DISTRIBUTED CONTEXT MONITORING FOR CONTINUOUS MOBILE SERVICES

Claudio Bettini, Dario Maggiorini, and Daniele Riboni
DICo, University of Milan, via Comelico 39, I-20135, Milan, Italy

Abstract: Context-awareness has been recognized as a very desirable feature for mobile internet services. This paper considers the acquisition of context information for continuous services, i.e., services that persist in time, like streaming services. Supporting context-awareness for these services requires the continuous monitoring of context information. The paper presents the extension of a middleware architecture for the reconciliation of distributed context information to support context-aware continuous services. The paper also addresses optimization issues and illustrates an adaptive video streaming prototype used to test the middleware.

Keywords: Context-awareness, adaptation, continuous mobile services

1. INTRODUCTION

Internet services provided to mobile users can be divided into two broad categories: instantaneous (or one-shot) services and continuous services. Examples of services in the first category are web browsing, search for the closest pharmacy, and delivery of a message with the current balance of a certain bank account. Services in the second category persist much longer in time and are typically characterized by multiple transmissions of data by the service provider. Examples are multimedia streaming, navigation services, location-based recommendation services, and publish/subscribe services.

Services in the first category can be implemented with context-aware features if some context information can be obtained at the time of service request and if the service application logic can take this information into account when answering to a specific request. Context-awareness is much

more challenging for continuous services, since changes in context should be taken into account during service provisioning. As an example, consider an adaptive streaming service. Typically, parameters used to determine the most appropriate media quality include a number of context parameters, as, for example, an estimate of the available bandwidth and the battery level on the user's device. Note that this information may be owned by different entities, e.g., the network operator and the user's device, respectively. With a naïve approach, the application logic should constantly monitor these parameters, possibly by polling servers in the network operator's infrastructure as well as the user's device for parameter value updates. Moreover, the application logic should internally re-evaluate the rules that determine the streaming bit rate (e.g., "if the user's device is low on memory, decrease the bitrate"). This approach has a number of shortcomings, including: (*i*) client-side resource consumption; (*ii*) high response times due to the polling strategy; (*iii*) complexity of the application logic; (*iv*) poor scalability, since for every user the service provider must continuously request context information and re-evaluate its rules. An alternative approach is to provide the application logic with asynchronous notifications of relevant context changes, on the basis of its specific requirements. However, when context information must be aggregated from distributed sources which may possibly deliver conflicting values, as well as provide different dependency rules among context parameters, the management of asynchronous notifications is far from trivial. The **CARE** middleware (Agostini et al., 2004; Bettini and Riboni, 2004) was originally designed to support instantaneous context-aware mobile services in an environment characterized by distributed context sources. The main contribution of this paper is indeed the extension of the **CARE** middleware to include a mechanism of asynchronous notifications, enabling context-awareness also for continuous mobile services. Technically, the extension involves algorithms to identify context sources and specific context parameter thresholds for these sources, with the goal of minimizing the exchange of data through the network and the time to re-evaluate the rules that lead to the aggregated context description.

While several frameworks have been proposed to support context awareness (see e.g., Bellavista et al., 2003; Butler et al., 2002; Chen et al., 2004; Hull et al., 2004), we have extensively described elsewhere (Agostini et al., 2004) that **CARE** has some unique features in dealing with distributed and possibly conflicting context information. The extension to the support of asynchronous context change notifications still preserves these unique features. The work on stream data management has probably the closer connection with the specific problem we are tackling. Indeed, each source of context data can be seen as providing a stream of data for each context parameter it handles. One of the research issues considered in that area is the

design of filter bound assignment protocols with the objective of reducing communication cost (see, e.g., Chengy et al., 2005). Since *filters* are the analogous of *triggers* used in our approach to issue asynchronous notifications, we are investigating the applicability of some of the ideas in that field to our problem.

The rest of the paper is organized as follows: Section 2 describes the **CARE** middleware architecture and features; Section 3 contains the principles and technical details for managing asynchronous notifications; Section 4 shows how the extension has been implemented; Section 5 briefly describes a streaming service used to test the middleware; Section 6 concludes the paper.

2. THE MIDDLEWARE ARCHITECTURE

The **CARE** (Context Aggregation and REasoning) middleware has been presented in detail elsewhere (Agostini et al., 2004; Bettini and Riboni, 2004). Here we only describe what is needed to understand the extension to support continuous services.

2.1 Overview

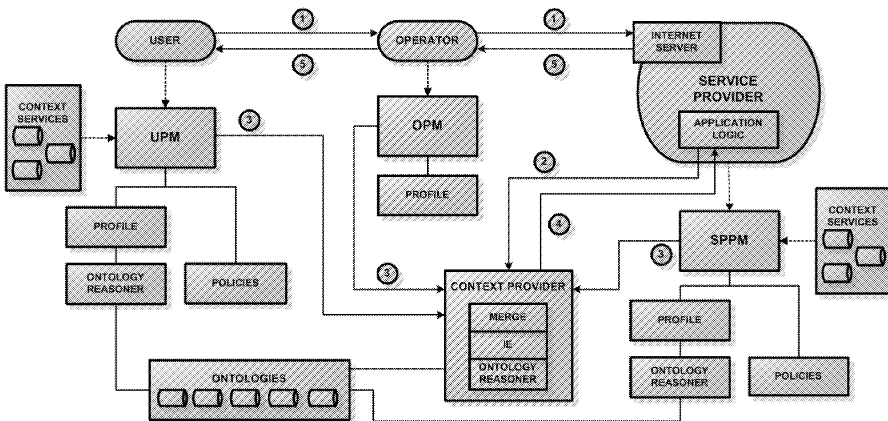

Figure 1. Architecture overview and data flow upon a user request

In our middleware, three main entities are involved in the task of building an aggregated view of context information, namely: the *user* with his/her devices, the *network operator* with its infrastructure, and the *service*

provider with its own infrastructure. Clearly, the architecture has been designed to handle an arbitrary number of entities. In **CARE** we use the term *profile* to indicate a set of context parameters, and a profile manager is associated with each entity; profile managers are named *user profile manager* (UPM), *operator profile manager* (OPM), and *service provider profile manager* (SPPM), for the user, the network operator and the service provider, respectively. Adaptation and personalization parameters are determined by policy rules defined by both the user and the service provider, and managed by their corresponding profile managers. In Figure 1 we illustrate the system behavior by describing the main steps involved in a service request. At first (step 1) a user issues a request to a service provider through his device and the connectivity offered by a network operator. The HTTP header of the request includes the URIs of UPM and the OPM. Then (step 2), the service provider forwards this information to the CONTEXT PROVIDER asking for the profile information needed to perform adaptation. In step 3, the same module queries the profile managers to retrieve distributed profile data and user's policies. Profile data are aggregated by the MERGE module in a single profile which is given, together with policies, to the Inference Engine (IE) for policy evaluation. In step 4, the aggregated profile is returned to the service provider. Finally, profile data are used by the application logic to properly adapt the service before its provision (step 5). Our architecture can also interact with ontological reasoners, but this aspect will not be addressed in this paper.

2.2 Profile Aggregation

In the following we show how possibly conflicting data can be aggregated into a single profile.

Profile and policy representation

Essentially, profiles are represented adopting the CC/PP (Klyne et al., 2004) specification, and can possibly contain references to ontological classes and relations. However, for the sake of this paper, we can consider profiles as sets of *attribute/value* pairs. Each attribute semantics is defined in a proper vocabulary, and its value can be either a *single value*, or a *set/sequence* of single values.

Policies are logical rules that determine the value of profile attributes on the basis of the values of other profile attributes. Hence, each policy rule can be interpreted as a set of conditions on profile data that determine a new value for a profile attribute when satisfied.

Example 1 *Consider the case of a streaming service, which determines the most suitable media quality on the basis of network conditions and available memory on the user's device. The* MediaQuality *is determined by the evaluation of the following policy rules:*

R1: *"**If*** AvBandwidth *≥ 128kbps **And*** Bearer *= 'UMTS' **Then Set*** NetSpeed*= 'high'"*
R2: *"**If*** NetSpeed*= 'high' **And*** AvMem *≥ 4MB **Then Set*** MediaQuality*= 'high'"*
R3: *"**If*** NetSpeed*= 'high' **And*** AvMem < *4MB **Then Set*** MediaQuality*= 'medium'"*
R4: *"**If*** NetSpeed! *= 'high' **Then Set*** MediaQuality*= 'low'"*

*Rules R2, R3 and R4 determine the most suitable media quality considering network conditions (*NetSpeed*) and available memory on the device (*AvMem*). In turn, the value of the* NetSpeed *attribute is determined by rule R1 on the basis of the current available bandwidth (*AvBandwidth*) and* Bearer.

Conflict resolution

We recall that, once the CONTEXT PROVIDER has obtained profile data from the other profile managers, at first this information is passed to the MERGE module which is in charge of merging profiles. Conflicts can arise when different values are provided by different profile managers for the same attribute. For example, suppose that OPM provides for the *AvBandwidth* attribute a certain value *x*, while the SPPM provides for the same attribute a different value *y*, obtained through some probing technique. In order to resolve this type of conflict, the CONTEXT PROVIDER has to apply a resolution rule at the attribute level. These rules (called *profile resolution directives*) are expressed in the form of priorities among entities, which associate to every attribute an ordered list of profile managers.

Example 2 *Consider the following profile resolution directives, set by the provider of the streaming service cited in Example 1:*

PRD1: setPriority AvBandwidth = (OPM, SPPM, UPM)
PRD2: setPriority MediaQuality = (SPPM, UPM)

In PRD1, the service provider gives highest priority to the network operator for the AvBandwidth *attribute, followed by the service provider and by the user. The absence of a profile manager in a directive (e.g., the absence of the OPM in PRD2) states that values for that attribute provided by that profile manager should never be used. The conflict described above is resolved by applying PRD1. In this case, the value x is chosen for the*

available bandwidth. The value y would be chosen in case the OPM *does not provide a value for that attribute.*

Once conflicts between attribute values provided by different profile managers are solved, the resulting merged profile is used for evaluating policy rules. Since policies can dynamically change the value of an attribute that may have an explicit value in a profile, or that may be changed by some other policies, they introduce nontrivial conflicts. The intuitive strategy is to assign priorities to rules having the same head predicate on the basis of its *profile resolution directive*. Hence, rules declared by the first entity in the *profile resolution directive* have higher priority with respect to rules declared by the second entity, and so on. When an entity declares more than one rule with the same head predicate, priorities are applied considering the explicit priorities given by that entity.

Example 3 *Consider the set of rules shown in Example 1 and* profile resolution directives *shown in Example 2. Suppose that R2 and R3 are declared by the service provider, and R4 is declared by the user. Since the service provider declared two rules with the same attribute in the head, it has to declare an explicit priority between R2 and R3. Suppose the service provider gives higher priority to R2 with respect to R3. Since the* SPPM *has higher priority with respect to the* UPM*, according to the* profile resolution directive *regarding* MediaQuality *(i.e., PRD2), if p(R) is the priority of rule R, we have that:*

$p(R2)>p(R3)>p(R4)$

The intuitive evaluation strategy is to proceed, for each attribute A, starting from the rule having $A()$ in its head with the highest priority, and continuing considering rules on $A()$ with decreasing priorities till one of them fires. If none of them fires, the value of A is the one obtained by the MERGE module on A, or *null* if such a value does not exist. A more in-depth discussion of conflict resolution can be found in (Bettini and Riboni, 2004).

3. SUPPORTING CONTINUOUS MOBILE SERVICES

In this section we describe a trigger mechanism for supporting continuous mobile services (Maggiorini and Riboni, 2005). This mechanism allows profile managers to asynchronously notifying the service provider upon relevant changes in profile data on the basis of triggers. Triggers in this

case are essentially conditions over changes in profile data (e.g., available bandwidth dropping below a certain threshold, or a change of the user's activity) which determine the delivery of a notification when met. In particular, when a trigger fires, the corresponding profile manager sends the new values of the modified attributes to the CONTEXT PROVIDER module, which should then re-evaluate policies.

3.1 Trigger-based Mechanism

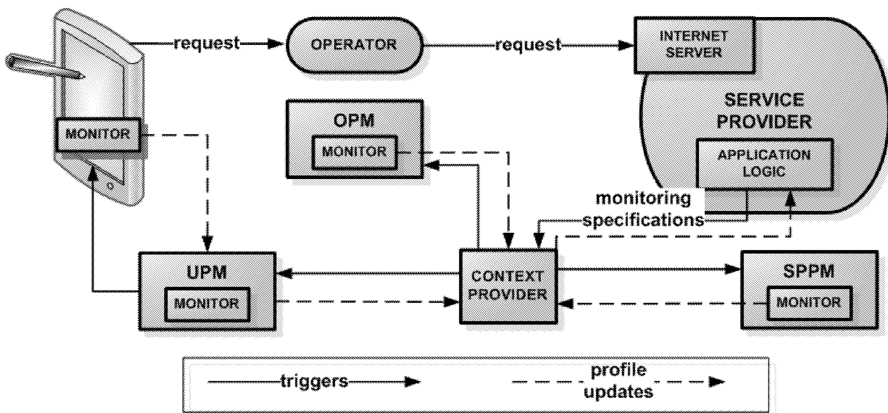

Figure 2. Trigger mechanism

Figure 2 shows an overview of the mechanism. To ensure that only useful update information is sent to the service provider, a deep knowledge of the service characteristics and requirements is needed. Hence, the context parameters and associated threshold values that are relevant for the adaptation (named *monitoring specifications*) are set by the service provider application logic, and communicated to the CONTEXT PROVIDER. Actual triggers are generated by the CONTEXT PROVIDER –according to the algorithms presented in the following of this section– and communicated to the proper profiles managers. Since most of the events monitored by triggers sent to the UPM are generated by the user device, the UPM communicates triggers to a light server module resident on the user's device. Note that, in order to keep up-to-date the information owned by the UPM, each user device must be equipped with an application monitoring the state of the device against the received triggers (named MONITOR in Figure 2), and with an application that updates the UPM when a trigger fires. Each time a profile manager receives an update for a profile attribute value that makes a trigger

fire, it forwards the update to the CONTEXT PROVIDER. Then, the CONTEXT PROVIDER re-computes the aggregated profile, and any change satisfying a monitoring specification is communicated to the application logic. In order to show the system behavior, consider the following example.

Example 4 *Consider the case of the streaming video service introduced in Example 1. Suppose that a user connects to this service via a UMTS connection, and that at first the available bandwidth is higher than 128kbps, and the user device has more than 4MB available memory. Thus, the CONTEXT PROVIDER, evaluating the service provider policies, determines a high* MediaQuality *(since rules R1 and R2 fire). Consequently, the service provider starts the video provision with a high bitrate. At the same time, the application logic sets a* monitoring specification *regarding* MediaQuality. *Analyzing policies, profile resolution directives, and context data, the CONTEXT PROVIDER sets triggers to the OPM and to the UPM/device, asking a notification in case the available bandwidth and the available memory, respectively, drop below certain thresholds. Suppose that, during the video provision, the user device runs out of memory. Then, the UPM/device sends a notification (together with the new value for the available memory) to the CONTEXT PROVIDER, which merges profiles and re-evaluates policies. Since this time policy evaluation determines a lower* MediaQuality *(since rule R3 fires), the video bitrate is immediately lowered by the application logic.*

3.2 Monitoring Specifications

In order to keep the re-evaluation of rules to a minimum, it is important to let the application logic to precisely specify the changes in context data it needs to be aware of in order to adapt the service. These adaptation needs, called *monitoring specifications*, are expressed as conditions over changes in profile attributes. As an example, consider the provider of the continuous streaming service shown in Example 1. The application logic only needs to be aware of changes to be applied to the quality of media. Hence, its only *monitoring specification* will be:

$$MediaQuality(X),\quad X \neq \$old_value_MediaQuality,$$

where *$old_value_MediaQuality* is a variable to be replaced with the value for the *MediaQuality* attribute, as retrieved from the aggregated profile. *Monitoring specifications* are expressed through an extension of the language used to define rule preconditions in our logic programming language (Bettini and Riboni, 2004). This extension involves the introduction of the additional special predicate *difference*, which has the

obvious semantics with respect to various domains, including spatial, temporal, and arithmetic domains. For instance, the *monitoring specification*:

Coordinates(*X*), *difference*(*X*, *$old_value_Coordinates*) > 200 *meters*

will instruct the CONTEXT PROVIDER to notify changes of the user position greater than 200 meters.

3.3 Minimizing Unnecessary Updates

In general, allowing the application logic to specify the changes in context data it is interested to does not prevent that unnecessary updates are sent to the CONTEXT PROVIDER. We define an update to the value of a profile attribute as *unnecessary* if it does not affect the aggregated profile. In the context of mobile service provisioning, the cost of unnecessary updates is high, in terms of client-side bandwidth consumption (since updates can be sent by the user's device), and server-side computation, and can compromise the scalability of the architecture. In order to avoid inefficiencies, the application logic does not directly communicate *monitoring specifications* to the profile managers. Instead, *monitoring specifications* are communicated to the CONTEXT PROVIDER, which is in charge of deriving the actual triggers and performing the optimizations that will be described in the following of this section.

Baseline algorithm

The baseline algorithm for trigger derivation consists of the following steps: *a)* set a trigger regarding the attribute A_i for each *monitoring specification* c_{A_i} regarding A_i, *b)* communicate the trigger to every profile manager, and *c)* repeat this procedure considering each precondition of the rules having A_i in their head as a monitoring specification. As a matter of fact, if A_i is an attribute whose value can be possibly modified by policies (i.e., an attribute that appears in the head of some policy rule), it is not sufficient to monitor the single A_i attribute. For instance, consider rule *R*2 in Example 1. The value of the *MediaQuality* attribute depends on the values of other attributes, namely *NetSpeed* and *AvMem*. Hence, those attributes must also be kept up-to-date in order to satisfy a *monitoring specification* regarding *MediaQuality*. For this reason, the CONTEXT PROVIDER sets new triggers regarding those attributes. Note that this mechanism must be recursively repeated accordingly to step *c)*. For example, since *NetSpeed*

depends on *AvBandwidth* and *Bearer*, the CONTEXT PROVIDER would set two triggers regarding those attributes. Generally speaking, for each *monitoring specification* c_{A_i} regarding an attribute A_i whose value was set by rule r'_{A_i}, triggers must be set for checking that the preconditions of rule r'_{A_i} are still valid, and for monitoring the preconditions of the other rules that can set a value for A_i.

The use of the baseline algorithm would lead to a number of unnecessary updates, as will be shortly explained in Example 5. We devised two optimizations, presented in the following of this section, which avoid a large number of unnecessary updates while preserving useful ones.

Optimization based on profile resolution directives

Any update that does not affect the profile obtained after the MERGE operation is unnecessary. Indeed, since we assume that neither policies, nor *profile resolution directives* can change during service provision, the aggregated profile does not change as long as the profile obtained after the *merge* operation remains the same. Hence, the first optimization considers the profile resolution directives used by the *merge* operation. The semantics of *merge* ensures that the value provided by an entity e_i for the attribute a_j can be overwritten only by values provided by e_i or provided by entities which have higher priority for the a_j attribute.

Example 5 *Consider the* profile resolution directive *on the attribute* AvBandwidth *given in Example 2 (PRD1). Suppose that the OPM (the entity with the highest priority) does not provide a value for* AvBandwidth, *but the SPPM and the UPM do. The value provided by the SPPM is the one that will be chosen by the MERGE module, since the SPPM has higher priority for that attribute. In this case, possible updates sent by entities with lower priority than the SPPM (namely, the UPM) would not modify the profile obtained after the MERGE operation, since they would be discarded by the merge algorithm. As a consequence, the CONTEXT PROVIDER does not communicate a trigger regarding* AvBandwidth *to the UPM.*

Note that, if the application logic defines a *monitoring specification* regarding an attribute whose value is *null* (i.e., an attribute for which no profile manager provided a value), the corresponding trigger is communicated to every entity that appears in the *profile resolution directive.*

Optimization based on rule priority

The second optimization exploits the fact that an attribute value set by the rule r'_{A_i} can be overwritten only by r'_{A_i} or by a rule having higher priority than r'_{A_i}. As a consequence, values set by rules having lower priority than r'_{A_i} are discarded, and do not modify the aggregated profile. For this reason, the preconditions of rules r_{A_i} having lower priority with respect to r'_{A_i} should not be monitored.

Generally speaking, for each *monitoring specification* c_{A_i}, an implicit *monitoring specification* is created for each precondition of the rule r'_{A_i} that determined the last value for A_i, and for the preconditions of the other rules having A_i in their head, and having higher priority than r'_{A_i}. Rules with lower priority do not generate triggers. For each *monitoring specification*, the CONTEXT PROVIDER creates a trigger and communicates it to the proper profile managers, as explained in Section 3.3.2.

Example 6 *Consider rules R2, R3 and R4 in Example 1. We recall from Example 3 that p(R2)>p(R3)>p(R4), where p(R) is the priority of rule R. Rules are evaluated in decreasing order of priority. Suppose that R2 does not fire, while R3 fires. In this case, the preconditions of R2 (the only rule with higher priority in this example) must be monitored, since they can possibly determine the firing of this rule. Preconditions of R4 must not be monitored since, even if they are satisfied, R4 cannot fire as long as the preconditions of R3 are satisfied. The preconditions of R3 must be monitored in order to assure that the value derived by the rule is still valid. In case the preconditions of R3 do not hold anymore, rules with lower priority (R4 in this example) can fire, and their preconditions are added to the set of implicit* monitoring specifications.

4. SOFTWARE ARCHITECTURE

The software architecture used to implement our middleware is shown in Figure 3. We have chosen Java as the preferred programming language, switching to more efficient solutions only when imposed by efficiency requirements. With regard to the inter-modules communication we have preferred, where possible, the web service paradigm.

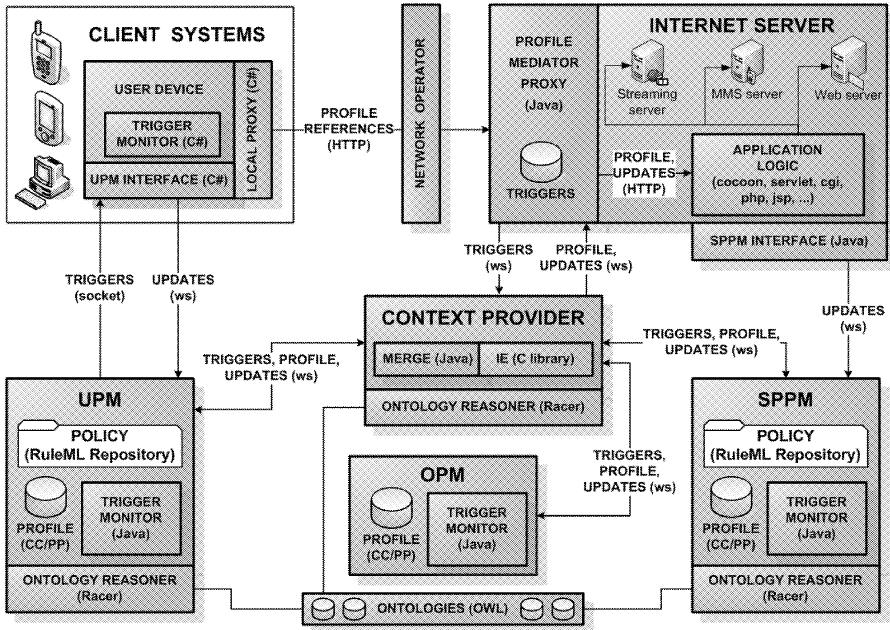

Figure 3. The software architecture

The PROFILE MEDIATOR PROXY (PMP) is a server-side Java proxy that is in charge of intercepting the HTTP requests from the user's device, and of communicating the user's profile (retrieved from the CONTEXT PROVIDER) to the application logic, by inserting profile data into the HTTP request headers. In this way, user profile data is immediately available to the application logic, which is relieved from the burden of asking the profile to the CONTEXT PROVIDER, and of parsing CC/PP data. The PMP is also in charge of storing the *monitoring specifications* of the application logic. When the PMP receives a notification of changes in profile data, it communicates them to the application logic by means of an HTTP message. Given the current implementation of the PMP, the application logic can be developed using any technology capable of parsing HTTP requests, including JSP, PHP, Cocoon, Java servlets, ASP .NET, and many others. The application logic can also interact with provisioning servers based on protocols other than HTTP. For instance, in the case of the adaptive streaming server presented in Section 5, profile data are communicated to the streamer by a PHP script through a socket-based protocol.

CC/PP parsing is performed using RDQL, a query language for RDF documents implemented by the Jena Toolkit[1] . User and service provider policies are represented in RuleML (Boley et al., 2001). The evaluation of the logic program is performed by an efficient, ad-hoc inference engine developed using C.

Profile data, policies and triggers are stored by the profile managers into ad-hoc repositories that make use of the MySQL DBMS. Each time a profile manager receives an update of profile data, the TRIGGER MONITOR evaluates the received triggers, possibly notifying changes to the CONTEXT PROVIDER. The UPM has some additional modules for communicating triggers to a server application executed by the user device. The communication of triggers is based on a socket protocol, since the execution of a SOAP server by some resource-constrained devices could be unfeasible.

The TRIGGER MONITOR module on the user's device is in charge of monitoring the status of the device (e.g., the battery level and available memory) against the received triggers. The LOCAL PROXY is the application that adds custom fields to the HTTP request headers, thus providing the CONTEXT PROVIDER with the user's identification, and with the URIs of his UPM and OPM. At the time of writing, modules executed on the user device are developed using C# for the .NET (Compact) Framework.

5. AN ADAPTIVE MULTIMEDIA STREAMING SERVICE

As already discussed in previous work (Maggiorini and Riboni, 2005), multimedia streaming adaptation can benefit from an asynchronous messaging middleware. In order to demonstrate the effectiveness of our solution, we implemented a streamer prototype based on the middleware described in this paper. We chose the VideoLan Client (*VLC*)[2] as a starting point to develop a customized client system, because it is an open platform and multiple operating systems are supported. The client is intended to run on windows workstations and windowsCE PDAs in order to achieve the largest possible population of users. VLC contacts the streaming service provider performing an HTTP request that has been modified by a local proxy adding in the HTTP headers the URIs of UPM and OPM. Then, the client waits for the video feed to come on a specific port. The HTTP request from VLC is received by the PMP module, which, as explained in Section 4, asks the CONTEXT PROVIDER for the aggregated profile information. The

[1] `http://jena.sourceforge.net/`
[2] `http://www.videolan.org/`

returned attribute/value pairs are included in the HTTP request header and the request is forwarded to the streamer application logic. Upon receiving the request, the streamer opens all the video files with the different encodings. Based on the context parameter values, the application logic selects an appropriate encoding, and the streamer starts sending over the network UDP packets containing frames belonging to the selected encoding. The streamer has been implemented on a Linux system. Network streaming is performed thanks to a specific file format, in which video data is already divided in packets, and a network timestamp is associated to each packet. Moreover, this streaming file format supports any kind of encoding, thus making the service independent by any specific format or codec.

We now illustrate how changes in context are detected and notified by the middleware to the streamer application logic. When the PMP module receives the user request, it recognizes that is directed to a continuous service, and retrieves from the media description all the monitoring specifications related to the requested feed, which in the case of our streamer prototype consist only of the *MediaQuality* parameter. Using the specification, the CONTEXT PROVIDER computes the set of required triggers, accordingly to the algorithms reported in Section 3, and illustrated by Example 6. Triggers are then set on the OPM, UPM/device to monitor available bandwidth and battery level, respectively. Upon firing of one of the triggers, the new value is forwarded to the CONTEXT PROVIDER, which recomputes the value for the *MediaQuality* parameter. If the new value differs from the previous one, it is forwarded to the PMP which issues a special HTTP request to the streamer application logic. The application logic selects a different encoding based on the new value; the feeder process is notified and forced to change the file from which the video frames are taken. The preliminary experiments performed with the current prototype are based on a full implementation and demonstrate the viability of our solution.

6. CONCLUSIONS

We presented the extension of the **CARE** middleware to support context-aware continuous services. In particular, we focused on optimizations to avoid unnecessary remote context updates which would strongly affect scalability. An adaptive video streamer has been used for a practical evaluation of the functionality of our middleware.

The natural extension of the **CARE** architecture will be related to session handoff. We experience a session handoff every time a user switches device and/or network connection. The main issue here is to efficiently describe all context data regarding the current session in order to guarantee a seamless

migration to the new device. In the case of our video streaming prototype, session handoff will be easy to accomplish, since session data can be easily described as the current frame number and media quality. In a more general scenario, involving a possibly huge number of context data, session handoff is much more challenging, and is the subject of future work.

References

Agostini, A., Bettini, C., Cesa-Bianchi, N., Maggiorini, D., Riboni, D., Ruberl, M., Sala, C., and Vitali, D., 2004, Towards highly adaptive services for mobile computing, *Proc. of IFIP TC8 Working Conference on Mobile Information Systems (MOBIS)*, Springer, pp. 121–134.

Bellavista, P., Corradi, A., Montanari, R., and Stefanelli, C, 2003, Context-aware middleware for resource management in the wireless Internet, *IEEE Trans. Soft. Eng., Special Issue on Wireless Internet*, **29**(12):1086–1099.

Bettini, C., and Riboni, D., 2004, Profile aggregation and policy evaluation for adaptive Internet services, *Proc. of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous)*, IEEE, pp. 290–298.

Boley, H., Tabet, S., and Wagner, G., 2001, Design rationale of RuleML: a markup language for Semantic Web rules, *Proc. of the International Semantic Web Working Symposium (SWWS)*, pp. 381–401.

Butler, M., Giannetti, F., Gimson, R., and Wiley, T., 2002, Device independence and the Web, *IEEE Internet Comp.*, **6**(5):81–86.

Chen, H., Finin, T., and Joshi, A., 2004, Semantic Web in the context broker architecture, *Proc. of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004)*, IEEE, pp. 277–286.

Chengy, R., Kaox, B., Prabhakary, S., Kwanx, A., and Tu, Y., 2005, Adaptive stream filters for entity-based queries with non-value tolerance, *Proc. of VLDB 2005, International Conference on Very Large Databases.*

Hull, R., Kumar, B., Lieuwen, D., Patel-Schneider, P., Sahuguet, A., Varadarajan, S., and Vyas, A., 2004, Enabling context-aware and privacy-conscius user data sharing, *Proc. of the 2004 International Conference on Mobile Data Management*, IEEE, pp. 187–198.

Klyne, G., Reynolds, F., Woodrow, C., Ohto, H., Hjelm, J., Butler, M. H., and Tran, L., 2004, Composite capability/preference profiles (CC/PP): structure and vocabularies 1.0. W3C recommendation, http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115/.

Maggiorini, D., and Riboni, D., 2005, Continuous media adaptation for mobile computing using coarse-grained asynchronous notifications, *2005 International Symposium on Applications and the Internet (SAINT 2005), Proc. of the Workshops*, IEEE, pp. 162–165.