Enabling Data Quality Notification in Cooperative Information Systems through a Web-service based Architecture

C. Marchetti, M. Mecella, M. Scannapieco, A. Virgillito
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113 - 00198 Roma, Italy
{marchet,mecella,monscan,virgi}@dis.uniroma1.it

1. Introduction

Cooperative Information Systems (CISs) are often characterized by a high degree of data replication; as an example, in an *e*-Government scenario, the personal data of citizens are stored by almost all administrations. In such scenarios, organizations typically provide the same information with distinct quality levels and this enables providing users with data of the highest available quality. Furthermore, the comparison of data values might be used to enforce a general improvement of data quality in all organizations. In the DaQuinCIS project[1], we propose an architecture for the management of data quality in CISs; this architecture allows the diffusion of data and related quality and exploits data replication to improve the overall quality of cooperative data.

In this paper, we present an overview of a component of the DaQuinCIS architecture, namely the Quality Notification Service (QNS), which is used to inform interested users when changes in quality values occur within the CIS. QNS can be used to control the quality of critical data, e.g. to keep track of quality changes and to be always aware when quality degrades under a certain threshold. The interaction between the QNS and its users follows the publish/subscribe paradigm: a user willing to be notified for quality changes *subscribes* to the QNS by submitting the features of the events to be notified for, through a specific subscription language. When a change in quality occurs, an event is *published* by the QNS i.e., all the users which have a consistent subscription receive a notification. However, as shown in [6], currently available pub/sub infrastructures do not allow to meet all the requirements that a QNS implementation should satisfy, in particular scaling to a large number of users and coping with platform heterogeneity.

[1] "DaQuinCIS - Methodologies and Tools for Data Quality inside Cooperative Information Systems" (`http://www.dis.uniroma1.it/~dq/`) is a joint research project carried out by DIS - Università di Roma "La Sapienza", DISCo - Università di Milano "Bicocca" and DEI - Politecnico di Milano.

QNS addresses both these problems through a layered architecture that (i) encapsulates the technological infrastructure specific of each organization, (ii) adopts the standard Web-service technology to implement inter-organization communications, and (iii) embeds solutions and algorithms (namely, *merge subscriptions* and *diffusion trees*) to reduce the use of physical and computational resources.

The remainder of this paper is organized as follows: we first introduce some preliminary concepts and the QNS specification (Section 2); then we motivate (Section 3) and describe (Section 4) the internal architecture of the service. Due to the lack of space, explanations are given at a very high abstraction level. Interested readers can find technical and formal details, as well as running examples, in [9].

2. The Quality Notification Service

Preliminary Concepts. In the DaQuinCIS architecture, organizations export data and quality data according to a data model, referred to as *Data and Data Quality (D^2Q) model*, which includes *(i)* constructs to represent the data of each organizations in a semi-structured way according to a semi-structured global schema [5] (data classes, data objects, attributes and links), *(ii)* a set of data quality dimensions (accuracy, completeness, currency, and internal consistency), *(iii)* constructs to represent their values, and *(iv)* the association between data and quality data [7]. Quality values are declared at attribute-level: each attribute of a data object has associated a set of quality values, one for each quality dimension. The D^2Q model is described in [9].

For the purpose of this paper, let consider a simple global schema with a single data class `Citizen`. Each citizen's attribute (`Name`, `Address`, `FiscalCode`) is associated to a set of data quality dimensions, whose values range in {`Low`, `Medium`, `High`}. Each organization of the CIS maps its own data concerning citizens to such a global schema and labels attributes with quality dimensions. To do so, each organization hosts a Quality Factory component that evaluates and exports quality data. This component is

Table 1. Examples of subscription to QNS

#	Subscription
S_1	$\langle Citizen, \perp, \perp\rangle$
S_2	$\langle Citizen, \perp, Name.Accuracy \leq \texttt{medium}\rangle$
S_3	$\langle Citizen, FiscalCode = \text{“}MCLMSM73H17H501F'', \ldots$ $\ldots Name.Accuracy \leq \texttt{medium}\rangle$
S_4	$\langle Citizen : \{INPS, INAIL\}, FiscalCode = \text{“}MCLMSM73H17H501F'', \ldots$ $\ldots Name.Accuracy \leq \texttt{medium}\rangle$

designed in a way that depends on the internal policies and infrastructures of the organization [2].

A *quality change* event occurs within an organization each time the value of a given quality dimension varies. Upon each quality change, the Quality Factory reports to the QNS the data object involved in the change (according to the data representation of the global schema) along with the associated quality data (see also Figure 1). As aforementioned, the QNS is thus in charge of notifying the new data to all interested users that express the features of the events of interest according to the syntax described below.

Subscription language. A generic QNS subscription is a triple in the form: $sub_{\Delta Q} = \langle \delta[: \mathcal{S}], expr_D, expr_Q\rangle$ where:

- δ is a data class of the global schema;
- $\mathcal{S}$ is an optional set of sources for the data class δ, i.e. organizations containing data objects belonging to δ according to the mapping rules. Users can specify the value of $\mathcal{S}$ in order to restrict the possible sources of interest for a given data class;
- $expr_D$ is an expression used to specify the set of data objects of class δ whose quality changes are relevant to the user. It is a conjunction of constraints on some attributes of δ[2];
- $expr_Q$ is an expression used to select notifications for quality changes occurred on data objects selected by $\langle \delta[: \mathcal{S}], expr_d\rangle$. This expression is a conjunction of constraints on some quality dimensions.

At least δ must be specified, while other parameters can be left empty (i.e., set to $\perp$).

Once a user has subscribed for some notifications, she will receive them in the same form they are produced by the Quality Factory of each organization, that is described below.

Notifications of quality changes. The Quality Factory component of each organization pushes quality change events to the QNS in the following form: $\Delta_{Q_{ev}} = \langle \delta,\ d,\ \{\ldots\langle qd,\ v\rangle_i \ldots\}\ \rangle$ where δ is a data class of the global schema, d is a data object of class δ, and the last element is a set of pairs $\langle qd,\ v\rangle$, where qd is a quality dimension and v is the value of the associated quality data, that express the new values of the quality dimensions of data object d.

[2]A constraint is a triple $\langle p, op, v\rangle$ where p is a property of δ, op is an operator (e.g. $=, <, >, \ldots$) depending on the type of p, and v is a (possibly empty) value of the same type of p.

Subscription matching and containment. As aforementioned, users will receive only those notifications that *match* their subscriptions. Informally, a notification N matches a subscription S if the data contained in N belongs to the data class of S and if it also satisfies the expressions in S. Notification matching against a set of subscriptions can be efficiently evaluated using well known algorithms, e.g., [1, 8].

Concerning subscription containment, we give here just the intuitive notion: a subscription S_1 contains another subscription S_2 if all notifications that match S_2 also match S_1. Subscription containment is useful in order to reduce both the use of memory and the network traffic due QNS, as outlined in the remainder of this paper. Formal definitions of matching and containment are available in [9].

Table 1 shows a set of example subscriptions for quality notifications of data objects belonging to the class `Citizen`[3]. It should be easy to check that subscription S_2 is contained in subscription S_1 and it contains S_3 that in turn contains S_4. Therefore S_1 contains all others.

Concerning matching, a valid quality change event generated by a Quality Factory might be:

$$\langle Citizen, Name = \text{“}M.\ Mecalla'' Address = \text{“}Via\ dei\ Gracchi\ 71,\ Roma'' FiscalCode = \text{“}MCLMSM73H17H501F'',\ \{\langle \texttt{name.Accuracy} = low\rangle\}\ \rangle$$

In the case of the example subscriptions given in Table 1, it is easy to see that the above notification matches S_1, S_2, and S_3. Matching with subscription S_4 depends on the organization in which the quality change occurs.

3. Overview of QNS Design

QNS is designed as a distributed service. Each organization hosts an independent instance of the QNS that adopts the architecture described in this section. A QNS instance accepts subscriptions only from users inside the organization it resides in (i.e. its *local users*) and receives notifications from the local Quality Factory and from other QNSs. QNS instances communicate among each other in a peer-to-peer fashion. That is, a QNS acts as a *producer* of information when it has to propagate to other QNSs the notifications produced by the Quality Factory of its organization. On the other hand, QNS acts as a *consumer* of information when it receives notifications produced in other organizations' QNSs, to be dispatched to its local users.

[3]INPS and INAIL are the acronyms of two agencies of the Italian Government.

The design of the internal architecture of the QNS has to face two main problems: *(i)* to cope with the heterogeneity of platforms and network infrastructures building the system and *(ii)* to scale to the large size we can expect in a CIS context.

Concerning heterogeneity, a standard technological solution, i.e. Web Services, is exploited for the communications among QNS instances in order to achieve independence from the specific technological platform of each organization. Concerning scale, the high number of possible users, each possibly issuing several subscriptions, and the high rate of notifications possibly produced by the Quality Factories, are factors that could impact the performance of the system, i.e. computational resources and network bandwidth. In particular, the high number of subscriptions that could be issued throughout the whole system, could imposes a high memory and computational overhead for storing subscriptions and matching each distinct notification against each distinct subscription, while possible frequent notifications may generate huge network traffic.

Both problems of heterogeneity and scale are faced through a layered architecture, in which a layer (namely, the *External Diffusion Layer, EDL*) deals with diffusing subscriptions and notifications among all organizations, while another layer (namely, the *Internal Interface Layer, IIL*) deals with handling subscriptions and notifications for all local users. The subscriptions of local users in an organization are stored inside the IIL. That is, each QNS instance "knows" only its local users' subscriptions and has only a *partial* knowledge about the subscriptions issued by users of other organizations. Knowledge about non-local subscriptions is propagated *selectively* by exploiting *merge subscriptions* evaluated using the subscription containment relation. Furthermore, notification diffusion occurs using *diffusions trees*, i.e. paths rooted at the producer and spanning all the QNSs consumer of a given notification. This allows to limit memory consumption and network traffic, as outlined below.

Merge Subscriptions. Under a complete absence of "knowledge" about subscriptions, when a QNS acts as a producer, each notification should be blindly flooded to all other QNSs, in order to surely reach all interested users. This would cause much unnecessary network traffic as each notification would be received also by QNSs with no interested local users. To avoid this, the EDL of the QNS in the generic organization O_i maintains a particular subscription, namely the *merge subscription*, denoted as $\mathcal{M}_i$. $\mathcal{M}_i$ contains all and only the subscriptions of all the local users of O_i and is computed from the union of the latter. For example the merge of subscriptions S_2, S_3 and S_4 in the examples of Table 1 is the subscription S_2 itself. Clearly, $\mathcal{M}_i$ allows to receive all and only notifications matching some subscriptions issued by some local users of O_i.

The merge subscription of each consumer organization is sent only to the possible producers of notifications matching the merge subscription itself[4]. This limits the use of memory on the producer side. Upon a new notification, a QNS acting as a producer has to check whether the notification matches some merge subscriptions of some consumer QNSs. In this case, the producer QNS sends the notification only to consumer QNSs actually interested in the notification. Note that also subscription traffic is reduced due to the use of merge subscriptions. Indeed, only changes in subscription that trigger a change in a merge subscription of an organization have to be propagated to producers.

Though exploiting merge subscriptions allows to let flow between consumer and producer organizations only the *necessary* subscriptions and notifications, the traffic generated by notifications could still be enough to overload some QNSs. The following section describes a mean to reduce the probability of this event.

Diffusion Trees. If a large number of consumer QNSs is interested in notifications coming from the same producer QNS, the latter could be forced to open a large number of concurrent network connections to reach all of them, and in the presence of a high rate of notifications, the scalability of the system could result compromised. To reduce the impact of this phenomena, each producer QNS, once determined all the consumers of a notification, does not directly send it to them, but it rather calculates a diffusion tree, rooted at the producer and spanning all the consumers. The fan out of each node of the tree is evaluated on the basis of information about the network connectivity capabilities of each network node. Each notification is thus sent from a node of the tree to its sons along with the information about the subtree of further consumers interested in the notification.

4. QNS Internal Architecture

Figure 1 depicts the internal architecture of a generic QNS instance, and shows also the internal fine-grained functional components of each of the QNS layers. In the remainder of this section we briefly present these components. A detailed description is available in [9].

Internal Interface Layer (IIL). The IIL consumer side comprises the components for (i) receiving and interpreting subscriptions from local users (*Interpreter*), (ii) matching notifications for local users (*Local Matcher*) and (iii) implementing the dispatching mechanisms used to actually send notifications to local users (*Local Dispatcher*). On the producer side, the IIL receives quality change events from the local Quality Factory of the organization. The data is first

[4] On the consumer side, the schema mapping is exploited to determine the set of possible producers for a merge subscription. This allows for further reducing the QNS instances involved in the communication with a given consumer only to those that can actually generate *interesting* notifications.

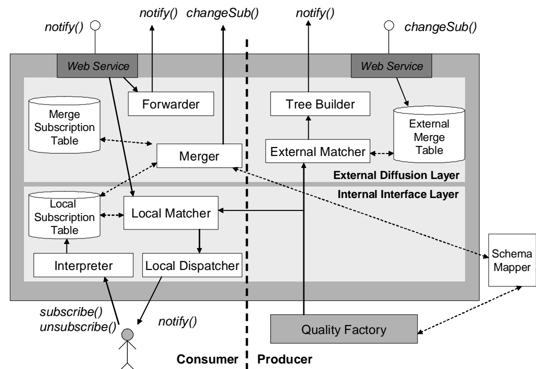

Figure 1. Quality Notification Service

checked inside the organization via the Local Matcher to be dispatched to possibly interested local users. The data is also transferred to components of the EDL to propagate it outside the organization.

External Diffusion Layer (EDL). As aforementioned, this layer is responsible for inter-organization notification diffusion. The communication among organizations is based on Web Services technology. This means that each EDL instance exposes a Web Service interface, containing two methods, namely `notify()` (on the consumer side) and `changeSub()` (on the producer side), that are used to receive from other QNSs a notification and a change of merge subscription, respectively.

The producer side of the EDL of a QNS instance comprises the components for (i) checking the QNS instances to which a notification has to be sent (*External Matcher*), and (ii) determining the actual diffusion path followed by notifications and starting the notification diffusion (*Tree Builder*). Notifications are propagated by invoking the `notify()` method on the Web Services of consumer QNSs. Each invocation contains also the specification of the remaining parts of the diffusion tree to which the notification has to be forwarded by the receiver.

Finally, the consumer side of the EDL of a QNS instance comprises the components for (i) determining the merge subscription for the organization, when a subscribe/unsubscribe request occurs (*Merger*), (ii) sending the notification to all the QNSs at lower levels of the diffusion tree, by invoking their `notify()` method (*Forwarder*).

5. Conclusions and Ongoing Work

The DaQuinCIS architecture provides services that enable the continuous enhancement of data quality in CISs by exploiting data replication. In this setting, QNS enables users to be notified for variations in the quality dimensions of data exported by some organizations. In this paper we have given an overview of this component. In essence, QNS is a *content-based* publish/subscribe system [3], i.e. users can subscribe by specifying information about the content of the notifications they are interested in. Implementing content-based pub/sub systems in large scale distributed systems is a hard task, in particular when aiming to implement efficient *matching* and *routing* algorithms [4]. To the best of our knowledge, the QNS is the first content-based pub/sub infrastructure based on Web-services to account for the peculiarities of CISs. QNS design also benefits from the overall DaQuinCIS framework: differently from "generic" content-based pub/sub systems, QNS is enabled to exploit other services and available tools, i.e. the D^2Q model, the Schema Mapper, and the Quality Factory, which simplify the design and improve scalability.

We are currently implementing QNS, along with other building blocks of the DaQuinCIS framework. Interested readers can refer to the project web site for further information (`http://www.dis.uniroma1.it/~dq/`).

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching Events in a Content-Based Subscription System. In *Proceedings of The Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[2] C. Cappiello, C. Francalanci, B. Pernici, P. Plebani, and M. Scannapieco. Data Quality Assurance in Cooperative Information Systems: a Multi-dimension Quality Certificate. In *Proceedings of the ICDT'03 International Workshop on Data Quality in Cooperative Information Systems (DQCIS'03)*, pages 64–70, January 2003.

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Notification Service. *ACM Transactions on Computer Systems*, 3(19):332–383, Aug 2001.

[4] A. Carzaniga, D. Rosenblum, and A. Wolf. Challenges for distributed event services: Scalability vs. Expressiveness. In *Engineering Distributed Objects '99*, Los Angeles CA, USA, May 1999.

[5] M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS 2002)*, pages 233–246, 2002.

[6] C. Marchetti, M. Mecella, M. Scannapieco, and A. Virgillito. Data quality notification in cooperative information systems. In *Proceedings of the ICDT 2003 International Workshop Data Quality in Cooperative Information Systems (DQCIS 2003)*, pages 47–54, January 2003.

[7] M. Mecella, M. Scannapieco, A. Virgillito, R. Baldoni, T. Catarci, and C. Batini. Managing data quality in cooperative information systems. In *Proceedings of Tenth International Conference on Cooperative Information Systems (CoopIS 2002)*, 2002.

[8] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient Matching for Web-Based Publish/Subscribe Systems. In *Proceedings of Eighth International Conference on Cooperative Information Systems (CoopIS 2000)*, pages 162–173, 2000.

[9] M. Scannapieco, A. Virgillito, C. Marchetti, and M. Mecella. The DaQuinCIS Architecture: a Platform for Exchanging and Improving Data Quality in Cooperative Information Systems. Technical Report Technical Report #18-03, Dipartimento di Informatica e Sistemisticaq, 2003. also available at: `http://www.dis.uniroma1.it/~dq`.