PERSEND : Enabling Continuous Queries in Proximate Environments

David Touzet, Frédéric Weis, Michel Banâtre

To cite this version:

David Touzet, Frédéric Weis, Michel Banâtre. PERSEND : Enabling Continuous Queries in Proximate Environments. [Research Report] RR-4780, INRIA. 2003. inria-00071806

HAL Id: inria-00071806

https://hal.inria.fr/inria-00071806

Submitted on 23 May 2006

PERSEND : Enabling Continuous Queries in Proximate Environments

David Touzet, Frédéric Weis, Michel Banâtre

N˚4780

March 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4780--FR+ENG

PERSEND : Enabling Continuous Queries in Proximate Environments

David Touzet, Frédéric Weis, Michel Banâtre [*]

Thème 1 — Réseaux et systèmes
Projet ACES

Rapport de recherche n˚4780 — March 2003 — 20 pages

Abstract: In the mobile computing area, short-range wireless communication technologies enable to envision direct interactions between mobile devices. In the scope of data access, devices can now be considered as both data providers and data consumers. Thus, each device can be provided with a remote access to data its neighbours agree to share. Such a service enables applications to consult a set of data providers which dynamically evolves according to the mobility of the neighbouring devices. The set of data sources an application may access by this way is therefore representative of its physical neighbourhood. For this purpose, we propose to design a tool enabling the continuous consultation of neighbouring shared data. We present, in this paper, the PERSEND system we develop in this scope. Based on relational databases systems, PERSEND enables applications to define continuous queries over neighbouring data.

Key-words: Mobile computing, Databases, Continuous queries, Proximate querying

(Résumé : tsvp)

[*] {dtouzet,fweis,banatre}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

PERSEND: Requêtes Continues de Proximité en Environement Mobile

Résumé : Dans le domaine de l'informatique mobile, les technologies de communication à courte portée rendent aujourd'hui possible la mise en œuvre d'interactions directes entre calculateurs mobiles. Dans le cadre de l'accès aux informations, les calculateurs ne sont plus confinés au simple rôle de consommateurs de données mais doivent également être considérés comme des fournisseurs autonomes d'information. Chaque calculateur peut ainsi disposer d'un accès distant aux données que ses voisins souhaitent partager. La mise en place d'un tel service permet aux applications d'accéder à un ensemble de fournisseurs de données dont la composition évolue dynamiquement en fonction des mouvements des calculateurs voisins. L'ensemble des sources de données accessibles à une application à un moment donné est donc représentatif du voisinage physique de cette dernière. Dans ce cadre, nous proposons de développer un outil permettant la consultation continue des données partagées avoisinantes. Nous présentons dans cet article le système PERSEND que nous avons conçu dans cet objectif. Reposant sur un système de bases de données relationnelles, PERSEND permet aux applications de définir et d'exploiter des requêtes continues portant sur des données avoisinantes.

Mots-clé : Informatique mobile, Bases de données, Requêtes continues, Interrogation de proximité

1 Introduction

The recent development of powerful mobile devices has made mobile computing a more and more popular paradigm. Typical mobile environments are today composed of mobile devices accessing data by the mean of a fixed infrastructure (such as 802.11b cells). These environments are based on non-symetrical interactions s-ince the infrastructure is the only data provider, the mobile devices being confined to a role of data consumers. These client/server exchanges mainly bear on structured data (such as visiting cards, buses timetables, ...) which are usually stored in database systems. Using wireless communication channels, mobile devices can download information as long as they are situated in the network communication area. Disconnections from the fixed network occur as soon as mobile devices move away from the network communication area. Although, in such environments, they are supposed to be temporary events, disconnections raise many challenges in the data management domain. Some approaches, such as hoarding and optimistic replication [1], have been introduced in order to address data access issues.

Recently, in the area of pervasive environments, the rise of short-range communication technologies, such as Bluetooth [2], has enabled the emergence of a new type of mobile environments dealing with direct and proximate interactions between devices. Considering each device as a potential data provider, they aim to promote direct exchanges between physically close enough devices. Due to their limited communication range, considered devices can only directly communicate with their closest neighbours. Thus, two mobile devices are declared to be neighbour as soon as they are able to directly communicate one with the other. In the remaining of this paper, such environments are called *proximate environments*. As proximate interactions are based on direct communications between mobile devices, fixed networks are no longer necessary. Such an approach enables us to envision new kinds of applications.

In proximate environments, each device sharing some local data has to be considered as a data provider. In such a context, the data set each device can access at a given time includes both local data and data shared by the neighbouring devices. Mobility, which makes devices coming closer or going away from the others, breaks and establishes neighbourhood relationships. Due to this mobility, the set of neighbours of each device evolves in time. Therefore, the data set each entity can access evolves according to its set of neighbouring data providers. As devices may update their own shared data, the data set a device can access also has to evolve according to the modifications processed in its neighbourhood. Consequently, data which are available to a device in a proximate environment not only depend on the neighbouring devices but also on the updates these devices perform on the data they share.

Whereas mobile systems attempt to enhance data availability for mobile devices despite intermittent connections with the fixed network, proximate environments aim to enable neighbourhood interactions in which wireless connections are used to define the set of data providers to be considered. Accordingly, techniques developped for mobile systems do not suit to proximate environments. Thus, no tools were developped for nomadic systems which enables to manage the concept of visible space as previously defined. In mobile environments, mobile devices usually communicate with a single entity at once: the base station which acts as a gateway to the wired network. Moreover, whatever its position, and more generally whatever its physical context (including its physical neighbours), a mobile device can access the same data, assuming it is situated in a covered area.

In this paper, we propose a system enabling applications to access available neighbouring data in the scope of proximate environments. As a large part of data is today managed by databases, our system has been developped using relational databases systems (RDBMS). In this context, shared data can be modified by the mean of the three data handling commands: data insertion, data removal and data update. Rather than providing a complete view of the available data, our system is designed to enable users to query these data for some specified subsets. Just as the whole set of available data, the data subsets queried by users evolve according to the set of neighbouring data providers. However, they also have to reflect the conditions users specify. In order to enable applications to be continually aware of their currenlty available data, our system provides them with persitent data sets which match the users' conditions. For this purpose, it enlarges some works previously performed in the *continuous queries* area.

This paper is organized as follows. In Section 2, we present the concept of continuous queries and highlight the main issues to be addressed in order to process such queries in proximate environments. Section 3 details the semantics we have considered to develop proximate continuous queries. In Section 4, we describe PERSEND (PERsitent SEnsing for Neighbouring Data), the continuous query system we designed for proximate environments. We discuss, in Section 5, some implementation issues. Section 6 deals with related works. Finally, Section 7 presents some conclusions and future works.

2 Continuous querying challenges in proximate environments

In this section, we first review some of the main existing continuous queries systems. We briefly present the context of these studies and the architectures which have been developped. Then, we explain how proximate environments differ from these works and what kind of specific constraints they have to face.

2.1 Continuous queries: goals and design

Users querying continually changing databases may want to be notified of data updates which occured since a query has been submitted. The simplest way to provide this service is to process the query again each time the database is updated and to return to the user the corresponding data set. This approach can prove to be extremely ineffective. Given a user's running continuous query CQ bearing on a single table, let us consider the insertion of a record r which is relevant to CQ. Then, providing the user with an up-to-date data set can be achieved by adding selected fields from r to the current data set.

Terry introduced the continuous query concept in order to manage such challenges. They are defined as queries that continually run once issued [3]. Considering a model limited to append-only tables (tables only accepting new insertions), Terry designed a system enabling the continuous querying of the Tapestry messaging system. By the mean of continuous queries specified using the SQL querying language, users can define some messages filters. Thus, they are notified as new messages matching their filters are received by the system and inserted in storage tables. Submitted continuous queries are recorded by the system and are processed so as to build corresponding incremental queries enabling to efficiently get up-to-date results. Notification frequency can be defined on a per user basis: for example, once a day, once a week or as soon as inserts are processed (this last option enables to have a continuous up-to-date view of requested data).

The previous continuous query model is highly centralized since a front-end server has to store all managed data and to perform the whole querying process. In the sensors database area, on going works on *long-running queries* are extending this restricting model [4]. Observing that interconnected sensors are now widely deployed [5], sensors database systems aim to enable to efficiently collect data from this kind of information providers. Centralized schemes proved to be unsuited to sensors database interrogations:

- sensors continually have to send captured data to the front-end server, thus overloading the communication network
- answering a query on a single sensor is performed by searching through the entire database: this process includes data from non-relevant sensors

Typical queries submitted to sensors database systems ask for values currently measured by some sensors. For example, a user can ask temperature sensors situated in a building for the current temperature every ten minutes. As each sensor is assumed to embed storage, computing and networking capabilities, distributed query processing schemes can be used. For this purpose, a front-end server is used to store a description of managed sensors. Each sensor is associated with an ID and some physical attributes (such as its location). Thus, queries execution can be distributed on sensors specified by users' conditions: non-relevant sensors are not included in the query execution plan. Moreover, only data which are relevant to the query are transmitted from concerned sensors to the front-end server. Otherwise, the concept of *virtual relation*, introduced by Bonnet [4], enables users to interrogate sensors database using the SQL syntax. Data scanned by sensors are indeed represented as append-only relational tables in which new measures are inserted associated with a time stamp.

Beyond this distributed model, the moving objects databases deal with continuous queries which involve mobile objects, such as cars. Usual storage schemes are not suited to manage such objects. As the value of their location continually evolves, keeping an up-to-date representation of mobile objects requires databases to be continually updated. Observing that the description of a mobile object motion is updated less frequently than its position, these systems chose to associate motion vectors to objects representations [6]. Thus, each mobile entity is associated with its last known location, its motion vector and the time stamp of its last update. Assuming a mobile object has kept an unchanged trajectory since its last update, its actual position can be calculated at any time without requiring its stored position to be explicitly updated. Continuous queries submitted to moving objects databases may involve several mobile entities. For example, a user can ask for the devices which are less than one hundred meters away from him. The described storage scheme enables systems to compute at once the data set currently associated to a continuous query, and further ones. For this purpose, a set of tuples $(r, begin, end)$ is built, where the record r belongs to the data set between time *begin* and time *end*.

2.2 Querying proximate environments

The architecture of proximate environments fundamentaly differs from those of s-tudied continuous queries systems. Because proximate environments are totally distributed, each mobile device potentially has to be considered as both a data provider and a query transmitter. As opposed to this model, continuous queries over Tapestry are based on a centralized scheme. In sensors database systems, the continuous querying process is distributed between two different kinds of entities. The sensors are the data providers whereas the front-end server acts as the query transmitter. Moving objects databases offer a more flexible architecture. Queries can be processed over multiple mobile objects which store each a subset of required data [6]. These systems however assume a global connectivity between all mobile objects by the mean of a wireless communication infrastructure. Such an assumption is no more considered as valid in a proximate context. Due to their short-range communication capabilities and to their unconstrained mobility, devices participating in proximate environments can be disconnected one from the others with no undertaking for further reconnections.

Beyond the developped architectures, many differences can be observed between proximate environments and existing continuous queries systems. Contrary to moving objects databases, we do not assume any knowledge about devices motion. This implies that the only computable data sets are those that currently satisfy the continuous queries. Otherwise, the data model proximate environments have to consider is more flexible than those previously described. Consider users sharing information stored in their adress book. Insertions, removals and updates should be allowed by a proximate continuous queries system. Such handlings are not managed by the Tapestry continuous query system which is limited to append-only tables. Likewise, data scanned by sensor databases are modelized by virtual relations which are also append-only.

Proximate environment querying has to deal with more constraints than described querying systems. Its objectives are also different. Whereas most of the systems attempt to enable transparent querying of data providers, whatever their physical location, data providers a device can query in proximate environments are restricted to the device's vicinity. Thus, a continuous query submitted in a proximate environment provides the user with a continuous view of available data matching the query in his physical neighbourhood. We call such a data set a *Continuous Result Set (CRS).* This model implies that data stored by a device should be required to answer continuous queries issued by some of the device's neighbours. Consequently, devices participating in proximate environments have to watch for local data updates

nu_cd	cd_title	cd_price
1	War	8
2	Transformer	16
3	Animals	20
4	Kind Of Blue	32

Table 1: The cd_to_sell table

in order to notify interested neighbours of the occured modifications. Notifications have to enable interested neighbouring devices to keep continuous result sets up-to-date.
In order to enable efficient continuous queries over proximate environments, we have brought out three main challenges to address.

Data providers management. In a proximate environment, each continuous query is submitted to the data providers located in the vicinity of the query transmitter. Each device has to know which are its neighbouring devices. Reminding that considered devices communicate by the mean of short-range wireless technologies, and that they are mobile, the neighbours set of a device can evolve. Consider two devices A and B. As B leaves the vicinity of A, continuous queries issued by A have no longer to take data from B into account. Conversely, as a new neighbour C gets closer from A, continuous queries submitted by A have to deal with data stored by C.

Assessment of the data updates impact. In proximate environments, continuous queries have to return the data set both being stored in the device's vicinity and matching the user's expressed conditions. Let a *querying device* be a device which has issued a continuous query. Applications initiating such queries are called *querying applications*. Devices neighbouring a querying device are called *queried devices*. As data stored on a queried device are modified, the associated querying device has to reflect the processed modifications, assuming that they bear on data relevant to the continuous query. In order to highlight these problems, let us study an example. Let A provide its neighbours with the list of audio CD its user sells (see Table 1). Now, consider a neighbouring device B having issued the continuous query CQ_B: *I'm looking for audio CD which price is between 10 and 20.* Let $CRS(CQ_B)$ be its associated continuous result set. Three kinds of events have to be considered:

- data removal. If removed data are relevant to CQ_B (*i.e.* their price is between 10 and 20), they also have to be removed from $CRS(CQ_B)$.
- data insertion. If some of the inserted rows match CQ_B's condition, they have to be included in $CRS(CQ_B)$.
- data update. For example, A decides to divide its prices by 2 (*price=price/2*). This update has three consequences. First, some of the rows which were relevant to CQ_B may no longer match its conditions. Second, some of the non-relevant rows may now be relevant to the query's conditions. Third, data from $CRS(CQ_B)$ which are still relevant to the query have to reflect the executed update. In our example, the price of *Animals* has to be set to 10 in the continuous result set. Moreover, *Transformer* has to be removed from $CRS(CQ_B)$ whereas *Kind Of Blue* has to be added to (with a price of 16).

Notification of data modifications. A querying device has to be notified when remote data involved in a continuous query it has issued are modified. Queried devices are responsible for this task: they have to notify their querying devices of the modifications to perform on their continuous result sets. Let us consider the update operation of the previous example. Assuming that it knows what data are handled by CQ_B, device A is able to deduce what modifications B has to perform in order to keep its continuous result set up-to-date. For this purpose, A can send to B a message containing three *update commands*:

- remove *Row*2 from $CRS(CQ_B)$
- add $(4, Kind\ Of\ Blue, 16)$ to $CRS(CQ_B)$
- set *price* to 10 at *Row*3

Now we have highlighted the challenges to be addressed, let us define some specific semantics for proximate environments.

3 Defining semantics for vicinity continuous querying

In this section, we present some data querying semantics which are compatible with the specific constraints relative to proximate environments. We first study those related to the vicinity management. Then, we investigate the impact of the duration parameter introduced by continuous queries.

3.1 Vicinity relative issues

Querying neighbouring information supposes that involved devices share some of their local data. Two different data modes are considered: *private* and *shared*. Data in private mode only accept local accesses. Conversely, shared data can be read by any neighbouring device. The data mode is defined at the table level: data from a shared table can be queried by remote devices whereas those from private tables not.

In such an environment, several devices can simultaneously store some data representing a same physical object (or person). These data can be concurrently updated in different ways according to the devices storing them. Due to the absence of a centralized control, the consistency of these co-existing copies can not be insured. Likewise, and for the same reasons, no global identification scheme is available: stored data are identified in an independent way on each device. Therefore, we consider that each database entry locally describes a unique object. In order to prevent any interference between remote identification schemes, objects are globally identified by the couple $(DeviceID, LocalID)$.

In this context, join queries have to be carefully managed. Joining two remote tables has indeed no sense since they rely on different schemes to identify the data they contain. Meaningfull join queries have therefore to be processed on a single device. Consequently, distributed join queries have to be independently processed on each neighbouring device before merging the computed results.

3.2 Duration relative issues

Considering continuous queries introduces some temporal issues. Indeed, built continuous result sets evolve according to data which are available in the devices' vicinity. Common querying languages, such as SQL, have been designed to define and build static data sets. Some of the tools and functions they provide do not suit to manage data sets subject to variations. Thus, SQL enables users to call some aggregation functions (such as `max`, `sum`, `count` ...) in the queries they define. These functions usually compute a single value from a static data set.

In order to manage changing data sets, dynamic semantics have been associated to these functions. Thus, the value to be returned now has to mirror the current state of the continuous result set. For this purpose, this value has to be evaluated again each time the continuous result set is updated. However, in most cases, such computations are not necessary. For example, the value returned by a call to `count(*)` can be easily managed: it as to be incremented each time a row is inserted in the CRS and to be decremented for each removed row.

Likewise, we have defined a dynamic version of the SQL `distinct` keyword. When specified, this keyword ensures that returned data sets are composed of distinct rows. Consider a continuous query CQ using the `distinct` keyword. Two CRS are associated to CQ: the primary one (with no unicity consideration) and the one to be returned. Each time the primary CRS is updated, two questions have to be considered:

- have new distinct rows appeared ?
- have some rows occuring only once been removed ?

We have introduced theorical issues which are specific to proximate environments. We now present the PERSEND querying system.

4 Design of a vicinity querying system

Besides classical queries, the PERSEND querying system enables the continuous querying of proximate environments. In this section, we detail the architecture of this system. First, we introduce some SQL extensions which enable the definition of continuous and proximate queries. Then, we describe the different components composing PERSEND. Finally, we present the way the PERSEND system manages proximate continuous queries.

4.1 Vicinity querying with SQL

SQL has been designed to handle data stored in relational databases. It enables users to issue instantaneous queries, that is queries which return data matching expressed conditions just as they are executed. However, its syntax provides no way to define continuous queries. We have therefore introduced the keyword `continuous` in order to distinguish continuous consultations from instantaneous ones. Positioned at the beginning of a consultation query, it indicates that the query has to be considered as continuous (see Query 1).

Query 1 *Continually querying local audio CD to sell which price is between 10 and 20.*

```
continuous select cd_title, cd_price
from cd_to_sell
where cd_price is between 10 and 20;
```

In SQL queries, data sources are identified by naming the implied tables and databases. In the querying model we consider, neighbouring tables can be implied in the queries a device processes. The system therefore has to know when to only consider local tables and when to distribute a query. For this purpose, we introduce the `vicinity` keyword. It has to be placed at the beginning of a consultation query, after the `continuous` keyword (if specified). It indicates that the following query, continuous or not, has to be distributed among all neighouring devices. We call such queries *proximate queries*. Query 2 extends the previous example by querying all neighbouring devices.

Query 2 *Continually querying proximate audio CD to sell which price is between 10 and 20.*

```
continuous vicinity select cd_title, cd_price
from cd_to_sell
where cd_price is between 10 and 20;
```

Now our SQL-based querying language is presented, let us study the architecture of the PERSEND system.

4.2 Overview of the PERSEND architecture

The figure 1 presents the global architecture of the PERSEND querying system. PERSEND is based on a Relational Database Management System (RDBMS). Besides the neighbourhood manager, which provides applications with information on neighbouring devices, PERSEND is organized around four main components: the query interface, the query parser, the update supervisor and the query descriptors manager. The remaining of this section is dedicated to their description.

The proximate continuous query interface. Queries, whatever their type, are transparently submitted by the way of the continuous query interface. Two primitives are currently provided. The first one, `executeQuery(QueryText)`, is used to submit a query to the system. The type of submitted queries is determined by the query parser. The instantaneous local consultations are processed as usual. Once parsed, modification queries are transmitted to the update supervisor. Continuous queries are, as for them, inserted in the query descriptors list. According to the type of the submitted query, `executeQuery` returns a result set (instantaneous consultations of local data), a query status (data modifications) or a continuous query handler (continuous queries). Data currently matching a continuous query are stored in the CRS

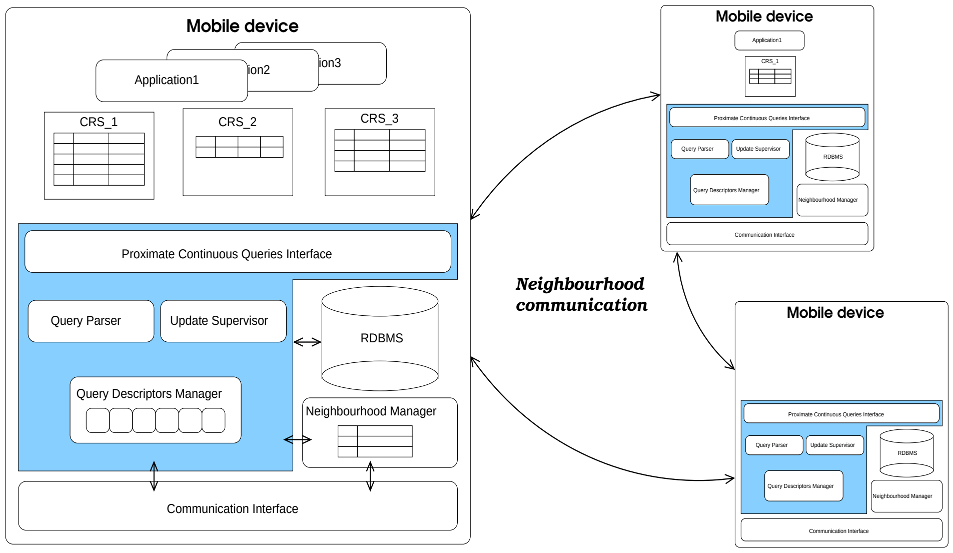

Figure 1: Architecture of the PERSEND querying system

associated to the query. Handlers enable applications to access CRS associated to the continuous queries they have issued. They also provide a global idendification of continuous queries with the couple $(DeviceID, CQueryID)$. The second primitive, `closeContinuousQuery(CQHandle)`, enables applications to close a given on-going continuous query.

The query parser. The analysis of a query associates the query with a descriptor. Besides its type (`select`, `delete`, `insert` or `update`), a descriptor contains all available information on a query: its range (local or proximate), its duration (continuous or instantaneous) and the data it handles. Data handled by consultation queries are identified by $(db\_name, table\_name, field\_name)$ tuples. Descriptors associated to continuous queries are indexed, with their handler, in the descriptors list. Data handled by modification queries are coded in specific ways. Thus, an `insert` descriptor just provides the targeted table, identified by a $(db\_name, table\_name)$ couple, and the row to be inserted. Descriptors associated to modification queries are transmitted to the update supervisor before the queries to be executed by the RDBMS.

The update supervisor. This component has to detect the consequences that submitted modification queries (`insert`, `delete` and `update`) may have on on-going continuous queries. Given the descriptor of a modification query Q_m, it examines all continuous queries in the descriptors list. In order to determine if the execution of Q_m interferes with the result of a continuous query CQ, it computes the intersection between data handled by Q_m and local data which currently match CQ. If the computed set is not empty, the device having issued CQ has to be notified of the modifications to be performed on $CRS(CQ)$.
Remind the example presented in Table 1. We assume the user has sold the CD *Transformer*. Now, he wants to remove it from the table with Query 3.

Query 3 *Removing the 'Transformer' CD from* `cd_to_sell`.

```
delete from cd_to_sell where cd_title = 'Transformer';
```

Consider the continuous query CQ issued by a neighbouring device is still running (see Query 2). When Q_3 is submitted, the update supervisor computes the intersection between data that Q_3 handles ($Row2$) and those which locally match CQ ($Row2, Row3$). As the computed intersection is not empty, and according to the type of the modification query (in this case, `delete`), the querying device has to be notified of the deletion of data locally identified by (`cd_to_sell`, $Row2$).

The query descriptors manager. It manages the list of the continuous queries which are running in the device's vicinity. This list contains two kinds of descriptors: those associated to locally submitted continuous queries (proximate or not) and those associated to proximate continuous queries issued by neighbouring devices. A descriptor is removed from the list when an explicit call to `closeContinuousQuery` occurs. When a device leaves the neighbourhood, the system closes all the continuous queries the device has issued.

The neighbourhood manager. The aim of this component is to provide applications with an up-to-date list of neighbouring devices. This list is stored in the device's *neighbourhood table*. Each device is associated in the table to a unique handler and the time stamp of its insertion. Applications can access the neighbourhood table and read its content each time they require information about their physical vicinity. Those frequently requiring such information may issue redundant readings as the table remains unchanged between successive accesses. The neighbourhood manager therefore provides applications with a notification service. As they subscribe to the service, applications receive the current content of the table. Afterwards, they are notified each time a device leaves the vicinity or a new neighbour is detected. The PERSEND querying system subscribes to the notification service to get information about its physical neighbourhood.

4.3 Managing continuous queries with PERSEND

We have presented the architecture of the PERSEND querying system. We now study how proximate continuous queries are managed: first, on the device having issued it and then on its neighbouring devices. Note that, save the communication issues, local continuous queries are managed in the same way than proximate ones are.

Locally submitted proximate continuous queries. A continuous query is submitted using the `executeQuery` function. As every query, it is transmitted to the query parser. If it is not syntaxically correct, `executeQuery` returns an error to the querying application. Otherwise, an empty CRS is associated to the continuous query and `executeQuery` returns a handler enabling the application to access the CRS.
The analysis of a query provides the system with its associated descriptor. Continuous queries' descriptors are indexed in the descriptors list. Then, PERSEND broadcasts to its neighbourhood the query associated to its descriptor. In the same time, it computes the set of local rows which match the continuous query (by submitting the query to the RDBMS) and inserts them in the associated CRS. As data sets associated to the query are received from neighbouring devices, they are merged to the CRS according to the semantics defined in Section 3.
As a device leaves the vicinity, the rows it has provided are removed from the CRS. A device entering the vicinity is sent all current proximate continuous queries which have been locally issued. When data modifications are performed, those interfering with the continuous query are detected by the update supervisor. The supervisor then generates the *update commands* to be performed on the CRS. Likewise, when *update commands* relevant to the query are received from a queried device, the CRS associated to the query has to be updated according to the transmitted commands. Finally, the querying application can close the continuous query by calling `closeContinuousQuery`: the command is then broadcasted to the neighbourhood before the descriptor to be removed from the descriptors list.

Remotely submitted proximate continuous queries. A device participates in a remote proximate continuous query as soon as it is notified of the existence of

such a query. Two cases have to be considered: a neighbouring device creates a new proximate continuous query or a new device, currently running such queries, enters the vicinity. In both cases, the queried device receives the continuous query to be executed and its descriptor. Note that, in the second case, the queried device receives all the on-going proximate continuous queries of its new neighbour.
The descriptor of a received proximate continuous query is indexed in the local descriptors list. The query is locally executed (on the local RDBMS) and the obtained result is returned to the querying device. Note that if the local execution of the query returns an empty set, no message is sent back.
When data modifications are locally performed on data involved in, at least, one remote proximate continuous query, the update supervisor has to notify the querying device about it. For this purpose, it generates a message containing the suited *update commands* and broadcasts it to its neighbourhood. Finally, a remote proximate continuous query is stopped, and its descriptor removed from the descriptors list, when the querying device either leaves the vicinity or explicitly closes the continuous query by calling `closeContinuousQuery`.

5 Implementation issues

A first prototype of the PERSEND querying system has been implemented. The experimentation platform we used is based on PocketPC PDAs running Windows CE 3.0 and equipped with IEEE 802.11b communication cards. Users' data are accessed by the mean of the Windows ADOCE 3.1 library.

The neighbourhood manager uses a simple discovery protocol, based on UDP sockets, in order to build and maintain the *neighbourhood table*. Devices announce their presence by periodically broadcasting a *Hello* message. When an announcement message is received, its sender is inserted in the local *neighbourhood table* associated with the current time stamp. If the sender is already in the table, the neighbourhood manager simply sets its associated time stamp to the current time. When a fixed period has elapsed since the last annoucement of a neighbour, its entry is removed from the *neighbourhood table*. As devices constituting our platform are equipped with homogeneous communication facilities, we currenlty assume a symetrical discovery (a device seeing a neighbour is also seen by this neighbour).

Each device runs a PERSEND server which uses the ADOCE interface to execute SQL queries. Communications between remote PERSEND servers are also based on UDP sockets. As ADOCE intern features are not available, we have implemented our own query parser. Having no knowledge of the queried databases structures, this parser is not able to associate fields defined in a query to their respective tables. Therefore, we assume users to prefix each declared field with either the name of table it is issued or any defined alias (see Query 4).

Query 4 *Rewriting Query 2 to deal with the query parser's limitations.*

```
continuous vicinity select C.cd_title, C.cd_price
from cd_to_sell C
where C.cd_price is between 10 and 20;
```

Finally, we implement a basic continuous query viewer in order to experiment our system. The viewer enables users to submit all types of queries to the PERSEND server and displays the results of on-going continuous queries. Displayed data are periodically read from opened CRS.

6 Related works

The PERSEND querying system considers the neighbouring devices as the only relevant data sources. So, the physical neighbourhood of a device can be seen as its current context. This notion of context is widely used in the pervasive computing area: pervasive systems aim to provide users with contextual services [7]. Since a few years, some of these studies have focused on neighbouring interactions in proximate environments. Some systems have been specifically designed in order to initiate casual meetings when mobile users meet. Thus, Proxy Lady triggers an alarm when a person within a pre-defined list is physically close enough [8]. When such a meeting occurs, Proxy Lady spontaneously provides the user with documents it has previously specified. Likewise, Proem performs some exchanges of users' profiles in order to initiate such encounters [9]. When a user-defined condition (such as mutual interests, common friends) is met, Proem triggers the action associated to the condition. The Side Surfer prototype was designed to enable spontaneous exchanges of relevant information between mobile users [10]. Based on the keywords used to describe its personal documents, Side Surfer automatically generates each user's profile. During physical encounters, these profiles enable a fast discovery of mutual interests.

Some pervasive studies more particularly deal with data access in proximate environments. Thus, the SPREAD system provides a spatial programming model [11] in which data can only be accessed in the physical space associated to the device which manages them. Data are published by the mean of tuples and are queried using some tuples patterns. By associating a physical space with each device, SPREAD provides a larger model than the one PERSEND considers. However, compared to databases systems, the tuples used to define data only enable to handle basically structured information. Moreover, SPREAD does not deal with data storage issues. MoGATU is another system which aims to enable proximate data accesses [12]. Managed data and submitted queries are defined by the mean of a semantic web language. The MoGATU system only enables to run simple queries, that is, in a database model, queries involving data stored in a single table. Based on the profile of the user, and according to its current context, implicit queries can also be processed. However, and contrary to PERSEND, the MoGATU system allows queries to be routed to non-neighbouring devices. As devices can, by this mean, access non-neighbouring data, the notion of physical neighbourhood is partially lost. Finally, MoGATU does not consider the storage issues.

In the database area, PERSEND is of course close to the studies on continuous queries. Besides works presented in Section 2.1, we can mention the Alert system [13]. Alert aims to build an active RDBMS based on a classical RDBMS. It defines the notion of *active table* which is an append-only table. *Active queries* can be run on active tables: they provide an append-only result set in which new relevant rows are added at the end. Such result sets are read using the *fetch-wait* primitive. This primitive is a blocking read: once the last row of the result set has been returned, the reading process is blocked until a new row is inserted in the result set.

Finally, the Microsoft ADOCE library [14] enables users to open data sets which are dynamically linked to the queried tables. When queried data are issued from a single table, the obtained data set behaves as a continuous result set by reflecting the updates performed on the data source. However, the library enables neither to manage continuous result sets associated to join queries nor to define proximate result sets.

7 Conclusion

In this paper, we presented the design and the implementation of the PERSEND querying system. This system allows applications running in a proximate environment to define and access continuous result sets (CRS). These data sets can involve both local data and data which are stored by current neighbouring devices. The PERSEND system is based on RDBMS and the continuous result sets are expressed using the SQL querying language. We defined, in this scope, new semantics for SQL agregation functions which are suited to proximate environments. We also introduced two new keywords enabling the definition of continuous and proximate queries with SQL. The PERSEND system associates each continuous query with a CRS which can be read by the querying application. Managed CRS are kept up-to-date by supervising the data updates performed on the neighbouring data sources. In order to demonstrate our system, we developped a first prototype and implemented a continuous query viewer application.

During the prototype implementation, additional challenges have arisen. Thus, in order to make result sets easily readable, users consulting the query viewer application may want displayed rows to be sorted according to their time of presence in the data set. For this purpose, a time stamp has to be associated with each row of CRS. Likewise, in order to be aware of the last modifications, applications currently have to periodically scan by themselves the content of the CRS they have opened. Just as for the neighbourhood table, this scheme is not satisfactory: applications can miss important updates and perform some unnecessary readings. We therefore plan to associate CRS with a notification mechanism enabling a querying application to be warned when its CRS is updated. This mechanism can be provided by the mean of a blocking event-based primitive. Each time a CRS is updated, such a primitive could return to the querying process the ID of the updated row and the description of the performed update.

References

[1] J. Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2):117–157, June 1999.

[2] J. Haartsen, M. Naghshineh, J. Inouye, O. Joeressen, and W. Allen. Bluetooth: Vision, Goals, and Architecture. *Mobile Computing and Communications Review*, 2(4):38–45, October 1998.

[3] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries over Append-Only Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 321–330, June 1992.

[4] P. Bonnet, J. Gehrke, and P. Seshadri. Querying the Physical World. *IEEE Personal Communications*, 7(5):10–15, October 2000.

[5] Deborah Estrin, Ramesh Govindan, John S. Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking*, pages 263–270, 1999.

[6] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Quering Moving Objects. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 422–432, April 1997.

[7] G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Department of Computer Science, Dartmouth College, 2000.

[8] Per Dahlberg, Fredrik Ljungberg, and Johan Sanneblad. Proxy Lady: Mobile Support for Opportunistic Interaction. *Scandinavian Journal of Information Systems*, 15, 2000.

[9] G. Kortuem, Z. Segall, and T. G. Cowan Thompson. Close Encounters: Supporting Mobile Collaboration through Interchange of User Profiles. In *Proceedings of the First International Symposium on Handheld and Ubiquitous Computing (HUC'99)*, pages 171–185, September 1999.

[10] D. Touzet, J-M. Menaud, M. Banâtre, P. Couderc, and F. Weis. SIDE Surfer: Enriching Casual Meetings with Spontaneous Information Gathering. *ACM SigArch Computer Architecture Newsletter*, 29(5):76–83, December 2001.

[11] P. Couderc and M. Banâtre. Ambient computing applications: an experience with the spread approach. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS'2003)*, January 2003.

[12] F. Perich, S. Avancha, D. Chakraborty, A. Joshi, and Y. Yesha. Profile Driven Data Management for Pervasive Environments. In *Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA'2002)*, pages 361–370, September 2002.

[13] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)*, pages 469–478, September 1991.

[14] Microsoft ADOCE 3.1 documentation. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/adoce31/html/adowlcm.asp.

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399