User Profile Management in Converged Networks (Episode II): "Share your data, Keep your secrets".

Arnaud Sahuguet Bogdan Alexe Irini Fundulaki Pierre-Yves Laligand
Abdullatif Shikfa Antoine Arnail

Bell Laboratories/Lucent Technologies
600 Mountain Avenue, Murray Hill, NJ 07974, USA
{sahuguet, fundulaki}@research.bell-labs.com
{bogdan.alexe, pierre-yves.laligand, abdullatif.shikfa, antoine.arnail}@polytechnique.org*

1 Introduction

Two years ago [27], we described the problem of user profile management for converged networks. These networks are moving towards an all IP backbone (wireline/wireless voice, wireline/wireless data, TV, etc.) where new services can be built which interact with users wherever they are. Back then, the main concern of the industry was focused around identity management with initiatives such as Microsoft Passport[24] and Liberty Alliance[17]. The challenge was to offer a simple way for users to authenticate on all these networks.

Certainly not as a surprise to our community, the focus of the industry has now turned to data management issues. If being able to authenticate users is crucial, it is even more important to manage user information (e.g. access to bookmarks, phonebooks, calendar, etc.) in order to deliver rich and personalized services.

XML has naturally been chosen as the data model to describe and exchange user profile information. Web services have naturally been picked as the architecture used by these services to access and manage this data. So far, nothing really new.

But an interesting twist has also emerged. Users are certainly willing to share some of their personal information as long as they remain in control of what information will be accessed and for what purpose. Users want to able to share selectively their personal information, or, as mentioned in the title, *to share their data and keep their secrets.*

The problem is not a pure data integration problem but a more complicated one where integration and access control need to be deployed together, as a privacy-conscious integration solution.

In this paper, we describe the current status of the GUP^{ster} project initiated at Bell Labs two years ago, show how the vision we sketched then [27] has fleshed out and present some unanticipated and remaining challenges.

2 Summary of the previous episode

2.1 Problem statement

Numerous industry initiatives like Microsoft Passport [24] or Liberty Alliance [17] have been started to address the issue of user profile data management. The work we present in this paper is motivated by the 3GPP Generic User Profile (GUP) effort [1], a telecom-based initiative to aggregate user profile information relevant to network operators.

In this context, we can summarize the problem of privacy-conscious integration of user profile information as follows:

- a user profile consists of profile components

Work done during a summer internship at Bell Labs research, as part of their last year of study at Ecole Polytechnique.

Proceedings of the 2005 CIDR Conference

that are distributed across networks, on various data sources (relational, LDAP, XML, etc.). The user profile can be logically seen as an XML document made out of these components.

- user profile components are described in terms of an agreed upon XML schema (defined by standard bodies such as 3GPP, independently of the schemas of the sources).
- each data source exports XML data compatible with the agreed upon schema.
- distribution varies on a per user basis (e.g. a user's calendar lives in Yahoo! and a different user's calendar lives in MS Exchange inside her corporate intranet).
- data consists of static (e.g. identity information) and dynamic data (e.g. IM and wireless presence) and cannot be "warehoused" as a whole[1].
- queries request sub-documents of the distributed used profile document and do not require joins, or restructuring.
- access to data must comply with the access control rules defined by the owner of the data (this is the *privacy shield*).

2.2 Current approaches

In current architectures, when an end-user needs to access her personal information, the application acting on her behalf must contact independently a possibly large number of data sources and aggregate the information returned: a hard and also quite expensive task. Moreover, in order for the end-user to share her data with others in a privacy conscious way, she is forced to define *all-or-nothing* access control policies at the level of each data source. This architecture is illustrated in Fig. 1.

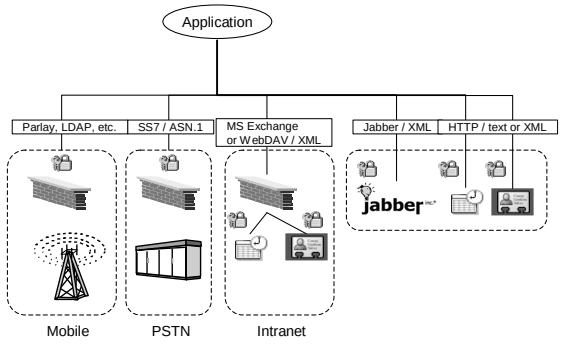

Figure 1: Before GUP[ster].

[1]There are also some ownership problems with some network operators refusing to share some information about users – like location – with other entities.

2.3 Our approach

In this context, we envision an architecture that will: (1) provide a *single point of access* to user profile information, hiding the syntactic and protocol heterogeneities of the various sources and (2) be responsible for enforcing *fine-grained access control* on the user profile data, as illustrated in Fig. 2.

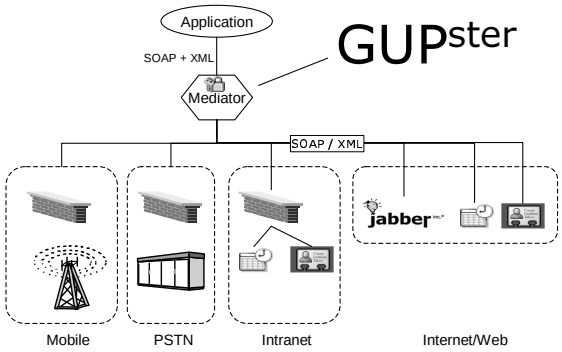

Figure 2: With GUP[ster]

The solution we advocate is embodied in the GUP[ster] system. It uses an XML mediator-based architecture where: (a) the mediator's backbone is an XML schema agreed upon by standard bodies [2]; (b) the user profile document is *virtual* with components (specified by *data mappings*) exported from distributed data sources; (c) access control rules are defined in terms of the XML schema, and specify *who* can *access what* part of the user profile document[3]; and (d) user queries are simple projection queries without joins or restructuring.

Ideally, we wanted (see [27]) a solution that would combine both access control and distribution within a single framework. In addition, our solution would (i) fetch only the necessary data from the sources to answer a user query and (ii) preserve the access control semantics, by sending to sources queries that ask only for *authorized data*. Such queries would be the result of (i) *rewriting the incoming user query with the access control rules* and (ii) *rewriting this new query with the mappings*, obtaining a new set of queries to be sent to the sources for evaluation.

The rest of this paper is organized as follows. We first (Section §3) present the XSQuirrel language, a new XML query language to describe and query sub-documents. XSQuirrel is at the core of our

[2]e.g. 3GPP GUP [1] in the telecommunications domain and Liberty Alliance [17].

[3]By default no access is granted and the user must define positive rules to grant access to parts of the profile.

GUP^{ster} framework. We then (Section §4) show how the language can be used to provide at the same time both integration and access control. We describe the architecture of the GUP^{ster} prototype in Section §5. We provide some details about the system's security aspects in Section §6. Section §7 describes some remaining challenges. Some related work is presented in Section §8, before we offer our conclusions.

3 One language to rule them all

As mentioned in the previous section, the main challenge of GUP^{ster} is to address both data integration and access control in an efficient way.

For GUP^{ster}, there are two key observations to make:

First, in GUP^{ster} we need to express the following three concepts:

- a **user query** (Q), defining the *requested portion of the user profile*
- a **data mapping rule**(M_i), that specifies the *portion of the user profile* that resides in data source i
- an **access control rule** (ACR_j), defining the condition under which a *portion of the user profile* can be accessed

Second, a *portion of the user profile* is nothing more than a sub-document of the original document that represents the user profile of a given user.

Based on these two observations, it becomes clear that for both data integration and access control, we need a language that can be used to describe a sub-document of the user profile document.

Given a user profile D, the portion of the user profile described by data source i is defined as a sub-document $M_i(D)$, where M_i defines the mapping for source i. D is defined as a disjoint union of the $M_i(D)$. The authorized part of the document is specified by $ACR(\cup_i M_i(D))$ where ACR is the access control rule. The result of a user query Q is obtained by evaluating Q on $ACR(\cup_i M_i(D))$.

In a highly distributed setting, the naive query processing (materialize the user profile, apply the access control and then evaluate the query) can be very inefficient when a) queries request an unauthorized portion of the profile document or b) the queries request a small portion of the user profile document.

We could do much better than the naive method should we compute the authorized query (by composing the user query with the access control rule) and then filter this authorized query with the data mapping rules. By doing this, we would send to the sources a query for the data that is needed for the query and that is visible by the requestor.

To achieve this form of rewriting, we need a language that is *compositional*. In this paper, we present such as language, XSQuirrel, that builds on XPath 1.0 syntax, returns sub-documents of the original document and is closed under composition.

3.1 The notion of sub-document

An XML *document* is a tree, defined by $D = (N, V, \lambda, <_d)$ where: (i) N is the set of nodes in the document with n_0 a designated node which is the document's root; (ii) $V \subseteq N \times N$ is the parent/child relationship between nodes; (iii) λ is a function that associates each node with a label; and (iv) $<_d$ is an ordering relation on the nodes of the document.

A *sub-document* $D' = (N', V', \lambda', <'_d)$ of an XML document $D = (N, V, \lambda, <_d)$ is defined as follows: (i) D and D' have the same root; (ii) $N' \subseteq N$; (iii) $V' \subseteq V$; (iv) $\lambda' = \lambda$ and (v) $<'_d = <_d$.

3.2 Syntax

XSQuirrel expressions are based on XPath 1.0 syntax (enhanced with nested union supported by XPath 2.0). They are built from a finite set of labels (e.g., tags, names) Σ of an XML schema S. The fragment of the language that we consider in this paper is syntactically defined as follows:

$$\begin{aligned} top &:= p \cup p \\ p &:= l \mid p/p \mid p/(p \cup p) \mid p[q] \end{aligned}$$

where l is a name in Σ, $\cup$ stands for *union*; '/' stands for XPath concatenation but here is also used as the XPath *child* axis. q in $p[q]$ is called a *qualifier* and is defined by: $q := p \mid label = v$.

For an in-depth description of the language we we direct the reader to [6].

EXAMPLE We give below some examples of XSQuirrel expressions.

$$\begin{aligned} q_1 &: /A/B/(D \cup H) \\ q_2 &: /A/B[H]/(D/DD \cup F) \\ q_3 &: /A/(B[C] \cup B[H]/(D/II \cup F/FF)) \\ q_4 &: /A/B[D/EE]/(D/DD \cup H \cup F) \end{aligned}$$

Query q_1 for instance returns `D` and `H` nodes, that are children of `B` nodes, themselves children of `A`

nodes. Along with these nodes, their descendants, and ancestors up to the root node of the document on which q_1 is evaluated are returned.

3.3 Semantics

Intuitively, the result of the evaluation of an XSQuirrel expression q on a document D is document $q(D)$ (sub-document of D) obtained as follows:
1. evaluate q using the usual XPath 2.0[4] semantics;
2. for each node n obtained from the previous step, get its *descendant* nodes, and its *ancestor* nodes up to the root of D;
3. finally, $q(D)$ is constructed by removing all nodes of D that are not in the set of nodes obtained from the previous step (note that the resulting document $q(D)$ is a sub-document of D).

The choice of taking the descendants is motivated by access control needs. A natural choice is that when a node is accessible, then all its descendants are too. Returning the ancestors of a node is helpful in the context of data integration, where the identifier of a node (persistent object identifier or semantic key) can be attached to it, allowing one to merge the data obtained by different sources.

EXAMPLE Consider the XML document D illustrated in Fig. 3 (ignore the grey marking for now) and query q_1 given in Example. 3.2. The result of evaluating q_1 over D is sub-document $q_1(D)$ where the nodes have been marked in grey. More specifically, this document is defined by the `D` and `H` nodes returned when evaluating q_1 as an XPath expression on D, their descendants and ancestors up to the root node of D. One can observe that although there are no `H` nodes for the first (in document order) node `B`, the latter's node `D` is returned (with its ancestors and descendants).

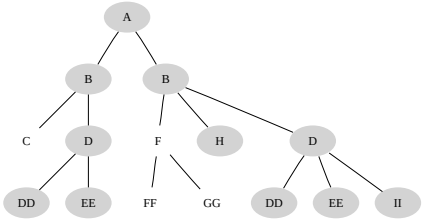

Figure 3: Original document D and $q_1(D)$ (grey marking).

More formally: An XSQuirrel expression is always evaluated against the root node of a document D. The result of evaluation XSQuirrel expression q against a document $D = (N, V, \lambda, <_d)$ is a sub-document $D' = (N', V', \lambda', <'_d)$ of D such that: (i) N' is defined as:

$$N' = [[D]]_q \bigcup_{n \in [[D]]_q} n[[D]]_{\Downarrow_* ::*} \bigcup_{n \in [[D]]_q} n[[D]]_{\Uparrow^* ::*}$$

where $n[[D]]_p$ denotes the set of nodes returned by evaluating XPath expression p on the node n of document D (n is omitted when it is the root), $\Downarrow_*$ and $\Uparrow^*$ are the XPath *descendant* and *ancestor* axis resp. and (ii) $V' = \{(n_1, n_2) \in V \mid n_1, n_2 \in N'\}$.

3.4 The true value of a new language

Why a new language People may question the need to define another query language for XML when there are already so many.

XPath [10] is not comparable to XSQuirrel since XPath expressions (i) *cannot be composed* (once an XPath expression is evaluated on a document D, the context is lost) and (ii) return sets of nodes instead of documents.

Another solution would be to use XQuery [8] for which a trivial composition algorithm [14] exists. XQuery is an extremely rich and powerful language, and the issue of how to optimize and efficiently compose XQuery expressions may remain open forever. Another issue is the fact that XQuery has no notion of sub-document and enforcing the sub-document semantics is not a totally trivial task, as explained below.

Sub-document queries, the easy way: The first advantage of the XSQuirrel language is that it permits to define in a concise way sub-documents.

As an illustration, we present one possible translation of a simple XSQuirrel expression into XQuery.

`/(a ∪ b[p1] ∪ c/d)` as XQuery

```
for $x1 in /*
return
  if $x1[self::a] then { $x1 }
  else if $x1[self::b[p1]][not(self::a)]
  then { $x1 }
  else if $x1[self::c[d]][not(self::a)][not(self::b[p1])]
  then {
  for $x4 in $x1/*
  return
  <c> if $x4[self::d] then { $x4 } else () </c>
      }
  else ()
```

The reason for such complexity in the translation is that the output must respect the structure

[4] XPath 1.0 considers only top level union while XSQuirrel (as XPath 2.0) considers union at any level of the expression.

of the original document. To enforce order, we need to iterate over the children (using /*) and then proceed, on a case by case basis. Moreover, to avoid children to appear more than once (e.g. when there is a union of overlapping elements), we need to add some predicates to make sure that each if statement corresponds to a unique case.

Sub-document queries the efficient way The second advantage of our new language it that it permits to handle sub-document queries efficiently. In GUP^{ster}, from a logical point of view, we need to handle documents as follows: from a document D, we need to keep the part that is accessible as defined by some access control rules; then we need to apply the query that will produce the answer. This corresponds to $Q\left((\bigcup_{xq}^{j=1..m} ACR_j)(D)\right)$. But as mentioned earlier, this implies that we materialize the accessible document before we apply the query. Alternatively, we would like to be able to perform the access control statically and write $\left(Q \circ (\bigcup_{xq}^{j=1..m} ACR_j)\right)(D)$. The difference between both ways is illustrated in Fig. 5 and Fig. 4.

This means that at the level of the language we need to define the composition ($\circ$) operator for which $\forall\, D,\ (Q_1 \circ Q_2)(D) = Q_1\,(Q_2(D))$.

For lack of space, we present the intuition for the composition using an example. The details of the composition algorithms for different fragments of the language are available in [6].

We distinguish between the inner (Q_i) and the outer expression (Q_o). The idea behind the composition algorithm is to find embeddings of the outer query Q_o into the inner query Q_i.

EXAMPLE Here is an example to illustrate the intuition.

Q_o	$/A/(B[C] \cup B[H]/(D/II \cup F/FF))$
Q_i	$/A/B[D/EE]/(D/DD \cup H \cup F)$
$Q_o \circ Q_i$	$/A/B[H][D/EE]/F/FF$

We see that node B[C] of Q_o does not appear in the composed query (the path $/A/B/C$ for node B[C] is not satisfied by the inner query). Node B[H][D/EE] is created from nodes B[H] and B[D/EE] of the outer and inner queries respectively. Node D (and its children) disappears from the resulting query since the outer query (Q_o) requests II nodes but the inner query Q_i returns only DD nodes. Finally, node FF requested by the outer query is added below node F (the inner query returns the subtree of F but the outer query requests only its FF subnodes).

4 XSQuirrel in action

In this section, we will present an end-to-end example that demonstrates how we can achieve access control and integration using XSQuirrel. Some design decisions about the language will hopefully appear clearer.

Let us consider a scenario where user Irini (username=fundulaki) tries to access some of Arnaud's profile (username=sahuguet) information related to his contacts. The query she issues is Q = /Gup/Contacts. Arnaud has defined some access control policies and some data mappings that are stored in the GUP^{ster} metadata repository.

For this incoming query, GUP^{ster} retrieves from the metadata repository the access control rules that apply to the request concerning resource /Gup/Contacts for user sahuguet and requestor fundulaki.

Let us assume that the following access control rules (R1, R2 and R3) are retrieved from the metadata repository. They respectively grant access to: the contact entries that are of type public along with the voicemail; the user identity; and the Jabber presence information, but only during working hours (9am to 6pm).

Relevant access control rules

```
user=sahuguet; requestor=fundulaki
R1: /Gup/(Contacts/Entry[@type="public"] # VoiceMail)
    condition: true
R2: /Gup/Self/Identity
    condition: true
R3: /Gup/Presence/JabberPresence
    condition: 9am < time-of-day < 6pm
```

For each rule with a condition that evaluates to true, GUP^{ster} then computes the union of the three expressions:

R1 ∪ R2 ∪ R3

```
/Gup/(Contacts/Entry[@type="public"]
     # Presence/JabberPresence
     # Self/Identity
     # VoiceMail )
```

GUP^{ster} then computes the accessible view Q_{acc} as $Q \circ$ (R1∪R2∪R3): /Gup/Contacts/Entry[@type="public"].

This is the end of the access control process with the accessible view. We now move to the data integration process. GUP^{ster} retrieves from the metadata repository the relevant mappings for Arnaud's profile. Let us assume that we have the following mappings M1 and M6 for Arnaud's profile, involving sources s1 and s6.

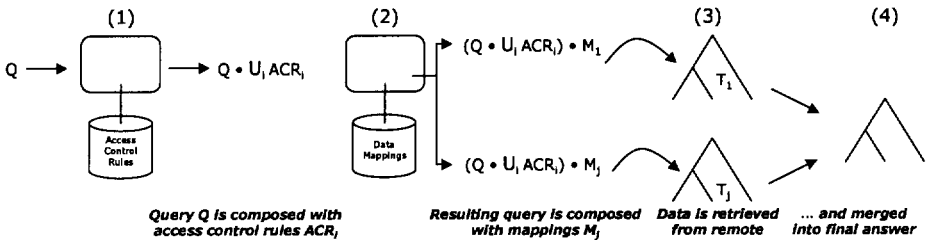

Figure 4: Query processing with static access control

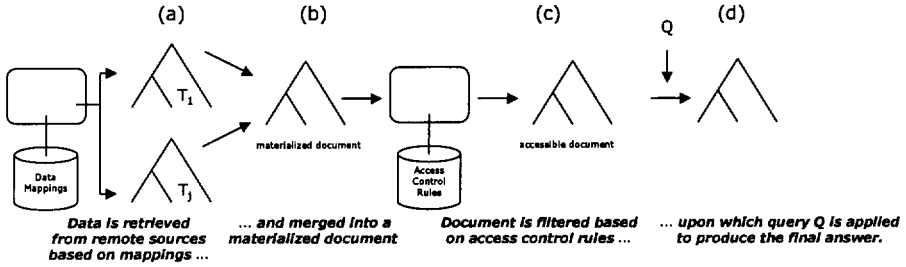

Figure 5: Query processing with non-static access control

```
M6   /Gup/Contacts/Entry[@type="private"]
M1   /Gup/(Self # Contacts/Entry[@type="public"])
```

For each data source, GUPster computes the query to be sent to the source by composing the mapping with the accessible view, i.e. $Q_{\text{acc}} \circ$ M1 and $Q_{\text{acc}} \circ$ M6, defined as follows:

```
s6   /Gup/Contacts/Entry[@type="public"][@type="private"]
s1   /Gup/Contacts/Entry[@type="public"][@type="public"]
```

The sub-documents retrieved from the various sources are merged to produce the final result. There are potentially many ways to merge documents. We currently merge documents based on schema information. When two nodes coming from two different sub-documents have the same path, they are added as siblings in the final result if they correspond to a list (or set) in the schema. Otherwise, one of them is kept and the other one ignored.

5 Our implementation

GUPster is a fully working prototype that has been demonstrated at [15, 2]. It is written in Java, with reuse of a number of open source components. All the interactions between components follow the web services paradigm.

5.1 GUPster architecture

There are basically three kinds of components in the GUPster system:

- SOAP clients
- the GUPster server that manages metadata (acces control rules and data mappings) and handles requests
- the GUPster wrappers that interface with legacy data sources to export via SOAP, XML data compliant with the GUPster schema.

The relationships between the various components is illustrated are Fig. 6.

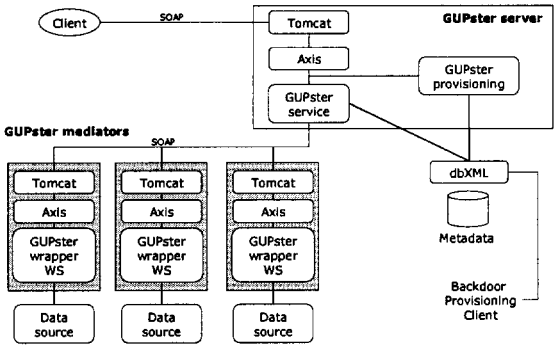

Figure 6: The GUPster architecture

The core the system consists of the XSQuirrel API that performs the query rewriting described previously. For the web services aspect, we are using Apache Axis, on top of Apache Tomcat. The GUPster web services is no more than a re-packaging of the GUPster core.

For metadata management, we are using the dbXML[5] native XML store. Support for XPath and XUpdate[31] along with a friendly provisioning interface makes it a perfect candidate. Moreover, access to dbXML is done via the XML:DB API[30] which means that the GUPster server could also use any XML:DB compliant native XML store.

For the SOAP clients, we are using a variety of SOAP stacks, including Axis itself, kSOAP[6], Mozilla SOAP, etc.

5.2 GUPster metadata

As mentioned earlier, in GUPster we handle two kinds of metadata information: data mappings and access control. The former associate portions of the user profile document (expressed as XSQuirrel expressions with data sources). The latter specify the authorized portions of the user profile document (again in terms of XSQuirrel queries). Both data mappings and access control rules are defined on a per-user basis. All metadata information is stored in our native XML database, on a per user basis. A provisioning interface makes it possible to modify this metadata information (see Fig. 6).

When a query comes in, the GUPster server sends a request to the metadata store to retrieve access control rules and mappings corresponding to a given subscriber. We present some example of metadata stored in the database in Figures 7 and 8.

Sources are simply described by the web service that needs to be invoked (end point and SOAP action). Note that sources do not necessarily support XSQuirrel. In this case, the query will be translated to whatever the source supports (e.g. XQuery, XSLT). For data mappings, some extra information can be added (e.g. username and password to access the remote data store).

For each user, an access control policy (set of access control rules) is defined. A rule can have a list of requestors (users or groups of users), a resource (an XSQuirrel expression), the type of action (e.g. read) and an optional condition. In our current implementation, we support a very limited condition language with boolean operators and variables such as *time of the day, day of the week.* Note that, because of GUPster processing flow (see below), replacing our condition language by something more elaborate such as XACML[23] or Vortex[16] would be totally transparent for GUPster.

[5] http://www.dbxml.com/

[6] http://ksoap.objectweb.org/

```
<policies>
  <policy name="main-policy">
    <rule ruleID="2">
      <requestors>
        <requestor groupRef="coworkers"/>
      </requestors>
      <resource>
        /Gup/Contacts/Entry[@type='public']
      </resource>
      <action type="read"/>
    </rule>
    <rule ruleID="5">
      <requestors>
       <requestor groupRef="friends"/>
       <requestor groupRef="family"/>
      </requestors>
      <resource>
        /Gup/Presence/JabberPresence
      </resource>
      <action type="read"/>
      <condition>
      9pm  &lt; time-of-day &lt; 5pm
      </condition>
    </rule>
...
  </policy>
...
</policies>
```

Figure 8: Access control policies

5.3 GUPster processing flow

For each incoming query from a client (i.e. a 3-uple consisting of a requestor identity, a resource defined as an XSQuirrel expression and the identity of the owner of data), the GUPster server performs the following actions:

1. it authenticates the requestor based on the requestor identity (see Section §6). If the authentication fails, an empty answer is sent back.
2. it retrieves the access control rules that the owner has defined for the requestor
3. it computes the accessible query by taking the composition of the initial query with the union of access control rules
4. it retrieves the data mappings for the owner
5. for each source, it computes the query to be sent to the source, by taking the composition of the accessible query and the mapping rule
6. for each source, it sends the SOAP request to the remote source and gets back a sub-document of the user profile
7. it merges the various sub-documents to produce the final answer
8. it sends back the final answer to the client

Note that in the case where the requestor has asked an unauthorized query, , GUPster sends directly an empty answer without having to retrieve

```
<sources>
  <source id="3" name="LocalSource_3">
   <sourceCapability>
    <service url="http://..."
             type="xsquirrel"/>
   </source_capability>
  </source>
...
</sources>
```

```
<dataMappings>
  <dataMapping id="2" sourceId="2">
    <params><param name="id" value="fundulaki"/></params>
    <resource>/Gup/(Interests # Locale # Contacts/Entry)</resource>
 </data_mapping>
 <dataMapping id="3" sourceId="3">
   <params><param name="id" value="fundulaki"/></params>
   <resource>/Gup/Money/BankAccounts/Bank[@name="CitiBank"]</resource>
...
</dataMappings>
```

Figure 7: Source description and data mappings

any data from the remote sources.

5.4 Data sources we federate

We now list the various data sources we have *wrapped* and that offer data access through our GUP[ster] interface.

- Jabber[7] presence; access via a home-made Jabber module that overrides the *only buddies can see each other's presence* restriction; Jabber XML converted into a GUP[ster] compliant XML.
- voicemail; access to Lucent Audix voicemail via JavaMail/POP interface; mail headers converted into GUP[ster] compliant XML.
- Lucent directory information; access Lucent corporate directory; generic translation from LDAP to XML.
- user location; access to Lucent miLife ISG SDK[8] simulating cell phone users moving around, via Parlay interface; ad-hoc translation to GUP[ster] compliant XML.
- various synthetic XML data; stored in dbXML.
- Microsoft MS Exchange data; access via webDAV; ad-hoc translation to GUP[ster] compliant XML.
- Palm PIM data; access through JSyncManager[9] plug-in; ad-hoc translation from PIM data (e.g. vCard, vCal) to GUP[ster] compliant XML;
- Sony Ericsson T610 phone data; access via Bluetooth; ad-hoc translation from phone data to GUP[ster] compliant XML.

5.5 GUP[ster] clients

Just like for data sources, we have implemented various different applications living on various platforms.

- GUP[ster] browser client using Sarissa[10]; SOAP messages are hand-crafted.
- rich internet application built using Laszlo[11] (Flash client).
- J2ME client application; SOAP support using kSOAP[12] and homemade serializer.

The first thing we learned from developing clients running on devices with limited capabilities is that XML support in J2ME usually requires some indecent amount of memory. For instance displaying the entire profile of a given subscriber raises an `OutOfMemoryException`.

The second thing is that current J2ME implementation do not offer access to the internal "PIM" datastores of the device (e.g. address book, calendar, todo list for a PDA). This means that it is not possible to export PIM data to the outside and that the only thing we can do with whatever XML data we retrieve from the network is to display it on the screen (which is not particularly useful).

6 Security issues

One aspect that we have not discussed yet is the issue of security. In the GUP[ster] context, we want to make sure that (1) data exchanged between the various parties cannot be intercepted by malicious attackers and (2) requestors are properly authenticated, otherwise everyone could pretend to me in order to access my full profile.

6.1 SSL certificates

Over the last two years, various solutions have been proposed to address the issues of network identity and single sign-on. For GUP[ster], we have taken a rather conservative approach by using SSL certificates[26]. Every time you access a web site using a secure (i.e. https) connection, your web client authenticates the server by checking that its

[7] http://www.jabber.org
[8] http://www.lucent.com/developer/milife/
[9] http://www.jsyncmanager.org/
[10] http://sarissa.sourceforge.net/
[11] http://www.laszlosystems.com/
[12] http://ksoap.objectweb.org/

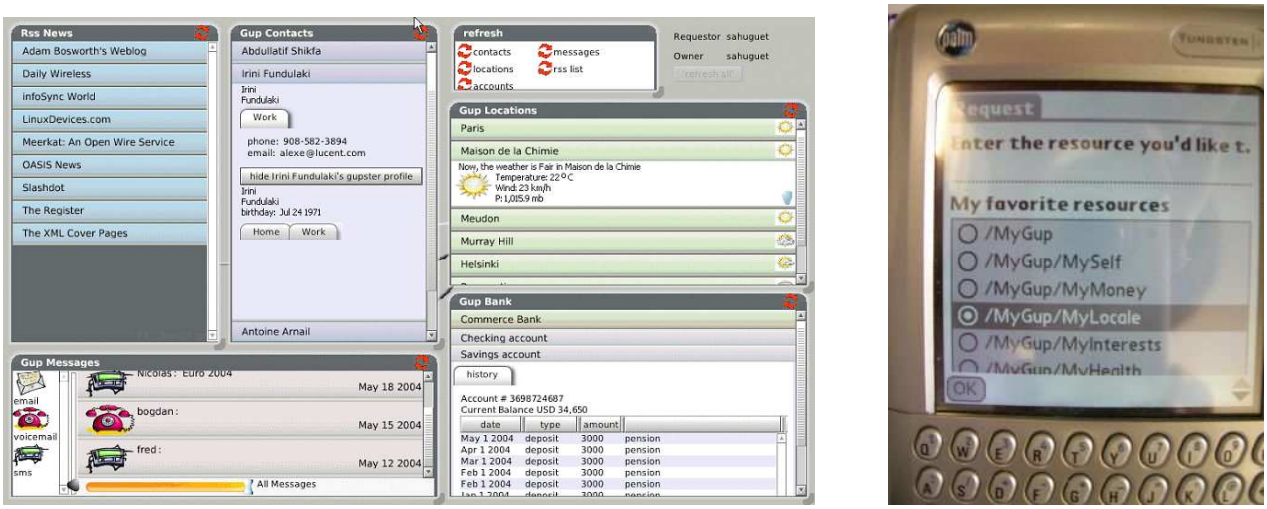

Figure 9: Some GUPster clients: Flash-based (left); J2ME-based (right) running on Tungsten C.

certificate has been signed by a trusted *certification authority* (CA). This is *server-side authentication.* Conversely, the server can authenticate the client by first asking the client to send a certificate as well and by checking that the certificate has been signed by a trusted CA. This is *client-side* authentication.

Client-side certificates are not as widely used as their server-side counterparts. The main reason is that the management (issuance, distribution, revocation) of millions of certificates is complicated. Moreover certificates usually need to be purchased.

Opponents often put forward the following concerns about certificates:

- they are complicated to issue
- they are complicated to use
- they cannot be remembered and are hard to manage (especially on devices)
- they require expensive computing power (public key cryptography is CPU expensive)

In the rest of this section we will show how we have addressed these issues, especially how clients authenticate to the GUPster server.

6.2 GUPster with SSL

To secure GUPster using SSL certificates, we assume that the various entities involved (clients, server and data sources) have received some valid certificates from a pool of trusted CAs for which the public keys are known to each entity (e.g. built-in in the client). End users are identified by the same unique name present inside the certificate.

When the server receives a request from a client, the authentication proceeds as follows:

1. the client opens a secure connection to the server (Tomcat)
2. the server retrieves from the connection the client certificate
3. the server checks that the certificate is valid (signed by a trusted CA, not expired, not revoked); if the certificate is not valid, the request is rejected.
4. the server forwards the request to the web service handler (Axis in our case)
5. Axis adds to the request context some information about the certificate before it dispatches the query to GUPster web service.
6. the web service extracts the identity from the certificate and compares it with the identity mentioned in the query; if they match, the query is processed

To make this work, we have created a special `SOAPCertificateHandler` for Axis that enriches the SOAP `MessageContext` with some information from the certificate and registered it in the processing flow of our GUPster web service. See [28] for more details.

6.3 SSL and devices

When moving to devices (i.e. J2ME as opposed to J2SE), numerous problems occur.

The first one is that J2ME does not offer an implementation of the SSL protocol with support for both client and server certificates. Fortunately, there exists a Java package (iSaSiLk-ME[13]) that of-

[13] `google:iSaSiLk`

fers a pure Java implementation of the SSL protocol (client and server). Because it is written in Java, it is kind of slow. Hopefully future version of J2ME will offer a native support (i.e. SSL support in the JVM itself).

The second one is that J2ME does not permit to manage certificates. To overcome this restriction, we propose two solutions. Our GUPster client for J2ME is deployed as part of a MIDlet suite (a bundle of multiple J2ME MIDlets). The user can install her MIDlet suite with her certificate already stored in the suite. Or she can install one MIDlet that securely connects to a server where the certificate is securely stored. In both cases, after this initial step, the user has on her device her certificate ready to be used.

When tried on real devices, the overhead of running SSL is substantial (a few seconds), but in line with the latency expected on wireless networks.

7 Remaining Challenges

We present in this section some challenges we have faced so far (some of which were already mentioned in [27]).

7.1 GUPster and open standards

One key aspect of the effort presented in this paper is that it is motivated by an urgent need from the telecom industry. As a result, any solution proposed must be agreed upon by all the parties involved. This usually translates into reliance on open standards. This sometimes dictates technical decisions.

Architecture For GUPster, we have had the chance to start from a situation where there was no de-facto standard. Today on the IT side, Liberty Alliance [17] has emerged as the standard for network authentication and user profile management in the context of web services. On the telecom side, 3GPP GUP [1] is trying to achieve the same by reusing as much from Liberty Alliance as possible (same interfaces, same signatures, etc.).

Serendipitously, our GUPster architecture maps naturally to the functional architecture of Liberty Alliance. The GUPster server naturally plays the role of a *Discovery Service* [21] and *Data Service* [20]. The GUPster wrappers are *Data Service* themselves. Alignment with the specification will simply consist in reusing the exact same web service interfaces.

XSQuirrel At the core of GUPster is this new

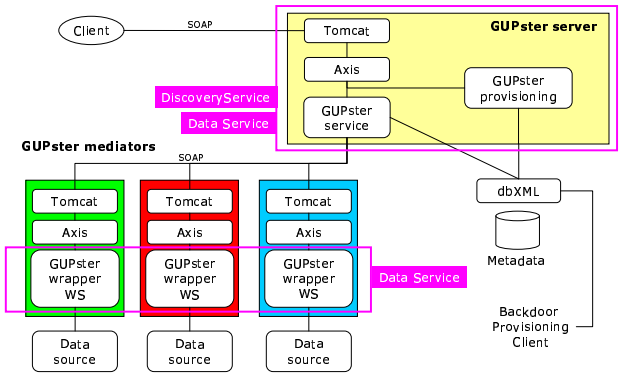

Figure 10: GUPster and Liberty Alliance

XSQuirrel language. Even though we have a strong motivation for its existence (see Section §3.4), people often don't like it because it is not (yet) standardized. Our answer is two-fold. From a syntactic point of view (see Section §3.2), XSQuirrel is a subset of XPath 1.0 extended with union at arbitrary levels. From a semantic point of view see Section §3.3), we have presented a clear semantics based on XPath 1.0.

People often complain that with our new language, already existing tools cannot be used and new ones need to be developed. To answer this criticism, we have written translators from XSQuirrel to XQuery and XSL-T. For lack of space, we cannot present the full details of the translation. The challenge in the translation is to preserve the sub-document semantics. Using these translators, XSQuirrel can be evaluated using one's favorite XML tools.

Schema Clearly, an agreement needs to be reached about the exact schema that will be used to represent user profiles. But whatever the schema looks like, it will have no major influence on our work. Some schema constructs might be friendlier to the static aspect of access control though. For instance, structural information such as private vs public contact entry can be encoded in the schema itself (e.g. `<publicEntry>` vs `<privateEntry>` or in the data (e.g. `@type="public"`).

Another area where standardization will play an important role is the language used to describe conditions in our access control rules. In our current prototype, this language is very limited. But users may want to express more elaborate conditions using languages such as P3P[14] or XACML [23].

[14] `http://www.w3.org/P3P/`

7.2 Support of updates

Another big challenge is the support for updates. Updates have been a major headache for industry and academia. As of this writing there is no standard for XML updates.

In the context of GUP^{ster}, updates are even trickier when we bring access control to the picture, mostly because there are so many ways to understand them. Allowing me to update a node in a document, what does it mean exactly? Can I delete the node, can I delete any of its descendants, etc. Unlike queries that leave the document unchanged, updates may radically modify the content. Updates need to take into account the context of the update (which node or set of nodes), the nature of the update (delete, replace, append, etc.) and the nature of the data actually used for the update (e.g. restriction on what can be appended). There is also a thin line between access control and validity checking.

7.3 Performance

A big challenge that we have not yet tackled is performance. In our telecom context, network operators are willing to migrate to a solution à la GUP^{ster} if their already existing services are not disrupted. They want to the benefits without the overhead.

With GUP^{ster}, we already see today some major benefits. As mentioned previously, because access control is done statically, we don't need to access any data when we know that this data is not visible to the requestor.

We need to make sure that our rewriting algorithms are really fast. There is a trade-off between the flavor of XSQuirrel we propose and the complexity of the algorithm (e.g. support for negation, support for descendant axis, etc.).

We also need to make sure that the evaluator we use for XSQuirrel expressions, either at the server or at the data source, either native or via translation, is really efficient in terms of fast execution, low memory requirement, etc.

Finally, in order to measure performance, we probably need a benchmark suite (data and metadata) that is representative of the application domain we are dealing with.

7.4 Provisioning

Database research usually assumes that the data is already in the database. In real life, this is not the case. Users often painfully need to enter the data by hand (e.g. calendar entry on a PDA, number in a phone book) and then later on modify it. In our context, provisioning is absolutely critical if we hope to see users share their information with others. It takes two flavors: provisioning of data and of metadata.

For metadata, provisioning will require some ad-hoc targeted interfaces and process flow. For instance, access control requires to be able to navigate the user profile schema, select some parts of it, add conditions and predicates. We can also imagine tools to check rules for consistencies.

For data, the provisioning should be simpler, but on a much larger scale since the schema representing the user profile can be grown at will. In this case, the provisioning should be automated. For instance web interfaces (e.g. forms) could be generated automatically based on schema and schema annotations.

8 Related work

In this paper we have presented the GUP^{ster} framework for privacy conscious integration of user profile data. At the core is the XSQuirrel language, a simple XPath-based query language for XML data that is *composable* and returns documents. The idea of retrieving the subtree of a returned node for a given query appeared in the context of distribution and replication of XML documents in [3]: the fact that a subtree of a node should be returned has to be explicitly defined in the XPath expression and is not inherent in the semantics of the language as in our case.

Authors in [25] follow a similar approach to [3] where XPath expressions return documents instead of sets of nodes. However, the authors neither define formally the semantics of the language, nor discuss algorithmic issues for operators such as composition and union.

A strength of XSQuirrel is that it offers a unified framework for addressing the issue of both access control and data integration for XML data; in this sense, it is (to the best of our knowledge) unique. The work on Hippocratic databases [4] addresses also both aspects but in a relational and centralized context, where access control rules are not defined by end users, just approved from a set of policies defined by the database administrator.

We give an overview of how it compares to approaches relevant to the two topics considered indi-

vidually.

Data Integration In GUP^{ster}, we use *local-as-view* [18] mappings to specify the data exported by each source. LAV-based query rewriting is a hard problem. In our approach we advocate a situation in which distinct data sources contain disjoint data. Based on that, the user profile document is computed by performing disjoint union of the sources data. Having these simple data integration views, LAV query rewriting boils down to query composition. In general, LAV descriptions (see [19, 18] for a survey) allow greater modularity in their descriptions, but introduce new complexities – e.g. mappings specifications may be nondeterministic or inconsistent, query rewriting is not thoroughly understood, even in the relational case. In the XML context, mappings have been used to describe an XML source to XML [11] or entity relationship schemata [5]. Finally in [32] the authors discuss mappings from and to generic nested structures. In contrast, our work looks at simpler mappings, for which more efficient algorithms can be obtained.

Access Control A significant amount of work has been done on access control for XML data over the last few years. In most approaches [7, 12, 22, 13] and standards [23, 29], the access control rules are specified as XPath [10] expressions. In [23, 29] access control is enforced by an all-or-nothing procedure where the query is rejected if the result contains non-accessible nodes. On the other hand, in [7, 12] access control is enforced by a tree labeling algorithm that computes the *authorized view* of the XML document. In [9] a different approach is undertaken. Schema nodes specify the conditions under which security annotations exist in data nodes. They also give an algorithm to compute the rewritings of twig queries (XPath without union and wildcards) using the DTD schema information. Generally, the above approaches use different languages for the access control rules than for the queries or algorithms that evaluate them, unlike in XSQuirrel.

[22] uses static analysis of queries and access control rules to check whether a query is safe (i.e., requests only accessible data). Queries and access control rules are translated into string automata and their intersection is performed to decide whether the query is safe. In their approach, the result of performing access control is either yes or no (all or nothing).

The work presented in [13] is closely related to ours: access control rules are expressed in XPath but enforcement of access control is done by means of a *security view*.

9 Conclusion and Future Work

In this paper we have presented GUP^{ster}, a unified framework for data integration and access control over distributed XML data. The core of GUP^{ster} is based on the XSQuirrel language, a new XML language for sub-document queries. The fact that the language is closed under composition, allows one to perform most of the query processing statically and minimize the data that is shipped from the sources to compute the result of a query. These ideas have been implemented in the GUP^{ster} prototype that we have also described.

As for some future work, the first aspect is to convince users to provide their data and share it via GUP^{ster}. This implies performance and scalability. This also involves working with on-going standards. The second aspect is our XSQuirrel language itself that seems to be useful beyond the scope of GUP^{ster}, as a general purpose language to define XML views and reason about distributed XML processing.

Acknowledgment: We would like to thank Guillaume Giraud, Nicola Onose and Nicolas Pombourcq who contributed to the first version of the GUP^{ster} prototype that was demonstrated at the SIGMOD 2004 conference.

References

[1] The Third Generation Partnership Project (3GPP). `http://www.3gpp.org`.

[2] S. Abiteboul, B. Alexe, O. Benjelloun, B. Cautis, I. Fundulaki, T. Milo, and A. Sahuguet. An Electronic Patient Record "on Steroids": Distributed, Peer-to-Peer, Secure and Privacy-conscious. In *VLDB*, 2004. (Demo).

[3] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. In *SIGMOD*, 2003.

[4] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, 2002.

[5] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Querying XML sources using an Ontology-based Mediator. In *CoopIS*, 2002.

[6] M. Benedikt and I. Fundulaki. Specification and composition of xml subtree queries. Internal Report, Bell Labs, June 2004. Available upon request.

[7] E. Bertino and E. Ferrari. Secure and Selective Dissemination of XML Documents. *TISSEC*, 5(3):290–331, 2002.

[8] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and L. Stefanescu. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery, February 2001.

[9] S. Cho, S. Amer-Yahia, L. V. S. Lakshmanan, and D. S. tava. Optimizing the Secure Evaluation of Twig Queries. In *VLDB*, 2002.

[10] J. Clark and S. D. (eds.). XML Path Language (XPath) Version 1.0, 1999. http://www.w3c.org/TR/xpath.

[11] S. Cluet, P. Veltri, and D. Vodislav. Views in a Large Scale XML Repository. In *VLDB*, 2001.

[12] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. A Fine-Grained Access Control System for XML Documents. *TISSEC*, 5(2):169–202, 2002.

[13] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML Querying with Security Views. In *SIGMOD*, 2004.

[14] M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A framework for publishing relational data in XML . *TODS*, 27(4):438–493, 2002.

[15] I. Fundulaki and A. Sahuguet. Share your data, keep your secrets. In *SIGMOD (Demo)*, 2004.

[16] R. Hull, B. Kumar, D. Lieuwen, P. F. Patel-Schneider, A. Sahuguet, S. Varadarajan, and A. Vyas. Enabling context-aware and privacy-conscious user data sharing. In *Mobile Data Management (MDM)*, 2004.

[17] Liberty Alliance Project. http://www.projectliberty.org.

[18] M. Lenzerini. Data Integration : A Theoretical Perspective. In *PODS*, 2002.

[19] A. Levy. Answering queries using views: a survey. *VLDB Journal*, 2001.

[20] Liberty Alliance ID-WSF Data Services Template Specification, Version 1.0. http://www.projectliberty.org/specs/liberty-idwsf-dst-v1.0.pdf, 2002.

[21] Liberty Alliance ID-WSF Discovery Service Specification, Version 1.1. http://www.projectliberty.org/specs/liberty-idwsf-disco-svc-v1.1.pdf, 2002.

[22] M. Murata, A. Tozawa, and M. Kudo. XML Access Control using Static Analysis. In *CCS*, 2003.

[23] OASIS. XACML, Feb 2003. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[24] Microsoft Passport. http://www.passport.net.

[25] M. Petropoulos, A. Deutch, and Y. Papakonstantinou. Query Set Specification Language (QSSL). In *Informal Proc. WEBDB*, 2003.

[26] E. Rescorla. *SSL and TLS Designing and Building Secure Systems*. Addison Wesley, 2003.

[27] A. Sahuguet, R. Hull, D. Lieuwen, and M. Xiong. Enter Once, Share Everywhere : User Profile Management in Converged Networks. In *First Biennial Conf. on Innovative Data Systems Research*, Asilomar,California, USA, January 2003. Online Proceedings.

[28] A. Shikfa. Identity Management in Converged Networks. Technical report, Ecole Polytechnique, 2004. Rapport de Stage d'Option.

[29] XML Access Control. http://www.trl.ibm.com/projects/xml/xacl/.

[30] XMLdb API, Sept 2001. http://www.xmldb.org/xapi/index.html.

[31] XUpdate: XML Update Language, 2000. http://www.xmldb.org/xupdate/.

[32] C. Yu and L. Popa. Constraint-Based XML Query Rewriting for Data Integration. In *SIGMOD*, 2004. To appear.