Towards Context-Aware and Resource-Driven Self-Adaptation for Mobile Handheld Applications

Davy Preuveneers
Department of Computer Science
Celestijnenlaan 200A
B-3001 Leuven, Belgium

davy.preuveneers@cs.kuleuven.be

Yolande Berbers
Department of Computer Science
Celestijnenlaan 200A
B-3001 Leuven, Belgium

yolande.berbers@cs.kuleuven.be

ABSTRACT

Mobile handheld computing is gaining momentum as more and more wireless handheld devices are being used to accomplish various computing tasks. Context-awareness can help to adapt and personalize applications or to optimize resource usage. However, many existing context infrastructures by themselves impose heavy resource requirements for deployment or highly affect the limited autonomy of the mobile device. In this paper we describe how resource-driven self-adaptation is used in our layered context-driven application middleware in order to enable deployment on a mobile device and optimize resource usage. Our evaluation shows that the introspection and intercession capabilities of the self-adapting middleware provide the necessary flexibility to achieve a balanced resource usage between the context-driven application middleware and the context-aware applications, and that the resource-driven self-adaptation is able to more than double the battery lifetime in real life scenarios.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; H.3.4 [**Information Storage And Retrieval**]: Systems and Software

Keywords

Context-awareness, adaptation, mobile computing

1. INTRODUCTION

The rapid advancements of wireless networking and the proliferation of portable and mobile handheld devices promise to change drastically the human-computer interaction [10]. Context-awareness [3] is one of the important key drivers of the growing middleware support for pervasive services. It provides the intelligence backbone to enable non-intrusive access to personalized services and information in public areas, at work, as well as in the home environment. As Charles

Darwin stated earlier: "*It is not the most intelligent of the species that survive the longest, it is the most adaptable.*" This statement holds as well for pervasive computing applications. In order to be successful, applications need to adapt continuously to their environment and therefore require information from the environment for the adaptation to be effective. However, mobile devices have not benefited to the same extend as other high-end systems from the latest advancements in context toolkits, frameworks and architectures. The main reason is that the supporting infrastructure is not adapted to the capabilities and limitations of the mobile device. We identify the following issues:

- In many cases the design of the context framework only allows an all-or-nothing deployment without any support for adaptation whatsoever. There is no way to free up allocated resources for functions of the framework that are not required or not being used.

- The minimum resource requirements for deploying the context framework are often far beyond of what a mobile device can offer today. The dependencies on many heavyweight libraries for the parsing and processing of context information do not help either.

- Many context frameworks are not built with the limited autonomy of a mobile handheld in mind. Processing power and memory are becoming less of a problem, but energy consumption is still a bottleneck on mobile devices. Having a permanent wireless connection might not be a good idea on a mobile device.

In order to achieve a resource-efficient context-driven middleware for mobile handheld applications that overcomes the previously mentioned issues, we describe an application middleware that self-adapts in order to reduce its own resource consumption to a minimum. The middleware itself has a layered design and provides introspection and intercession capabilities within each layer to support a more flexible self-adaptation strategy at runtime.

In section 2 we discuss the deployment of our context-driven application middleware on a PDA and elaborate on how its component-based design combined with introspection and intercession are used to support behavioral and structural self-adaptation at runtime. In section 3 we detail how self-adaptation of the middleware is achieved in order to minimize resource consumption. In section 4 we conduct experiments to validate the strategies for resource driven context-awareness on a personal handheld device. Section 5 provides an overview of related work. We end with conclusions and future work in section 6.
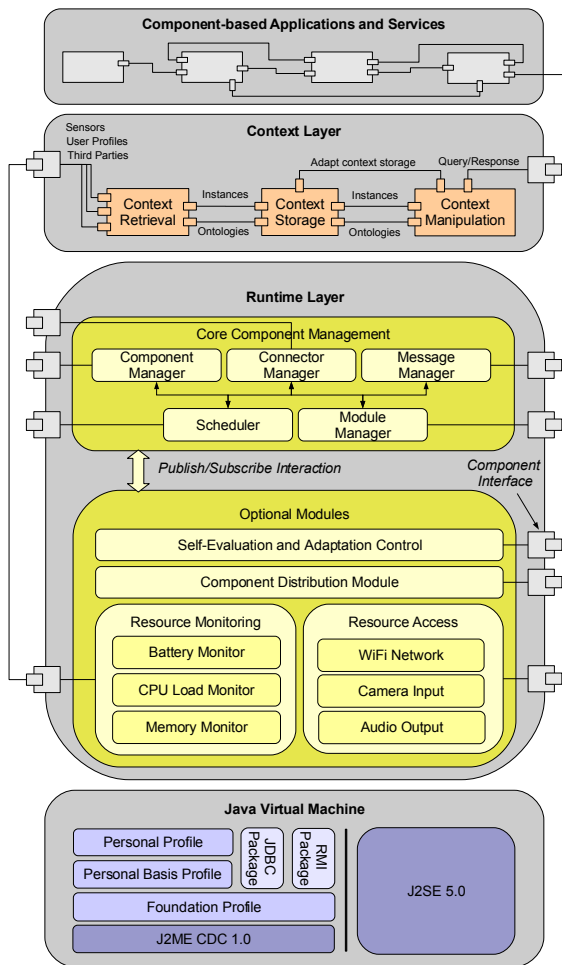
Figure 1: The context-driven application middleware for component-based applications

Figure 2: Building blocks of a component-based conferencing client

2. BEHAVIORAL AND STRUCTURAL SELF-ADAPTATION

We use *components* [9] as modular building blocks for the design and deployment of adaptable services [7] and context management [6]. The component paradigm makes it possible to add, remove or replace a component at runtime when the execution context changes. An overview of our Java-based context-driven application middleware for component-based applications is given in Figure 1. The layered architecture separates the applications, the management of context information and resources, the component runtime and the underlying Java virtual machine.

2.1 Self-adaptation in the application layer

Applications are modeled as a composition of mandatory, optional and alternative inter-connected components that send asynchronous messages to one another to cooperate. Figure 2 shows the composition of a conferencing client that supports Jabber-based instant messaging, web based document sharing and playing multimedia files.

Behavioral self-adaptation
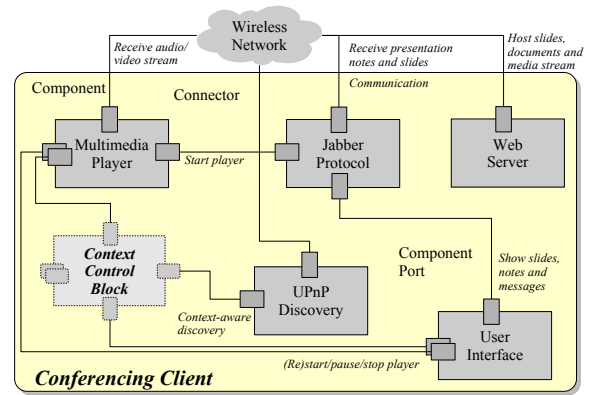Behavioral self-adaptation does not change the composition of the application, but changes the way an application behaves due to a changing context. Some examples:

- At home, the media player will automatically increase the volume when playing one of your favorite songs.
- During meetings the application will change the online status of the user and redirect incoming calls.

Context information is required to initiate non-intrusive behavioral self-adaptation: location awareness, user preferences, devices and resources. This information is delivered by the layer below, the *Context Layer*. The self-adaptation is based on the appropriate messages being sent from one component to another (e.g. to change the volume).

Structural self-adaptation
Structural self-adaptation is the ability of the application to reconfigure the composition of the application itself:

- Add a *Web Server* component for sharing files.
- Replace the *Jabber Protocol* component with another instant messaging protocol component.

Preconditions on the incoming messages trigger structural self-adaptation. For example, the *Jabber Protocol* component will replace itself with an alternative instant messaging component based on the address of the user.

2.2 Self-adaptation in the context layer

The *Context Layer* acquires and aggregates data from various sources and is self-adaptable. It can (un)load its own context managing components whenever appropriate. The following context managing components are available:

- **Input:** Encapsulate data gathering from sensors, user preferences and profiles, and other public databases.
- **Filter:** Filter out irrelevant or old data or compare the accuracy of different input components.
- **Persistency:** Provide a repository for context values as well as relationships between context concepts.
- **Transformation:** Translate a concept from one representation to another, e.g. *GPS position* to *city*.
- **Reasoning:** Combine known context concepts with derivation rules to derive new information.

A simplified overview is given in Figure 1 and shows three composite components for context retrieval (combining *input* and *filter*), storage (*persistency*) and manipulation (combining *transformation* and *reasoning*).

Behavioral self-adaptation
If the storage capacity of the context repository is limited so that not all context information can be retained for future use, then the *Context Layer* will self-adapt:

- Free up memory by purging old, rarely used or redundant key-value pairs and ontologies.

- Reduce the rate at which certain context concepts are sensed and retrieved.

Note that the same context concept can be stored multiple times in the context repository, as its value may be provided by different sources or at different times.

Structural self-adaptation
The *Context Layer* provides many optional components of the *input* and *transformation* type and will structurally self-adapt as follows:

- When a context-aware application requests specific context information, the *Context Layer* will add the appropriate components into the processing chain.

- If a context concept is no longer used, then all the relevant *input* and *transformation* context components are removed from the processing chain.

Obviously, when an application exits then the relevant context managing components are removed as well.

2.3 Self-adaptation in the runtime layer

The *Runtime Layer* provides the basic functions to deploy and run component compositions and acts as a hardware abstraction layer. Self-adaptivity takes place at this level to provide uniform access to platform specific I/O features, hardware resources and resource monitors. The *Runtime Layer* provides the following functions (see Figure 1):

- **Core Component Management:** Deploys and connects components, schedules and delivers asynchronous messages from one component port to another.

- **Self-Evaluation and Adaptation Control:** Evaluates and adapts to resource usage and controls on-demand activation of computational resources.

- **Resource Access:** Provide uniform access to platform specific I/O features (audio, backlight), persistency (hard disk, flash memory), communication (WiFi, Bluetooth, GPRS), power management, etc.

- **Resource Monitors:** Keep track of the CPU load, memory usage, disk space allocation, network bandwidth, battery status, etc.

Behavioral self-adaptation
Behavioral self-adaptation in the *Runtime Layer* mainly involves resource-awareness:

- Decrease the CPU clock frequency when no application is active or when the system is idle.

- Reduce the display backlight to extend the autonomy of the device when running on battery.

```
<componentstructure>
  <component type='composite'>
    <instance name='server' opt='Y' impl='WebServer.jar' />
    <instance name='jabber' opt='N' impl='Jabber.jar'
        alt='MSN.jar, Yahoo.jar' />
    <instance name='ui' opt='N' impl='GUI.jar'
    <instance name='player' opt='Y' impl='Player.jar' />
    ...

    <connection src='jabber/Control' dest='player/Control' />
    <connection src='jabber/Msg' dest='ui/IO' />
    <connection src='ui/Control' dest='player/Control' />
    ...

  </component>
</componentstructure>
```

Figure 3: Deployment descriptor of the *Conferencing Client* application specifying mandatory, optional and alternative components

- Increase concurrent behavior by changing the number of messages that can be processed in parallel.

Structural self-adaptation
The *Runtime Layer* provides access to several optional *monitors*, *I/O devices* and *hardware resources*. These modules are enabled on demand when required by the applications or the *Context Layer* and again disabled when no longer active and after a resource specific timeout.

3. RESOURCE-DRIVEN INTROSPECTION AND INTERCESSION

Applications are implemented as a composition of components and accompanied by a deployment descriptor specifying all the mandatory, alternative and optional components (see Figure 3). The *Runtime Layer* provides the means for introspection to find out which of the alternative components has actually been deployed. The state of a component is modeled explicitly as a list of state variables and their values, not only for inspection purposes, but also to support replacement of components at runtime while preserving the state. A component's state can be retrieved by sending a state requesting message to the dedicated *State* port of the component (not shown in Figure 2 for legibility reasons).

Behavioral self-adaptation is performed by sending messages to a particular component port. Structural self-adaptation is initiated within each layer independently. Each layer determines the state of its components and the new configuration and instructs the *Runtime Layer* to activate new components and/or to reconnect the composition. The old component state retrieved by introspection is forwarded to the new component's *State* port.

3.1 Resource-driven component deployment

The whole resource-driven component selection algorithm is entirely based on the processing of resource information. It consists of the following steps:

1. **Hardware** Process the hardware of the device to discover processing power, network capabilities, and other input providers, such as a GPS module.

2. **Resource-awareness** Request the current available resources on the device, including the CPU load, the memory and bandwidth usage, and the battery status.

```
@sense = {}
@subsume = {}
for each  a  in  @applications
    // Get for each application the context concept dependencies
    for each  c  in  a.concepts()
        if @sense subsumes  c  then
            @subsume += c
        else
            for each  s  in  @sense
                if c subsumes s
                    @sense -= s
                    @subsume += s
            @sense += c

@sensors = {}
for each  s  in  @sense
    // Sort all components sensing s by energy efficiency and
    // start using the first 'sensor' in the list
    @sensors[s] = s.components().sort('energy')
    @sensors[s].lastused = 1

for each  s  in  @sense
    idx = @sensors[s].lastused
    @value[s] = @sensors[s][idx].sense()
    r = random()   // Returns a random value in [0.0..1.0]
    // The sensor accuracy to measure s goes from [0.0..1.0]
    if r > @sensors[s][idx].accuracy then
        v = @sensors[s][idx+1].sense()
        if @value[s] = v then
            v = @sensors[s][idx-1].sense()
            if @value[s] = v then
                @sensors[s].lastused -= 1
        else
            @value[s] = v
            @sensors[s].lastused += 1

for each  s  in  @subsume
    @values[s] = @values.subsume(s)
```

Figure 4: Resource-driven context component selection

3. **Context dependencies** Check which context information is required by the active applications for personalization and non-intrusiveness behavior.

4. **Context component selection** Determine for each context concept the minimum resolution and the resource requirements, and select the most energy efficient one as outlined in the algorithm.

5. **Application component selection** Determine a feasible application composition based on its components' resource requirements, and select the most resource efficient ones.

The algorithm in Figure 4 outlines the approach that is being used to retrieve all the contextual concepts that the applications have requested and choose the most energy efficient context components among the alternatives that fulfill the context dependencies of the applications. The algorithm ensures when multiple applications request similar concepts but with a different resolution (e.g. city and office floor) that only the most fine-grained context concept is sensed. The other contextual concepts are retrieved by subsumption, using ontologies to provide the requested information in the right format (country, city, zip code, address, building, etc.). Subsumption is less expensive than sensing similar concepts multiple times. Note that the algorithm has been simpli-
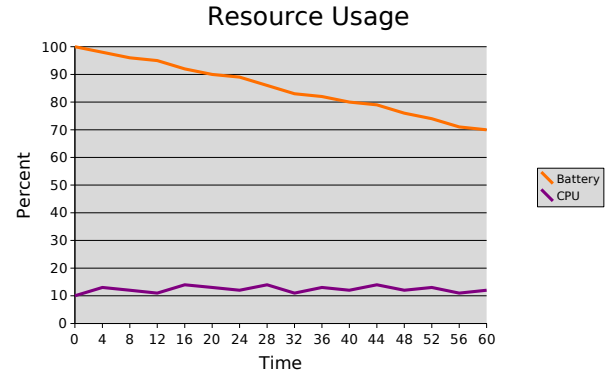

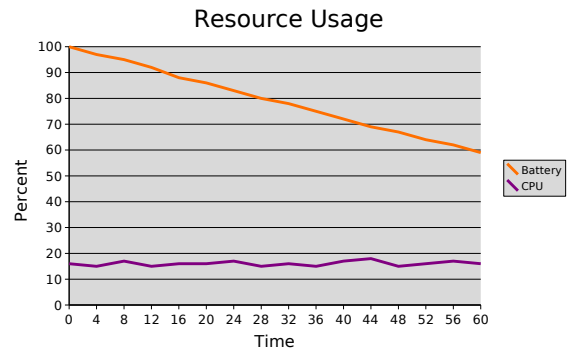**Figure 5: Resource usage for GPS-based location-awareness.**

Figure 6: Resource usage for WiFi signal strength-based location-awareness.

fied and does not include any checks on boundary crossing whatsoever.

Also note that the sorting on energy efficiency can only be done at runtime. For example, using WiFi RSSI triangulation for positioning is very cheap if wireless network access is already required for a particular context-aware application.
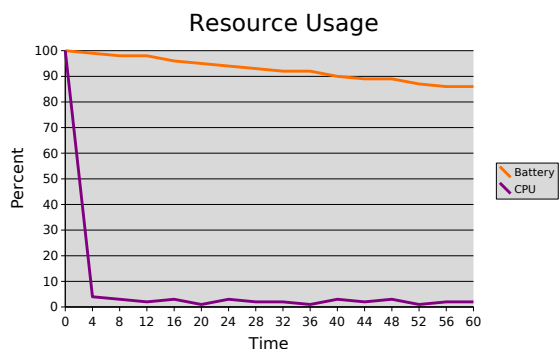
4. PERFORMANCE EVALUATION AND EXPERIMENTAL VALIDATION

Our current mobile test platform, the Qtek 9090 handheld, is equipped with several wireless communication protocols (WLAN 802.11b, GPRS, Bluetooth, Infrared irDA) for information retrieval. In our test setup we also use a Bluetooth-enabled external GPS module (GlobalSat 338 Bluetooth GPS) for outdoors location-awareness. This module has a default operation time in continuous mode of 17 hours after having been fully recharged, and more that 20 hours when power saving is enabled. The energy consumption on the PDA only reflects the Bluetooth communication to read the GPS receiver's NMEA-0183 [5] output containing *time, longitude, latitude, speed* and *direction* information.

Before the experiment we profiled several location-sensing context components for their energy efficiency and CPU usage. The difference in memory consumption was negligible. Table 1 provides an overview of the characteristics and the profiled resource requirements of the different location sens-

Locating Technique	Type	Reference	Sensing Accuracy	Limitation	Sensing Frequency	Battery Usage	CPU Load
Bluetooth GPS	physical	absolute	5-10 m	outdoors	1 sec	30% / h	12 %
WLAN RSSI	physical	relative	< 10 m	indoors, ≥ 3 AP	1 sec	41% / h	17 %
WLAN MAC	physical	relative	< 100 m	≥ 1 AP	30 sec	38% / h	13 %
IP Address	symbolic	absolute	country/city	network access	4 min	34% / h	5 %
Personal Agenda	symbolic	absolute	user		4 min	12% / h	2 %

Table 1: Characteristics and resource requirements of different location sensing techniques

Figure 7: Resource usage for PIM-based location-awareness.

Figure 8: Performance evaluation for the WiFi and GPS-based scenario.

Figure 9: Performance evaluation for the adaptive resource-aware scenario.

ing techniques. To provide a fair comparison, the sensed values had all to be converted to the most accurate symbolic representation. The performance results therefore include the transformation from one location specification (address, geographical coordinates, city, country) to another. The CPU load in the last column depends on:

- The amount of processing to read the sensor data.
- The enhancing of the physical coordinates to the corresponding symbolic location information.
- The sensing frequency, which depends on the inherent precision and accuracy of the technique.

Figures 5, 6 and 7 illustrate the CPU load and the battery consumption for the GPS, WiFi and PIM-based location-awareness technique respectively over a period of 60 minutes. Obviously, the power consumption for WiFi-based techniques is the highest. However, if WiFi network access is already used for the IM application, then the extra overhead is negligible. Also, for the GPS-based technique the battery usage does not include the power consumption of the external Bluetooth GPS module. This module has its own battery which uses about 5 to 6 % per hour of its total capacity. The least expensive method from a power and CPU perspective is the PIM-based technique (using the current time and entries in the personal agenda) as it requires no wireless network access and already provides the information in the most appropriate form for the user. However, this method can only be used when the agenda specifies an event for which location information is present. The high peak for CPU usage in the beginning is the result of parsing the personal agenda.

In the experiment, a test person carried around two identical PDAs during a whole day, going from home to the office and back. Each PDA was running the application mentioned
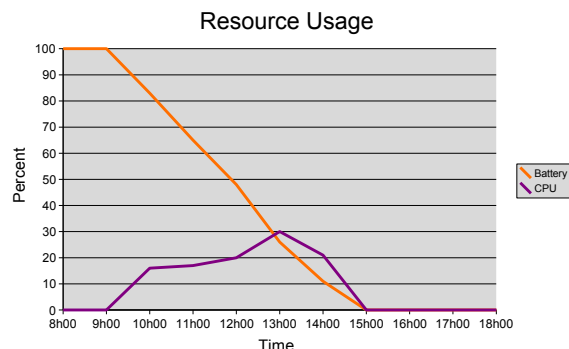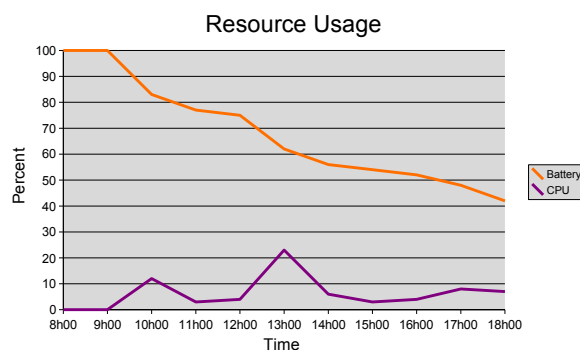
in the introduction. Location-awareness is used to automatically set the volume of the media player and set the online status of the user if connected to the Internet. One of the PDAs was continuously using the WiFi triangulation and GPS methods for indoors and outdoors location-awareness, whereas the second PDA used the adaptive context-driven resource-aware approach. Neither of both PDAs was turned off or suspended during the experiment. The results of the performance evaluation are shown in Figures 8 and 9. The CPU load shown here is averaged over small time intervals (therefore no high peaks of short duration are shown) and includes the load of all applications, including the *Context Layer* and *Runtime Layer*. The results in the first figure show that the first PDA did not make it through the day. The PDA ran out of battery power around 15h00. The second PDA in Figure 9 experienced an overall lower CPU load and had plenty of time left to recharge at the end of the day.

5. RELATED WORK

In recent years, many researchers have addressed the issue of context-awareness and self-adaptive middleware. This research has greatly improved our knowledge of how context-aware and self-adaptive middleware can accommodate to new functionality requirements, performance optimization, variable runtime conditions and changing environments. Providing a detailed review of the state of the art on context-awareness and self-adaptability is beyond the scope of this paper. Instead, we focus on those contributions that are most related to the work presented in this paper.

Cakmakci et al. [1] propose statistical modeling of raw sensor data to provide context-awareness in system with limited resources. Their approach exists in keeping the data processing closer to the sensors in order to save power and is able to distinguish simple contexts at this level. Their approach is fundamentally different compared to ours, as the context information is tightly coupled to the device and the application. Our approach is more complex and uses more resources, but adds layers of abstraction to manage context information in a uniform and interchangeable way.

Puppeteer [2] is a system for adapting component-based applications in mobile environments. Compared to our work Puppeteer also takes advantage of the component-based nature of the applications to perform adaptation. The difference with our work is that their goal is to adapt applications in order to achieve reductions in user-perceived latencies in two office applications without modifying the applications themselves.

The BBN QuO system [4] also supports applications that adapt to resource variability allowing users to define operation regions. The runtime systems monitors the application and execution environment and activates application handlers when the application changes operation region. Compared to our work, context-awareness is not being considered for application adaptation.

Sousa et al. [8] compare their work with other systems that adapt their computing strategies in reaction to bandwidth, memory, CPU and power variation. They claimed their work is an improvement as it was not restricted to adaptation of one component at a time, but they also tackled multi-component integration, configuration and reconfiguration. Our work also deals with multi-component reconfiguration, but subdivides these components into three layers of abstractions to separate self-adaption concerns into layer specific reconfigurations.

6. CONCLUSIONS

This paper presents a resource-aware and context-driven application middleware for mobile devices. The modular design and the small resource footprint of the middleware allow deployment on mobile devices with limited resources and autonomy. The number of deployable components depends on the available resources. The necessary flexibility for deployment on such devices is achieved through self-adaptation in each of the layers of the application middleware. Runtime adaptation ensures a resource efficient deployment of components. Experiments have shown that our self-adapting application middleware can more than double the autonomy of a mobile handheld device by intelligently adapting the approach for location-awareness depending on the resolution required by the applications. The result is an increased au-

tonomy of the handheld device and a better resource usage trade-off between the application middleware and the regular applications.

Ongoing work looks at how to make use of resource rich systems in the environment of the mobile device by distributing applications or single components to other devices. However, this requires a clear view on the message flow of each components as a good trade-off needs to be found between, for example, memory and CPU consumption on the one hand and network bandwidth and energy efficiency on the other hand.

Future work will focus on the modeling of resource requirements for other indoor and outdoor locating methods and integration of other positioning systems into our context middleware. The major focus in this paper with respect to resource usage was on battery lifetime and CPU load. This will be further enhanced by modeling the interdependencies of CPU load, the battery lifetime, the memory and bandwidth usage. This should result in a better trade-off when relocating context information and processing components from one system to another, as relocating data also has an effect on the CPU load, the autonomy of the device, and the overall quality of service of the regular applications.

7. REFERENCES

[1] CAKMAKCI, O., COUTAZ, J., LAERHOVEN, K. V., AND GELLERSEN, H. Context awareness in systems with limited resources, 2002.

[2] DE LARA, E., WALLACH, D. S., AND ZWAENEPOEL, W. Puppeteer: Component-based adaptation for mobile computing. In *USITS* (2001), USENIX, pp. 159–170.

[3] DEY, A. K. Understanding and using context. *Personal Ubiquitous Comput. 5*, 1 (2001), 4–7.

[4] LOYALL, J. P., SCHANTZ, R. E., ZINKY, J. A., AND BAKKENL, D. E. Specifying and measuring quality of service in distributed object systems. In *The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (1998), p. 43.

[5] NMEA. The NMEA 0183 Interface Standard. http://www.nmea.org/pub/0183/, 2003.

[6] PREUVENEERS, D., AND BERBERS, Y. Adaptive context management using a component-based approach. In *Proceedings of 5th IFIP International Conference on Distributed Applications and Interoperable Systems* (June 2005), vol. 3543 of *LNCS*, Springer, pp. 14–26.

[7] PREUVENEERS, D., AND BERBERS, Y. Automated context-driven composition of pervasive services to alleviate non-functional concerns. *International Journal of Computing and Information Sciences 3*, 2 (August 2005), 19–28.

[8] SOUSA, J. P., POLADIAN, V., GARLAN, D., AND SCHMERL, B. R. Capitalizing on awareness of user tasks for guiding self-adaptation. In *CAiSE Workshops (2)* (2005), J. Castro and E. Teniente, Eds., FEUP Edições, Porto, pp. 83–96.

[9] SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming, 2nd edition.* Addison-Wesley and ACM Press, 2002.

[10] WEISER, M. The world is not a desktop. *Interactions* (Jan. 1994), 7–8.