Query Management over Ad-Hoc Communities of Web Services

Apostolos Zarras, Panos Vassiliadis, Evaggelia Pitoura
Department of Computer Science, Univ. of Ioannina
45110 Ioannina, Hellas, http://dmod.cs.uoi.gr/
{zarras, pvassil, pitoura}@cs.uoi.gr

Abstract

In this paper, we present CONSERV – a middleware infrastructure for the development of virtual databases in pervasive computing environments. CONSERV provides an SQL front-end for posing and processing queries on information provided by ad-hoc communities of web services hosted by peers that arbitrarily join and leave the system. The cornerstone of the proposed infrastructure is the fact that we replace the traditional treatment of databases as persistent collections of records by the assumption that a database relation is a collection of records dynamically compiled from such ad-hoc sets of peers. Each peer offers data to the relations through a workflow of web services. Another aspect of our approach is that we confine query processing over specific sets of peers that we call communities. Communities are defined based on the current context of the peer initiating each query. Since our infrastructure departs from the traditional query processing strategies, we discuss query processing as customized in CONSERV.

1. Introduction

Today it is immediately visible that the future of distributed systems is aligned with the general idea of pervasive and ubiquitous computing, which consists of the gradual disappearance of stationary workstations and the distribution of information and computational power in the environment where the users of those systems live and work. Typical applications are met in places like airports, railway stations and shopping centers.

Passing from conventional distributed systems to mobile distributed systems for pervasive computing involves the collaboration of a number of novel technologies such as handheld computers, wireless networks and sensor devises. This large variety of technologies imposes the need for high interoperability amongst the *services* that are provided and used by the constituents of pervasive systems. Several emerging technologies for achieving interoperability rely on the standard web service architecture [1]. An equally important requirement for pervasive systems is *adaptability*. The services provided should be capable to *adapt* appropriately to the constantly changing pervasive execution environment. Achieving adaptability enables the continuous fulfillment of the functional and the quality requirements of the users. Taking an example, changes in the location of a user may result in changes in the availability of the services provided by the pervasive environment.

The term *context* is quite broad and it is defined in [2] as *anything that may influence the state of an entity playing a particular role in a system*. An entity may be a human being, a location, a system element, etc. In our case, we see *a pervasive system as a collection of peers providing a set of web services. Hence, the context of a particular peer, in its broadest sense is the state of the peer itself and the states of the peers that communicate with it.* This particular definition of context is still aligned with the most generic one that is given in [2]. In our case, the entities are peers. Naturally, the state of a peer `p` is influenced either by the peer itself (if, for instance, the peer performs some internal computation), or by the other peers that have access to the services provided by `p`.

In this paper, we present the basic concepts of *CONSERV - a middleware infrastructure, which aims at providing context-aware querying of information, provided in a pervasive computing environment that consists of ad-hoc communities of web services*.

To achieve interoperability, the proposed infrastructure relies on the standard web service architecture for the realization of primitive services that are provided and required by the peers of the pervasive system. In principle, these primitive services can be composed by the users of the system towards the realization of complex workflows.

Workflows are realized using standard XML-based languages such as BPEL [3] and WSFL [4].

However, in a typical pervasive situation, the users will not be sitting comfortably in front of their workstation, having the ability to write down BPEL or WSFL code. On the contrary, they will most probably be in a situation where they will have to quickly compose a workflow that satisfies an urgent need by providing necessary information regarding the users' pervasive execution environment. Moreover, the typical users of pervasive systems will not be experienced developers, familiar with BPEL or WSFL. Hence, we must provide them with more simple means, which shall allow them to efficiently perform their tasks. To this end, we rely on ideas from the field of traditional databases. Typical database query languages provide a classical declarative way for managing and exploring information stored in a database. In our case, *the database is substituted by information, stored everywhere in the pervasive execution environment*. Hence, it is challenging to keep the same classical front-end for managing and exploring this contextual information.

In a nutshell, CONSERV explores two basic concepts:

- Providing a declarative, SQL-based front-end on top of web-services, and
- Confining the query results to the current context of the user through the concept of communities of web services.

Specifically, CONSERV consists of two main subsystems, deployed on the side of each peer that contributes in the community of web services:

- The *query processor* subsystem is in charge of parsing and executing user queries. The query processor takes contextual information into consideration, in order to construct, optimize and eventually execute corresponding execution trees over peer communities of web services.
- The responsibilities of the *context manager* subsystem comprise the management of a constantly changing directory of peers, the identification of the peers that constitute the targets of the user's queries and the determination of web service workflows that must be performed towards performing these queries.

The goal of this paper is to explore the research issues raised by such a system. Rather than delving into one specific technical issue, the focus is on the general infrastructure and the interplay of the various components.

The remainder of this paper is structured as follows. In Section 2, we present a motivating example, which is used throughout this paper to exemplify our approach. In Section 3, we present the general architecture of CONSERV. In Sections 4 and 5, we discuss, respectively, the query processor and the context manager subsystems. In Section 6, we discuss several implementation issues of CONSERV. In Section 7, we present related work. Finally, in Section 8, we summarize our contribution and outline our future research directions.

2. Motivating example

Several kinds of vehicles are driving on the highway from Marseilles to Barcelona. Each vehicle comprises web services, providing dynamically changing information regarding the vehicle's location, velocity and fuel deposit. Moreover, each vehicle comprises services that offer static information concerning its type and technical characteristics. On the highway, there exist exits to parking areas, which may include facilities such as gas stations, fast food restaurants, medical help, and shopping centers. Each one of these facilities comprise web services, which range from simple ones, reporting the existence of the facility, to more complex ones providing information regarding for instance the price lists, the availability of certain goods or the number of patients waiting for medical help.

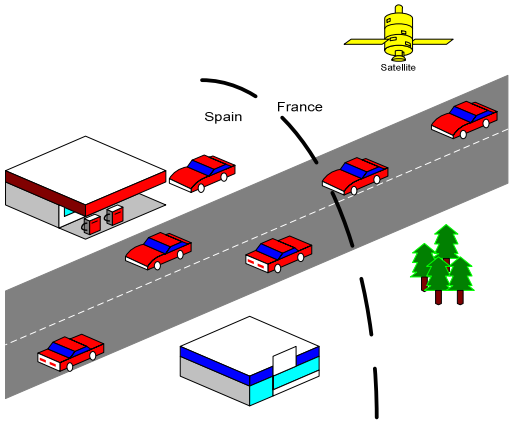

Figure 1. Motivating example.

The drivers of the vehicles may use/query several of the services provided during the drive. For instance, they may be interested in obtaining the following information:

(a) The closest gas-station with a price of gasoline under 2 €/gallon.

(b) The closest Italian restaurant.
(c) Notification for the average speed of all the cars ahead.

3. The CONSERV architecture

3.1. Overview

In a broader sense, we view a *pervasive computing environment* as a collection of peers, *i.e.,* units of data or computation, distributed over the web. The peers may execute on either stationary workstations or mobile devises. A peer *provides* web services to the pervasive environment. It further *requires* using web services provided by other peers.

The peers of the system are organized into *communities*. Communities can be seen as groups of relevant peers. Peers are connected to other related peers through links, thus forming an *overlay network of peers*. The links between two peers do not necessarily correspond to physical communication links, that is, two peers connected to each other may be far away in the physical network. Instead, the distance between peers may be a characterization of their relevance; the smaller the distance, the more relevant the peers. A *community* consists of peers that may use each others' services through a path in the overlay network of a maximum of *n-links*. The peers in the community of a peer are called its *neighbors*.

The number of links is a property that may be customized for each particular application, developed on top of CONSERV. The definition of community is generic. It may be based on actual network reachability between the peers. For instance, in an ad-hoc network, we may consider as the community of a peer the set of all peers that can be reached by it in *n*-hops.

The main objective of the CONSERV architecture is the facilitation of the answering of queries over communities of peers. *The cornerstone of the CONSERV architecture is the fact that we replace the traditional treatment of databases as persistent collections of tuples by the assumption that a database relation is a collection of tuples dynamically compiled from an ad-hoc community of peers, each offering tuples to the relation through a workflow of web services.*

3.2. Peer databases

In our framework, a peer may support database management facilities. To this end, each peer comprises a database, several relations, along with their schemata and data. Naturally, a database relation is instantiated by a set of tuples (i.e., a relation *instance*). A fundamental difference of relations in our framework, as opposed to their traditional treatment, involves their classification as (a) locally stored (i.e., in the traditional fashion), (b) virtual, and (c) hybrid relations. *Locally stored relations* are materialized and lie within a peer's permanent storage device. *Virtual relations* are collections of tuples populated at runtime through the invocation of the appropriate web services of peers in the community. *Hybrid relations* comprise a locally stored part and a virtual part, that is, they consist of both data residing locally and data coming from other peers.

Formally, a database is defined as a set of relations. As usual, a relation is characterized by (a) a *name*, (b) a *schema*, i.e., a finite set of attributes, (c) an *instance*, i.e., a finite subset of the Cartesian product of the domains of the attributes of the schema, and (d) a *type*, ranging in the set of values {`local`, `virtual`, `hybrid`}. No global schema is assumed, and each peer has its own relations. Whereas the local schema is fixed, the contents of the relations are not. Coming back to our reference example, we assume that the peer p_0 carries the database of Fig. 2.

Virtual relations
CARS(ID,PLATE,BRAND,VEL)
Locally Stored relations
BRANDS(BRAND,COUNTRY,METRICS_SYSTEM)
MAPS(X-UP,Y-up,X-LOW,Y-LOW,IMAGE)
Hybrid relations
SITES(SID,NAME,X,Y)
GAS-STATIONS(SID,PRICE,FACILITIES), SID references SITES(SID)
HOTELS(SID,PRICE-SINGLE,PRICE-DOUBLE,FACILITIES), SID references SITES(SID)
RESTAURANTS(SID,TYPE,MENU,PARKING), SID references SITES(SID)

Figure 2. Database scheme of peer p_0.

The `CARS` relation describes neighboring cars by a CONSERV-generated identifier, their plate number, their brand and their current velocity in Km/h. A typical tuple in this relation would be the following: `(12345, HPX7864, RENAULT, 105)`. The `BRANDS` relation describes technical characteristics of different kinds of cars that may circulate along with p_0. These characteristics include the brand that uniquely identifies a particular kind of cars, the country that constructs these cars and the metrics

system (e.g., SI, CGS, etc.) used in these cars towards measuring car related features such as velocity. A typical tuple in this relation would be the following: `(RENAULT, FRANCE,SI)`. The `CARS` relation is virtual because its contents depend on the contextual information that comes from p_0's neighboring cars. Information stored in the `BRANDS` relation does not depend on the pervasive environment; it simply encapsulates a constant part of p_0's knowledge on the pervasive environment. The `MAPS` relation describes the parts of a map that are depicted over the screen of the car. Each part comprises a set of coordinates for its upper and lower points and a bitmap figure stored as a binary large object in the database of the car. Finally, a set of hybrid relations involving useful sites is stored in the database of the peer. The relations are hybrid, since the database has a set of locally stored, well-known sites (e.g., monuments, well-known hotels, etc.) as well as a set of sites discovered as the peer travels. Also, some of the information of the locally stored tuples (e.g., hotel prices) can be updated when the sites are in the neighborhood of the peer. The relation `SITES` is acting as a super-class table of the three other hybrid relations; still, the scheme that we employ is a pure relational one.

3.3. Processing queries

Queries are posed against the database of a peer. The "user" that issues a query need only know the names and schemata of the relations being used; the nature of the relations and the workflows necessary for the collection of the values of the virtual relations are transparent to the user.

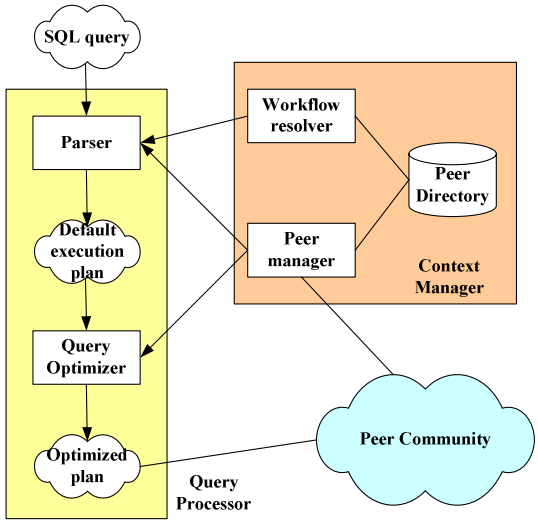

Figure 3. The CONSERV architecture.

The query is expressed in standard SQL (*i.e.*, the nature of the involved relations is transparent to the user) and the collection of tuples is automatically performed by the system. Figure 3 depicts the different stages of query processing for user queries. First, the SQL query is parsed by the *query processor*. As in traditional DBMSs, the query processor receives a declarative SQL query and produces a procedural *execution tree* to be issued against the underlying data. In our case, the execution tree involves the integration of information coming from different peers.

There are several steps to be taken towards the construction of the execution tree:

1. Identification of the peers to be probed for tuples. To facilitate this task, there is a *directory of known peers* in the community of the peer serving the question and a *peer manager* that ultimately determines which peers are to be contacted.

2. Identification of the workflows of web services that need to be invoked for each peer. In the simplest case, each relation in the local database is linked to the execution of one or more web services in remote peers. Each of these web services, in turn, returns a message that corresponds to one, several, or all the attributes of the relation that we wish to populate. In more complicated cases, it is quite possible that we need to transform, merge, cleanse or, in any case, process this incoming information before propagating it further towards the local relation. Hence, in general, we need a workflow of web service operations in order to obtain the tuples from each peer. The complexity of the workflow may vary along with the overhead introduced during its execution. The determination of this workflow is performed by the *workflow resolver*. In CONSERV, we treat such workflows as connected digraphs comprising at least a fountain start node and a sink end node.

The peer's directory, the peer's manager and the workflow resolver form the context manager subsystem, which together with the query processor constitute the overall CONSERV architecture. In the remainder of the paper, we discuss further details and various policies supported by these subsystems.

4. Query processor

The query processor constructs a first execution tree, by employing information regarding which peers are to be contacted and how. The algorithm for achieving this is given in Table 1.

Table 1. Constructing an execution tree.

Algorithm `ConstructExecutionTree`
Input: an SQL query `Q` over hybrid relations and a finite set of web service workflows **F**
Output: an execution tree
Begin
For each hybrid relation in the `FROM` clause of the query construct a sub tree as follows:
1. The leaves of the tree are (a) web services at the peers, or (b) the locally stored parts of the hybrid relation
2. The root of the subtree is a `UNION` operator that collects the information from the different parts
3. Local parts of the hybrid relation are directly connected to the root
4. For each peer leaf `p`, the end node of the workflow f_p (belonging to **F**) is connected to the `UNION` node and the start node is connected to the peer `p`.

The rest of the tree is constructed following the traditional SQL parsing algorithm. First, joins are placed on top of the UNION nodes, followed by selections, projections, groupings and orderings.
End.

Coming back to our motivating example, assume that a peer p_0 is driving from Marseilles to Barcelona. The database relations at peer p_0 are as depicted in Fig. 2. The relation `CARS` is virtual; it describes neighboring cars of p_0, while the relation `BRANDS` is local since information stored in the `BRANDS` relation does not depend on the pervasive environment.

In Fig. 4, we depict an example of how the query processor works. At the left part of the figure we see the original SQL statement. At the middle of the figure, the original execution tree is depicted. Observe that the tree has a distributed part (under the UNION node) and a local part (everything above the UNION part).

While the collection of data is realized in a distributed manner, joins, selections, projections, groupings and orderings are performed locally by the query processor. Nevertheless, there are chances for obtaining the answers a lot faster than this, by distributing some of the operations. Therefore, once the originating execution tree over the different peers is constructed, it is propagated to the query optimizer. Then, the final query execution tree is produced and executed over the involved peers. The results are calculated and returned to the issuing peer. An optimized tree for our example is shown at the right part of Fig. 4.

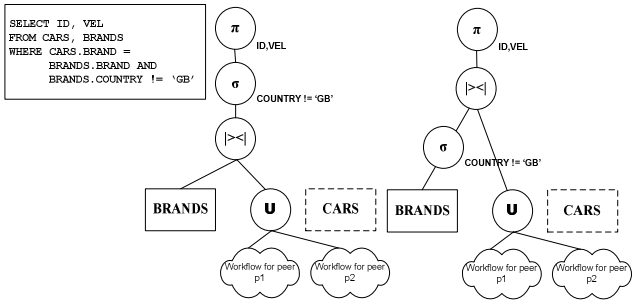

Figure 4. Query processing example.

Assume now that two neighboring cars p_1 and p_2 are in the same community with p_0. Peer p_1 provides a web service that includes the `get_state()` operation, returning an XML message with the attributes `[PLATE_NUM,BRAND,VELOCITY]` and peer p_2 provides a web service that includes the `get_velocity()` operation that provides its velocity in miles/hour and the `get_brand()` operations that provides its brands.

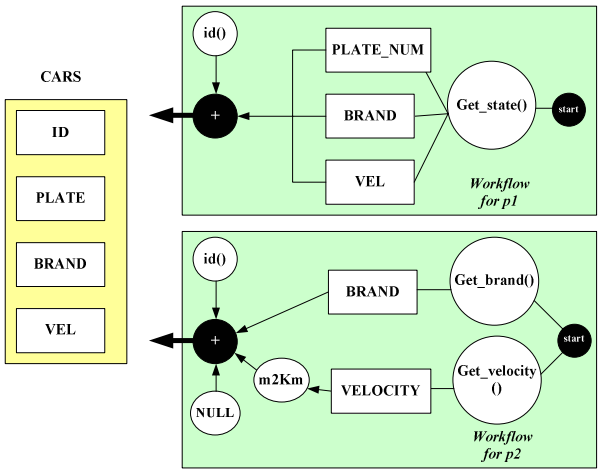

Figure 5. Answering a query by employing different workflows, per different peer types.

The mapping between the local peer's relation and the web services provided by the peers handles the following technical problems.

- As far as peer p_1 is concerned, a CONSERV-generated id is generated and added to the incoming tuple through a tuple constructor.
- As far as peer p_2 is concerned, a tuple constructor is generated at p_0's side as a

placeholder for the results of the invocation of the two services of p_2, miles/hour are converted to Km/hour and a NULL value along with a system generated id must be produced and added to the incoming tuple.

The first from the above points is realized by the upper workflow of Fig. 5, while the second point is realized by the lower workflow of Fig. 5. Selecting the appropriate workflows for each kind of peers is a responsibility of the *workflow resolver*, which is part of the context manager, discussed in Section 5.

5. Context manager

There are several technical issues regarding the way communities of web services are formed to answer queries. In this section, we discuss these issues and the approach followed in the CONSERV infrastructure. Moreover, we explain the role of communities in their management.

5.1. Managing communities

In the CONSERV infrastructure, the result of a query depends on which peers belong to the community of the peer that poses the query. Thus, determining the list of neighboring peers to be bound for answering the query is a central part of query processing.

Various policies regarding how information about the members of each community is maintained may be followed by the peer managers (Fig. 3). At one end, there is a single peer per community that comprises a peer manager, which maintains the peer directory, that is, the members of its community. At the other end, each peer may comprise a peer manager that maintains a local directory with the set of its neighboring peers. Between these two extremes, membership information may be distributed among all members of the community.

Another technical issue that also arises here is the refreshment of the content of the peer directory. Several policies may be followed in CONSERV to accomplish this task:

- *Always-update*. The peer directory is kept up-to-date. When a peer leaves its current community or enters a new one, the peer directory is updated to reflect the new membership information. This is a form of push-based update, in the sense that it is initiated by the departing (entering) peer.
- *Lazy-update*. The peer directory is updated only on demand when a peer poses a query and its neighboring peers must be determined.
- *Periodic-update*. The peer directory is refreshed at pre-specified time intervals. This can be either push or pull-based. In the pull-based case, the directory update is initiated by the peer holding the directory to be refreshed, while in the push-based case, the update is initiated by each peer that communicates its position to the related directories.

5.2. Timing of queries

Given that the members of each community (and thus the results of a query) change during query evaluation and execution, we need to specify the validity of each result set. Let p_q be the peer issuing the query at time instance t_{init} and t_{result} be the time the results are received by p_q. Let $C_t(p_q)$ denote the community of p_q at time instance t and let S_R be the set of peers that participated in the execution of the query. We introduce the following quality measure:

$$Validity = |S_R \cap C_t(p_q)| \; / \; |S_R \cup C_t(p_q)|.$$

for $t \in [t_{init}, t_{result}]$. The validity measure characterizes how far is the set of peers that actually participated in the computation of the results from the set of peers that belonged to the community of p_q at some time instance t during query execution. For instance, t may be set equal to t_{init} or t_{result}. In the former case, we evaluate the query validity with respect to the time the user issued the query, whereas in the latter, we evaluate the query with respect to the time that the results are presented to the user.

The tracing of the members of a community is done in a peer directory. There are two ways to manage the peer directory: (a) a single centralized directory keeps track of the peers in each community and (b) each peer retains a local directory for this purpose. Although in CONSERV we follow the latter approach, we provide a generic computation model that abstracts from the particularities of such a choice.

Providing an estimation of the validity of query results that can be attained in a system such as CONSERV further involves identifying the members of S_R. To achieve this, we rely on the three validity metrics, namely snapshot, interval and single-site validity, introduced in [5] for computing aggregate queries in dynamic networks, such as in peer-to-peer (p2p) and sensor networks. *Snapshot validity* requires that the set of peers S_R contributing to a virtual relation are members of the community of p_q at some point instance t during the execution of the query,

that is $S_R = C_t(p_q)$ for some $t \in [t_{init}, t_{result}]$. *Interval validity* requires that S_R includes peers that were members of the community of p_q at some time instance (not necessarily the same for all of them), during query execution. *Single-site validity* restricts S_R to those peers that were reachable during query execution. It can easily be shown that neither snapshot nor interval validity can be attained in a setting such as ours where dynamic updates are possible. Thus, we aim at single-site validity.

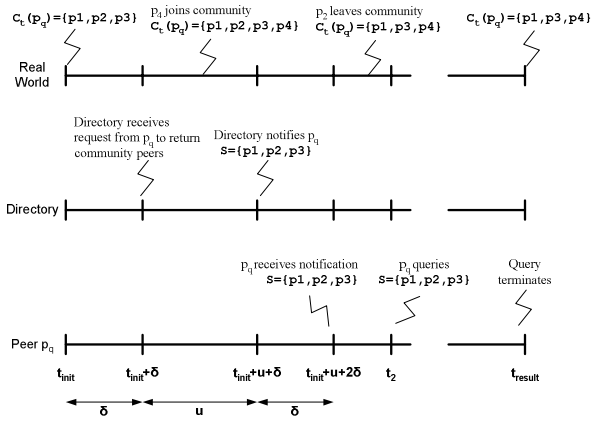

Figure 6. Time axis for the real world, the directory and the querying peer.

We assume that there are two distinct phases in the evaluation of a query following the construction of the optimized query execution tree. In Phase 1, the set S_q of peers that are involved in the execution of the query is identified. During Phase 2, each peer in S_q is contacted, evaluates locally the corresponding web service workflow and returns the results to the issuing peer.

After the completion of each phase, it is possible to compute the validity of the query results attained using the three update policies (always, lazy and periodic). The three policies affect the set S_q computed during the first phase. We assume a relaxed asynchronous model, where there is a maximum delay δ between any two peers. Let t_1, $t_1 > t_{init}$, be the time instant, Phase 1 starts (i.e., the optimized execution tree is constructed).

For Phase 1, we get singe-site validity with respect to peers reachable by the directory. In particular, the set S_q includes peers reachable from the directory that were members of the community of p_q during some interval $[T_1, T_2]$, where T_1 and T_2 depend on the update policy. Next, we compute T_1 and T_2 for each update policy. With the always-update policy, the set S_q corresponds to the peers that were members of the community during $[t_1, t_1 + \delta]$. With the lazy-update policy, there is an overhead for updating the directory, let this be u time units. The set S_q is the set of peers that are members of the community during $[t_1 + u, t_1 + u + \delta]$. Let P be the period of the periodic update policy. In this case, the set S_q is the set of peers that are members of the community during $[t_1 - (t_1 + \delta) \bmod P, t_1 + \delta - ((t_1 + \delta) \bmod P)]$.

Phase 2 is initiated with the appropriate S_q as defined above. We assume a simple evaluation model, where each site in S_q is contacted directly by p_q and returns the results to it. Let t_2 be the time instant when Phase 2 starts. For the always-update and the periodic policies, $t_2 = t_1 + 2\delta$. For the lazy-update policy we have, $t_2 = t_1 + u + 2\delta$. During this phase, a subset of peers in S_q is contacted namely those peers that were reachable from p_q during $[t_2, t_{result}]$. Thus, $S_R = \{p: p \in S_q$ and p reachable from p_q during $[t_2, t_{result}]\}$.

In Fig. 6, an example is presented for the lazy-update policy. It depicts a peer p_q, the real world (referring to the community of peer p_q), and the peer directory that this community (including p_q) employs. At time point t_{init}, peer p_q contacts the directory to learn the members of its community. We assume that p_q takes an interval δ to reach the directory. The directory responds to p_q after a processing time u with a list of neighboring peers. Then, p_q contacts these peers with a query, at time point t_2. Unfortunately, in the meanwhile, the configuration of the community has changed, with peer p_4 joining and peer p_2 leaving the community. By the time the query is completed at time point t_{result}, its validity has decreased to 50%.

Another technical issue that results from the dynamic nature of the environment is *query termination*. Typically, a query is considered to be successfully completed when all relevant peers reply. However, as we have just shown, the fundamental assumption that each query will eventually terminate does not necessarily hold. Consequently, in the CONSERV framework, a query requires a termination condition for each of the operators that constitute the query expression. This can involve a timeout, the number of tuples collected, the percentage of virtual relations that were successfully populated or a combination of the above.

Additional issues related to the timing of queries arise in the case of *continuous queries*. For a continuous query, the contents of the relation are

continuously modified according to the state of the community peers. We discriminate two cases:

- *Periodic (or pull-based) refreshment: In this case,* the peer posing the query periodically polls its community to collect tuples. Note that this corresponds to resubmitting the query and that the community, i.e. the set of peers, that each time receives the query may be different.
- *Push-based querying,* where each of the contributing peers notifies the peer that posed the query, whenever one of the tuples that it contributes changes value.

5.3. Workflow resolution

A central issue in query processing is determining how the virtual and hybrid relations are to be populated. This relates to how the peers carrying fragments of the virtual parts of the relations will be contacted, that is, how the workflows of web services are formed. In our setting, this resolution is facilitated through the usage of profiles and rules for peers.

Peers with common interfaces (i.e., peers exporting the same set of web services) are organized into *peer classes*. Each peer is aware of the interface of a specific set of classes. Such information is maintained in a *class profile*. A *rule* is a mapping of the form:

(relation fragment, peer class) => *web service workflow*

The semantics of the mapping is that once the profile of a neighboring peer is resolved, then, depending on the internal relation to be queried, a specific web service workflow is employed. It is possible that instances of a peer class can populate a subset of the attributes of an internal relation. The term *relation fragment* refers to the subset of the attributes of the internal relation that pertains to the information returned by the web service workflow of a rule.

Naturally, this kind of meta-information would probably be impossible to hard-code for all potential peers. Therefore, the role of classes is crucial, since they act as classes of peers, all exporting the same interface.

In our motivating example, in Fig. 7, we have seen that peer p_0 contacts its two neighboring peers differently. This is due to the fact that peer p_1 belongs to the class *European car* and peer p_2 belongs to the class *American car*.

- Peer p_1 provides a web service that includes the `get_state() operation`, returning an XML message with the attributes `[PLATE_NUM,BRAND,VELOCITY]`. The rule for the mapping is:

```
(CARS(PLATE,BRAND,VEL),"European car")
                              => get_state()
```

- Peer p_2 provides a web service that includes the `get_velocity()` operation that provides its velocity in miles/hour and the `get_brand()` operations that provides its brands. The rule for the mapping is:

```
(CARS(VEL,BRAND),"American car")
     =>tuple[get_velocity(),get_brand()]
```

Rules and profiles are managed by the peer manager and used by the workflow resolver. Naturally, other schemes can also be devised, especially if we keep scalability and evolution in mind. For example, a super-peer acting as a web service repository can inform interested peers on the interfaces of the peers of their community.

Another issue that needs to be clarified is the locality of the execution of the workflows of web services. Workflows comprise specific operations to be executed; therefore, once the operations involved in a workflow are resolved, the only pending issue is the monitoring of the whole execution. By default, this is handled by the local peer, although one can envision schemes where a third party is employed for this task.

Still, if a certain service can be obtained by more than one peer (e.g., both the local and the queried peer can perform a translation of miles to kilometers), a decision must be made about which particular operation will ultimately be invoked. The role of the workflow resolver further includes resolving such ambiguities.

6. Implementing CONSERV over JXTA

The development of the first prototype of the CONSERV infrastructure is based on JXTA [6]. JXTA is a lightweight platform, capable of supporting the organization of p2p communities. JXTA peers may execute on either stationary or mobile devices and they communicate through the exchange of XML messages. In the particular case of CONSERV, the messages further conform to the SOAP format, as imposed by the web service standard architecture [1]. JXTA provides a variety of primitive services on top of which we build CONSERV. Specifically, CONSERV relies on the following:

- The *Membership* service, which allows peers to create, join and leave communities (groups is the exact term used in JXTA).
- The *Pipe* service, which allows peers to create pipe connections between them. A pipe is the basic message exchange mechanism provided by JXTA.
- The *Discovery* service, which enables peers to discover peer groups and pipes.

In our first CONSERV prototype, we assume that each peer contains a local peer directory, managed by a locally deployed peer manager. Whenever a peer intends to join a community it follows the steps below:

- First, the peer queries, through the peer manager, the JXTA Discovery service for available communities.
- Based on the results of this query, the peer selects one (or more) community to join.
- Finally, the peer joins the community, using the Membership service.

Once these steps are completed, the peer is a member of the selected community. However, the peer's local directory must be organized into classes of peers with compatible interfaces. This functionality is not directly supported by JXTA. To work around this problem, we use the notion of pipes. First, we describe the initialization of the local directory of a peer. Each peer that joins a community creates a different input pipe for every web service it provides. Each input pipe is identified by the JXTA peer identifier and by the URI of the WSDL service specification. Each input pipe is then published using the JXTA Discovery service. Following, the peer queries the JXTA Discovery service for input pipes that provide compatible interfaces, *i.e.*, input pipes that have the same URI. The peer issues one query per different class of peers and the results are stored in different sub directories within the local peer directory.

Once the local directory of a peer has been initialized, it is ready for use. As time passes, the community around a peer is modified. As already discussed in Section 5.1, the update of the local directory may take place according to three different policies. JXTA provides primitive support that facilitates their realization. To implement the always-update policy, CONSERV peer managers register in the Discovery service as Discovery listeners, providing a particular call-back service, which is invoked upon the occurrence of a new pipe advertisement. The realization of the periodic-update and the lazy-update policies is much simpler. The CONSERV peer managers contact the Discovery service for new pipe advertisements, either periodically or right before issuing a query for contextual information.

7. Related work

To our knowledge, CONSERV is the first attempt to employ web services over peers as a substitute to relational databases. Therefore, in this section, we review research results that serve more as a background, rather than as alternatives to our approach.

Data integration is an old problem, addressed by the database community for a long time [7, 8, 9]. Although several research efforts exist, we believe that the focus of the research community so far is not directly relevant to the CONSERV approach, since the fundamental assumptions underlying earlier approaches focus on data sources as data providers and not ad-hoc, autonomous peers in a p2p framework. Still, we mention a few prominent works that appear to be closer to the CONSERV framework. An excellent survey of problems typically occurring in the context of data integration is found in [10] (still, with a special focus on data warehousing). In [11] the author discusses the issue of generic model management, with a focus on mapping different models to each other.

Query processing in a distributed setting has been extensively studied in the database community [12, 13]. An important research problem is how to combine the workflows involved in the execution of the queries and the algorithms employed in distributed query processing in our setting. The workload balancing/optimization of the peers executing services is also an area where existing results [14, 15] can be exploited by the CONSERV architecture.

Communities of peers are usually organized in terms of *Semantic Overlay Networks* (SONs). Different kinds of SONs have been proposed so far. Structured SONs [16, 17] assume that the peers are conceptually connected into specific topologies. Based on these topological constraints, they provide guarantees on the overall cost for performing queries. Unstructured SONs [18, 19, 20] provide no such guarantees as they do not impose any topological constraints. On the other hand, they are simpler to manage in the presence of peer join and leave actions. Extending the CONSERV technique for constructing communities of peers with SONs is a challenging issue.

8. Conclusions and future work

In this paper, we have presented *CONSERV – a middleware infrastructure for the development of virtual databases that enable the management of contextual information, spread in pervasive computing environments*. The contextual information we deal with is provided by ad-hoc communities of web services offered by peers that arbitrarily join and leave the pervasive computing environment. CONSERV provides an SQL front-end for the answering of queries. The cornerstone of the proposed infrastructure is the fact that we replace the traditional treatment of databases as persistent collections of tuples by the assumption that a database relation is a collection of tuples dynamically compiled from an ad-hoc set of peers. Each peer offers tuples to the relation through a workflow of web services.

CONSERV raises many issues for future work. In terms of performance, a central issue is deriving efficient workflow execution plans, coupled with appropriate algorithms for updating the peer community. In terms of dependability, it is important to construct workflow execution plans by taking into account the mobile nature of peers, which may cause failures. Regarding the management of context, we plan to extend the notion of community exploring various aspects of context.

Acknowledgement

This research was funded by the program "Pythagoras" of the Operational Program for Education and Initial Vocational Training of the Hellenic Ministry of Education under the 3rd Community Support Framework and the European Social Fund.

References

[1] W3C. Web Services Architecture. http://www.w3.org/TR/ws-arch/

[2] A. K. Dey. Understanding and Using Context. In *Personal and Ubiquitous Computing*, vol. 5, no. 1, pp. 4-7, 2001.

[3] Business Process Execution Language for Web Service (BPEL4WS) v.1.0. http://www.ibm.com/developerworks/webservices/library/ws-bpel/

[4] WSFL. http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[5] M. Bawa, A. Gionis, H. Garcia-Molina, R. Motwani. The Price of Validity in Dynamic Networks. In *Proceedings of ACM SIGMOD* 2004.

[6] SUN. Project JXTA v2.0: Java Programmer's Guide. http://www.jxta.org/docs/JxtaProgGuide_v2.3.pdf.

[7] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer,* vol. 25, no. 3, pp. 38-49, 1992.

[8] A. Y. Levy, D. Srivastava, T. Kirk. Data Model and Query Evaluation in Global Information Systems. *Journal of Intelligent Information Systems,* vol. 5 no. 2, pp. 121-143, 1995.

[9] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems,* vol. 8, no. 2, pp. 117-132, 1997.

[10] D. Calvansese, G. deGiacomo, M. Lenzerini, D. Nardi, R. Rosati. Source Integration. In Matthias Jarke, Maurizio Lenzerini, Yannis Vassiliou, Panos Vassiliadis (eds.). *Fundamentals of Data Warehouses*. Springer-Verlag, 2000.

[11] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research* (CIDR'03), pp. 5-8, 2003.

[12] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, vol. 32, no. 4, pp. 422-469, 2000.

[13] M. T. Özsu, P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991

[14] M. Gillmann, G. Weikum, W. Wonner. Workflow Management with Service Quality Guarantees. In *Proceedings of ACM SIGMOD*, pp. 228-239, 2002.

[15] M. Gillmann, J. Weißenfels, G. Weikum, A. Kraiss. Performance Assessment and Configuration of Enterprise-Wide Workflow Management Systems. Enterprise-Wide and Cross-Enterprise Workflow Management, pp. 18-24, 1999.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM*, 2001.

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. Chord. A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, 2001.

[18] E. Pitoura, S. Abiteboul, D. Pfoser, G. Samaras, M, Vazirgiannis. DBGlobe: A Service Oriented P2P System for Global Computing. In *Sigmod Record*, vol. 32, no. 3, 2003.

[19] G. Koloniari and E. Pitoura. Filters for XML-based Service Discovery in Pervasive Computing. *Computer Journal: Special Issue on Mobile and Pervasive Computing*, in press.

[20] G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. In *Proceedings of EDBT*, 2004.