Experiences from Developing a Distributed Context Management System for Enabling Adaptivity

Nearchos Paspallis, Avraam Chimaris, and George A. Papadopoulos

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

Abstract. Today, one can observe an ever increasing trend in the use of mobile systems. This change inevitably affects the software running on such devices by necessitating additional functionality such as context awareness and adaptive behavior. While some developers design their systems to be fully self-reliant with regard to context awareness, others aim for more synergistic approaches by allowing context sharing across devices. This paper describes our experience with first designing and implementing a basic context management system, and then with extending it to allow context distribution. In the proposed architecture, the developers define the context dependencies for their software independently of the availability of context information in their corresponding devices. An automated mechanism is then used to match these needs to the corresponding providers, even when those reside across distributed devices. This approach enables them to utilize shared context information at runtime thus reducing both development efforts and hardware costs.

Keywords: Context-awareness, Middleware, Distributed architectures.

1 Introduction

Today, one can observe an ever increasing trend in the use and proliferation of mobile systems. This change has inevitably affected the design and the implementation of software running on such devices. For instance, additional functionality in terms of context awareness and adaptive behavior is now a common feature desired and frequently found in such systems. While the adaptive-behavior implies the capability of a system to run in a number of different configurations or modes, context-awareness refers to its ability to dynamically perceive the characteristics of its surrounding environment. The ultimate benefit is provided in mobile systems which are capable of monitoring and exploiting the contextual information, and infer decisions on choosing the optimal adaptation. This process is guided by the aim for maximizing the quality of the service as it is perceived by the users.

In this work it is assumed that an adaptive, mobile system monitors its environment and dynamically chooses an optimal configuration, thus adapting itself on demand. While the context information which is monitored can be theoretically of unbound variability, in practice only a small fraction of the available context data is delegated as input to the adaptation decision-making component. Naturally, the more context information is available to such a decision maker, the better the decision can be. In most context-aware systems, acquired information is retrieved from sensors or the client side of services. Consequently, the available context information types are restricted by the limited mobile device size and resources which render the hosting of unlimited context sensors intolerable. This limitation highlights the importance of enabling sharing of context information between distributed sources. In this way, the distributed context sources can further eliminate the related costs (e.g. battery consumption, memory use, etc.) while providing mobile nodes with richer context information which otherwise would be impossible to have access to.

This paper describes the architecture of a distributed context management system which is used to drive the adaptation reasoning process in the *mobility and adaptation enabling middleware* (MADAM) [1, 2]. Besides the architecture design this paper's contributions also include a review of requirements for the design and implementation of such a system, as well as a list of related experiences and findings.

The rest of this paper is organized as follows: First, section 2 describes the basic aspects of context-aware systems, followed by section 3 which analyzes a number of requirements for distributed context management systems. Then the proposed architecture is analyzed in section 4, along with a description of its implementation. Following that is a discussion of experiences and related work presented in section 5, and finally, section 6 concludes with a review of the contributions of this paper.

2 Context Awareness

Context-aware computing is an area which studies methods and tools for discovering, modeling and consuming contextual information. Such information can include any information affecting the interaction of a user with a system, such as user location, time of day, nearby people and devices, user activity, light or noise conditions, etc. A more formal and widely used definition specifies context as "*any information that can be used to characterize the situation of an entity; an entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves*" [3, 4].

Context can also be classified in more fine-grained categories: *physical*, *computing* and *user* context information types [5]. The physical context type is related to environmental factors which can usually be evaluated by using specialized hardware mechanisms. The light, noise, and temperature are examples of physical context data types. The computing context refers to the information which describes the resources available in the computing infrastructure. This includes information such as the network connectivity and its characteristics (e.g. bandwidth, latency, etc.), nearby resources (such as printers, video projectors, etc), and details concerning the memory availability, the processor use, etc. Finally, the user context refers to the user's profile by focusing on the user needs, preferences, mood, etc. For example these can include information concerning the user's occupation (e.g. driving, studying, etc.) or the user's choice for preferring, say, to use a desktop computer rather than a PDA while at work.

Furthermore, it is argued that any system that aims to be minimally intrusive must be context aware, in the sense that it should be cognizant of its user's state and environment [6]. In other words, context-aware mobile systems are expected to utilize such information in order to adapt their behavior, based on a predefined set of adaptation rules. These rules are usually monitored by a system which dynamically adapts the system's operation based on the contextual information sensed.

In this paper, the context awareness is treated as an independent concern, where the applications can separately and independently register for particular context change events, without having to be involved in the collection or management of contextual information. Because of this separation of concerns, it is possible to treat the context awareness support mechanism independently of the adaptation mechanism. I.e. from a developer's point of view, the two mechanisms can evolve independently, thus improving on both the development and the maintenance effort required.

3 Requirements for Distributed Context Management

The main responsibilities of a context-aware, adaptive mobile system include acquiring context information, reasoning on the acquired information, and performing adaptations as a result of these changes. In many cases the acquired information is retrieved locally (e.g. through attached sensors) but frequently this information is insufficient for performing the required adaptation reasoning. In a distributed context management system, additional context information can be shared among a set of distributed mobile devices. This enhances the process of making adaptation reasoning decisions by offering context information which would otherwise not be accessible.

3.1 General Requirements

The implementation of a distributed context-aware framework should address many of the requirements of traditional distributed systems such as *heterogeneity*, *mobility*, *scalability*, and *tolerance* to system and network failures. Heterogeneity is required because systems are inevitably developed by different teams and target many different platforms. However, these systems are still expected to collaborate with each other and share context information. Distributed context management systems are also naturally expected to enable mobility, and thus it should be possible to disseminate context information independently of the communication protocols, the underlying network infrastructure or the location of the nodes. The requirement for scalability is a natural consequence of the distributed nature of the desired context management system. This requirement dictates that the performance of the system is not severely downgraded as the number of participating nodes increases. Finally, and although not critical from a functional point of view, the *ease of deployment and configuration* is also an important requirement for such a system. These requirements were considered in our implementation, as it is discussed in sections 4 and 5.

3.2 Requirements for the Distribution of Context

Typical context management systems adhere to the publish/subscribe model, where providers asynchronously provide their information, and clients subscribe for notification when such events occur. This approach however, is further extended in the case of distributed systems, as the providers and the subscribers can reside on different, network connected nodes. The additional requirements are:

Service Discovery: The service discovery requirement refers to the need for discovering context providers (i.e. nodes capable of sharing context information). Suitable approaches include two main categories: centralized and ad-hoc service discovery. Centralized approaches include services which provide context at well known locations (e.g. a URL), or advertise their capabilities in directories. Contrary to these, ad hoc approaches utilize services which dynamically form partnerships for context exchange. Their communication can be realized by using combinations of infrastructure-based, wireless and ad hoc-based networks.

Modeling and Semantics: The context modeling refers to the requirement for formatting the information so that it encapsulates both the required data and metadata. Context modeling is important for guaranteeing compatibility among the possibly heterogeneous devices (i.e. mobile nodes, context sensing mechanisms, etc.). This is particularly important in ad-hoc configurations, where the nodes participate to context exchanges without being *a priori* aware of each other, and consequently of the methods they use to abstract (model) and interpret (semantics) context information.

Scope and Privacy: When sharing context information in a distributed environment, it is important to define its scope. For example, context information which is limited to local use should be prevented from being generally distributed. Rather, suitable methods should be used to limit its dissemination within a local area in which it is more likely to be valid. As most of the context information is expected to be of local interest only, this requirement seeks to ensure that an explosion of context information is prevented and rather a form of localized scalability is enabled. On the other hand, the dissemination of context information should also be controlled so that no sensitive information can be leaked to the wrong hands. Similar to the context scope, the privacy is another important parameter which must be taken into account when defining the access to context information. In particular, the access to sensitive context information must be explicitly defined so that only the context information which is intended to be public is shared with other devices.

4 The Architecture of the Context Management System

The main concept of the implemented architecture is the separation between *context clients* and *context providers* [7,8]. In this respect, all nodes act as both context providers and context consumers, as part of a membership group which is formed using a loosely coupled protocol. Furthermore, while individual nodes are free to access context information from any possible provider (i.e. even context servers located at remote geographical locations), it is nevertheless assumed that in most cases context sharing is limited to a local area only. In this respect, the locality refers to groups formed by nodes which can directly communicate with each other, e.g. over a wireless link by forming an ad-hoc WiFi or Bluetooth network (i.e. a piconet).

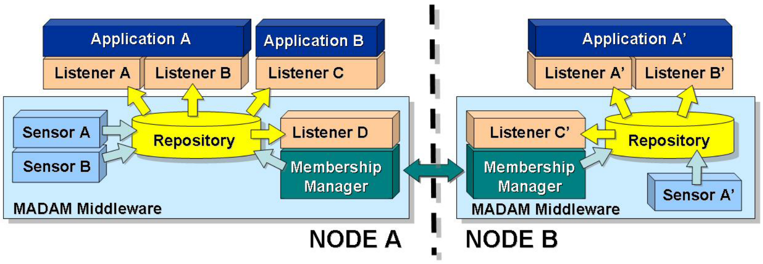

Fig. 1. Distributed Context Management System Architecture

This approach has the important advantage of assigning higher importance to local context and consequently enabling localized scalability [6]. The first one refers to the fact that it is more likely that two neighboring nodes will share a common interest on the same context as opposed to nodes at different geographical locations. This is true for example in most pervasive computing applications where applications aim to utilize the infrastructure which is embedded in the surrounding environment. In another example, it would be more likely that an application would be more interested in the temperature information provided by nearby nodes (and thus residing in the same environment) as opposed to the temperature information provided by distant nodes. Second, localized scalability is achieved by preferring local sources (and respectively consumers) for sharing context information with. In this approach, the use of mainstream links is avoided as most of the communication is carried out over local (i.e. direct) network links. The following paragraphs describe the basic ideas of this approach, along with the algorithms required to support it.

4.1 Context Management in Centralized Environments

As it has already being mentioned, the implemented architecture is based on the separation of roles between context *providers* and *consumers* [7]. Even if all nodes can interchangeably act as both clients and producers, at the underlying layer there are specialized architectural components which can either support context production or consumption. These components are the Context Sensors that are used to *produce* context, and the Context Listeners which can be registered to *listen* for context changes (Fig. 1). When the monitored context type changes, the listeners inform the linked applications (e.g. *Application A* is informed for context changes for the monitored context of *Listener A* and *Listener B*). The Context Sensors generate context elements that are stored in local repositories. This centralized architecture is quite simple and is based on the requirements defined in the context-aware section.

4.2 Membership and Distributed Context Management

In a distributed context-aware system the intention is for the information in the local repositories to be shared between nodes. In order to enable this, we implemented a loosely coupled communication protocol between the distributed nodes which is based on the transmission and handling of *heartbeat* messages. This architecture is based on the requirements that were identified in section 3. In the analysis of the required communication protocol we discovered that not all context information was suitable for sharing. For example, context information describing the battery status of a device is generally useless to other, neighboring devices. Furthermore, as per the privacy requirement, we detected a need for excluding some context information from being shared. In this respect, two *properties* were defined for characterizing the context element types: *scope* and *privacy*. The first property refers to whether the context element value is appropriate for distribution or not. The possible values that can be assigned to this property are: *public* (i.e. can be distributed without restrictions), *local* (i.e. useful only within a small range around the providing node; such information is typically directly communicated across devices) and *private* (meaningful only within the device itself). The *privacy* property describes how sensitive is the context information and consequently whether it is suitable for sharing or not. This property can be assigned two values: *public* (i.e. the information can be shared unrestricted), and *private* (i.e. the information is not subject to distribution outside the local device).

Once the context information is appropriately annotated with properties, the next step is to define an appropriate mechanism to first enable the dynamic discovery of nodes, and second to physically enable information sharing among them. In this work, we have purposely aimed for a completely ad-hoc approach, which has the benefit of not requiring the set-up of context servers and, additionally, it provides better access to neighboring information which is much more likely to be relevant to collaborating nodes. The used protocol is based on a loosely coupled method, which is enabled by periodically broadcasting and handling heartbeat messages. Furthermore, the overall system is based on a push/pull hybrid approach. While pull approaches attempt to retrieve context information without *a priori* being aware if the requested data is available or not, push approaches proactively communicate context information to peer nodes regardless of whether the context was requested or not. In our hybrid approach, the distributed context is transmitted (pushed) from the providing nodes to the requesting ones. Additionally, the requesting nodes do not keep track of the remotely provided context, but rather they notify nearby nodes of their needs.

The distributed context needs are defined inside the heartbeat messages which are broadcasted by the underlying network layer. The broadcasted messages also encode the types of the desired context data. When received, the context data is decoded to form a list of the required values by all nodes in the neighborhood. Then, from an individual node's point of view, requested context types that are available are subsequently broadcasted to the local network (push approach) also by being encoded in the corresponding heartbeat messages that are periodically broadcasted. On the receivers' side the heartbeats are decoded and the corresponding context values are used to generate a local context change event, as if the changes were sensed locally.

In practice, the push mechanisms are more efficient than their pull counterparts, as the pull mechanisms need local meta-data in order to select the proper provider to request for, and to construct the request message. In push architectures, there is no need to keep local information about remote providers because as soon as a nearby node receives a context request an appropriate heartbeat message is immediately constructed and communicated back to the requestor.

We argue that this architecture satisfies the detected required features. The use of a broadcasting mechanism for the heartbeat messages reduces the communication overhead (especially as the required context information is piggy-backed into these messages). Another alternative would be to have nodes announcing their offered context information, but this imposes significant overhead for updating local tables mapping context offerings to context requestors. Instead, in the proposed architecture there is no need for storing such information because the requests are handled directly by context producers. Consequently, this architecture provides the benefits of better scalability and consistency, while at the same time requiring fewer resources.

This architecture is heavily based on the periodic broadcast of special heartbeat messages which serve two purposes: first they are used to update the membership status of the individual nodes and they communicate basic information about context required by the sender. Additionally, the heartbeat messages are used for transmitting context change events from providing nodes (using the discussed push approach) to the requesting nodes. This approach also enables a loosely-coupled synchronization method which is based on periodic broadcast of heartbeat messages. These messages are intended to both form and maintain a *membership group*, as well as to update the individual nodes of the context information required by the senders. Similar protocols have also been proposed and tested in commercial environments (e.g. the Bonjour [9] and the Bluetooth technologies [10]). In the proposed approach however, the aim is specialized on the exchange of context information rather than of general data.

The membership manager: In this architecture, the most important component is the *Membership Manager* (see Fig. 1). The Membership Manager is part of the context management system of the MADAM middleware. Its main responsibility is to periodically multicast the heartbeat messages and to handle the received ones.

The periodic multicast of heartbeats aims at achieving mainly two goals: first, to enable the formation of a loosely coupled membership group, and second, to inform the neighboring nodes (i.e. the group) about possible context needs which cannot be locally satisfied. Additionally, the heartbeats are also used to encapsulate context data so that they can be shared with other nodes. On the receiving side, the membership manager exploits this information exactly for forming this loosely coupled group and for decoding possible context change events which are of interest to the local node.

The membership manager's functionality is supported by two table-like data structures: the *membership table* which is used for managing the membership status and the *context requestors table* to maintain the context requests from the remote nodes. When a heartbeat message is received, the membership table is updated with the provided information. For example if the heartbeat was sent by a node which is not already present in the membership management table, a new entry is created for it. At the same time, an event is generated indicating the addition of the new member. If the node is already present in the membership table, then its context requirements are examined for changes, and appropriately update the context requestors table. In this way, the requesting nodes notify the nearby context-provider nodes of their newly required context in order to adjust their remote context listeners.

In order to detect when a node has left the membership, a simple algorithm is also used which is based on a predefined, globally agreed timeout period: the *heartbeat interval*. In simple words, this algorithm periodically checks the table with the current members and ensures that all members have a recent heartbeat timestamp. When a

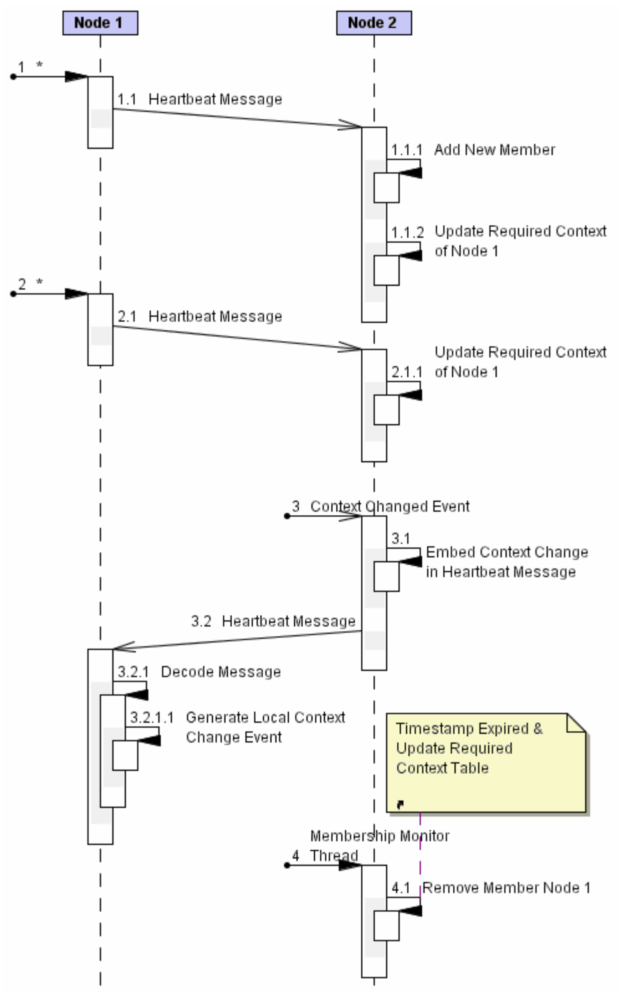
Node 1
Node 2
1 *
1.1 Heartbeat Message
1.1.1 Add New Member
1.1.2 Update Required Context of Node 1
2 *
2.1 Heartbeat Message
2.1.1 Update Required Context of Node 1
3 Context Changed Event
3.1 Embed Context Change in Heartbeat Message
3.2 Heartbeat Message
3.2.1 Decode Message
3.2.1.1 Generate Local Context Change Event
Timestamp Expired & Update Required Context Table
4 Membership Monitor Thread
4.1 Remove Member Node 1

Fig. 2. Sequence diagram of a typical message interchange in a group membership

member misses a predefined number of consecutive heartbeats, it is assumed to have left the group. At that point, an appropriate event is generated indicating the fact that the member in question has left the group. Because the departed node was possibly also included in the context requestors table, an appropriate update takes place there too, so that all context entries requested by that node only, are removed.

As a result of the heartbeat messages, two main events are triggered by the corresponding membership management mechanism: the *new member added* event and the *existing member left* event. An additional event concerning context updates (pushed context changes) can also be raised: the *context updated* event. All events encapsulate information about the identity of the node involved, as well as information on its requested context. To better explain the used algorithm, the following paragraphs explain how these events are handled by the context manager:

- **New member added event:** This type of event is generated when a heartbeat message is received from a node not previously registered with the membership manager. Once detected, the new node is also automatically considered for its needed context. For each remote requestor, a local context listener counterpart is instantiated. This listener automatically pushes context information to the remote requesting node when the respective context changes (sequence 1 in Fig. 2).
- **Existing member left event:** This event is triggered when a node is detected to have left the membership group. Each heartbeat timestamp is updated whenever a heartbeat message is received from the specific node. In this way, when the heartbeat timestamp of a node in the context providers table is found to be outdated, the corresponding node is assumed to be disconnected. At that time, the listeners that are pushing information to this remote node are considered obsolete, and thus are removed from the table (sequence 4 in Fig.2).
- **Context requirements updated event:** Finally, a context change event occurs when an existing node is found to have changed its needed context. In that case, the membership manager iterates through the context requestors table and updates the corresponding entries (i.e. removes obsolete entries and add newly required ones). This is depicted by sequence 3, in Fig. 2.

Besides generating these events, the membership manager also reacts on them, by adding and removing context listeners (to itself). The actual context information is communicated through the heartbeat messages, as piggy-backed context information. Thus, beyond updating the membership status when a heartbeat is received, the membership manager also parses the heartbeats and passes possible context change events to the context repository (see Fig. 1) for further distribution.

4.3 Implementing the Architecture

The described architecture was designed and implemented as part of a broader adaptation enabling middleware (MADAM). The system was implemented in the Java language and tested on both a laptop computer running the Windows XP operating system and an iPAQ PDA computer running the Windows Mobile operating system. Regarding the JVM, in the first case we used the mainstream implementation provided by Sun Microsystems, while in the case of the PDA we used the CreMe JVM by NSI.com. Finally, the MADAM middleware provides a context visualizer (a simple context client) which allows a user or a developer to dynamically monitor and edit (simulate) the context information (shown in Fig. 3, when deployed on a PDA).

During the implementation, some of our main goals were interoperability, platform independence, and extensibility. To facilitate the first two goals, we used the Java system while refraining from using *native* (i.e. platform dependent) libraries. However, lower-level layers of the MADAM middleware (and especially the resource management component) do extensive use of native libraries, which are platform-dependent (e.g. two different implementations are made available by the MADAM consortium targeting both Windows-based PCs and PDAs). However, extensive coverage of the resource management layer is beyond the scope of this paper. The interested readers are rather referenced to the MADAM website [1].

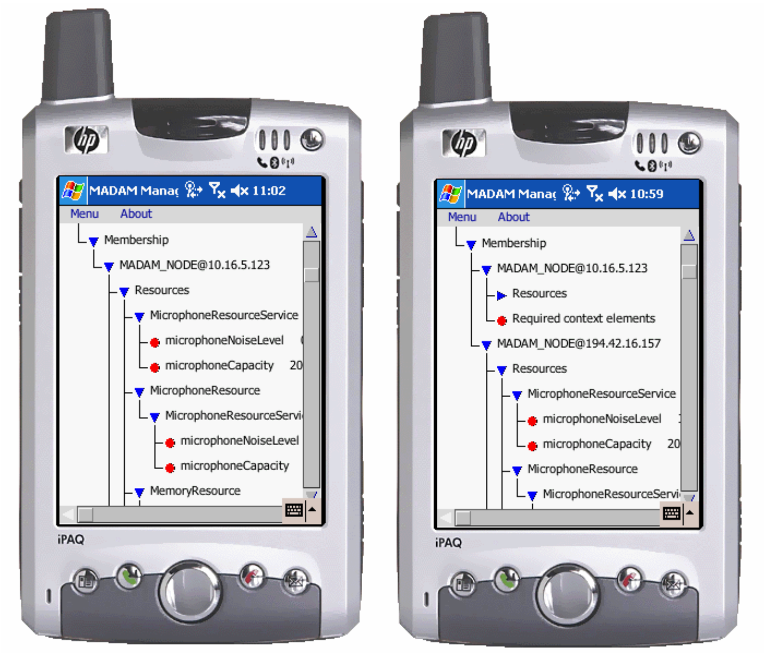

Fig. 3. The left diagram depicts the context view in the case of a single node, while the right diagram depicts a situation where two individual nodes form a membership

For the extensibility goal, we used an approach which allows to interchangeably selecting different networking technologies. In this respect, we defined a *Broadcast Service* interface which provides methods for *broadcasting* generic, serializable messages and for subscribing (and unsubscribing) for the reception of such messages. The membership manager is only aware of this interface, thus allowing a developer to provide different implementations.

At this time, we have tested a default implementation of the broadcast service which has successfully demonstrated message broadcasts on both wired and wireless networks, on both Windows XP and Windows Mobile-based systems. Furthermore, we developed a simulated version of this service, which uses plain TCP communication messages and a simulation hub, with the intention of enabling the middleware to function even behind firewalls or simply when on devices which do not support multicasting. Finally, a Bluetooth-based implementation is also underway.

5 Experiences from the Development of the Context System

The process of first designing a basic context management system and then extending it to enable distributed context sharing has provided us with many valuable insights that we attempt to document in the following paragraphs:

Non-functional nature of context should remain as such: When designing context aware systems, the aim is usually to optimize the operation of the system, rather than extend its capabilities. For example, an intelligent agenda could exploit GPS information so that when a “lunch at 12pm” entry is activated, a list of nearby restaurant options, compatible with the user's taste, are automatically displayed to inform the user about them. However, in this case the availability of context information (i.e. GPS coordinates) is completely optional and does not prevent the software from performing its basic goals. Rather, it simply limits its functionality to some extent, with also a possible decrease in the quality of the offered service. It has been our experience with the development of the context management system, but also with the development of the MADAM adaptation-enabling middleware, that the context information should be used as such and never being allowed to become a part of a *critical path*, i.e. its absence should never cause a system to stop functioning. In this respect, the MADAM middleware suggests the designers to provide a set of possible adaptations (i.e. configurations) for their applications, along with a set of *properties* and *utility functions* which always allows the computation and the selection of a minimal configuration, regardless of the availability or absence of (possibly distributed) context information. This experience is in accordance to a common distributed computing fallacy[1]: *the network is reliable*.

Modeling of context should provide support for distribution: While designing the basic context management system, one can be easily mislead to the assumption that the context information is both generated and consumed at the same node. However, in real distributed systems, sharing of context information imposes additional requirements for identifying both the nature and the origin of the context information. For example, information about the memory availability of a node becomes useless, unless the actual node association is explicitly or implicitly defined. This also implies that unless the context information is generated and consumed by the same system (e.g. the MADAM middleware), then a set of semantics metadata must accompany the actual context data to allow for better optimization of the context data (e.g. the metric system used for the measurements, the methods used to acquire the data, and even the accuracy of the communicated information). Last but not least, distributed dissemination of context data requires that the distributed peers *trust* each other and they are capable of securing that the communicated data is handled as it is intended.

Plug-and-Play architecture support for context sensors: Assuming that a device will require a constant set of context information types is erroneous. In practice, different applications are dynamically started and stopped. Additionally, in the case of adaptive, component-based applications different variants of the same application might impose different context requirements. Having the maximum context information provided at all times is not an optimal solution, especially in mobile systems where resource consumption is an important concern! In this respect, the design of a plug-and-play architecture enables dynamic reconfiguration of the context manager’s architecture, which can greatly improve the system’s efficiency and autonomy. In our context system’s architecture, we maintain a dynamically updated list with the registered context listeners (consumers) along with their corresponding

[1] http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing.

needs. This allows the system to periodically and dynamically evaluate the situation which concerns the need for context information and dynamically activate and deactivate the corresponding context sensors. Additionally, while some of the sensing functionality (such as the memory and CPU monitoring) can only be embedded in the middleware system, others depend on software and hardware sensors, both native to the device and newly added ones. For example, a system might be originally designed with a GPS device only, but in the future it might be equipped with a temperature and barometer sensor as well. Such an addition should not require any updates to the middleware, but simply the addition of new software context sensors which would make the new information available to the middleware as well. This is combined with the general middleware's pluggable architecture which allows dynamic loading and unloading of applications and components together with the corresponding (software) context sensors and reasoners. In effect, this enables the context system to extend its domain of covered context information at runtime while at the same time conforming to the actual needs of the hosted applications.

6 Related Work and Conclusions

A plethora of related work studies both centralized and distributed issues of context management. This section discusses a number of achievements established already, but also detects open problems which are not addressed by existing approaches yet.

Centralized context-aware systems use a local service which provides applications with contextual information. Such infrastructures encapsulate these services as part of a middleware which acquires raw contextual information from sensors and provides interpreted context to applications via a standardized API. Furthermore, the middleware is assigned to monitor particular context changes and dispatch relevant events to interested applications when required.

In contrast to centralized approaches, distributed context-aware applications allow the generation of context information at several locations, thus avoiding potential bottlenecks and unnecessary hardware duplication. Despite the fact that decentralized architectures increase the communication cost, they are more resilient to errors as they do not require a central server to maintain the context information.

An approach which is partly based on message multicasts is described in [12]. In this approach clients broadcast their location queries to all the members of a group and interested parties anonymously listen to the queries. When they match a query and their privacy policy allows it they reply to the query. Just like in our approach, the main disadvantage lies in the increased computation and communication cost. Unlike that approach though, our proposed mechanism aims at limiting the communication cost by minimizing the heartbeat message size. Furthermore, both the computation and communication costs can be minimized by increasing the heartbeat interval if that can be tolerated by the applications. Finally, the computation cost is further limited by using the context update timestamp which prevents the nodes to perform unnecessary computations when there are no context changes encoded in the heartbeat.

The Context Toolkit [13] provides a component framework for acquiring and handling context using three key abstractions: widgets, interpreters, and aggregators.

The context widgets are the most important components of this framework because they provide applications with access to the context information while hiding the details of context sensing. The context interpreters convert or interpret context to higher level information and the context aggregators collect context relevant to particular entities. Similar to our approach, the Context Toolkit provides support for storing historical context data, and then reusing them to estimate their value trend.

Other systems, like Jini [14], use coordination model infrastructures to implement well-formed shared repositories. This technology is usually used in the background, such as for example in the Smart Map project [15], which enables position-aware applications by using the Jini technology for implementing a registry. The registry is used by service providers to register themselves for context availability and the service consumers use the registry to discover them. The Context Fusion Networks (CFN) [16] project is implemented as a context-aware middleware which handles context information by realizing sources, sinks and channels. The context sensors are represented by sources because they are responsible for constructing contextual information. The applications which use this information are represented by sinks. Furthermore, more recent approaches exist which aim at enabling generic data sharing between neighboring devices. A notable approach is described in [17] where support is provided for developing efficient solutions for sharing data in the neighborhood.

In contrast to most of these approaches, which do not explicitly tackle fault tolerance, our approach provides limited fault tolerance. As the context manager has a minimum state, any failures can be tolerated by simply re-instantiating the context manager and allowing some time for the corresponding context producers and consumers to recover by processing their periodic messages. However, our approach is not tolerant to malicious attacks such as message flooding, which is a common limitation of broadcasting-based approaches. Finally, unlike most other works, our approach implements and promotes localized scalability as an effective measure to optimize the consumption of resources and maintain the system performance.

In conclusion, this paper proposes a distributed context management mechanism which aims at driving the decision making in the *adaptation enabling middleware* (MADAM). We have detected a number of both general and more specific requirements imposed by the distribution aspect. In this respect we have proposed an approach which is based on the periodic communication of heartbeat messages for forming loosely coupled membership groups and for advertising their required context. We argue that this approach satisfies the detected requirements to a great extend. Furthermore, this architecture has been implemented, tested, and evaluated in real pilot applications, on both resourceful (laptops) and small (PDAs) computers, with significant success. Further work is underway, aiming at specifying a more structured context model, as well as extending its application domain to ubiquitous computing (i.e. embedded in addition to mobile devices).

Acknowledgments. The authors would like to thank their partners in the MADAM-IST and the MUSIC-IST projects and acknowledge the financial support given to this research by the EU (6th Framework Programme, contract numbers 4169 and 35166).

References

1. IST MADAM (Mobility and Adaptation Enabling Middleware), http://www.ist-madam.org
2. Floch, J., Stav, E., Hallsteinsen, S., Eliassen, F., Gjørven, E., Lund, K.: Using Architecture Models for Runtime Adaptability. IEEE Software 23(2), 62–70 (2006)
3. Dey, A.: Providing Architectural Support for Building Context-Aware Applications, PhD Thesis, College of Computing, Georgia Institute of Technology, pp. 170 (2000)
4. Dey, A.: Understanding and Using Context. Personal Ubiquitous Computing 5(1), 4–7 (2001)
5. Chen, G., Kotz, D.: A Survey of Context-Aware Mobile Computing Research, Technical Report: TR2000-381 Dartmouth College, Hanover, NH, USA (2000)
6. Satyanarayanan, M.: Pervasive Computing: Vision and Challenges, IEEE Personal Communications Magazine, pp. 10–17 (2001)
7. Mikalsen, M., Paspallis, N., Floch, J., Stav, E., Papadopoulos, G.A., Ruiz, P.A.: Putting Context in Context: The Role and Design of Context Management in a Mobility and Adaptation Enabling Middleware, International Workshop on Managing Context Information and Semantics in Mobile Environments (MCISME'06). In: conjunction with the 7th International Conference on Mobile Data Management (MDM'06), Nara, Japan, May 9-12, 2006, pp. 76–83. IEEE Computer Society Press, Washington, DC (2006)
8. 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 2006), Chicago, IL, USA, Sept. 17-21, 2006, pp. 299–306. IEEE Computer Society Press, Washington, DC (2006)
9. Bonjour: Connect Computers and Electronic Devices Automatically without any Configuration http://images.apple.com/macosx/pdf/MacOSX_Bonjour_TB.pdf
10. Draft Bluetooth Core Specification v2.1 + EDR https://www.bluetooth.org /spec/
11. Want, R., Schilit, B., Adams, N., Gold, R., Petersen, K., Goldberg, D., Ellis, J., Weiser, M.: An Overview of the PARCTAB Ubiquitous Computing Experiment. IEEE Personal Communications 2, 28–43 (1995)
12. Spreitzer, M., Theimer, M.: Providing location information in a ubiquitous computing environment. 14th ACM Symposium on Operating Systems Principles, Asheville, NC, USA, December 5-8, pp. 270–283. ACM Press, New York (1993)
13. Dey, A., Salber, D., Abowd, G.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human Computer Interaction 16(2-4), 97–166 (2001)
14. Sun Microsystems, Jini Network Technology, http://www.sun.com/software/jini/
15. Urnes, T., Hatlen, A., Malm, P., Myhre, O.: Building Distributed Context-Aware Applications. Personal Ubiquitous Computing 5(1), 38–41 (2001)
16. Chen, G., Li, M., Kotz, D.: Design and implementation of a large scale context fusion network. 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous), Cambridge, MA, USA, Aug. 22-25, 2004, pp. 246–255. IEEE Computer Society Press, Washington (2004)
17. Lachenmann, A., Marrón, P.J., Minder, D., Saukh, O., Gauger, M., Rothermel, K.: EWSN 2007. LNCS, vol. 4373, pp. 1–16. Springer, Heidelberg (2007)