Transactional Issues in Sensor Data Management

Levent Gürgen, Claudia Roncancio, Cyril Labbé, Vincent Olive
France Telecom R&D — LSR-IMAG
Grenoble, France — Grenoble, France
{levent.gurgen,vincent.olive}@orange-ft.com
{cyril.labbe, claudia.roncancio}@imag.fr

ABSTRACT

This paper presents a novel research direction in the field of sensor data management. It concerns transactional support in heterogeneous large scale sensor systems. Besides well-known continuous queries on sensor data, system management queries should be supported in these systems. Indeed, with increasing capacity and diversity of sensors, new applications which require complex read-only and update queries are likely to appear. Those applications will require challenging properties such as ACID properties. This paper discusses the relevance of ACID properties in sensor data management context. It then focuses on the isolation property and proposes a concurrency control mechanism to support concurrent execution of continuous queries and update transactions on sensor properties.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems — Distributed applications; H.2.4 [**Database Management**]: Systems — Concurrency, Distributed databases, Query processing, Transaction processing

General Terms

Algorithms, Design

Keywords

sensor networks, sensor stream data, heterogeneity, scalability, transactional data management, concurrency control

1. INTRODUCTION

Traditionally, sensor querying had been made in an application dependant way, i.e. each sensor application had its own sensors, querying scripts, and sensor data processing techniques. However, with the emergence of new generation sensors, and the multitude of "sensor-aided" applications, a need for a more generic, reusable way of sensor querying had risen. *Sensor database systems* [6, 21, 17] attempt to fulfill this need. In fact, the reason of this evolution is similar to that of emergence of database systems a couple of decades ago, i.e. transition from application dependant data files to the application independent databases.

Proceedings of the 3rd International Workshop on Data Management for Sensor Networks (DMSN'06), Seoul, South Korea, 2006

However, these tiny yet intelligent devices came with some challenges addressed to different domains of computer science such as networks, operating systems, and databases. The database community has been investigating new data management techniques [18, 10, 21] such as *continuous queries*, *in-network aggregation*, *approximate query answers*, and *resource sharing*. However, in spite of the fact that reliable transaction processing is the core functionality of a database management system (DBMS), transactional aspects are not explored for sensor database systems.

There are two main reasons why transactional data management has not been studied in the sensor database context. The first one is that, limited sensor resources (e.g. energy, computing, communication, storage) prevent them from implementing complex protocols for reliable data processing. And secondly, existing sensor systems were conceived for read-only sensor data querying purposes. System management queries were not considered in those systems. Therefore there was no need for an explicit control mechanism in order to guarantee sensor data consistency. However, with increasing capacity and decreasing cost of sensors, we can therefore ask the following question: would sensors which henceforth are involved in issues such as self-organizing routing protocols, continuous query processing, and in-network aggregation, also involve in transactional sensor data management? In addition, various types of sensors (e.g. temperature, pressure, and humidity sensors; current and voltage readers; GPS devices; visual and auditory sensors; RFID readers; chemical sensors) have already started to contribute to the anywhere anytime information processing paradigm. Thus, large scale heterogeneous sensor systems have become a reality. These systems would include, besides continuous queries on sensor data, system management queries such as: update all sensors in a particular location with a new sampling rate; which sensors have the energy level lower than 10%; modify measuring unit from Celcius to Fahrenheit. These queries may require some transactional properties such as atomicity (update all sensors or none) or isolation (isolate continuous read-only queries from update queries). Therefore, can we talk about a certain *sensor database management system* which guarantees ACID properties as conventional DBMSs do? We believe that, a revision of classical transactional schemes is required for sensor transactions in order to propose new schemes addressing to unique characteristics of sensor applications; as it has been done for new generation information systems (e.g. mobile transactions [24]).

This paper aims to expand the open questions mentioned above, and tries to find some answers. It particularly deals with concurrency control problem which arises with coexistence of update transactions and continuous queries. Firstly, section 2 gives our vision of large scale sensor database systems, and introduces different sensor transactions that can exist in these systems. The relevance of traditional ACID properties for sensors context is also given in that section. Section 3 proposes a temporally nested transactional model to represent continuous queries in a finer way, and then proposes a concurrency control mechanism based on this model. Finally, section 4 concludes and gives directions for future work.

2. TRANSACTIONS IN SENSOR DATABASES

Before introducing sensor transactions, let us first give our sensor database vision and an application scenario.

2.1 Sensor Databases

We can classify existing sensor database solutions into three categories according to the place where query processing takes place: distributed [4, 21, 6, 17], centralized [10, 5, 2, 7], and hybrid systems [20, 11, 3, 9]. We adopt the distributed hybrid approach presented in [11] which is an integrated view of sensor networks and data stream management systems (DSMS). Our architecture is composed of three main levels: control sites, gateways and sensors (see Figure 1). A control site receives queries from users or applications, decomposes the query, and sends the sub-queries to the concerned gateways. Gateways are distributed according to their location. They group different kinds of sensors, more precisely their proxies. A proxy is the software controlling one or more sensors. There is also one adapter per proxy which is the interface between the sensor specific proxy and our sensor management system. Sensors are physically distributed in an environment and send their measures to their proxies in a periodic or aperiodic manner. Sensors may have some query processing and storage capabilities. Different parts of the queries can be evaluated at control sites, gateways, proxies, or sensors. The hybrid approach aims, firstly to be scalable by distributing query evaluation among different levels of the architecture, and secondly to integrate heterogeneous sensor data.

As in conventional relational database systems, sensor data is represented by tuples which conform to a data schema. Queries are formulated according to that schema. Mostly, queries pertain to three parts of sensor data: meta-information of sensor (identification, location, type, unit of measure, etc.), sensor's measurement (temperature, pressure, GPS coordinates, RFID tag Id, etc.), and timestamp of the measurement. Continuous query operators execute on sensor measurement (e.g. sensors **measuring less than 10**). However, in order to localize sensors whose data will be interrogated, a part of the query is executed on the sensor meta-information (e.g. **temperature** sensors **in room A** measuring less than 10 **Celsius**). Finally, time is also concerned by most of the queries (e.g. temperature sensors in room A measuring in average less than 10 Celsius **in a sliding window of 5 minutes**). Hence, we differentiate three types of sensor data attributes: properties, measurement and timestamp.

2.2 Application Scenario

Consider a factory equipped with two kinds of sensors, temperature sensors and RFID readers. In the factory, each product passes by a certain number of sections during its lifecycle. Each section has a gateway responsible for the sensors present at that section. At every section there are, several sensors which measure temperature of the section, and one RFID reader detecting product tags (see Figure 1).

Sensor data is distributed at different levels of the architecture (see Figure 2). Meta-information is distributed on control sites, gateways, and proxies (in *gateways*, *proxies*, and *sensors* table). Sensor stream data is represented by a virtual table called *measures*. Queries are formulated according to a common global schema. A simple schema example could be as $sensor\_stream = < sensorId, location, type, rate, unit, measurement, timestamp >$. This schema actually represents a view over different materialized databases and non-materialized data stream of sensors:
$sensor\_stream = \pi_{list-attr}(gateways \bowtie_{GId} proxies \bowtie_{PId} sensors \bowtie_{sensorId} measures)$
where $list - attr =< sensorId, location, type, rate, unit, measurement, timestamp >$

The first five attributes form the *properties* of sensor data. The *measurement* field represents the measurement made by the sensor at the time indicated by the *timestamp* attribute.

2.3 Sensor Transactions

Generally speaking, we can define a sensor transaction as a set of operations performed on a sensor database. We will differentiate three types of sensor transactions:

One-time query transactions. These are *one-time* queries in the sense that the query will be executed on the current state of sensor data. Mostly, these queries will be evaluated on the sensor properties and not on the sensor *measurements*. Typical queries include: Where are the temperature sensors? Are there any sensors with memory full? How many RFID readers are there in building A? Which sensors measure with the Celsius unit? Etc.

Update transactions. These transactions modify sensor "*property*" attributes: set the name of the temperature sensor having the Id number "1", to the value "temp_sensor"; change the location attribute value "Section A" of the gateways to the new value "Section I"; double the sampling rate of temperature sensors in section B; update firmware of the sensors; etc. Update transactions also include arrival/departure of gateways, proxies, or sensors (i.e. insert/delete queries).

Continuous query transactions. These are the continuous queries over sensor stream data. Such queries involve reading *measurements* of sensors. They include filters, sliding window aggregates, joins, etc. Mostly, they succeed one-time queries which are used to localize the sensors to query. For instance, consider a continuous query asking the average **temperature** of the **section A** every 5 seconds. Firstly, the gateway of the section A, and then the proxy of temperature sensors are localized by a *one-time query*. Next, data stream of the chosen sensors is initiated and finally continuous query can be evaluated, every 5 seconds, on the sensor stream data.

As in traditional database systems, we argue that executing queries or update operations in transactional mode could lead us to keep a consistent state of the sensor information system. However, due to the unique characteristics of sensor systems, additional challenges are likely to occur. These points are discussed in the next section.

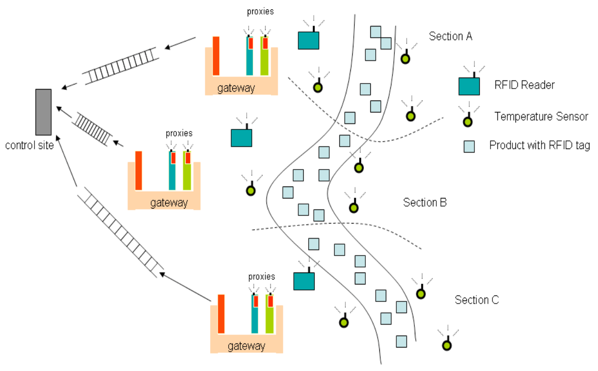


Figure 1: Architecture and application scenario

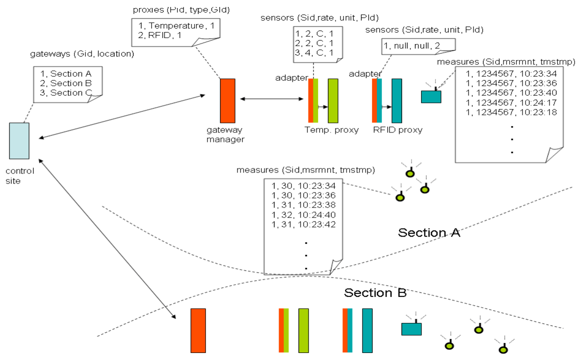


Figure 2: Schema examples

2.4 ACID properties for sensor transactions

This section discusses relevance of conventional ACID properties in the sensor database context. This is a first step of a work including more detailed analysis of the problems that could be encountered while dealing with sensor transactions.

2.4.1 Atomicity

Challenges: Consider the case where a transaction wants to update the sampling rate of all temperature sensors in the factory. The atomicity property implies that, either all sensors are updated, or none. However, as sensors are mostly prone to the failures, it is strongly probable that several sensors among hundreds can not successfully complete the operation. What to do in this case? If we abort the entire transaction, the successful changes made on the sensors should be undone and this can cause a performance degradation of the system. On the other hand, if we tolerate the violation of the atomicity property, and let some sensors continue with their old sampling rate, this can cause some incoherence of the measures made by the sensors with different sampling rates.

Possible solutions: As we can notice in the example, a flexible transaction management is necessary for sensor transactions. Such flexibility is proposed by nested transaction models [8], in which a transaction can split into several sub-transactions that can execute relatively independent one from the other. Therefore, commit decisions on a set of sub-transactions can be made independently by the coordinators at different levels (e.g. gateways, proxies). If we take back the updating example, we can imagine a sub-coordinator at each gateway which is responsible for partial commit decisions. With the assumption that sensors at different sections can sample with different rates, the global transaction initiated on the control site can commit even if one of the sub-coordinators does not commit. Relaxing atomicity can lead us to find a trade-off between system performance and data incoherence.

In conclusion, we argue that due to the large number of sensors and unreliable nature of sensor systems, a commit protocol adapted to sensors context (e.g. few message exchanges), on top of a flexible transaction model should be chosen for sensor transactions.

2.4.2 Consistency

Challenges: In sensors context, in addition to the *logical* consistency, *temporal* consistency may also need to be maintained. Temporal consistency ensures that sensor data indeed reflects the current state of the environment (*absolute temporal consistency*). In addition, a *relative temporal consistency* may be required. It concerns maintaining a temporal consistency "between" the measures made at quasi-same instant by different sensors, in order to guarantee the accuracy of query results concerning several sensors.

Possible solutions: Real-time database systems (RTDBS) have made a significant research on temporal consistency [23]. In these systems, transactions have temporal constraints to terminate. Hence, blocking protocols are avoided. Priority-based protocols are chosen instead to let most critical transactions terminate earlier. Although most sensor database systems are not considered as much "real-time" as RTDBSs, time-cognizant protocols should be used in order to reflect the temporal dimension of sensor data [12].

In conclusion, temporal consistency, just as logical consistency, may be required for sensor databases. Results issued from RTDBS community will certainly be helpful in answering temporal requirements of sensor transactions.

2.4.3 Isolation

Challenges: Isolation guarantees the correctness of data even in presence of concurrent access to the resources. In order to illustrate the necessity of a concurrency control mechanism, consider, in the factory example, a continuous query which asks for the average temperature in Celsius unit for each section. The result of this query carries a significant importance, because if the average passes a certain threshold, an alarm will be triggered. During the execution of the query, consider an update transaction which modifies the measuring unit of sensors in Section A to Fahrenheit. If there's no concurrency control mechanism in the system, the modification will be done and therefore the calculated average temperature would be wrong, which may eventually cause a false alarm.

Possible solutions: Mainly, there are two families of concurrency control protocols: pessimistic and optimistic protocols [22]. Pessimistic protocols detect conflicts before executing the transactions and conflicts are resolved by locking the resources. Clearly, this is not an optimal solution for sensor transactions as continuous queries can last during long and eventually undetermined time. On the other hand, in optimistic protocols transactions are allowed to progress until a validation phase where conflicts are detected. If a conflict is detected, conflicting transactions are restarted according to a validation scheme. However, optimistic protocols can lead useless redundant restarts of update transactions when they are in conflict with *continuous* queries. Hence, a finer view of continuous queries is needed in order to increase the concurrency with the update transactions (see section 3.1).

In conclusion, modifications over sensor properties can lead inconsistent data reads for the concurrent transactions (e.g. continuous queries). A concurrency control mechanism taking into account sensor specificities (continuous queries, real-time nature, limited resources, etc.) should be used (see section 3.2).

2.4.4 Durability

Challenges: Durability property is maintained by commit protocols including recovery and termination procedures. As failures are likely to be occurred in sensor systems, these protocols gain more importance. Consider an update transaction modifying the sampling rates of sensors of a particular section. When the transaction arrives to the proxy, it sends the update command to its sensors. The update message addressed to the sensor or the sensor's action acknowledgement (or rejection) message to the proxy can be lost resulting from the unreliable wireless connectivity. In addition, some sensors may fail or be blocked during a certain time after receiving the transaction.

Possible solutions: Due to the fast response requirements of sensor systems, the termination protocol implemented on the proxy should avoid long-time blocking. Besides, for recovery purposes, sensors may need to provide a *logging* facility which is used to undo (or redo) the transaction. However, this additional log information can cause overhead on the storage of the sensor as its storage capacity is limited. In addition, message exchanges required by the protocol introduce communication overhead, therefore a waste of energy, precious resource for sensors. On the other hand, in some cases the proxy of a sensor may execute the transaction on behalf of the sensor. This happens when the sensor would not necessarily be aware of the modifications made on some of its meta-information such as its name or location attribute. In these cases the recovery protocol can be implemented on the proxies.

In conclusion, the balance between little storage requirement, few message exchanges, and high concurrency should be found for recovery purposes of sensor transactions. "Lightweight" logging and rollback mechanisms should be chosen for sensor transactions.

3. CONCURRENCY CONTROL

This section focuses on the isolation property. It firstly gives a temporally nested model for continuous queries, and then introduces a concurrency control mechanism based on this model to support coexistence of continuous queries and update transactions.

3.1 Temporally Nested Model for Continuous Queries

Nested transactions provide the flexibility required for sensor transactions in particular for update and one-time query transactions. Nevertheless for continuous queries, an even finer model will be useful to support the coexistence of update transactions and continuous queries on the *same site*. As mentioned earlier, continuous queries have two parts: one-time and continuous. One-time part localizes the sensors whose data will be used to evaluate the continuous part of the query. Continuous part concerns timely (periodic or aperiodic) execution of a set of operations on sensor stream data. We model each execution of these operators as a single tiny transaction. These transactions represent temporally nested (TN) sub-transactions of the continuous query:

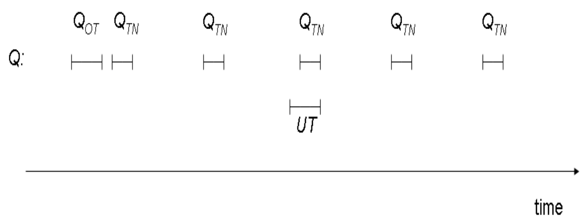
DEFINITION 1. *Let Q be a continuous query transaction to be executed at one particular site (e.g. control site, gateway, proxy), then the temporally nested sub-transaction Q_i^t represents the i^{th} execution of the continuous part of Q at instant t, where t belongs to a discrete time domain T.*

These tiny transactions are executed on sensor stream data, mostly on the *measurement* and *timestamp* fields. However, they inherit the read-set of the one-time part of the query. Modifications by update transactions on the attributes contained in this read-set causes conflicts. To illustrate, consider a continuous query transaction asking "the measures of **all temperature** sensors measuring **in Celsius**". Its read-set contains the *type* and *unit* attribute. If another transaction modifies the sampling unit of sensors of "Section A" from Celsius to Fahrenheit (i.e. its write-set contains the *unit* attribute), there will be a read/write conflict. Sensors in "Section A" are no more in the scope of the continuous query. Or, if a new temperature sensor appears in the system, it should also be included in the set of sensors being queried. Therefore, one-time part of the query should be re-executed in order to rediscover the sensors concerned by the query, and then the continuous part should be retaken.

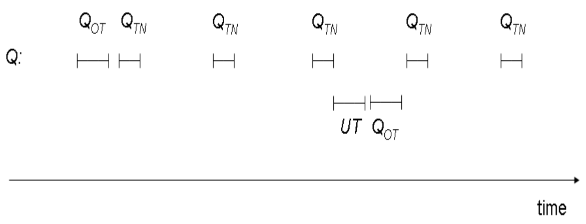
Our objective by defining temporally nested model is to be able to execute conflicting update transactions "during" continuous queries, with as less as possible interruption to the continuity of queries. For instance, let t_1 and t_2 be time instants of two consecutive execution of the continuous part of a query Q, Δt_{UT} and Δt_{OT} be, respectively, durations of an update transaction and the one-time part of Q, then if $\Delta t_{UT} + \Delta t_{OT} \leq t_2 - t_1$, we can execute the update transaction in a transparent way to the continuous query user (see Figure 3). However, we don't always have the necessary information to realize this (e.g. aperiodic queries). For these cases, next section proposes a priority-based optimistic concurrency control mechanism.

3.2 Concurrency Control Mechanism

SDBSs and RTDBSs [23] have similarities in that, both have to deal with temporal issues. As SDBSs are mostly conceived for monitoring sensor data in "quasi-real time", timely deliverance of sensor data gains importance. We believe that concurrency control protocols proposed for RTDBSs can certainly have reusable aspects for transactional sensor data management. In RTDBSs, optimistic protocols are mostly preferred against blocking protocols in order to deal with temporal constraints [16, 25, 15]. Similarly, we adopt the optimistic approach [19] for concurrency control. The transactions have three phases: work, validate, and commit. During the work phase, transactions read data, perform operations, and pre-write the result to a local variable. In the validate phase, conflicts are detected. If there is no conflict, in the commit phase the results are made permanent. Otherwise, the conflict resolution policy is applied. For instance, the work phase for a TN transaction consists



(a) Q conflicted with a UT



(b) UT is executed between two consecutive Q_{TN}

Figure 3: Q: A continuous query transaction, Q_{OT}: One-time part of Q, Q_{TN}: Temporally Nested sub-transactions of Q, UT: A conflicting update transaction.

of dequeueing the tuple(s) from the input stream, performing operations, and writing the result to a local variable. In the validate phase, conflicts with update transactions are checked. Finally, commit phase consists of enqeueuing the result to the output stream.

3.2.1 Conflict Detection

Optimistic protocols differ in two ways according to their conflict detection scheme: backward and forward validation scheme [13]. In backward validation, conflicts are checked against already committed transactions. At the validation phase, a read transaction checks if it has read data modified by a write transaction before the commit of this latter. If this is the case, the read transaction is aborted and restarted. In forward validation, conflicts are checked against currently executing transactions. During the validation of a write transaction, write set of the transaction is compared with the read set of currently active transactions. A non-empty intersection of these sets implies a conflict which is resolved by aborting and restarting one or more transactions. Forward validation scheme is chosen by most RTDBSs due to its early conflict detection property and flexibility of choosing the transaction to abort in case of conflicts [15, 14, 16]. This flexibility also provides the possibility of applying priority-based conflict resolution schemes [14]. For the same reasons our concurrency control mechanism uses a forward validation scheme. We deal with *read/write* conflicts. Write/write conflicts don't occur as we assure that there is only one transaction in *validate* phase; and *validate* and *commit* phases occur in one critical section.

According to the forward validation scheme, the conflicts will be detected during the validate phase of **update transactions** (UT). An UT firstly checks if there is an already validating transaction[1]. If it is the case, UT is enqueued to a waiting queue where it will wait until the end of the validating transaction. When its turn comes to validate, UT checks if it is conflicted (read/write) by another earlier updating transaction while its waiting period. If it is the case, conflict resolution decision (discussed in the next section) will be applied. Otherwise, it checks, if among the currently active transactions (e.g. TN transactions in *work* phase), there are some that conflict with it. Its write set is compared with the read sets of active transactions. If they intersect, the conflicting transactions are marked as conflicted by the UT. If there is no conflict, UT can continue with its commit phase. At the end of the commit phase, UT's commit result is notified to the waiting queue[2]. The completed transaction is then removed from the *active transactions* list. Note that, entering to the commit phase does not necessarily signify that transaction will commit successfully. For instance, due to sensor related problems (communication latency, unavailability, etc.) transaction can decide to abort after a certain number of retries of failed operations on the sensor.

Similarly, query transactions (OT or TN transactions), in their validate phase, check if there is an already validating transaction. If there is, the transaction waits. When its validation turn comes, it checks if it is marked as conflicted by an updating transaction. If it is, the conflict resolution decision is applied. Otherwise it passes to the commit phase.

3.2.2 Conflict resolution

As mentioned earlier, our main objective for defining TN transactions is to be able to "insert" the update transaction between two consecutive executions of TN transactions (see Figure 3). However, if this is not possible, either the update transaction must wait until the termination of the continuous query, or the update transaction will delay the continuous query. In order to deal with these cases, we will adopt a priority-based approach. Priorities can be assigned to transactions by users or by system. For instance, higher priorities could be assigned to update transactions whose operations should imperatively be executed. On the other hand, users can assign higher priorities to continuous queries if they are considered as "not interruptible".

Our conflict resolution mechanism is a variant of OPT-SACRIFICE and OPT-WAIT [14]. Validating update transaction UT, checks if there is at least one conflicted high priority (CHP) transaction. If it is the case, then we adopt a policy similar to OPT-SACRIFICE. UT is immediately aborted. However, it is not immediately restarted (as it would be with OPT-SACRIFICE). This is because eventual conflicting continuous queries would cause useless redundant restarts of UT. Thus, instead, its restart is scheduled for after the completion of the continuous query transaction having the longest execution time with the highest priority. Usually, lifetime of continuous queries are defined by the queries. This can inform the UT about how long it should wait. If such information is not provided, then it is restarted once there is some completed CHP continuous query. If there are no conflicting continuous queries, then it is restarted immediately.

Although this can be seen as a conventional locking based approach, this is made in the worst case, thus conforms to

[1]There is only one transaction in validate and commit phase

[2]Note that, a transaction, until it terminates its commit phase, is considered as "validating".

the philosophy of optimistic mechanisms. However, note that this later scheduling doesn't guarantee that the transaction will commit, as at the moment of restart there can be more conflicting transactions in the system with higher priority. Besides, its parent transaction can decide to abort (e.g. due to expiration of a timer), therefore cancel the later scheduling. Another solution for the starvation problem could be to increment the priority of the transaction at each restart in order to ensure its execution.

In the case that validating transaction is decided to be committed (there are not any CHP transaction), the measure taken is based on a waiting approach similar to the OPT-WAIT policy. However, according to our policy, instead of the validating transaction, the conflicting transactions will wait until the validating transaction terminates. If it is successfully committed, then waiting transactions are notified, aborted and restarted. If it can not commit successfully, then the conflicting transactions in the waiting queue is notified (they are not anymore conflicted), therefore the next transaction can pass to its *validate* phase.

4. CONCLUSION AND PERSPECTIVES

This paper discussed transactional issues of sensor data management. These issues not yet explored gain importance with the emergence of large scale heterogeneous sensor systems. We believe that revision of classical transactional issues, already done for mobile transactions, will also need to be done for sensor transactions. This paper particularly focused on the isolation property. Concurrent execution of update transactions and read-only continuous queries may cause conflicts. Continuous queries should be handled in a finer way for a more efficient conflict resolution. We proposed a temporally nested transaction model to represent continuous query transactions, and introduced a priority-based optimistic concurrency control mechanism inspired by solutions from RTDBS domain.

We have developed a sensor querying prototype based on a service-oriented approach for the PISE project [1]. The aim of this project is to monitor electric power materials in real-time. Indeed, in this project some need of transactional properties appeared. The implementation of transaction management services, as well as the analysis of our proposal under different scenarios is our future work.

5. REFERENCES

[1] PISE Project, http://www.telecom.gouv.fr/rnrt/rnrt/projets/PISE.htm.

[2] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 2003.

[3] D. Abadi, W. Lindner, S. Madden, and J. Schuler. An integration framework for sensor networks and data stream management systems. In *VLDB*, 2004.

[4] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4), 2002.

[5] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[6] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. *Lecture Notes in Computer Science*, 2001.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[8] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[9] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An architecture for a world-wide sensorweb. *IEEE Pervasive Computing*, 2003.

[10] L. Golab and M. T. Ozsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[11] L. Gurgen, C. Labbé, V. Olive, and C. Roncancio. A scalable architecture for heterogeneous sensor management. In *MDDS'05, DEXA Workshops*, pages 1108–1112, Denmark, 2005.

[12] L. Gurgen, C. Labbé, V. Olive, and C. Roncancio. SStreaM: A model for representing sensor data and sensor queries. In *International Conference on Intelligent Systems And Computing: Theory And Applications (ISYC)*, Cyprus, 2006.

[13] T. Harder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, 1984.

[14] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *IEEE Real-Time Systems Symposium*, 1990.

[15] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *PODS '90*, pages 331–343, NY, USA, 1990.

[16] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB'91.*

[17] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, 2003.

[18] N. Koudas and D. Srivastava. Data stream query processing. In *ICDE*, page 1145, 2005.

[19] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. In *VLDB*, 1979.

[20] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[21] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[22] D. A. Menascé and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Inf. Syst.*, 7(1), 1982.

[23] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[24] P. Serrano-Alvarado, C. Roncancio, and M. Adiba. A survey of mobile transactions. *Distrib. Parallel Databases*, 16(2):193–230, 2004.

[25] X. C. Song and J. W. S. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, 1995.