Distributed Context Monitoring for the Adaptation of Continuous Services *†

Claudio Bettini Dario Maggiorini
Daniele Riboni
DICo - University of Milan, Italy
{bettini,dario,riboni}@dico.unimi.it

Abstract

This paper describes a middleware designed for distributed context acquisition and reconciliation intended to support the adaptation of *continuous* Internet services, like e.g., multimedia streaming. These services persist in time, and are characterized by multiple transmissions of data by the service provider, as a result of a single request from the user. Adapting these services to the current context requires the continuous monitoring of context data, and a real-time adjustment of the adaptation parameters upon the detection of a relevant context change. The proposed solution is based on asynchronous context change notifications, and specific techniques have been designed to minimize the number of unnecessary updates and the re-evaluation of policies. The paper also provides experimental results obtained by developing an adaptive video streaming system and running it on top of the proposed middleware.

1 Introduction

Various forms of adaptation of Internet services both in terms of *content* and *presentation* are currently performed by several commercial systems with different approaches. The amount and quality of context data available at the time of service provision highly influences the effectiveness of

*This work has been partially supported by Italian MUR (FIRB "Web-Minds" project N. RBNE01WEJT_005).

†The original publication is available at www.springerlink.com

adaptation, and the possibility to deliver services through multiple channels. Context data includes but is not limited to user profile and preferences, service provider current policies, user device capabilities, user device status, network resources availability, user location, local time, speed, as well as the content of the service request. This data comes from different sources (e.g., user, sensors, devices, transaction logs) and multiple sources may also provide conflicting values for the same type of data (e.g., location can be provided by the GPS on the user device as well as by the mobile phone operator). In the last years we have been working on the design and implementation of the *CARE* middleware [2, 7]. *CARE* supports the acquisition of context data from different sources, the reasoning with this data based on distributed policies, and the reconciliation of possibly conflicting information. The final goal of *CARE* is the delivery to the service application logic of a consistent and accurate description of the current context, so that the most appropriate adaptation can be applied during service provisioning.

Consistently with the approach taken by many adaptation systems, *CARE* was initially designed to compute the current context at the time of a user request for a specific service. This model is adequate for the large class of services that provide a single response from the server to each request from the user; examples are adaptive Web browsing, and location-based resource identification.

In this paper, we consider the particular class of *continuous* services. These services persist in time, and are typically characterized by multiple transmissions of data by the service provider as a result of a single request by the user. Examples are multimedia streaming, navigation services, and publish/subscribe services. 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. A straightforward solution is to constantly monitor these parameters, possibly by polling servers in the network operator 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 and network resource consumption, (*ii*) high response times due to the polling strategy, (*iii*) complexity of the application logic, and (*iv*) poor scalability, since for every user the service provider must continuously request context data and re-evaluate its policy rules.

The alternative approach we follow is to provide the application logic with asynchronous notifications of relevant context changes, on the basis of its specific requirements. However, when context data must be aggregated from distributed sources that 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 straightforward strategy of monitoring all context data, independently from the conflict resolution policies, and from the rules affecting the adaptation parameters, would be highly inefficient and poorly scalable. Indeed, computational resources would be wasted by communicating with certain context data sources when not needed, by monitoring unnecessary context data, and by re-computing dependency rules not affecting the adaptation parameters.

The main contributions of this paper are:

- The design and implementation of a middleware for context-awareness supporting continuous services through asynchronous context change notifications;
- Algorithms to identify context sources to be monitored 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;
- Theoretical proofs of the correctness of the proposed optimization algorithms, and an experimental evaluation of their effectiveness;
- Experimental results with a prototype of the new middleware coupled with an adaptive video streaming system. The resulting system has been extensively tested proving the efficiency of the context monitoring solution and the effectiveness of streaming adaptation. Experiments also include a comparison with a state-of-the-art commercial solution for adaptive streaming.

The rest of the paper is organized as follows. Section 2 discusses related work; Section 3 describes the *CARE* middleware architecture and features; Section 4 presents our architectural solution and the algorithms for the setup of asynchronous notifications and the minimization of unnecessary updates; Section 5 shows the software implementation; Section 6 presents the experimental evaluation; Section 7 concludes the paper.

2 Related work

While several frameworks have been proposed to support context awareness (see e.g., [3, 8, 9, 11, 14, 18]), our *CARE* middleware has as distinctive main features a distributed model for context acquisition and handling, and a sophisticated mechanism to deal with conflicts on data and policies provided by different sources. A more detailed comparison regarding these aspects can be found in [2]. The extension to the support of asynchronous context change notifications still preserves these unique features.

Many proposed architectures for context-awareness do not explicitly support adaptation for continuous services. On the other side, some existing architectures supporting this feature are bound to specific platforms. As an example, an extension of Java for developing context-aware applications by means of code mobility is proposed in [1]. Moreover, various commercial products (e.g., RealNetworks Helix server [19], TomTom Traffic [21]) adopt some forms of context-aware adaptation, requiring external entities (typically, applications running on the user device) to cooperate providing asynchronous updates of context data (e.g., available bandwidth and current location). These approaches are quite common in practice, but they do not provide a general solution to the addressed problem, since they are bound to specific applications.

Other frameworks try to optimize and adapt the behavior of applications running on the user device on the basis of context data, reacting to context changes (typically, availability of resources on the device, user location, and available bandwidth). One such architecture is described in [13]. In that architecture, context data are aggregated by modules running on the user device, and kept up-to-date by a trigger mechanism. Users can define policies by specifying priorities among applications as well as among resources of their devices. These policies are evaluated by a proper module, and de-

termine the applications behavior. Being limited to user-side adaptation, similar proposals cannot be applied to complex Internet services.

A proposal for a flexible mechanism for intra-session adaptation presenting many similarities with our approach can be found in [18]. That architecture includes a module devoted to apply adaptation policies on the basis of context changes. Adaptation policies are represented as ECA (event, condition, action) rules. Actions correspond to directives that modify the service behavior on the basis of conditions on context. Our middleware essentially extends this approach by considering a scenario in which context data and adaptation policies are provided by different entities. Hence, we provide formal techniques for aggregating distributed context data and for solving conflicts between policy rules. As a further contribution, we address the problem of minimizing the number of context change notifications, and the subsequent evaluation of adaptation policies by the architecture. These optimizations are intended to improve the scalability of the framework, especially when considering context data that may continuously change, such as the user location and the available bandwidth.

A more recent proposal for a flexible architecture supporting intra-session adaptation is presented in [17]. This proposal includes sophisticated techniques for resolving inconsistencies between conflicting context data; however, mechanisms for minimizing exchange of data and re-evaluation of rules are not specifically taken into account.

The work on stream data management has also a close 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., [12]). Since *filters* are the analogous of *triggers* used in our approach to issue asynchronous notifications, we are investigating the applicability to our problem of some of the ideas in that field.

3 The middleware architecture

The *CARE* (Context Aggregation and REasoning) middleware has been presented in detail elsewhere ([2, 7]). Here we only describe what is needed to understand the extension to support continuous services.

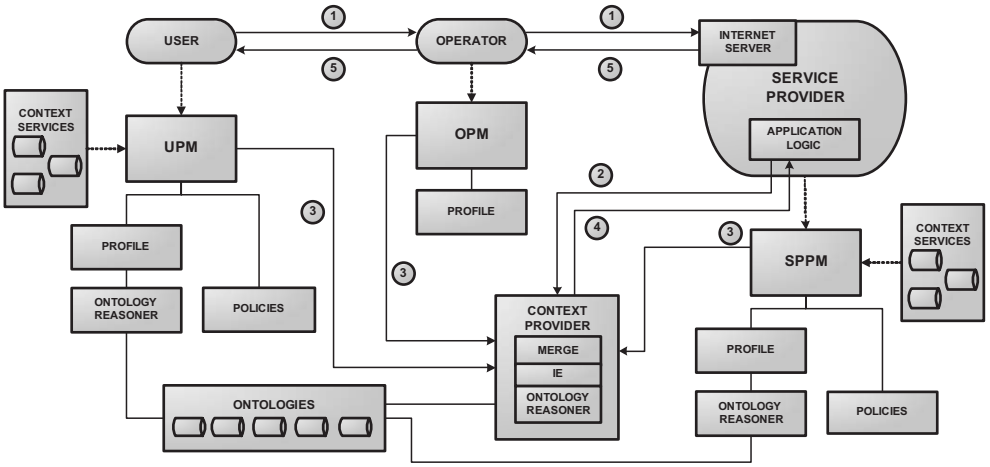


Figure 1: Architecture overview and data flow upon a user request

3.1 Overview

In our middleware, three main entities are involved in the task of building an aggregated view of context data, namely: the *user* with 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 UPM, OPM, and 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. Referring to a common business model, in the current implementation of CARE we do not associate the network operator profile manager with reasoning modules. However, the architecture can be easily extended for supporting different business models.

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 her device and the connectivity offered by a network operator. The HTTP header of the request includes the URIs of UPM and OPM. Then (step 2), the service provider forwards this information to the CONTEXT PROVIDER asking for the profile data needed to perform adaptation. In step 3, the CONTEXT PROVIDER 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.

3.2 Profile aggregation

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

3.2.1 Profile and policy representation

Essentially, profiles are represented adopting the CC/PP [15] 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 $\geq$ *128kbps **And*** Bearer = *'UMTS'*
Then Set NetSpeed=*'high'"*

R2: *"**If*** NetSpeed=*'high'* ***And*** AvMem $\geq$ *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 consid-

*ering 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*.*

3.2.2 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 that 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 the 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.*

The semantics of priorities actually depends on the type of the attribute. A more in-depth discussion of the *merge* mechanism can be found in [7].

Once conflicts between attribute values provided by different profile managers are resolved, 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. Details on rule conflict resolution can be found in [7].

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 in Example 2), 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.

4 Supporting continuous services

In this section we describe a trigger mechanism for supporting continuous services. This mechanism allows profile managers to asynchronously notify 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) that determine the delivery of a notification when met. In particular, when a trigger fires, the corresponding profile

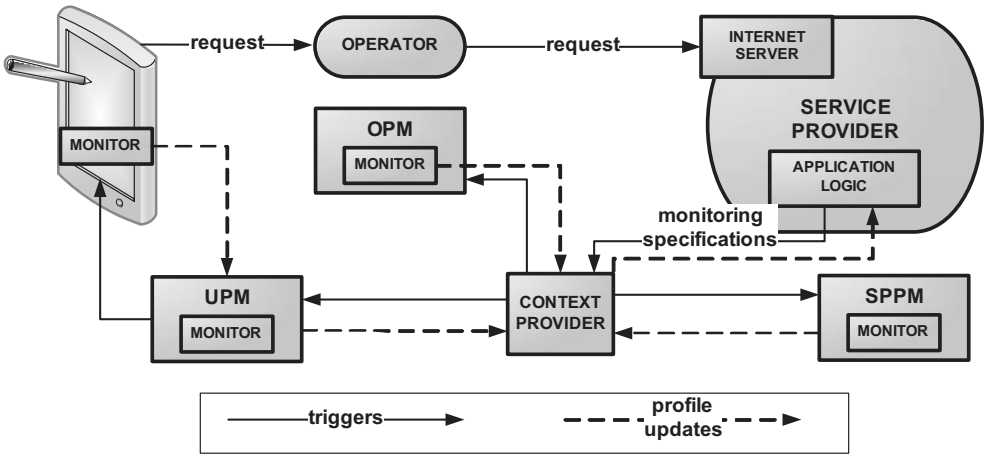


Figure 2: Trigger mechanism

manager sends the new values of the modified attributes to the CONTEXT PROVIDER module, which should then re-evaluate policies.

4.1 Trigger-based mechanism

Figure 2 shows an overview of the mechanism. In order 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 Section 4.3 – and communicated to the proper profile 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 the UPM receives an update for a profile attribute value that makes a trigger fire, it forwards the update to the CONTEXT PROVIDER. Finally, 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 of 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. This time, policy evaluation determines a lower* MediaQuality *(since rule R3 fires). Thus, the* CONTEXT PROVIDER *notifies the application logic, which immediately lowers the video bitrate.*

4.2 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:

$$MediaQuality(X),\ X \neq \$old\_value_{MediaQuality}.$$

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 [7].

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*:

$$Coordinates(X),\ difference(X,\ \$old\_value_{Coordinates}) > 200 \text{ meters},$$

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

4.3 Minimizing unnecessary updates

In general, allowing the application logic to specify the changes in context data it is interested in does not guarantee that unnecessary updates are not sent to the CONTEXT PROVIDER. We define an update of 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, *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 Sections 4.3.2 and 4.3.3.

4.3.1 Baseline algorithm

The baseline algorithm for trigger derivation is shown in Algorithm 1, and 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.

The completeness of Algorithm 1 is shown by the following proposition:

Proposition 1 *Given a monitoring specification c_{A_i} regarding attribute A_i, and a set of policy rules P, the baseline Algorithm 1 calculates a set of triggers t that is sufficient to detect any change in the value of A_i that satisfies the monitoring specification c_{A_i}.*

Algorithm 1 Baseline algorithm for trigger derivation

Input: Let C be the set of *monitoring specifications*; c_{A_i} be a *monitoring specification* regarding the attribute A_i; R_{A_i} be the set of rules r_{A_i} having A_i in their head; r'_{A_i} be the rule that determined the value for A_i in the aggregated profile (if such rule exists); $b(r_{A_i})$ be the set of preconditions pc of r_{A_i}.

Output: A set of directives regarding the communication of triggers to profile managers.

```
1:  for all c_{A_i} ∈ C do
2:    for all r_{A_i} ∈ R_{A_i} do
3:      for all pc ∈ b(r_{A_i}) | pc ∉ C do
4:        if r_{A_i} ≠ r'_{A_i} then
5:          C := C ∪ pc
6:        else
7:          C := C ∪ ¬pc
8:        end if
9:      end for
10:   end for
11:   trigger t := "if c_{A_i} then notify_update(A_i)"
12:   communicate(t, ProfileManagers);
13: end for
```

As a matter of fact, the value of an attribute A_i in the aggregated profile can change in three cases: *(i)* the value of A_i is changed by a profile manager; *(ii)* the preconditions of the rule $r'_{A_i} \in P$ that set the value of A_i are no more satisfied; *(iii)* the preconditions of other rules $r_{A_i} \in P$ possibly setting a value for A_i are satisfied. Case *(i)* is addressed by Algorithm 1 in lines 11 and 12. With regard to cases *(ii)* and *(iii)*, for each *monitoring specification* c_{A_i} regarding an attribute A_i whose value was set by rule r'_{A_i} the algorithm creates new monitoring specifications pc for checking that the preconditions of r'_{A_i} are still valid (line 7), and for monitoring the preconditions of the other rules r_{A_i} that can possibly set a value for A_i (line 5). The algorithm is recursively repeated for the newly generated monitoring specifications pc.

Example 5 *Consider rule R2 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 monitoring specifications regarding those attributes. This mechanism is recursively repeated. For example, since* NetSpeed *depends on* AvBandwidth *and* Bearer *(rule* $R1$*), the* CONTEXT PROVIDER *generates new monitoring specifications for those attributes.*

The use of the baseline algorithm would lead to a number of unnecessary updates, as will be shortly explained in Example 6. We devised two optimizations, one exploiting *profile resolution directives* (see Section 3.2.2), and the other exploiting priorities over rules. These optimizations, presented in the following of this section, avoid a large number of unnecessary updates while preserving useful ones.

4.3.2 Optimization based on profile resolution directives

A number of unnecessary updates are the ones that do not affect the profile obtained after the MERGE operation, as shown by the following lemma.

Lemma 1 *Given an aggregated profile* p*, a set of policy rules* R*, and a set of* profile resolution directives PRD*, any changes to profile attributes that do not affect the result of the* MERGE *operation do not affect* p*.*

Proof. The aggregated profile p is obtained through the evaluation of the logic program P against the profile obtained after the MERGE operation. We recall that P is obtained from R adding a fact $A_i(\overline{x})$ for each context data $A_i = \overline{x}$ obtained after the MERGE operation. P, in turn, is transformed before evaluation considering the *profile resolution directives* PRD. Since we assume that neither policies R, nor *profile resolution directives* PRD can change during service provision, the logic program obtained after these transformations does not change as long as the profile obtained after the MERGE operation remains the same. Since in [7] we proved the program model uniqueness of policies expressed in our language, we have that different evaluations of the same logic program determine the same aggregated profile p. □

Our 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. Hence, if the application logic defines a *monitoring specification* regarding an attribute a_j, the first optimization consists in communicating the trigger only to e_i and to those entities having higher priority than e_i for a_j. Note that, if no profile manager provided a value for a_j, the corresponding trigger is communicated to every entity that appears in the *profile resolution directive*.

Example 6 *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*.*

The algorithm corresponding to the first optimization is shown in Algorithm 2.

Theorem 1 *The optimization applied by Algorithm 2 preserves the completeness of the trigger generation mechanism.*

Proof. The demonstration easily follows from Lemma 1, since Algorithm 2 determines the communication of triggers to and only to those entities whose updates can modify the result of the MERGE operation (lines 3 to 5, and lines 7 to 9). □

4.3.3 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} with respect to the head predicate 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.

Algorithm 2 Communicating triggers to the proper profile managers.
Input: Let $(\{\mathbf{E_3}\{, \mathbf{E_2}\{, \mathbf{E_1}\}\}\})$ be the priority over entities for the attribute A; $\mathbf{E_r}$ be the entity among $E = \{E_1, E_2, E_3\}$ providing the value obtained by the MERGE module for A; p be the aggregated profile, $A(X) \in p$; let the application logic set a *monitoring specification* c involving A.
Output: A set of directives regarding the communication of triggers to profile managers.

1: trigger t := "if c then notify_update(A)"
2: **if** $A(X)$, $X = null$ **then**
3: **for all** $E_i \in \{E_3\{, E_2\{, E_1\}\}\}$ **do**
4: communicate(t, E_i)
5: **end for**
6: **else**
7: **for** $j = r$ to 3 **do**
8: communicate(t, E_i)
9: **end for**
10: **end if**

Lemma 2 *Given an aggregated profile p, a set of policy rules P, and an attribute A_i whose value was set by rule $r'_{A_i} \in P$, any rule $r_{A_i} \in P$ having lower priority than r'_{A_i} with respect to the head predicate A_i does not affect p, as long as the preconditions of r'_{A_i} hold.*

Proof. Rules in P having the same attribute A_i in their head are evaluated in decreasing order of priority, and when a rule fires, rules having lower priority are discarded. As a consequence, the evaluation of r'_{A_i} precedes the evaluation of r_{A_i}. If the preconditions of r'_{A_i} hold, rule r'_{A_i} fires, and rule r_{A_i} is discarded. Thus, r_{A_i} cannot affect p. □

Algorithm 3 is the optimized version of the baseline Algorithm 1. 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 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 new monitoring specification. For each monitoring specification, the CONTEXT PROVIDER creates a trigger and communicates it to the proper profile managers according to Algorithm 2.

Algorithm 3 Derivation of implicit monitoring specifications.

Input: Let C be the set of *monitoring specifications*; c_{A_i} be a *monitoring specification* regarding the attribute A_i; R_{A_i} be the set of rules r_{A_i} having A_i in their head; $p(r_{A_i})$ be the priority of the rule r_{A_i}; r'_{A_i} be the rule that determined the value for A_i in the aggregated profile (if such rule exists); $b(r_{A_i})$ be the set of preconditions pc of r_{A_i}.

Output: A set of directives regarding the communication of triggers to profile managers.

```
1: for all c_{A_i} ∈ C do
2:   for all r_{A_i} | p(r_{A_i}) ≥ p(r'_{A_i}) do
3:     for all pc ∈ b(r_{A_i}) | pc ∉ C do
4:       if r_{A_i} ≠ r'_{A_i} then
5:         C := C ∪ pc
6:       else
7:         C := C ∪ ¬pc
8:       end if
9:     end for
10:  end for
11:  Apply Algorithm 2 to c_{A_i}
12: end for
```

Example 7 *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 than R3 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 monitoring specifications.*

Theorem 2 *The optimization applied by Algorithm 3 preserves the completeness of the trigger generation mechanism.*

Proof. The demonstration follows from Lemma 2. As a matter of fact, for each monitoring specification c_{A_i} regarding attribute A_i whose value was set by rule r'_{A_i}, Algorithm 3 generates new monitoring specifications considering the preconditions of r'_{A_i} and the preconditions of every other rule r_{A_i} such that $p(r_{A_i}) > p(r'_{A_i})$ (line 2). □

An important complexity result regarding Algorithm 3 is shown by the following theorem.

Theorem 3 *The time complexity of Algorithm 3 is $O(N)$, where N is the total number of rules in the joined logic program.*

Proof. We call P the logic program obtained by joining user and service provider policy rules. According to Algorithm 3, triggers are generated by traversing part of the rule dependency graph of P. We recall that, given a logic program P, $RDG(P)$ is a directed graph whose nodes are the rules forming P. The graph contains an edge from R to R' iff the head predicate of rule R' belongs to the set of body predicates of R. Since we guarantee that the rule dependency graph of the logic program P is acyclic [6], Algorithm 3 terminates in at most K steps, where K is the number of rules in P, $K \leq N$. □

5 Software architecture

The software architecture used to implement our middleware is shown in Figure 3. The current implementation improves the one presented in [5]. In particular, part of the CONTEXT PROVIDER modules have been rewritten in order to optimize computationally intensive tasks such as profiles merge and policy transformation. Moreover, more efficient protocols have been adopted for the communication between the distributed modules of *CARE* with respect to the previous implementation.

We have chosen Java as the preferred programming language. However, the most computational intensive algorithms have been developed in C, and integrated into the corresponding modules using JNI.

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

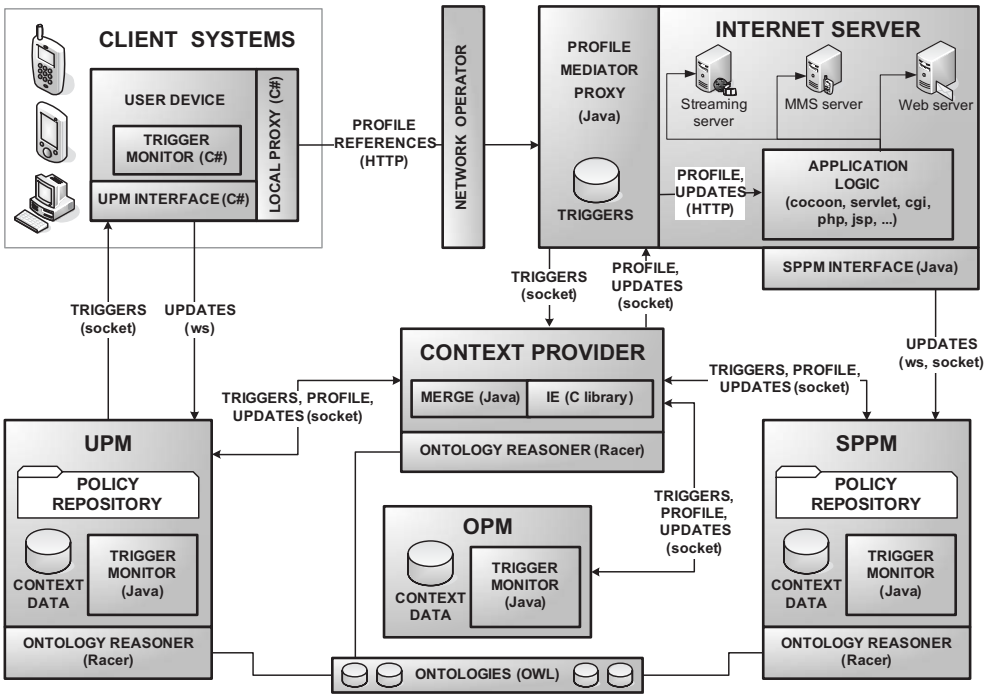


Figure 3: The software architecture

headers. In this way, user profile data is immediately available to the application logic, which is relieved from the burden of asking the remote 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 HEAD 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. As an example, in the case of the adaptive streaming server presented in Section 6.2, profile data is communicated to the streamer by a PHP script through a socket-based protocol.

CC/PP profiles are represented by means of Java classes, and communicated by profile managers to the CONTEXT PROVIDER by means of the socket-based binary serialization of Java objects. The evaluation of the logic program is performed by an efficient, ad-hoc inference engine [7] developed

using C.

Context 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 infeasible.

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 her UPM and OPM. At the time of writing, modules executed on the user device are developed using C# for the .NET (Compact) Framework. A command-line proxy is also available for Linux clients.

6 Experimental evaluation

In order to evaluate our solution, we have performed both experiments with the optimization algorithms for trigger derivation, and experiments with an adaptive video streamer.

6.1 Experiments with the optimization algorithms

The aim of the optimization algorithms presented in Section 4.3 is to avoid the generation of triggers that produce unnecessary updates of context data. The correctness of the algorithms has been proved by Theorems 1 and 2. In this section we report the results of extensive experiments intended to evaluate the reduction rate of the number of triggers that are generated by Algorithms 2 and 3 (called *reduction rate* in the rest of this section).

The reduction rate obtained by applying the optimization algorithms depends on various parameters affecting our rulesets. Rulesets in the experiments were randomly generated; each rule contained a random number of preconditions between 1 and 7, the maximum depth of rule chaining was 3, and each ruleset contained a random number (up to 7) of conflicting rules

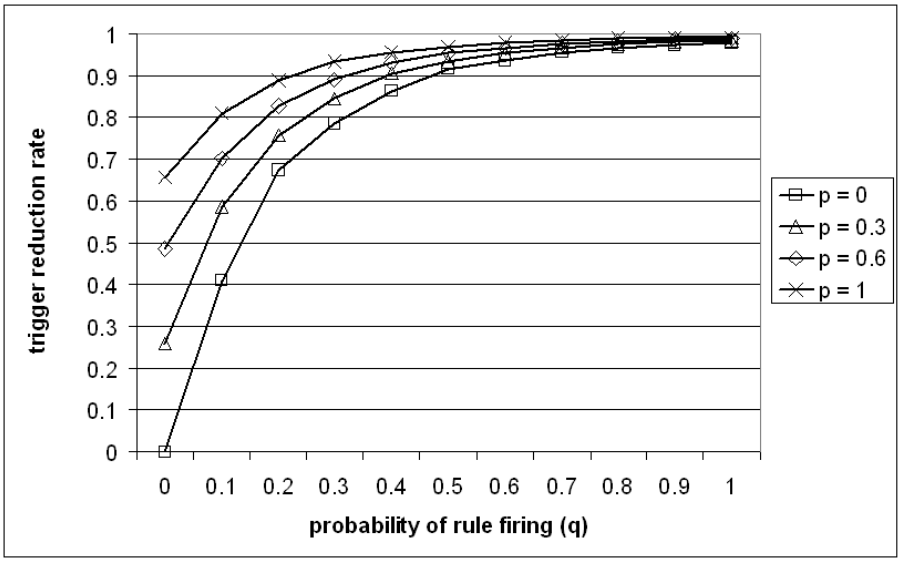


Figure 4: Experimental results with the optimization algorithms: p is the probability that an entity provides a value for a context attribute

for each head predicate. Consistently with the current implementation of our architecture, we assumed that at most 3 entities can provide a value for each context attribute.

The reduction rate also depends on the probability p that a particular entity provides a value for a certain context data, and on the probability q that a particular rule fires. On the one hand, p impacts on the reduction rate obtained by Algorithm 2: the higher the probability p, the higher the probability that the value of a context attribute a is set by an entity e_j with high priority for that attribute. Then, the reduction rate obtained by Algorithm 2 grows with p, since – according to Algorithm 2 – triggers are communicated only to e_j and to those entities having higher priority than e_j with respect to a.

On the other hand, q affects the reduction rate obtained by Algorithm 3: the higher q, the higher the probability that the value of a is set by a policy rule r_j having high priority for a. Then, the reduction rate obtained by Algorithm 3 grows with q, since – by applying Algorithm 3 – the rules' preconditions to be monitored are those of r_j, and those of the rules having the same head predicate of r_j but higher priority.

Since it is almost impossible to estimate realistic values for p and q, we

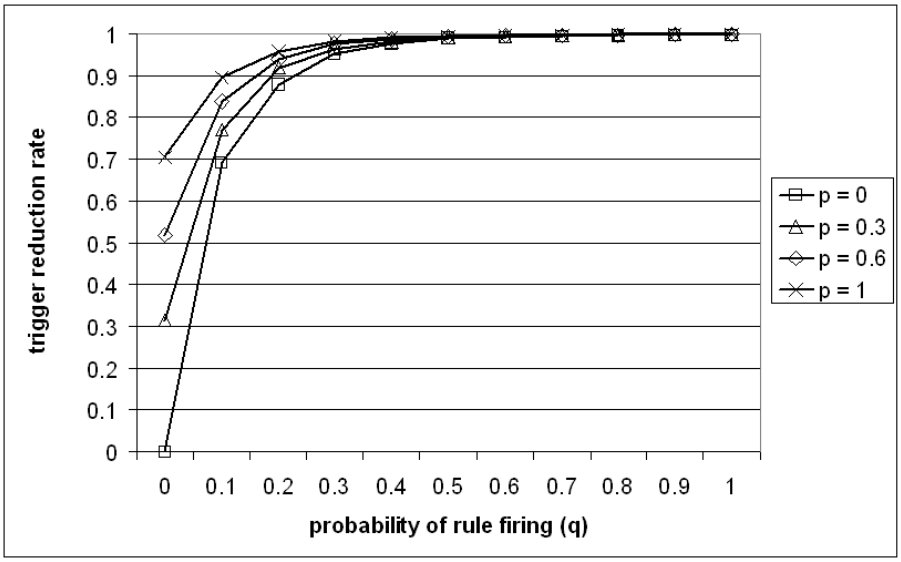


Figure 5: Experimental results with the optimization algorithms and more complex rulesets: p is the probability that an entity provides a value for a context attribute

have studied the behavior of our algorithms with respect to these parameters. Experimental results are shown in Figure 4. Results are averages of 100 runs with different rulesets, and different sets of context data provided by each entity. As expected, our algorithms do not produce any optimization in the particular case in which both p and q are equal to 0. However, with more reasonable values for p and q, the reduction rate obtained by our algorithms is sufficient to avoid most of the unnecessary updates. For instance, with $p = q = 0.3$, our optimization algorithms produce a reduction of 84% on the number of the generated triggers with respect to the baseline Algorithm 1. The reduction rate grows to 96% with $p = q = 0.6$.

We have repeated the same experiments using more complex rulesets, in which the maximum depth of rule chaining was 4. In this series of experiments, each rule contained up to 10 preconditions, and each ruleset contained up to 10 conflicting rules for each head predicate. The other parameters had the same values as in the previous experiment. The results, reported in Figure 5, show that the reduction rate obtained by our optimization algorithms is higher with more complex rulesets. For instance, with this setup it is sufficient to have $p = q = 0.3$ to obtain a reduction rate of 96%.

6.2 Experiments with an adaptive video streaming service

As already discussed in previous work [16], 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.

6.2.1 Our adaptive streaming system

We have implemented our novel adaptive streaming system using the C^{++} language on the Linux platform. Our streamer is able to concurrently read data from a variable number of different video files while keeping frames synchronized. Switching between video files is performed only upon request by the underlying middleware. Upon receiving a streaming request, the streamer loads into memory all the video files associated with the requested content. Based on the context parameter values, the application logic selects an appropriate encoding for the specific request. Frames are then sent to the client application by means of the UDP protocol.

Network streaming is performed thanks to an ad-hoc file format. Our custom format is a streamer-friendly version of the AVI encapsulation format. Metadata are removed from the original AVI file, while frames are isolated and divided into fixed size chunks that will be used as UDP payload. Packets are then tagged with timestamps and other relevant information (i.e., frame size). Hence, our streaming-friendly encapsulation format is essentially codec-independent.

With regard to the client application, we chose the well-known VideoLan Client (*VLC*) [22] as a starting point to develop a customized client system. Our choice is motivated by the fact that VLC is an open-source product, and it supports multiple operating systems. Our ad-hoc client is intended to run on Linux systems, Windows workstations, and WindowsCE PDAs, in order to achieve the largest possible population of users. In addition to graphical user interface modifications, we have implemented two new components for VLC: a network adapter and a demuxer. The network adapter is in charge of communicating context data retrieved by *CARE* to the streaming server, by means of an HTTP request. The HTTP request coming from the VLC network adapter is intercepted by the PMP module, which, as explained in Section 5, is in charge of requesting the aggregated context data to the CONTEXT PROVIDER. Context data are then included

into the HTTP request header as attribute/value pairs, and the request is forwarded to the streamer application logic. The demuxer is in charge of collecting frame segments coming from the network component, reconstructing the frame, and communicating the resulting frame to the correct decoder.

6.2.2 The interaction between *CARE* and the streamer

We now illustrate how changes in context are detected and notified by the *CARE* middleware to the streamer application logic. When the PMP module receives the user request, it retrieves the monitoring specifications related to the requested video – that in the case of our streamer prototype consist only of the *MediaQuality* parameter. According to this specification, the CONTEXT PROVIDER computes the set of required triggers, applying the algorithms reported in Section 4, and illustrated by Example 7. Triggers are then communicated to the OPM and to the UPM/device, in order to monitor available bandwidth and battery level, respectively. Upon firing of a trigger, 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 streamer process is notified and forced to change the file from which the video frames are read. The transmission restarts from the beginning of the last frame being transmitted, and the client will discard any UDP packets belonging to a partially transmitted frame.

6.2.3 Experimental Setup

The experiments performed with the current prototype are based on a full implementation and demonstrate the viability of our solution. During our experiment we setup a testbed and emulate network congestion in order to observe the perceived quality of streamed media under different conditions. At first, we run a set of experiments without performing adaptation, and then another set while using *CARE* to adapt the streamer media quality on the basis of context. The streaming behavior is evaluated by measuring the number of frames per second received by the decoder. In the ideal case this number should always be 25 (since we are using PAL). When the number of frames per second drops below the expected value we start to experience

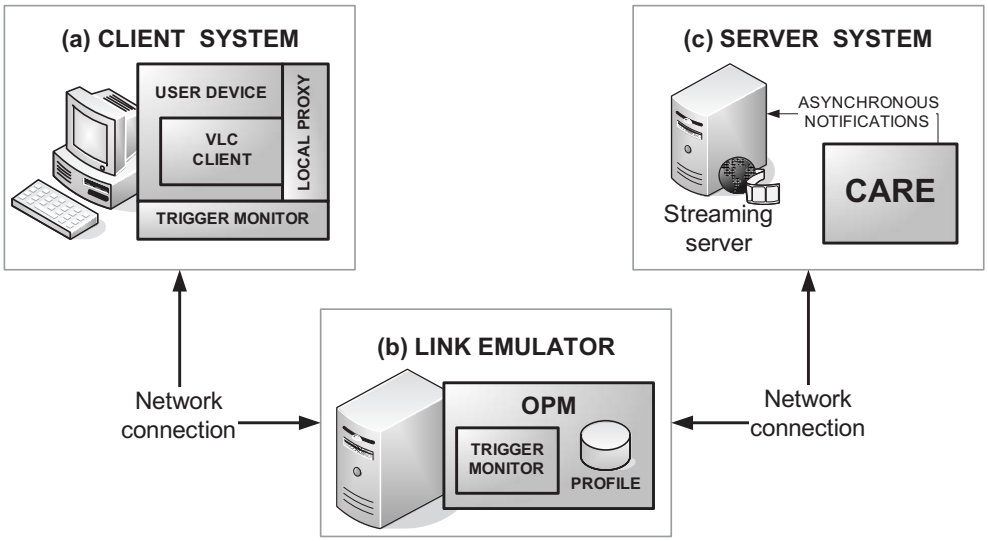


Figure 6: Experimental setup

mosaic effects, stop-and-go video, and random artifacts.

The experimental setup is shown in Figure 6. In order to simulate changes in the available bandwidth, client/server communications are mediated by a link emulator machine (*b*). This machine runs a live distribution of Unix. The Dummynet [20] link emulator software is used to limit available bandwidth in both directions, in order to simulate network congestions. This machine also hosts the OPM module, which is in charge of notifying the CONTEXT PROVIDER with changes in the available bandwidth according to our trigger mechanism.

The client system (*a*) hosts the VLC client, as well as the *local proxy* used to identify the user and her profile managers. A trigger monitor is in charge of monitoring the device resources (e.g., available memory) according to the received triggers.

The server system (*c*) is dedicated to host the streaming server as well as the remaining components of the *CARE* middleware.

6.2.4 Observed results

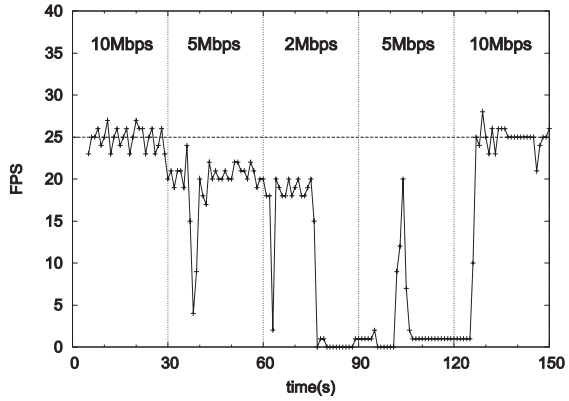
The video streaming sent to the VLC client is encoded using XviD [23] – an open-source MPEG-4 codec – using three different sets of parameters for average bandwidth request and quantization matrix size. In order to ensure a clear visual feedback during the experiment, we kept the difference between

Figure 7: A detail of the same frame at three different levels of encoding (high, medium, and low resolution, respectively)

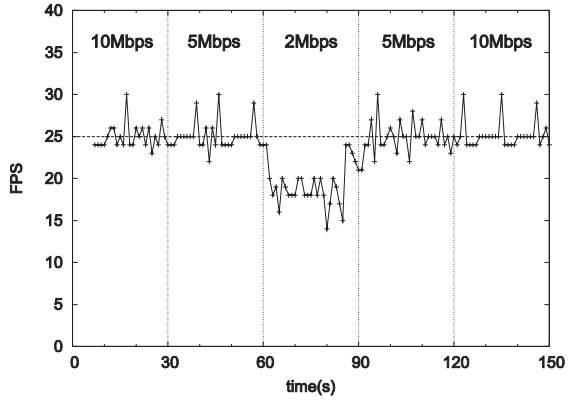
encoding parameters wide. The first encoding (high quality) has an average bandwidth of 512 KBps and a quantization of 4 bits, while the other two (medium and low quality, respectively) have 128 KBps with 16 bits and 64 KBps with 64 bits, respectively. Figure 7 shows the appearance of a detail of the same frame, using the different encodings. Due to the VBR encoding and some buffering effects in the kernel protocol stack, the aforementioned bandwidths cope nicely with a medium bandwidth on transmission of 10, 5 and 2 Mbps, respectively. These three bandwidth levels are meaningful for the experiment since they are three possible values for link quality after handshake in WiFi networks.

The experiment has a duration of 150 seconds divided into five 30-seconds segments that we will call *slots*. The bandwidth allowed by the link emulator is changed at every slot transition. During the first slot the network link is unconstrained (i.e., available bandwidth is 10 Mbps, due to our setup); in the second slot the allowed bandwidth is limited to 5 Mbps, and in the third it is limited to 2 Mbps. Then, we scale up again first to 5 Mbps, and then to unconstrained in the last slot.

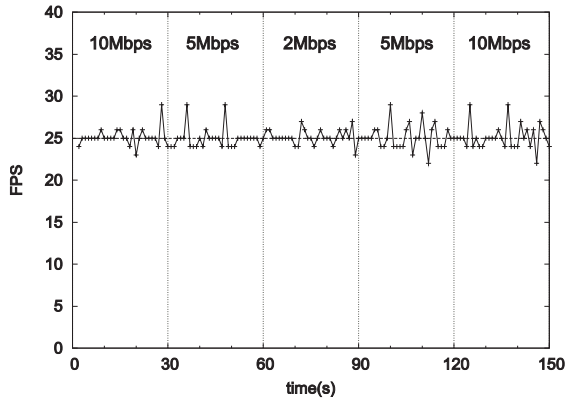
At first, we ran the experiment without performing adaptation. The resulting frames per second (FPS) perceived by the player for the three



(a) High-resolution video



(b) Medium-resolution video



(c) Low-resolution video

Figure 8: Experimental results with no adaptation

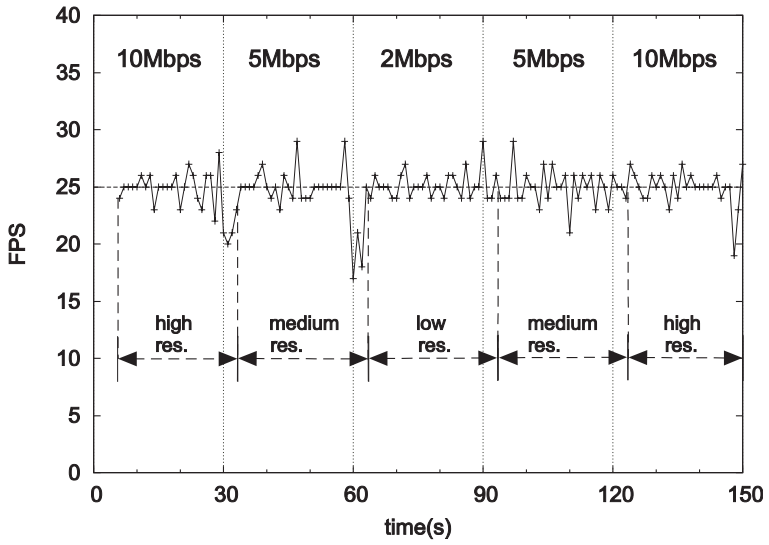


Figure 9: Results with our adaptive streamer with variable bandwidth

video encodings are reported in Figure 8. As we can see in Figure 8(a), using the high bitrate video, the streamer is able to provide the required 25 frames per second during the first slot. Then, when the available bandwidth drops to 5Mbps the streamer receives around 20 frames per second. In the third timeslot congestion is too strong, and frames per second drop to nearly zero. Strangely enough, even if the link is less congested during the forth slot, video quality does not improve. This behavior has been encountered in a significant number of experiments; we suppose that this is due to VLC internal buffering and decoder issues. Only in the last timeslot – when bandwidth is unconstrained – we can observe a normal playout again. Streaming the medium bitrate video (Figure 8(b)) we can observe that network congestion becomes a problem only in the third slot. From the point of view of the user experience the drop to 20 FPS is definitely perceivable and the resulting quality would not be normally acceptable if not for a short period of time. Figure 8(c) shows that the streaming of the lowest bitrate video is not affected by the network congestion. However, its quality is not comparable with the other two video encodings.

The improvement obtained by means of adaptation can be seen in Figure 9. The adaptation of media quality to changes of network bandwidth is almost immediate. The asynchronous notifications sent by the network operator profile manager (OPM) have been sent introducing a three seconds

delay in order to be more realistic. This delay is intended to model an upper bound for the network latency in the notification of context updates, as well as possible high loads at the profile managers. Without this delay, due to the buffering effects of the streaming system, it is almost impossible to perceive the data loss when switching between the video encodings.

From the point of view of the visual experience of the final user, experimental results have been very positive. After a small data loss at the beginning of the second and third slots, the playout proceeds smoothly at the adapted bitrate. A video capture of the experiments can be found at the project Web site [10].

6.2.5 Comparison with a commercial solution

In real production environments a number of proprietary software solutions are available for video streaming. One of the most widely adopted solutions at the time of writing appears to be the Helix Server from RealNetworks [19].

From an architectural point of view, the RealMedia system is codec-dependent; this means that every clip needs to be converted in order to get streamed. On the contrary, our streamer supports multiple codecs: hence, any kind of MPEG-like frame-oriented video content inside an AVI encapsulation can be sent to the client. Moreover, while RealMedia encoding is proprietary, MPEG is an open standard, and many implementations exist (both open- and closed-source). With respect to the video quality, when dealing with low bitrates the RealMedia format is generally better suited than MPEG. As a matter of fact, when encoding for very low bandwidth, MPEG usually exposes a mosaic-like effect, while RealMedia adopts fuzzy wavelet-style degradation, thus rendering images that are more pleasant for the final user. On the other hand, in our experiments, when encoding is performed for high bitrates the rendering quality of MPEG clips seems to be better than the one of the RealMedia format.

In order to compare our solution with a commercial product, we have repeated the same experiment proposed in the previous section using Helix server, the RealPlayer client, and the same content re-encoded in RealMedia format. Multiple runs of the experiment have been performed to ensure consistent results. The results obtained with Helix, reported in Figure 10, are difficult to compare with the ones obtained with our streamer, since the RealPlayer client is closed-source and we have not been able to produce FPS

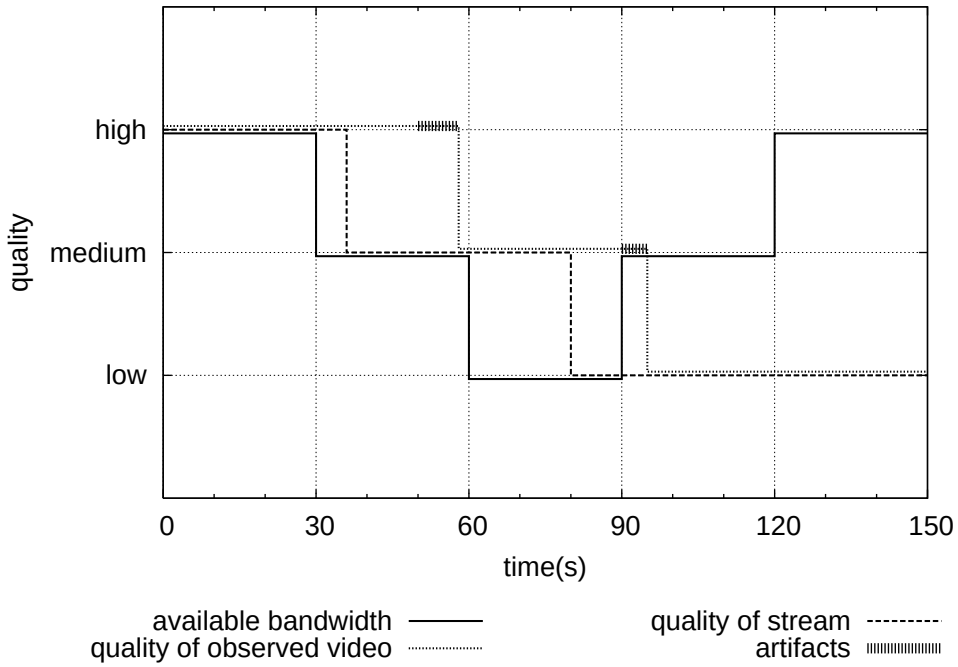


Figure 10: Results with the Helix streaming server with variable bandwidth

statistics. Hence, we will report observations based on user experience, and some technical observations based on the available knowledge on Helix.

We have observed that during the first slot the playout proceeds smoothly. At the beginning of the second slot, the link becomes congested, and after 6 seconds the RealPlayer information panel shows a quality drop, even if still not perceivable by the user due to buffering. Data loss becomes visible on the screen as artifacts at the end of the second slot, from second 50 to second 58. At the beginning of the third slot, the link becomes even more congested. This time, a quality drop occurs after 20 seconds; artifacts can be seen 10 seconds later, and last for several seconds. Interestingly, even if the available bandwidth increases at the fourth and fifth slots, the stream quality does not improve, and the lowest bitrate video continues to be streamed until the end of the experiment. To our knowledge, technical specifications motivating this behavior are not publicly available, and this may be a specific design choice.

From the point of view of the user experience, the adaptation performed by Helix is very effective. With respect to what observed using our streamer

with the same experimental setup, the artifacts appear a bit later when encodings are switched, last longer, and – due to the specific encoding – have less impact on the global perception of the video stream. On the other hand, our streamer is faster to react in the case of congestion, and it is also able to increase quality when more bandwidth becomes available, while Helix in slots 4 and 5 continues to produce low bitrate video.

We proposed the experiment to a group of users not involved in the project in order to evaluate perceptual comparison. From a perceptual point of view, users reported to like more the RealMedia artifacts produced by Helix; the MPEG mosaic effects produced by our streamer seem to be too invasive in the picture, and can be better tolerated in low-rate encodings, thus making RealMedia even more desirable for low bitrates. The average user does not pay attention to the fact that Helix does not switch back to high-quality when more bandwidth becomes available, probably because she is not aware of the bandwidth increase. However, a minority of users, noticing that the MPEG stream produced by our streamer does switch back to high-quality in slots 4 and 5, declared a preference for our solution.

Overall, the experiment showed that our prototype adaptive streamer coupled with the *CARE* middleware, despite the limited functionalities, offers a user experience close to that of a leading commercial solution.

7 Conclusions and future work

We presented the extension of the *CARE* middleware to support context-aware continuous services. In particular, we focused on the definition of optimization algorithms aimed at minimizing the exchange of data through the network, and the time to re-evaluate the adaptation rules.

An adaptive video streamer has been used for a practical evaluation of the functionality of the *CARE* extension. Despite our middleware is not specifically designed for streaming, and supports arbitrary services, experimental results have shown that the user experience achieved by our streaming system compares well with the one provided by a leading commercial streamer that adopts an ad-hoc technique to perform adaptation. We believe that streaming adaptation would benefit from using context acquisition middleware such as *CARE*. Indeed, a more effective adaptation may be obtained by considering multiple context parameters, including those that describe

the status of resources on the client device (e.g., available memory and battery level), and possibly the user's activity and surrounding environment. Other advantages are the decoupling of the streamer from context acquisition modules (like network probing modules) and the ability to acquire parameters from multiple sources.

We are investigating various possible extensions and enhancements. In particular, we are trying to improve the scalability of our middleware by means of caching techniques. We are also interested in testing the integration of numerous sources of context data (i.e., sensors) into our framework.

As already mentioned in Section 3, our architecture can also interact with ontological services in order to reason with complex context data (e.g., user activities, and surrounding environment). The main issue is to find a satisfying compromise between expressiveness and complexity of reasoning. In particular, we are investigating the possibility to adopt relational database techniques for efficiently reasoning with ontology instances.

Since the focus of our middleware is supporting adaptation in mobile environments, we are also developing various prototype services for mobile users. One of them, called POIsmart [4], provides location- and environment-aware resource discovery facilities.

References

[1] A. Acharya, M. Ranganathan, and J. H. Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. In *Proceedings of Mobile Object Systems - Towards the Programmable Internet, Second International Workshop, MOS'96*, volume 1222 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 1997.

[2] A. Agostini, C. Bettini, N. Cesa-Bianchi, D. Maggiorini, D. Riboni, M. Ruberl, C. Sala, and D. Vitali. Towards Highly Adaptive Services for Mobile Computing. In *Proceedings of IFIP TC8 Working Conference on Mobile Information Systems (MOBIS)*, pages 121–134. Springer, 2004.

[3] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Context-aware Middleware for Resource Management in the Wireless Internet. *IEEE Transactions on Software Engineering, Special Issue on Wireless Internet*, 29(12):1086–1099, IEEE Computer Society, 2003.

[4] C. Bettini, N. Cesa-Bianchi, and D. Riboni. A Distributed Architecture for Management and Retrieval of Extended Points of Interest. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems -*

Workshops (ICDCS 2005 Workshops), pages 266–272. IEEE Computer Society, 2005.

[5] C. Bettini, D. Maggiorini, and D. Riboni. Distributed Context Monitoring for Continuous Mobile Services. In *Proceedings of IFIP TC8 Working Conference on Mobile Information Systems (MOBIS)*, pages 123–137. Springer, 2005.

[6] C. Bettini, L. Pareschi, and D. Riboni. Cycle Resolution and Policy Evaluation for Adaptive Internet Services. Internal Technical Report, DaKWE Laboratory, University of Milan, 2006.

[7] C. Bettini and D. Riboni. Profile Aggregation and Policy Evaluation for Adaptive Internet Services. In *Proceedings of The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous)*, pages 290–298. IEEE Computer Society, 2004.

[8] M. Butler, F. Giannetti, R. Gimson, and T. Wiley. Device Independence and the Web. *IEEE Internet Computing*, 6(5):81–86, IEEE Computer Society, 2002.

[9] C. Cappiello, M. Comuzzi, E. Mussi, and B. Pernici. Context Management for Adaptive Information Systems. *Electronic Notes in Theoretical Computer Science*, 146(1):69–84, 2006.

[10] CARE middleware architecture Web site. `http://webmind.dico.unimi.it/care/`.

[11] H. Chen, T. Finin, and A. Joshi. Semantic Web in the Context Broker Architecture. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004)*, pages 277–286. IEEE Computer Society, 2004.

[12] R. Cheng, B. Kao, S. Prabhakar, A. Kwan, and Y.-C. Tu. Adaptive Stream Filters for Entity-based Queries with Non-Value Tolerance. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, pages 37–48. ACM Publishing, 2005.

[13] C. Efstratiou, K. Cheverst, N. Davies, and A. Friday. An Architecture for the Effective Support of Adaptive Context-Aware Applications. In *Proceedings of Mobile Data Management, Second International Conference (MDM 2001)*, volume 1987 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2001.

[14] R. Hull, B. Kumar, D. Lieuwen, P. Patel-Schneider, A. Sahuguet, S. Varadarajan, and A. Vyas. Enabling Context-Aware and Privacy-Conscious User Data Sharing. In *Proceedings of the 2004 IEEE International Conference on Mobile Data Management*, pages 187–198. IEEE Computer Society, 2004.

[15] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M. H. Butler, and L. Tran. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0. W3C Recommendation, W3C, January 2004. http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115/.

[16] D. Maggiorini and D. Riboni. Continuous Media Adaptation for Mobile Computing Using Coarse-Grained Asynchronous Notifications. In *2005 International Symposium on Applications and the Internet (SAINT 2005), Proceedings of the Workshops*, pages 162–165. IEEE Computer Society, 2005.

[17] D. Preuveneers and Y. Berbers. Adaptive Context Management Using a Component-Based Approach. In *Proceedings of DAIS 2005, Distributed Applications and Interoperable Systems, 5th IFIP WG 6.1 International Conference*, volume 3543 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2005.

[18] A. Rakotonirainy, J. Indulska, S. W. Loke, and A. B. Zaslavsky. Middleware for Reactive Components: An Integrated Use of Context, Roles, and Event Based Coordination. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218 of *Lecture Notes in Computer Science*, pages 77–98. Springer, 2001.

[19] RealNetworks. `http://www.realnetworks.com/`.

[20] L. Rizzo. Dummynet: a Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, 27(1):31–41, ACM Publishing, 1997.

[21] TomTom PLUS services. `http://www.tomtom.com/plus/`.

[22] VLC media player. `http://www.videolan.org/vlc/`.

[23] XviD Codec. `http://www.xvid.org/`.