An Optimization of Context Sharing for Self-adaptive Mobile Applications

Nearchos Paspallis and George A. Papadopoulos

Department of Computer Science, University of Cyprus
P.O. Box 20537, Postal Code 1678 Nicosia, Cyprus
{nearchos,george}@cs.ucy.ac.cy

Abstract. Because of the high potential of mobile and pervasive computing systems, there is an ongoing trend in developing applications exhibiting context awareness and adaptive behavior. While context awareness guarantees that the applications are aware of both their context and their own state, dynamic adaptivity enables them to react on their knowledge about it and optimize their offered services. However, because in pervasive computing environments there is also a need for enabling arbitrary synergies, such a behavior also requires appropriate algorithms implementing the adaptation logic required to reason on the sensed context and dynamically decide on the most appropriate adaptations. This paper discusses how utility function-based approaches can use context-awareness for that and, additionally, it shows how the decision-making process is improved with respect to both performance and resource consumption by using a more intelligent approach.

Keywords: Self-adaptive, Context-aware, Optimization, Mobile computing.

1 Introduction

Today, one can observe an ever increasing trend in the use of mobile systems and applications which are used to assist us with our everyday tasks. As these applications become more ubiquitous, developers are faced with both opportunity and challenge. Adaptive, mobile applications are designed to constantly adapt to the contextual conditions in an autonomous way, with the aim of optimizing the quality of their service. The complexity of self-adaptive software though, renders their development significantly more difficult. As Paul Horn has quoted in IBM's manifesto of autonomic computing [1], tackling the development complexity, which is inherent in modern autonomic systems, is the next grand challenge for the IT industry.

When aiming complicated, autonomous and adaptive software, one of the most important hurdles is to provide suitable software engineering methods, models and tools, to ease the development effort. Current approaches aim to achieve this by using architectural [2] and modeling tools [3]. Other approaches propose development methodologies such as the separation of the functional from the extra-functional concerns in the design and development of adaptive, mobile applications [4].

A. Bourgeois and S.Q. Zheng (Eds.): ICA3PP 2008, LNCS 5022, pp. 157–168, 2008.

Abstracting adaptive, mobile applications with compositions of individual and reusable components [5] offers many benefits, including the opportunity to delegate part of the adaptation responsibility to a different layer (middleware). This paper discusses the proactive and reactive approaches for sharing context information with the purpose of achieving distributed adaptation reasoning. Furthermore, it proposes an optimization which is shown to significantly improve distributed context-awareness in terms of number of needed message exchanges.

The rest of this paper is organized as follows: Section 2 introduces the basic terms of context-awareness and adaptation reasoning. Then, Section 3 presents a basic approach to adaptation reasoning and proposes an optimization, aiming at minimizing the number of context change messages to be communicated. Then, Section 4 describes a case study scenario and validates some of the approaches proposed in the previous section, and Section 5 discusses related work. Finally, the paper concludes with Section 6 which presents the conclusions and points to our plans for future work.

2 Adaptation Enabling Middleware

Often, applications featuring context-awareness and adaptivity, exhibit a common pattern: context changes are monitored and evaluated against the possible adaptation options so that the optimal choice is dynamically selected. This pattern naturally leads to the attempt of encapsulating and automating much of these tasks, in the form of appropriate middleware tools. The *Mobility and Adaptation-enabling Middleware* (MADAM) project [6] has aimed at providing software developers with reusable models and tools, assisting them in the design and implementation of adaptive, mobile applications. To facilitate the reusability of adaptation strategies, a middleware layer was proposed which can be used to encapsulate context monitoring, adaptation reasoning logic and reconfiguration tasks. Building on MADAM's legacy, the *Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments* (MUSIC) project [7] envisions to improve the results of MADAM and also to extend the application domain from mobile to ubiquitous computing.

As illustrated in Fig. 1, the middleware layer can serve by automating three basic functions: First, it monitors the context for changes and notifies the adaptation logic module when a relevant change occurs. Second, it reasons on the context changes and makes decisions about which application variant should be selected (different application variants refer to different component compositions providing the same functionality with different extra-functional properties). This step typically includes the dynamic formation of all possible application variants, as they are defined by corresponding component metadata. Finally, when an adaptation is decided, the configuration management instructs the underlying component framework to apply it (i.e. it reconfigures the application by setting adjustable parameters and by binding or unbinding the involved components and services).

The adaptation reasoning refers to the process where a set of possible variants are first formulated, based on the composition plans provided by the application [2], and then evaluated with the aim of selecting the adaptation which optimizes the utility for the given context. This process is triggered by changes to the context, which in this case includes the user context (preferences, activities, state, mood, etc), the computing

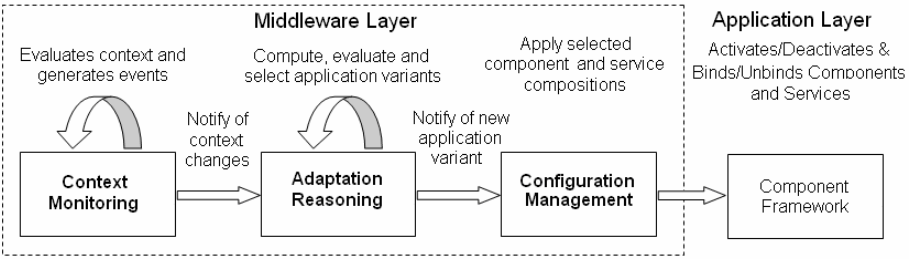

Fig. 1. High-level structure of a typical context-aware, adaptation-enabling middleware

context (devices, networks, UI options and capabilities, available composition plans and services, etc) and the environment (location, weather, light and noise, etc).

The adaptations are performed at the application layer, where different components and services can be interchangeably replaced (or [re]connect to each other in different configurations) in order to form different variants of the application. Although these variants are assumed to be characterized by different extra-functional properties, they are nevertheless assumed to offer the same functional service. This results in different application variants, which can offer different levels of *Quality of Service* (QoS) depending on their contextual conditions. To enable self-adaptation, these variants are then evaluated (e.g. using utility functions) and the optimal one is selected (for an example see [8]).

Assuming a centralized system, the decisions are taken locally (i.e. no networking interactions are required), and the decided application variants are limited to non-distributed ones. However, while the lack of networking requirements improves the system's robustness, it also prevents it from exploiting the opportunities arising when distributed compositions are available. More particularly it misses the opportunity of supporting distributed compositions, which allow hosts to better exploit resources and services offered by other hosts. This is particularly important in mobile and pervasive computing environments where frequent context changes and scarce resources render the exploitation of distributed resources extremely useful. For instance, a mobile device is enabled to delegate processor-intensive tasks (such as text-to-speech) to appropriate server nodes, thus better utilizing the globally available resources.

This paper discusses a basic approach which allows for distributed decision-making and distributed compositions (i.e. applications comprising of components residing on distributed nodes). The approach builds on the basic architectural-based model for runtime adaptability, as it is described by Floch *et al* [2].

2.1 Problem Description and Requirements

In its simplest form, a centralized architecture can be designed so that it supports the adaptation of a single, non-distributed application. The composition plans specify a set of possible variants, which are all evaluated whenever a relevant context change is sensed. A natural evolution of this approach is the support of composition plans where some of the components are allowed to be distributed. This implies the definition and use of distributed composition plans, i.e. plans defining compositions where some of the components are possibly deployed on distributed nodes.

Furthermore, an additional relaxation of the original form refers to the case where the distributed variants are formed and decided on a single central node or they are formed and decided in a distributed way. The latter approach is of course significantly more complex as it requires mechanisms to enable the nodes to reach trusted and fair agreements. For the latter, it is assumed that the common point of reference is the *utility*, as it is perceived by the end user [8].

In this context, the dynamic adaptation reasoning problem can be defined as the requirement for models and algorithms which can be used for the dynamic selection of the most suitable variant. In this case, the suitability of a variant refers to its fitness to the user needs, as it is measured by the *utility* offered to the end user. The next section discusses a straightforward approach for enabling adaptation reasoning and an optimization which minimizes the number of messages required to be communicated for distributed context sharing.

3 Adaptation Reasoning

We consider the case where adaptive, component-based applications are defined as collections of software components which can be configured to derive a number of variants according to a set of composition plans. These components are defined as self-containing modules of code, which can communicate with each other through a set of ports. In practice, many systems use computational reflection to reason and possibly alter their own behavior, as well as component-orientation to "*allow independent deployment, and composition by third parties*" [5].

The composition plans are defined at design time and they are used to dynamically construct different variants of the application. Individual variants are designed so that they offer an advantage (such as better resource utilization) compared to the others in varying context. Naturally, each variant is designed with the aim of maximizing the utility of the application for at least a subset of the context space.

In autonomic systems, the possible approaches for making adaptation reasoning are classified to *action-based*, *goal-based* and *utility function-based* [10]. In this work we consider the use of utility functions for two reasons: First they facilitate scalability and, second, they support dynamically available, arbitrary components.

Utility functions are simple computational artifacts which are used to compute the utility of an application variant: i.e. a quantifiable scalar value, reflecting the *utility* perceived by the end user. In this respect, the overall objective of the middleware can be defined as "the continuous evaluation of all possible variants with the aim of always selecting the one which maximizes the utility offered to the end user".

Assuming there is only a single application which is managed by the middleware the utility function can be implemented as a function which maps application variants, context conditions and user preferences to scalar values, as depicted in the following:

$$f(\mathrm{p},\mathrm{c}): (\mathrm{p}_1, \mathrm{p}_2, \ldots, \mathrm{p}_N) \cdot (\mathrm{c}_1, \ldots, \mathrm{c}_M) \rightarrow [0,1] \quad (1)$$

In this formula, the "$p_1, \ldots, p_N$" values correspond to the available variants, and the "$c_1, \ldots, c_M$" values correspond to the possible points in the context space (this includes the user preferences, as part of the user context). In other words, the utility function is used to map each combination of a composition plan and context condition to a scalar

number (typically in the range of [0, 1]). As this definition indicates, all the parameter types are subject to change. Thus, the aim is to always select a composition which maximizes the utility. The evaluation process is triggered whenever any of the arguments (i.e. context and available variants) changes.

Although it is assumed that the computed scalar utility reflects the benefit as that is perceived by the user, there are currently no general methods which can guarantee the precision of such an assignment. Rather, approaches such as the one used in the MADAM project [6] simply encourage the assignment of utilities to components and composition plans in an empirical manner (i.e. using the developers' intuition). When the application is sufficiently complex, there is no straight-forward method or approach which can guarantee that there is a perfect (or even close) match between the computed utility value and the actual user desires. Nevertheless, it is argued that constructing utility functions in an empirical manner, in combination with experimental evaluation, can result in reasonable solutions with moderate effort.

Finally, as it is evident from the definition of the utility functions, the performance of the selection process is inversely proportional to the number of possible variants. Naturally, the adaptation reasoning becomes less efficient as the number of composition plans increases. This becomes more evident with larger, distributed applications featuring large numbers of possible variants, especially as this number typically increases exponentially with the number of used components.

3.1 Developing Applications with Compositional Plans

In order to be able to define applications in a dynamic, compositional way, a recursive approach is defined as follows: The primary modeling artifacts defined, are the *component types* and the *component implementations*. A component type can be realized by either a component implementation, or by a well-defined composition of additional component types (i.e. a composite component type). The latter enables the dynamic formation of alternative compositions in a recursive manner (in this case the recursion ends when all the component types have been assigned to either a composite component type or to an actual component implementation). The application is defined by an application type, which is itself a component type.

Additionally, the composition plans are predefined (i.e. at development time rather than at runtime). For instance, the model which is defined in [3] specifies how to construct different composition plans (and thus variants) for an application, and thus it aims at the developers rather than the runtime system. The latter uses the composition plans to dynamically compose the possible variants during the evaluation phase.

The applications are also defined in a recursive manner: for each step, of which a new layer is defined, specifying how the abstract component type is implemented. Always, the first layer is a layer with a single composite component type, abstracting the whole application. Depending on whether the application interacts with other applications or not, the first layer includes a composition plan with possibly some input (dependency) and some output (offered) services (or ports in component-orientation terminology). Subsequent layers expose further details of the composition plan by specifying additional component implementations and component types. The recursion ends when a layer is reached where all component types are fully resolved with component implementations.

Evidently, variability is enabled by allowing the use of various alternatives for particular component types. Each such alternative adds to the total number of possible variants. During the adaptation process, all possible variants are computed with the purpose of being evaluated.

3.2 Adaptation Reasoning

A centralized adaptation reasoning approach implies that the decisions are taken locally, and that no negotiation with other peers is required [11]. On the other hand, a distributed approach allows coordination between the collaborating peers, thus allowing the proposition and agreement of mutually accepted decisions.

A typical centralized implementation, triggered by context changes, is expressed by the following pseudo-code:

```
1. Detect a relevant context change
2. For all application variants (including distributed
   ones), compute the utility value for the new context
3. If the optimal variant is different from the current
   one, then adapt (reconfigure the application)
```

First, the adaptation reasoning is triggered by a relevant context change event. In this case, the relevance is computed by analyzing the utility functions of the deployed applications and extracting the context types which affect their outcome. The next step simply iterates through all possible variants and computes their utility value. The last step evaluates the computed values and selects the variant which maximizes the utility. If that variant is different from the one already selected, then an adaptation occurs by applying the new, optimal variant. Although not shown in this algorithm, another optimization would be to evaluate *how much* does the newly selected variant improves on the current one. If the margin is too small, then it is usually better to skip the adaptation, as it typically incurs additional overhead cost (i.e. for reconfiguration). Ideally, the exact cost of each prospective adaptation should be taken into account when selecting on the reaction to a context change. However, when distributed context sharing is considered, the context change events can be distributed, which implies a higher cost for each message in terms of resources.

We assume an approach where the adaptation managers directly consult their corresponding context managers (instead of their remote adaptation manager peers), which subsequently provide them with access to the information that is required to assess all the possible application variants, including the distributed ones. In this way, the best variant can be efficiently selected and applied. In [16], two main strategies were discussed for optimizing the communication between the distributed devices: First, a *proactive* strategy which aims at communicating as much information as soon as it is available. In practice, with this strategy the nodes are always aware of as much context information as possible, which as a result minimizes the response time at the cost of increased messages communications carrying the required context updates. Alternatively, a *reactive* strategy aims at minimizing the number of communicated messages at the cost of slower reaction time. This strategy activates the adaptation reasoning process only when a context change is sensed, which subsequently triggers the exchange of all relevant context changes from the participating peers. This results

in less message communications of context events at the cost of increased response time. Hybrid approaches are also possible, one of which is presented in this paper with the purpose of achieving both minimal communication of messages and quick response times.

3.3 Optimizing the Adaptation Reasoning through Context Management

As it was argued in the previous subsections, adaptation reasoning can be solely based on offering component types and assigning a utility value to them (which on the client side appears as cost). Thus, practically, the distributed aspect of adaptation reasoning can be implemented exclusively through the use of appropriate context distribution mechanisms, facilitating the exchange of needed context data among the collaborating nodes. This subsection discusses an approach for optimizing distributed adaptation reasoning in the form of minimizing the number of messages required to be exchanged as a result of context change events.

Typically, the context management systems inform their peers about the subset of context data they are interested in, which as a result triggers a distributed context change event whenever a relevant change is detected. For example, if node A is interested in context elements "$c_1, \ldots, c_P$", which are not locally available but are offered by a peer node B, then node A can simply register for it. For example, this would occur if node A had no local sensors available for that particular context type, while some of its applications depend on it [9].

Naturally, the straight-forward approach includes node A sending an update message to node B every time *any* context change occurs to the registered context elements. However, this would be unnecessary, as not all context changes have a potential of causing an adaptation. Assuming that the two nodes share a copy of the relevant utility functions, then a natural optimization would be for node A to ask node B to further process context changes, and filter out any context change messages that are unlikely to cause an adaptation, before communicating them to A.

Of course, this also implies that node B will go through the same evaluation process for all possible variants as node A would, which as described earlier can be a quite heavy process, especially for a mobile device. However, it is argued that this process can still offer significant benefits with regards to resource usage. Assuming that the serving node is sufficiently powerful, it is expected that the gain of minimizing the communicated context messages dominates the cost of processing and filtering context change events.

4 Case Study Example and Experimental Evaluation

As a means of better illustrating the use of the optimization approach described in Section 3.3, this section describes a case study example and also provides an evaluation which arguably validates its potential. The gathered results are based on simulations and aim at identifying and measuring the improvements that could result from the application of the proposed approaches. Further details such as the actual overhead incurred when making a decision is not discussed, but nevertheless the primary

objective of this evaluation is to illustrate that performance can be improved by showing that the number of required context coordination messages is reduced.

In this respect, we have revisited the scenario discussed in [11], which describes an application used by onsite workers for assisting them into performing their everyday tasks. This application offers three primary modes of operation: Visual UI interaction, Audio UI interaction with local Text-to-Speech (TTS) and Audio UI interaction with remote TTS (illustrated in Fig. 2). Each of these modes is optimized for offering the best quality to the user under different context conditions. For the purposes of evaluating the context and selecting the optimal mode, the following property predictors and utility function have also been defined, as illustrated in Fig. 3.

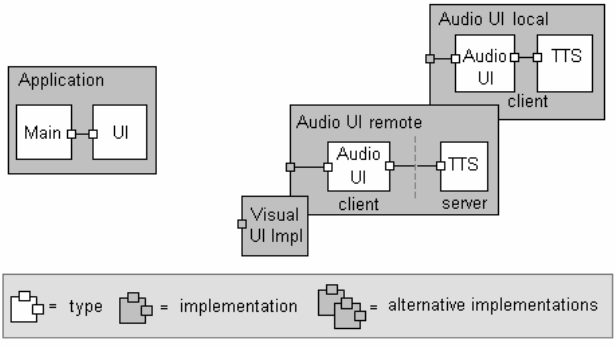

Fig. 2. The compositional architecture of the case study scenario

The composition of different variants is achieved through the exploitation of the offered component types and component implementations, as shown in Fig. 2 and discussed in Section 3.1 and Section 3.2. This figure illustrates the composition of a simple application. At the highest abstraction layer, an application consists of a single component type, which in this case is named *Application*. This component type is composite, and thus describes its architectural composition as the simple binding of two component types: *Main* and *UI*. The first is assumed to be an atomic component implementing the main application's logic, while the latter is assumed to be a component providing UI functionality. Although not depicted in this figure, the main component type is provided by a component implementation. The UI component type, however, is further decomposed in three possible variants: The first one is provided by a single atomic implementation, namely the *Visual UI*. The second and third are equivalent in terms of architecture (an *Audio UI* component type bound to a Text-to-Speech or *TTS* component type), but differ in their deployment plan as in one case the *TTS* component type is deployed *locally*, while in the other case *remotely*. Subsequent layers specify that the *Audio UI* and the *TTS* component types are provided as single, atomic component implementations (not shown in Fig. 2).

Given this composition plan, a utility function was also defined, along with a set of property predictors, which are used to dynamically evaluate the utility value for each possible variant, and for specific context values. In this case, we consider three simple context types only: *bandwidth* which refers to the available network bandwidth (as a

percentage), *response* which corresponds to the user's need for quick response, and *hands-free* which corresponds to the user's need for hands-free operation. The bandwidth and response context properties are constrained to numeric values in the range [0, 100], and the hands-free property is constrained to false or true values only. The exact configuration of the property values and the property predictors for each of the three variants is depicted in Fig. 3.

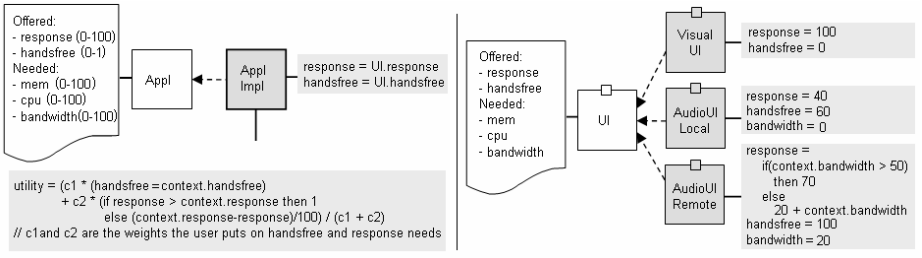

Fig. 3. The left side illustrates the application's dependence on the *response* and *hands-free* properties. It also describes the definition of the *utility function*. The right side, illustrates the three possibilities for implementing the UI role, which comprise the three primary modes of operation for the application. The utility of the latter is defined using a *property predictor*.

Given these metadata, and a set of context property values, one can compute the utility of any variant. However, not all context changes can affect the selected variant, i.e. a transition in the value of a context property does not imply that an adaptation will be triggered. It is exactly this fact that it is exploited in the optimization approach defined in Section 3.3. In order to validate its usefulness, we used this example and computed the ratio of context changes that could potentially trigger an adaptation.

Table 1 shows the results of our evaluation, which was performed as follows: First, different domains for the values of each of the three context properties were defined: the *bandwidth*, the *response* and the *hands-free*. In this case, the bandwidth value-set of (0:10:40) implies that the bandwidth is simulated with all values between 0 and 40 with a step of 10. Next, for each of these context combinations, we computed the number of different context settings that favor the use of each of the three possible variants. Then, the adaptation probability is computed as follows: It is assumed that each context setting corresponds to a different node in a fully connected graph. Furthermore, each of the graph nodes is colored with one of three colors, based on the variant which optimizes the utility for that context setting. Finally, the probability is computed by assuming that any context change can occur with the same probability, and by counting the number of node-transitions that cause an adaptation (i.e. connect nodes of different colors).

Equivalently, the probability for switching across different variants can be computed using the following probability formula:

$$p = p(A_1)\cdot(p(B_2)+p(C_2))+p(B_1)\cdot(p(A_2)+p(C_2))+p(C_1)\cdot(p(A_2)+p(B_2)) \quad (2)$$

Table 1. Adaptation evaluation outcomes as a result of different context settings; the last column depicts the probability that a context change can potentially trigger an adaptation

Bandwidth	Response	Hands-free	iVisual UI	Audio UI Loc	Audio UI Rem	Adaptation Probability
0:10:40	0:25:100	false:true	20	18	12	65%
0:10:40	0:20:100	false:true	30	18	12	62%
0:20:100	0:25:100	false:true	24	28	8	60%
0:20:100	0:20:100	false:true	36	28	8	58%
40:10:80	0:20:100	false:true	30	30	0	50%
40:10:80	0:25:100	false:true	20	30	0	48%

In this formula, the probabilities $p(A_i)$ refer to the probability for the corresponding event at step *i* (i.e. selecting the variant at step *i*). For instance, the probability for a change is equal to the sum of probabilities where the current variant is either A, B or C (*i=1*) and the next variant is one of the other two variants (*i=2*).

As it is shown in Table 1, a context change does not always imply an adaptation. Actually, the probability for an adaptation ranges from 65% down to 48% for the given scenario. The columns of the three variants illustrate the number of configurations for which that variant is *optimal*. The main lesson from this evaluation process is that when the distributed nodes coordinate at the context sharing level, the number of messages required for coordination can be significantly reduced (in this example by more than 50%). Notably, this experiment has assumed that the context properties were identical in both nodes (i.e. both devices refer to the same notion of bandwidth, response and hands-free requirements). Finally, the constants of the utility function were tuned to C_1=80 and C_2=20 respectively (see utility function in Fig. 3).

5 Related Work

There is a substantial amount of literature on adaptive, mobile systems. A very good description of composite adaptive software is provided by *McKinley et al* in [13]. This paper studies many basic concepts of adaptation, such as how, when and where to compose. One statement in this work is that the main technologies which are required for supporting compositional adaptations are Middleware, Separation of Concerns (SoC), Computational reflection and Component-based design. This is in agreement with the spirit of this paper. Applications are expressed in components, and SoC is achieved by defining utility functions which express the adaptivity properties of the compositions. Architectural reflection is used for enabling the actual reconfigurations required for adaptivity and a middleware is assumed in the background, collecting the distributed context management and distributed adaptation reasoning functionalities.

Another approach for enabling adaptivity from the coordination community is LIME, which enables coordination by means of logical mobility as it is described in [14]. In this case, the mobile hosts are assumed to communicate exclusively via transiently shared tuple spaces. LIME offers decoupling both in space and time and allows adaptations through reactive programming, i.e. by supporting the ability to react to events.

The Aura project [15], which built on the legacy of the Odyssey and Coda projects, also describes a relevant approach. Aura targets primarily pervasive applications. For this reason it introduced auras (which correspond to user tasks) as first class entities. To this direction, the same project categorizes the techniques which support user mobility into: use of mobile devices, remote access, standard applications (ported and installed at multiple locations) and finally use of standard virtual platforms to enable mobile code to follow the user as needed.

Unlike the existing literature, the approach which is described in this paper aims for self-adaptive applications which are constructed and dynamically adapted using architectural models. Additionally, this approach builds on previous work which described two alternative strategies for distributed adaptation reasoning: *proactive* and *reactive* approach [16]. Both of these offered significant advantages, depending on the deployment environment. However, the hybrid strategy proposed in this paper enables distributed adaptation reasoning merely through distributed context management. Furthermore, it combines benefits from both the reactive and proactive strategies, to achieve better results in terms of required communicated messages and response time, something that is illustrated and validated through the description and the examination of a case study example.

6 Conclusions

In this paper we have examined the problem of distributed context management and adaptation reasoning, and we proposed an approach for overcoming it. Building on two previous approaches, namely proactive and reactive adaptation reasoning, we proposed a hybrid approach which aims at optimizing the number and timing of communicated context change messages. This approach was illustrated and validated through a case study example, which highlights its potential.

In the future, we plan to investigate further approaches which can enable agile and efficient adaptation reasoning for distributed computing environments. Furthermore, we aim at further studying the relationship between distributed context-awareness and distributed adaptation reasoning, and propose approaches which further challenge it.

Acknowledgments. The authors would like to thank their partners in the MUSIC-IST project, and acknowledge the partial financial support provided to this research by the European Union (6th Framework Programme, contract number 035166).

References

1. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM Corporation (2001), http://www.research.ibm.com
2. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using Architecture Models for Runtime Adaptability. IEEE Software 23(2), 62–70 (2006)
3. Geihs, K., Khan, M.U., Reichle, R., Solberg, A., Hallsteinsen, S., Merral, S.: Modeling of Component-Based Adaptive Distributed Applications. In: 21st ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006, pp. 718–722 (2006)

4. Paspallis, N., Papadopoulos, G.A.: An Approach for Developing Adaptive, Mobile Applications with Separation of Concerns. In: 30th Annual International Computer Software and Applications Conference (COMPSAC), Chicago, IL, USA, September 17-21, 2006, pp. 299–306. IEEE Computer Society Press, Los Alamitos (2006)
5. Szyperski, C.: Component software: beyond object-oriented programming. ACM Press / Addison-Wesley Publishing Co (1998)
6. The MADAM Consortium: Mobility and Adaptation Enabling Middleware (MADAM), http://www.ist-madam.org
7. The MUSIC Consortium: Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), http://www.ist-music.eu
8. Alia, M., Eide, V.S.W., Paspallis, N., Eliassen, F., Hallsteinsen, S., Papadopoulos, G.A.: A Utility-based Adaptivity Model for Mobile Applications. In: 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW), Niagara Falls, Ontario, Canada, May 21-23, 2007, pp. 556–563. IEEE Computer Society Press, Los Alamitos (2007)
9. Paspallis, N., Chimaris, A., Papadopoulos, G.A.: Experiences from Developing a Context Management System for an Adaptation-enabling Middleware. In: 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Paphos, Cyprus, June 5-8, 2007, pp. 225–238. Springer Verlag, Heidelberg (2007)
10. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: Utility Functions in Autonomic Systems. In: International Conference on Autonomic Computing (ICAC), New York, NY, USA, May 17-18, 2004, pp. 70–77. IEEE Press, Los Alamitos (2004)
11. Alia, M., Hallsteinsen, S., Paspallis, N., Eliassen, F.: Managing Distributed Adaptation of Mobile Applications. In: 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Paphos, Cyprus, June 5-8, 2007, pp. 104–118. Springer Verlag, Heidelberg (2007)
12. Chen, G., Kotz, D.: A Survey of Context-aware Mobile Computing Research, Technical Report: TR2000-381, Dartmouth College, Hanover, NH, USA (2000)
13. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.: Composing Adaptive Software. IEEE Computer 37(7), 56–64 (July 2004)
14. Murphy, A.L., Picco, G.P., Roman, G.-C.: LIME: A Middleware for Physical and Logical Mobility. In: 21st IEEE International Conference on Distributed Computing Systems (ICDCS), Phoenix (Mesa), Arizona, USA, April 16-19, 2001, p. 524. IEEE Computer Society, Los Alamitos (2001)
15. Sousa, J.P., Garlan, D.: Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In: 3rd Working IEEE/IFIP Conference on Software Architecture, Montreal, Canada, August 25-31, 2002, pp. 29–43. Kluwer Academic Publishers, Dordrecht (2002)
16. Paspallis, N., Papadopoulos, G.A.: Distributed Adaptation Reasoning for a Mobility and Adaptation Enabling Middleware. In: 8th International Symposium on Distributed Objects and Applications (DOA). LNCS, vol. 4277, pp. 17–18. Springer, Heidelberg (2006)