

A Flexible Middleware for Service Provision Over Heterogeneous Pervasive Networks

Emerson Loureiro

Frederico Bublitz
Hygo Almeida

Nadia Barbosa
Glauber Ferreira

Angelo Perkusich

Embedded Systems and Pervasive Computing Lab
Center of Electrical Engineering and Informatics
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil

{eclf,fbublitz,nadia,ferreira}@dsc.ufcg.edu.br {perkusic,hygo}@dee.ufcg.edu.br

Abstract

Pervasive computing has gained much attention from the research community due to the possibility of deploying the first pervasive environments. Therefore, many software solutions are emerging, with the intent of facilitating the development of pervasive applications. Within this scope, in this paper, we introduce a service oriented middleware for pervasive computing, enhanced with runtime flexibility, extensibility for applications, and heterogeneous service provision. Our goal is to enable the middleware and its applications to be adapted to changing operational scenarios. Furthermore, different protocols can be used to discover and access services.

1. Introduction

Nowadays we are facing a new trend in computer science. Advances in microelectronics, and thus on hardware components are promoting the development of computers with reduced size. Also, wireless networking technologies are increasingly being made available on different computing devices that incorporates efficient energy consumption techniques. The mix of these advances is enabling the large-scale development of small mobile devices embedded with wireless networking features. Cellular phones and handhelds, both equipped with Bluetooth (<http://www.bluetooth.org>) and/or Wi-Fi (Wireless Fidelity - <http://www.wi-fi.org>) interfaces, are among these devices.

A consequence of these advances in mobile technology is the possibility to give the first steps towards a computing paradigm foresaw almost 15 years ago – the so-called ubiquitous, or pervasive, computing [15]. Introduced by Mark Weiser, pervasive computing defines a world where computing appliances are seamlessly integrated into our lives,

providing us with anytime, anywhere information.

Although simple at first sight, the pervasive computing vision raises a set of challenges in the context of application software development. The seamless or invisible integration of appliances in human life, requires them to act pro-actively on the behalf of the users [10]. To perform this task, pervasive computing systems need to be aware of the current context, such as the situation of a user (e.g., in a meeting), surrounding conditions (e.g., people in the vicinity), and the time of the day. More precisely, the interests of the users as well as the resources and information available both in the device and in the environment are the two main features in such scenario.

The interests of the users, which are viewed as the requirements of the application, are however, too dynamic. In one moment a user can be interested on a cheap fast food restaurant, and some minutes later on the weather forecast. Considering that it is not possible, or at least very difficult, to anticipate all the functionalities that satisfy the needs of a user, these changes in the requirements forces pervasive computing systems to adapt in an unanticipated way. In our example, the application will be faced with two functionalities which it was probably not conceived to deal with. Therefore, it is necessary to provide mechanisms to deploy “pieces” of software in the device in order to provide the missing features. Also, considering that such deployment should not bother users while executing their tasks, extensibility and runtime flexibility are thus mandatory features for any pervasive system.

Another point to be stressed is concerned with the way that pervasive applications access information and resources available in the environment. One way of enabling such access is through the provision of *services*. In such an approach, a lighting sensor, for example, could provide two services (e.g., *getLightingLevel* and *setLightingLevel*) for returning and setting up the current lighting level of a

room. These services are then used by a pervasive application to detect the level of illumination in the room, to check if it is according to the preferences of the user, and if it is not, adjust it accordingly. This idea of using services to access remote functionalities has been considered the next step in distributed computing [11] and can be applied as well to develop pervasive applications. This is due to the dynamic binding nature of the service approach, which allows an application to discover and use services on demand, possibly guided by the interests of the users.

Within this scope, the support for mobility is one of the key problems to be solved. When a user moves through different environments, it is likely that the service discovery protocols has to change. For example, in one environment services can be discovered based on UPnP (Universal Plug and Play - <http://www.upnp.org>), and in another one based on JXTA (<http://www.jxta.org>). Therefore, it is necessary to include capabilities to perform service discovery based on different protocols, in order to increase the number of services that a user can access. One approach to tackle this problem is to embed all available service discovery protocols in the mobile device. However, their storage and memory constraints suggests that this is not a reasonable decision. Besides, it is possible that some of these protocols will be never be used, thus only wasting storage space. A better approach is to embed only a core set of service discovery protocols in a device, and enable them to be dynamically added and removed, whenever needed.

Based on our discussion, in this paper we present a service provisioning middleware for pervasive computing, named *Wings*. Our main focus is on flexibility to perform service provision based on heterogeneous networks and to provide the middleware, as well as the applications running over it, with enough extensibility for facing pervasive environments. The remainder of this paper is organized as follows: in Section 2 we present the *Wings* pervasive computing middleware. We also present and discuss some related works in Section 3. Finally, in Section 4, we conclude the paper and review our future works.

2. The *Wings* Middleware

The basis of the *Wings* middleware lies on the concepts of *context*, *service*, and *peer*. *Context* encapsulates the information used by pervasive applications for improving their interaction with users [9]. A *service* is some functionality provided by a device. Audio streaming and printing are some examples of such services. Each service is defined by: a *name*, a *description*, a *parameter list*, and a *return type*. The first one defines the name of the service. The description provides information about the functionality of the service, and can be used in the discovery process, in order to determine whether a discovered service matches the needs of the user. The parameter list and the return type

describe the types of the arguments of the service and its result, respectively. Such information is useful, for example, when trying to find an alternative for a service which is no longer available in the environment. Finally, *peers* are defined as remote network hosts, providing two information: their names and the list of services they provide.

The architecture of the *Wings* middleware, which has been implemented over the Java CDC ¹ specification, is shown in Figure 1. Observe that, the concepts of service and peer are implemented in the *Pervasive Networking* module. The concept of context, on the other hand, is implemented in the *Context Awareness* module. In what follows each module of the *Wings* architecture is detailed.

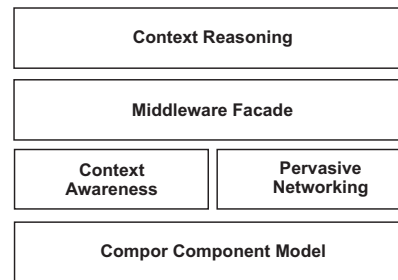


Figure 1. The architecture of the *Wings* pervasive computing middleware.

2.1. The *Compor* Component Model

For enabling the middleware to be updated on the fly, we have used a dynamic component model, named *COMPOR* Component Model [1]. Such a model defines a dynamic composition solution for inserting, removing, and changing components of an application without impacting the existing ones, even at runtime. More precisely, two entities are specified in such a component model; *containers* and *functional components*, which are organized in a hierarchical way. Functional components implement the functionalities of an application, making them available through *services*. On the other hand, containers do not implement functionalities. Instead, they are composed of functional components or other containers, and their function is to manage the access to the services provided by their child components.

Each container keeps two tables: one for the services provided by its child components and another for the events they are interested. In this way, after the insertion of a component, the list of services and events for each container up to the root of the hierarchy must be updated. This is also true also for the removal and update of components. We illustrate in Figure 2 the process of inserting a component into an application. Each step presented in such a figure

¹Connected Device Configuration - <http://java.sun.com/products/cdc>

is described next: 1) a component, named *X*, which implements the service “calculate” is added to *Container 2*; 2) *container 2* updates its service table redefining the services provided by its child components (the same occurs for the event table); 3) *container 2* asks its parent container, *Container 1*, to update its service table. This will redefine the services provided by its child components, as in the last step; 4) *container 1* updates its service table.

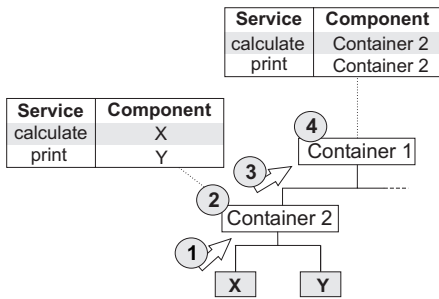


Figure 2. The insertion of a new component into an application.

After the execution of these steps the services provided by *component X* can be accessed from any component in the hierarchy without an explicit reference to it (Observe that a component has only the reference to its parent container). Therefore, assuming the existence of a service named “save”, implemented by a component *K*, the execution of this service can be requested by a component *X*, without an explicit reference to *K*. This feature enables to change the component that provides a service without modifying the rest of structure. In Figure 3 it is illustrated the process of invoking the “save” service. The steps described in such a figure are described next: 1) *component X* requests the execution of the service “save” to its parent container; 2) based on its service table, *container 2* verifies that no child component implements the service “save”; *container 2* forwards the request to its parent container, in this case *container 1*; *container 1*, according to its service table, verifies that one of its children, *container 3*, implements the service “save”. *Container 1* views *container 3* as the component that implements the requested service; *container 1* then forwards the service request to *container 3*; *container 3* does not implement the requested service but has a reference to the component that implements it, in this case *component K*; *container 3* forwards the request to *component K*; *component K* executes the “save” service and returns the result, following the reverse path, back to the requester.

2.2. The Pervasive Networking Module

This module implements features for enabling a device to discover peers, to be discovered by them, as well as provide services (i.e., advertise and discover). In *Wings*, these

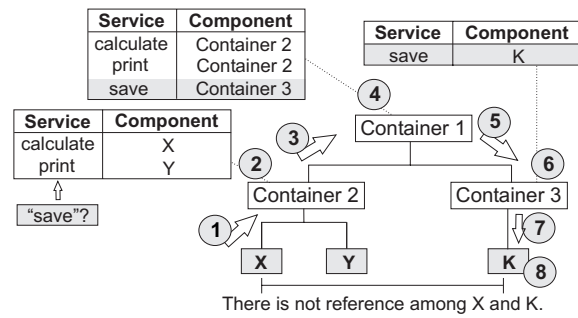


Figure 3. The invocation of a service in a Compor-based application

features are implemented by two different types of plugins, which we have been named of *Service Provision Plugins* (SPPs) and *Host Discovery Plugins* (HDPs). Both types of plugins are based on the *Compor* Component Model, extending the *functional component* entity. In this way, it is possible to insert, remove, and update SPPs and HDPs in a transparent way, both for the users and for the applications executing over the middleware. Furthermore, as it is possible to deploy different implementations of such plugins, at the same time, the discovery of peers and provisioning of services can be performed over heterogeneous networks. For example, both operations can be executed over UPnP, JXTA, and Bluetooth networks, increasing the number of services and peers a device has access to.

To this end, each SPP and HDP provides a specific set of services. The former provides four services: *discoverServices*, *stopServiceDiscovery*, *advertiseService*, and *unadvertiseService*. The *discoverServices* service searches for remote services available in the network. This service receives in the argument a set of keywords representing the desired functionality as well as a listener object, which is notified when a service matching the keywords is found. It returns an identifier that uniquely identifies the search. Such identifier should be provided to the *stopServiceDiscovery* service, whenever a peer or service search must be stopped. Finally, the *advertiseService* and *unadvertiseService* services are used for respectively advertising/unadvertising a service. HDPs, on other hand, only provide two services: *discoverPeers* and *stopPeerDiscovery*. The former starts the discovery of peers in the network associated with the HDP. It receives a listener as the argument, which is notified whenever a peer is discovered. Just like the *discoverServices* service, *discoverPeers* also returns an identifier for the search. Such an identifier should be provided to the *stopPeerDiscovery* service, when a search for peers should be canceled.

The *Pervasive Networking* module itself is implemented as a *container*, named of *Pervasive Networking Container*, which is also based on the *Compor* Component Model.

Therefore, SPPs and HDPs can be inserted and removed from the middleware without stopping and restarting it. It is through this container that all services of the SPPs and HDPs are invoked, hiding from the above module all the details involved in this process. To exemplify how this transparency has been achieved, we present in Figure 4 the service discovery process through *Pervasive Networking Container*. Each step presented in such a figure is described next: 1) a requester starts the service discovery from *Pervasive Networking Container*; 2) *Pervasive Networking Container* passes the service discovery request to each installed SPP; 3) once a SPP discovers a service, it notifies the listener associated with the search. As it can be viewed in Figure 4, SPPs 1 and 2 have discovered services and are about to notify the listener. The processes associated with the other functionalities are similar, and therefore, we have not presented them here.

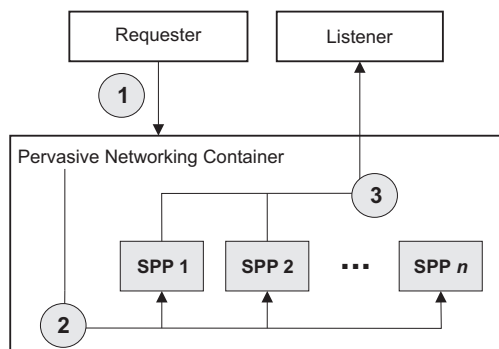


Figure 4. The process of discovering services through the installed SPPs.

2.3. The Context Awareness Module

The *Context Awareness* module provides a mechanism for delivering context information for applications and services running at the highest module of *Wings*. This process can be performed in two ways: using *key-value pairs* and *context events*. In the former, each context information is associated with a key, which is used to retrieve the current value of the information. The current battery level of the device and the number of peers in the vicinity are some examples of this kind of context information retrieval. Context events, on the other hand, associates a context information with a condition. Each condition is associated with a key, which serves to register a listener to it. Once the condition is satisfied, an event is fired, in order to notify all the listeners. As an example, an application could be interested in the *battery low* condition, in order to apply some energy saving technique or to store critical data before the battery be completely consumed.

Practically speaking, the provision of context information is performed by a specific type of plugin, named of *Context Awareness Plugins* (CAPs). The idea is that each CAP should be associated with a specific domain (e.g., intelligent homes and tourist information), and thus provide only the context information related to such a domain. To this end, each CAP provision two services, *retrieveInformation* and *registerContextCondition*, associated respectively with the key-value pair and event-based context information retrieval approaches. By using this approach, it is possible to easily increase the context sensitivity feature of the middleware, just by adding CAPs to it. Besides, as each CAP is based on the *Compor* Component Model, just like the SPPs and HDPs, this addition can be fulfilled at runtime.

Just like in the *Pervasive Networking* module, this one is implemented as a container according to the *Compor* model. Therefore, the retrieval of context information in the installed CAPs as well as their insertion and removal is performed through such a container.

2.4. The Middleware Facade Module

The *Middleware Facade* module simply provides a single interface so that applications and services, running on the above module, can easily use the middleware features. To this end, this module provides one method for each of the services provided by the middleware plugins, that is, SPPs, HDPs, and CAPs. In addition, it also keeps the root container of the *Wings* plugin hierarchy, that is, the one where the *Pervasive Networking* and *Context Awareness* modules are placed during the middleware execution.

2.5. The Context Reasoning Module

In this module are placed all the software which makes use of context information. Agents, services, and client applications are examples of what can be executed here, although the current implementation of *Wings* only provides native support for the last two.

Service development in *Wings*: The service framework of *Wings* is based on three entities: *Service*, *LocalService*, and *ServiceProxy*. *Service* defines, at the implementation level, the basic features of all *Wings* services, presented in the beginning of this section. The other two entities, *LocalService* and *ServiceProxy* extend *Service*, and represent, respectively, services provided locally and remotely. The former are the ones which can be advertised, and as we have seen, this is performed by the SPPs, through the *Middleware Facade* module. It is important to note that service advertising is only possible for local services. This approach forbids a peer to advertise a service not provided by itself (i.e., a *ServiceProxy*).

Winglets: *Wings*-based pervasive computing applications: In *Wings*, pervasive applications are named of

Winglets. They are supported by a plugin-based architecture, in order to be extended on demand. The plugins of a *winglet* executes, basically, accessing its core API. Such an API, which is released by the developer, should contain all the classes and resources that plugins will use to extend the *winglet*.

Plugins are packaged in zipped files, containing a properties file, a single folder, and a *jar* file. The properties file contains general information about the plugin, such as its name, description, vendor, amongst others. This file must be named of “descriptor.properties”. The folder should contain the plugin’s libraries, and must be named “lib”. Among these libraries it should be included the application’s core API. The *jar* file should contain the plugin’s .class files, and must necessarily be named of “plugin.jar”. Still, the manifest of this file should contain a *Plugin-class* entry, which indicates the main class of the plugin.

3. Related work

Some of the current work on the infrastructure for pervasive computing has been concentrated on solving the problem of context-sensitivity. An example is the RCSM (Reconfigurable Context Sensitive Middleware) middleware [17]. It provides a language, the context sensitive interface description language (CS-IDL), for applications specifying the actions to be triggered given some context condition. The action itself is implemented in programming languages like C++ and Java. Another examples are the JCAF [2] (Java Context Awareness Framework) and Scooby [14] infrastructures, which use events for notifying applications about changes in the context. In particular, JCAF, just like *Wings*, provides flexibility to the context awareness mechanism.

Efforts have also been made concerning the development of flexible and extensible pervasive computing middlewares, where the concept of plugins has been successfully applied. Basically, the idea is to deploy the middleware with a minimal set of functionalities, providing hooks where extensions can be plugged. In this context, examples of extensions are transport and remote procedure call protocols as well as context-awareness mechanisms. Examples of plugin-based pervasive middlewares are ReMMoC [7], Runes [6], and Plugin-ORB [8]. The ReMMoC middleware uses plugins for discovering services available in the environment regardless of the underlying protocol. The Runes middleware, on the other hand, is based on the concept of *component frameworks*. The idea of Runes is that each component framework should have a specific purpose, defining, for example, the types of plugins it accepts. Finally, the Plugin-ORB middleware is based on the idea of encapsulating other pervasive computing middlewares in plugins, and thus, access remote resources through different solutions.

At the application level, flexibility has also been considered. The purpose is to provide mechanisms for enabling applications to adapt according to the environment and the interests of the users. In other words, the main goal is to enable the insertion, removal, and update of features, providing pervasive applications with some degree of adaptability for facing an ever-changing environment. Works such as the one of Belaraman et al. [3] are classified in this category.

The mix of service oriented and pervasive computing has also been tackled by some middlewares. The main idea of using services within pervasive computing is to enable applications to dynamically bind to needed services. Therefore, in a general way, these approaches try to determine which services from an environment are relevant to the current context. For example, information about the users’ needs can be used for filtering relevant services. In this research branch we can place works like [5], [4], and [12]. Within this scope, we would like to give special attention to the SDIPP protocol [13], which defines a set of sub-protocols for enabling the discovery of services, either through Bluetooth or GPRS².

As it has been presented, a reasonable effort has been performed concerning the aspects of flexibility and service provision in pervasive computing middlewares. However, as it is possible to note, these features are scattered across different solutions. Whereas some middlewares provide inherent flexibility for atop applications, or for the middleware itself, service provision is not tackled. When services are used for dynamically finding needed functionalities, application and/or middleware flexibility is not considered. In this context, except for one solution, all the others perform service provision considering a single discovery protocol. Besides, none of them provides specific mechanisms for the discovery of hosts, which is an important feature for pervasive environments [16].

4. Conclusions and Future Work

In this paper we have presented a pervasive computing middleware focused on flexibility and service provision across heterogeneous networks. More precisely, we have provided it with dynamic flexibility and also extensibility for its applications. In addition, the possibility to advertise, discover, and use services based on any existing protocol is included. Within the scope of pervasive computing, these are important features, since they enable the unanticipated evolution of the middleware, the personalization of client applications, and the transparent access to heterogeneous services.

Another point to be considered is the simplicity of the introduced solution. By focusing on the features related to dynamic software composition and plugin based software architectures, we have developed the *Wings* middleware,

²General Packet Radio System

which incorporates adaptability features that can effectively support the development of applications for pervasive computing environments. Also, it is important to point out, that other efforts have been made to solve the problems discussed in this paper. However, such efforts do not address all the key features related to flexibility and heterogeneous service provision. Therefore, the solution introduced in this work can be seen as a more complete approach concerning service provision middlewares for pervasive computing.

The *Wings* middleware implementation is already finished and tested, and can be downloaded from <http://www.percomp.org/wings>. A C++ version, for the Symbian operating system, is also being developed. We are also putting our efforts towards the development of repositories for deploying SPPs, HDPs and CAPs, so that developers can download them. Still, such a repository can be useful for the middleware itself, when it needs to deploy a new SPP, HDP or CAP. Furthermore, with the intent of supporting the development of SPPs, HDPs, CAPs, services, *winglets* and their plugins, we are also working towards the development of an eclipse plugin.

5. Acknowledgements

The first, second, third, and sixth authors are with the Graduate Program in Computer Science (COPIN/UFCG), and the fifth is with the Graduate Program in Electrical Engineering (COPELE/UFCG), from the Federal University of Campina Grande. The research reported in this paper is partially supported by grants 481031/2004-9 and 550820/2005-1 from the Brazilian National Research Council (CNPq), scholarships from CNPq for the fifth author, and scholarships from CAPES for the first, second, third, and sixth authors.

References

- [1] H. O. Almeida, A. Perkusich, E. B. Costa, and R. B. Paes. Compor: a methodology, a component model, a component based framework and tools to build multiagent systems. *CLEI Electronic Journal*, 7(1), 2004.
- [2] J. E. Bardram. The java context awareness framework (JCAF) - A service infrastructure and programming framework for context-aware applications. In H. Gellersen, R. Want, and A. Schmidt, editors, *Proceedings of the 3rd International Conference on Pervasive Computing*, Lecture Notes in Computer Science. Springer Verlag, May 2005.
- [3] N. M. Belaramani, Y. Chow, V. W.-M. Kwan, C.-L. Wang, and F. C. Lau. *Intelligent Virtual World: Technologies and Applications in Distributed Virtual Environments*, chapter A Component-based Software Architecture for Pervasive Computing, pages 201–222. World Scientific Publishing Co., 1 edition, July 2004.
- [4] U. Bellur and N. C. Narendra. Towards service orientation in pervasive computing systems. In *International Conference on Information Technology: Coding and Computing - Volume II*, pages 289–295, Las Vegas, NV, EUA, April 2005.
- [5] T. Chaari, F. Laforest, and A. Celantano. Design of context-aware applications based on web services. Technical Report RR-LIRIS-2004-033, Laboratoire d'InfoRmatique en Images et Systèmes d'information, Lyon, France, September 2004.
- [6] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The runes middleware: A reconfigurable component-based approach to networked embedded systems. In *Proceedings of the 16th IEEE International Symposium on Personal Indoor and Mobile Radio Communications - PIMRC'05*, Berlin, Germany, September 2005. IEEE Communications Society.
- [7] G. Coulson, P. Grace, G. Blair, D. Duce, C. Cooper, and M. Sagar. A Middleware Approach for Pervasive Grid Environments. In *Workshop on Ubiquitous Computing and e-Research*.
- [8] A. d'Acierno, G. D. Pietro, A. Coronato, and G. Gugliara. Plugin-orb for applications in a pervasive computing environment. In *Proceedings of the 2005 International Conference on Pervasive Systems and Computing*, pages 140–146, Las Vegas, NE, USA, June 2005. CSREA Press.
- [9] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, February 2001.
- [10] E. Loureiro, L. Oliveira, H. Almeida, G. Ferreira, and A. Perkusich. Improving flexibility on host discovery for pervasive computing middlewares. In *Proceedings of the 3rd International Workshop on Middleware for Pervasive and Ad hoc Computing*, Grenoble, France, 2005. ACM Press.
- [11] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Proc. of Fourth International Conference on Web Information Systems Engineering*, pages 3–12, Rome, Italy, December 2003. IEEE.
- [12] P. Poupyrev, T. Sasao, S. Saruwatari, H. Morikawa, T. Aoyama, and P. Davis. Service discovery in TinyObj: Strategies and approaches. In *Proceedings of the Workshop on Pervasive Mobile Interaction Devices*, pages 19–22, Munich, Germany, May 2005. Springer Verlag.
- [13] N. Ravi, P. Stern, N. Desai, and L. Iftode. Accessing ubiquitous services using smart phones. In *Proceedings of 3rd IEEE International Conference on Pervasive Computing and Communications*, pages 383–393, Kauai Islands, Hawaii, Março 2005. IEEE Computer Society.
- [14] J. Robinson, I. Wakeman, and T. Owen. Scooby: Middleware for service composition in pervasive computing. In *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad hoc Computing*, Toronto, Canada, October 2004.
- [15] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991.
- [16] Y. Xiong, X. Lin, and J. A. Rowson. Estimating device availability in pervasive peer-to-peer environment. In *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems*, pages 254–260. IEEE Computer Society, May 2004.
- [17] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. S. Gupta. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. *IEEE Pervasive Computing*, 1(3):33–40, July 2002.