It also implies that you should limit context models to specific network domains because some context information might pertain to only a subset of applications or devices. We call this *context domain,* a logical boundary that establishes the context information's scope.

Each of these aspects also has an evolutionary component: when a new device type, application, or context type is introduced, context models and the software infrastructure should evolve to accommodate the extended environment's properties and requirements. The infrastructure should efficiently support this evolution.

Developers should work on context models and middleware in tandem to cope with heterogeneity and efficiency. Notably, a context model's complexity determines the computational complexity of the middleware's context-handling capability. For example, adopting a certain context model might require adopting more complex context management and storage mechanisms as well. Joëlle Coutaz and her colleagues present this relationship as a conceptual framework that interconnects an ontological foundation for context modeling with a runtime infrastructure (middleware).[1] Most middleware projects (see the related sidebar) have not explored such an interdependence. Consequently, most middleware have adopted context models restricted to a single context domain (for example, a device's local execution context[2,3] or location[4]), or the context-processing complexity has hindered the middleware's deployment to resource-limited devices. Our approach uses context meta-information to make middleware-level decisions that improve the provision, dissemination, and access to context information.

Other work, such as CoCo,[5] handles heterogeneity of context information services by adopting different context models and services in different network domains. For example, CoCo provides an infrastructure and language to describe context models in order to achieve interoperability among context information services. However, our work doesn't specifically focus on context service heterogeneity, so we assume that any context service and infrastructure are built on the same middleware.

Designing context management middleware

We extended MoCA's (*mo*bile *c*ollaboration *a*rchitecture) context information service[6] to support hardware and software heterogeneity, context evolution, and deployment at devices with different resource profiles. This new service provides a uniform view of context types and data so that the system can retrieve context information published by different sources and at different locations using a single primitive. It also permits the inclusion of new context-aware services without adapting (and redeploying) the middleware to support the new context types.

In our approach, two basic components interact with the context management infrastructure to create, disseminate, and use context: *context providers* and *context consumers*.

A context provider is an entity responsible for publishing a certain kind of context information; for example, it can probe raw data or it can (as an inference agent) aggregate or interpret basic context information into more complex context information. For example, MoCA's *monitor* is a context provider that publishes information about a device's local execution context such as its memory usage, energy consumption, or IEEE 802.11 RSSI (received signal strength indication). MoCA's *location inference service* (LIS),[7] another of its context providers, transforms IEEE 802.11 RSSI values published by the monitor into symbolic positioning information.

A context consumer is an entity interested in certain context information, including context-aware applications or context-processing services. An entity can act simultaneously as a context provider and as a context consumer. For example, LIS is both a consumer of wireless connectivity context and a publisher of location context. And MoCA's *context information service* (CIS) interconnects context providers and consumers, receiving and storing context information and disseminating it to consumers.

We provide a generic context access API for context providers and consumers, keeping the heterogeneity management at the middleware implementation level. To do this, we follow three design strategies:

- We propose an architecture that offers a set of components that help meet common requirements of context management in different execution environments.

- We define a runtime strategy for incorporating new context types into the model and middleware.

- We configure the access and evaluation of context information at runtime, enabling different options for efficient context handling.

Architecture

The CIS consists of the basic elements shown in figure 1. This organization is logical—that is, it doesn't describe each component's actual location or distribution. However, it establishes well-defined responsibilities for the following components:

- The *context event service* provides a mechanism for asynchronous communication, which disseminates *contextual events* and context information to context consumers. A contextual event represents a change of certain context information. The event service adopts a publish-subscribe paradigm and offers a specialized API to handle subscriptions for contextual events. It also maintains a trigger repository for notifying consumers about events that have occurred.

- The *type system manager* maintains context types and validates them at context deployment. The TSM also resolves a context provider's location within each context domain—in particular, when the context information consists of subcontexts placed at different network locations.

- The *context repository* maintains a database of several types of context data.

- *Subservices* are additional services that ease client application development and improve context management system performance. Examples of subservices include caching, quality-of-context, and query services. The query service translates client requests into context repository queries and delivers their results to the client.

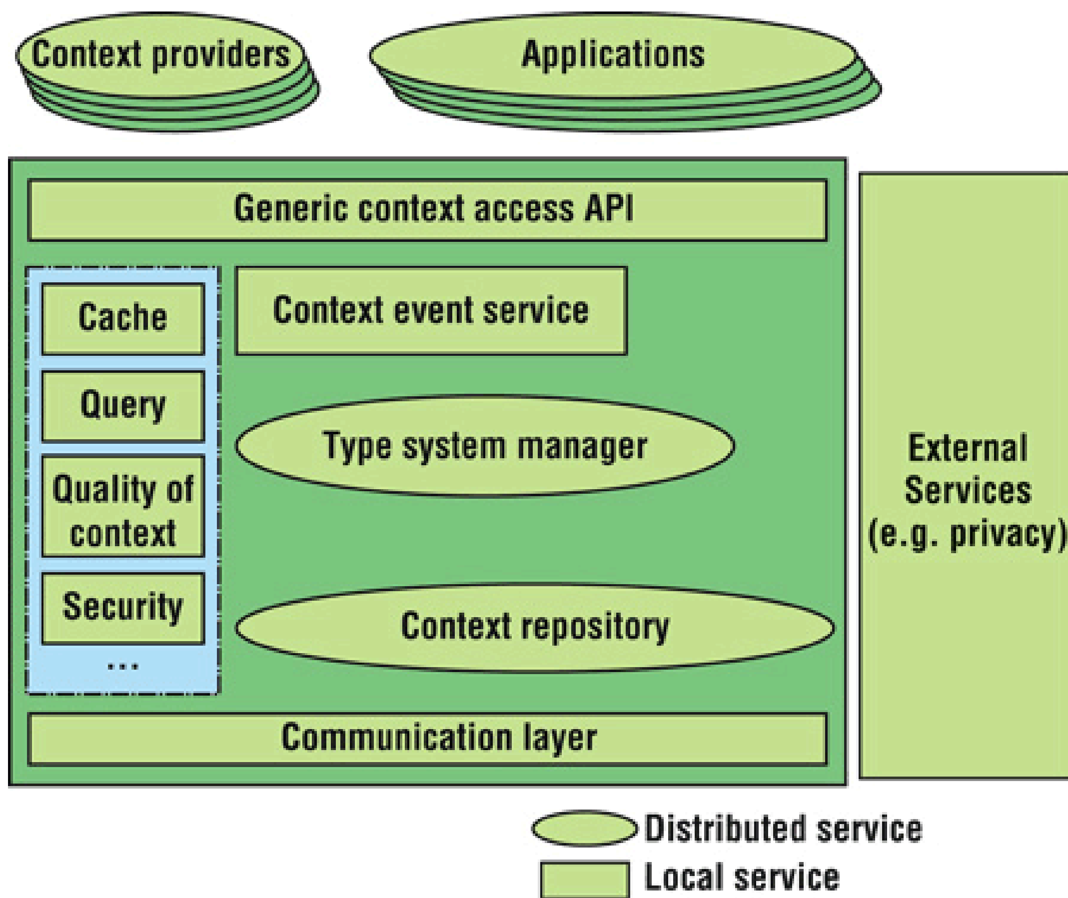

Figure 1. The context service architecture.

These components hide the complexity required to manage context in a heterogeneous environment. Each service's behavior and distribution can vary according to the target device where the middleware will be deployed. The middleware implementation comes in two flavors: *heavyweight* and *lightweight*, the latter targeting resource-limited devices. The lightweight instance defines more restrictive policies for each component and typically distributes some context-related tasks to other remote components. For example, with a lightweight instance, the context repository doesn't maintain a history of context

data. Moreover, the context repository would store only context published locally, delegating requests for nonlocal context to remote repositories, say, at workstations or servers. Also, the subservices available at each device depends on the type of middleware implementation—for example, a lightweight instance will likely have a caching service to cope with intermittent connectivity.

The external services implement services that either depend on or complement context management. For example, we implemented our privacy service[8] as an orthogonal service of MoCA's architecture. Adopting these complementary services helps simplify the interaction between the context management service and a client.

Deploying context types

When introducing a new type of context information into a context-aware system, a process that we call *deployment of context types*, you must model and make it available to all interested parts: sensors, inference agents, applications, and the middleware itself.

A fundamental principle of our approach is to use a strongly typed model for context data. That is, we define and resolve the context type system at development time and describe the context type through an XML-based model that contains:

- structural information—such as attributes and dependencies among context types;
- behavioral information—for example, if a context attribute has a constant or variable value; and
- context-specific abstractions—such as contextual events and queries.

Figure 2 shows the steps involved in deploying a new context type. First, we validate the context model: the *context tool* (CT) checks for inconsistencies between the new and existing context types, such as name conflicts or incorrect type dependency relationships. Next, the CT updates the context type system through the *type system manager*, which in turn initializes the repository for storing the new context information. Finally, the CT generates a library containing stubs that implement the interface for context access. This approach allows efficient context handling and interoperability among languages.

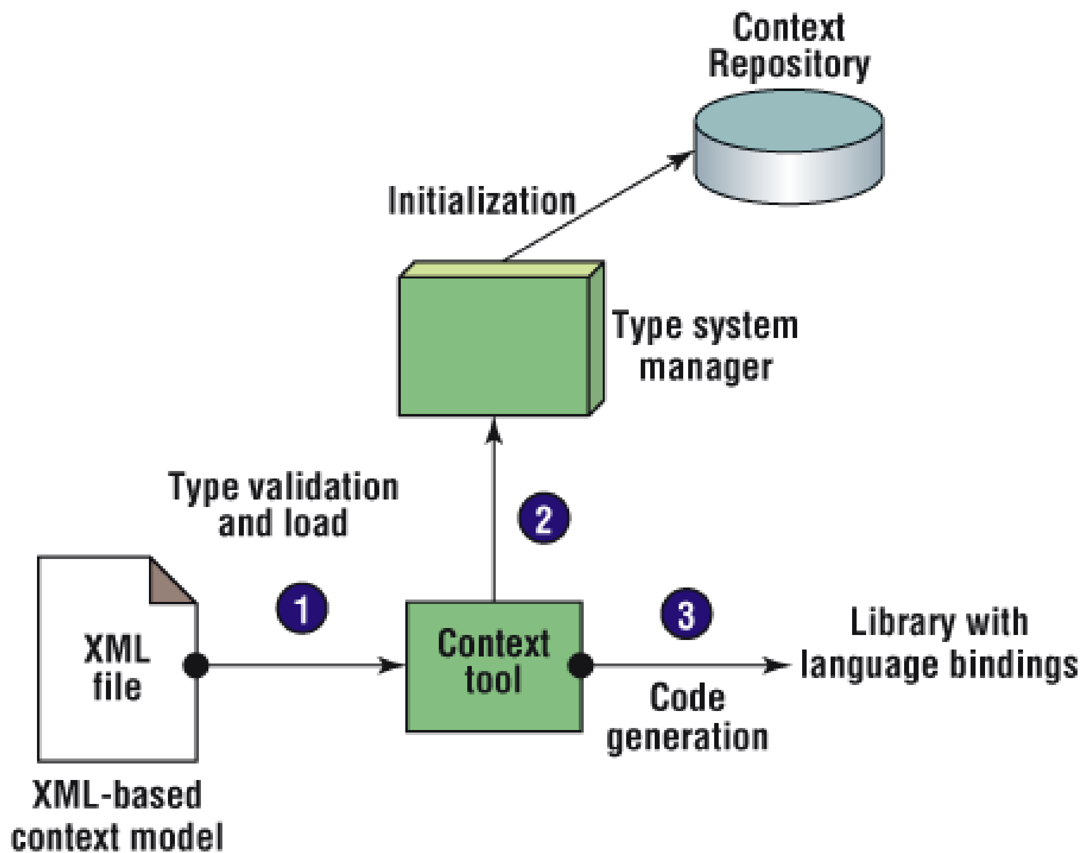

Figure 2. Context deployment steps.

Currently, we've only implemented a Java language binding for MoCA's context model. The generated library maps the XML-based context model to object-oriented language constructs that the applications use to access context information. We've decided to adopt an OO model for context handling instead of an ontology-based model because the latter requires resource-hungry ontology engines.

The context deployment task integrates context modeling with the development of the context-aware application. The library and the middleware keep context access transparent to the application, thus easing application development. We argue that context models should maintain the aforementioned information to help the infrastructure (middleware) make the right decisions about how to use context information efficiently. Finally, the context model helps developers better understand context usage semantics.

Configuring context access

To access and evaluate context information efficiently, we employ a middleware configuration comprised of static and dynamic parts, which adapt context evaluation and dissemination to the modeled context's characteristics and to the application's runtime requirements, respectively. For each configuration, the middleware selects the most appropriate policy for making the context available, according to policies we discuss elsewhere.[9]

Static configuration

To explore each context type's particularities and improve its access performance, a middleware should handle different types differently.

The context meta-information, obtained from the context model, lets the middleware choose the most suitable mechanisms to handle certain context information. When a CT processes a context model, it uses this meta-information to produce an efficient stub implementation for context access. Additionally, the middleware uses such meta-information to adapt its runtime behavior. For example, consider a *static* context attribute—that is, an attribute that has a constant value (for example, the OS type and version running on a device). When deploying this context, we configure the context management infrastructure to disseminate and update this attribute only the first time an application requests the context.

Context information management of local and nonlocal domains is another example of a task improved through static configuration. A local context domain comprehends context information provided by a device that describes its local execution context—CPU usage, available memory, and the operating system's type and version. On the other hand, a nonlocal context domain consists of context information provided by an external context provider. Hence, access to nonlocal context requires at least one hop on the network. The middleware stores local domain context in a local instance of the context repository, which improves the performance for context information access. Current middleware platforms usually don't distinguish between local and nonlocal context domains.

Table 1 shows some parameters for static configuration and alternatives for changes in the middleware behavior. We call this configuration static because it doesn't change at runtime.

Table 1. Parameters for static configuration.

Aspect	Modeling information	Middleware behavior
Distribution	Local vs. distributed	• Direct access to local context • Opportunistic approach for context dissemination
Upgrading	Static vs. dynamic	• Continuous caching for static context
Context service placement	Context domains	• Transparent selection of the most appropriate service instance for the context
Querying	Context-specific queries	• Strongly typed and optimized queries through the network • Abstraction sharing

Dynamic configuration

Besides static configuration based on context models, our middleware also supports dynamic configuration at application start-up time. When an application launches and registers itself at the middleware, it can define a specific policy for using certain context information. These policies are based on application requirements about the precision of the context information to be used. Table 2 shows some parameters for dynamic configuration (in terms of application requirements) and the corresponding influence on the middleware behavior. *Freshness* (how recent data must be) and *lazy* (on demand) evaluations are examples of access policies that applications can set.

Table 2. Parameters for dynamic configuration.

Aspect	Application requirement	Middleware behavior
Updating	Precision parameters	• Quality-of-context negotiation (freshness, precision) • Suitable cache policies
Evaluation	Eager vs. lazy	• On-demand context attribute evaluation
Querying	Eager vs. lazy	• On-demand query result retrieval
Distribution	Context views	• Decreased information overhead (handles just a portion of the context information)

Several applications using the same middleware instance can select different policies for accessing the same context information. In this case, the middleware chooses the most restrictive policy that satisfies all application requirements. Using such a dynamic configuration, the middleware can choose when to best publish context information and to execute context queries. For example, an application might be interested in location context changes only in terms of a symbolic location—for example, the *building name* where the device is located—and thus shouldn't receive notifications about location changes at a finer granularity (for example, coordinates). In this example, if a middleware delivers the location only in the selected granularity, it could decrease the amount of disseminated information, improving the context dissemination's performance.

We're investigating how quality-of-context modeling can provide a richer, dynamic middleware configuration. So far, we've decided to just implement freshness and context precision properties as parameters for setting up our middleware policies.

At this point, we've developed a prototype of a context service based on our architecture. We're now integrating this with other components and services of our MoCA context-provisioning middleware architecture—in particular, our privacy service for context access. Through this integration, we aim to validate the architecture's ability to support configuration of context access and context model evolution.

As part of our future work, we'll investigate extending our architecture to flexibly accommodate quality-of-context parameters. We're also planning to research how context consumers can specify and use *context views*—that is, selected portions or attributes of a complex context type. We believe that specifying different context views without changing the underlying context model could create new opportunities for enhancing context access efficiency.

Acknowledgments

Research grants from the Brazilian National Research Funding Agency (CNPq) (project numbers 552.068/02-0 and 479824/2004-5) support this work.

References

1. J. Coutaz , et al., "Context is Key,"*Comm. ACM,* vol. 48, no. 3, 2005, pp. 49-53.
2. A.T.S. Chan and S.N. Chuang , "MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing,"*IEEE Trans. Software Eng.,* vol. 29, no. 12, 2003, pp. 1072-1085.
3. H. Lei , et al., "The Design and Applications of a Context Service,"*Sigmobile Mobile Computing Comm. Rev.,* vol. 6, no. 4, 2002, pp. 45-55.
4. A. Ranganathan , et al., "MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications,"*Proc. 5th Int'l Conf. Middleware (Middleware 05),* Springer, 2004, pp. 397-416.
5. T. Buchholz , et al., "Dynamic Composition of Context Information,"*1st Ann. Int'l Conf. Mobile and Ubiquitous Systems: Networking and Services* (MobiQuitous 04), 2004, pp. 335-343.
6. V. Sacramento , et al., "MoCA: A Middleware for Developing Collaborative Applications for Mobile Users,"*IEEE Distributed Systems Online,* vol. 5, no. 10, 2004; http://csdl2.computer.org/comp/mags/ds/2004/10/ox002.pdf.
7. F. Ney da Costa Nascimento , "A Service for Location Inference of Mobile Devices Based on IEEE 802.11," master's thesis, Dept. de Informática, Pontifícia Universidade Católica, 2005 (in Portuguese).
8. V. Sacramento , M. Endler and F. Ney da Costa Nascimento , "A Privacy Service for Context-Aware Mobile Services,"*Proc. 1st Int'l Conf. Security and Privacy for Emerging Areas in Comm. Networks* (SecureComm 01), IEEE CS Press, 2005, pp. 182-193.

9. R. Couto Antunes da Rocha and M. Endler , "Evolutionary and Efficient Context Management in Heterogeneous Environments,"*3rd Int'l Workshop Middleware for Pervasive and Ad-Hoc Computing* (Middleware 05), ACM Press, 2005, pp. 1-7.

Ricardo Couto Antunes da Rocha is a PhD candidate in the Department of Informatics at the Pontifícia Universidade Católica. His research interests include context-aware computing, middleware, mobile and ubiquitous computing, and distributed systems. He received his MSc in computer science from the University of São Paulo in 2001. He is a member of the ACM and Brazilian Computer Society (SBC). Contact him at Rua Voluntários de Pátria, 330/601, Botafogo, Rio de Janeiro, RJ, Brazil, 22270-010; rcarocha@inf.puc-rio.br.

Markus Endler is an assistant professor at the Pontifícia Universidade Católica's Department of Informatics. His research interests include distributed algorithms and systems, mobile and ubiquitous computing, and middleware for mobile networks. He received his Dr.rer.nat. in computer science from the Technical University in Berlin and the title Professor Livre-docente from the University of São Paulo. He is a member of the ACM and Brazilian Computer Society (SBC). Contact him at Dept. de Informática, PUC Rio, Rua Marques de São Vicente 225, Office 503, Gavea, Rio de Janeiro, RJ, Brazil, 22453-900; endler@inf.puc-rio.br.

Related Work on Managing Heterogeneity

Heterogeneity is a classic problem addressed by mobile computing middleware. However, most middleware research efforts have managed heterogeneity in mobile and pervasive systems either simply as an interoperability issue or solved it at a higher level, using static and dynamic reconfiguration.[1] For example, some middleware adopt a uniform communication paradigm that hides the lower-level communication protocol's particularities.

The RCSM (reconfigurable context-sensitive middleware)[2] and Pace (pervasive autonomic context-aware environments)[3] middleware address heterogeneity with approaches similar to ours. Both RCSM and Pace provide tools for validating context models and generating stubs for different languages (to support interoperability), accessing context from different programming languages and platforms and separating models from implementation. RCSM middleware focuses on spontaneous interactions in ad hoc networks (instead of structured networks), aiming to support autonomous collaboration among peers. Pace middleware presents requirements closer to our service, such as offering support for runtime evolution of context type. However, these middleware have some drawbacks when deploying the context services in resource-limited devices because they use a common infrastructure that doesn't consider differences among devices. For example, because they don't aim for efficiency and scalability, they don't distinguish between local and nonlocal context.

To conform to device limitations, some middleware adopt an agent-based paradigm and use techniques such as remote execution of mobile code to decrease the local resources required for context management.[4,5]

Early work on context-aware middleware considered the evolutionary inclusion of context providers of different types and technologies. For example, the Context Toolkit[6] offers the abstraction of a widget, which applies to context providers and context interpreters. To support an evolutionary use of context, some middleware[7] explore context discovery. A middleware that supports this concept lets applications discover and use new context types at runtime.

However, you should still combine such an approach with the adoption of context models that enable the modeling of complex context information. In this respect, ontologies are a powerful tool for context modeling, offering both rich expressiveness and support for the evolutionary aspect of context modeling. To process ontologies, the system architecture must provide an ontology engine that can make inferences over ontologies. This requirement could impact local context management performance with resource-limited devices because it requires the ontology engine to run on a server instead of on the device. To address this limitation, some middleware use agents to transfer resource-hungry computing (handling ontologies) to servers on a wired network.[4,5]

Through a similar approach, we complement heterogeneity management by adopting a dual approach for context management in which we shape the distribution and behavior of context management services to mobile and desktop (or server) computers. We believe that this approach enables deployment and usage in more realistic scenarios because it focuses more on performance and scalability. Additionally, our approach uses context meta-information to explore opportunities for performance enhancement.

References

1. C. Mascolo , L. Capra, and W. Emmerich , "Middleware for Communications,"*Principles of Mobile Computing Middleware,* Q. Mahmoud, ed., John Wiley & Sons, 2004, pp. 261-280.
2. S.S. Yau , et al., "Reconfigurable Context-Sensitive Middleware for Pervasive Computing," http://doi.ieeecomputersociety.org/10.1109/MPRV.2002.1037720, *IEEE Pervasive Computing* , vol. 1, no. 3, 2002,pp. 33-40.
3. K. Henricksen , et al., "Middleware for Distributed Context-Aware Systems,"*On the Move to Meaningful Internet Systems 2005,* LNCS 3760, Springer, 2005, pp. 846-863.
4. H. Chen , *An Intelligent Broker Architecture for Pervasive Context-Aware Systems,* doctoral dissertation, Univ. of Maryland, Baltimore County, 2004.
5. M. Khedr and A. Karmouch, , "ACAI: Agent-Based Context-Aware Infrastructure for Spontaneous Applications," *J. Network and Computer Applications* , vol. 28, no. 1, 2005pp. 19-44.
6. A.K. Dey , "Providing Architectural Support for Building Context-Aware Applications," doctoral dissertation, College of Computing, Georgia Inst. of Technology, 2000.
7. G. Thomson , et al., "An Approach to Dynamic Context Discovery and Composition,"*Proc. System Support for Ubiquitous Computing Workshop* (UbiComp 03), 2003; http://ubisys.cs.uiuc.edu/papers/dynamic-context-discovery.pdf.

Related Links

- DS Online's Mobile and Pervasive Community, cms:/dsonline/topics/mobile/index.xml

- "MoCA: A Middleware for Developing Collaborative Applications for Mobile Users" (pdf), http://csdl2.computer.org/comp/mags/ds/2004/10/ox002.pdf

- "Guest Editors' Introduction: Context-Aware Computing", http://doi.ieeecomputersociety.org/10.1109/MPRV.2002.1037718

Cite this article: Ricardo Couto Antunes da Rocha and Markus Endler, "Context Management in Heterogeneous, Evolving Ubiquitous Environments," *IEEE Distributed Systems Online*, vol. 7, no. 4, 2006, art. no. 0604-o4001.