Ryu W, Hong B, Kwon J *et al.* A reprocessing model for complete execution of RFID access operations on tag memory. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 27(1): 213–224 Jan. 2012. DOI 10.1007/s11390-012-1218-z

A Reprocessing Model for Complete Execution of RFID Access Operations on Tag Memory

Wooseok Ryu[1], Bonghee Hong[1,*], *Member*, *ACM*, *IEEE*, Joonho Kwon[2] and Ge Yu[3] (于 戈), *Senior Member*, *CCF*, *Member*, *ACM*, *IEEE*

[1] *Department of Computer Engineering, Pusan National University, Busan 609-735, Korea*

[2] *Institute of Logistics Information Technology, Pusan National University, Busan 609-735, Korea*

[3] *School of Information Science and Engineering, Northeastern University, Shenyang 110004, China*

E-mail: {wsryu, bhhong, jhkwon}@pusan.ac.kr; yuge@ise.neu.edu.cn

Received April 21, 2010; revised July 18, 2011.

Abstract This paper investigates the problem of inconsistent states of radio frequency identification (RFID) tag data caused by incomplete execution of read/write operations during access to RFID tag memory. Passive RFID tags require RF communication to access memory data. This study is motivated by the volatility of RF communication, where instability is caused by intermittent connections and uncertain communication. If a given tag disappears from the communication area of the reader during the reading or writing of tag data, the operation is incomplete, resulting in an inconsistent state of tag data. To avoid this inconsistency, it is necessary to ensure that any operations on tag memory are completed. In this paper, we propose an asynchronous reprocessing model for finalizing any incomplete execution of read/write operations to remove inconsistent states. The basic idea is to resume incomplete operations autonomously by detecting a tag's re-observation from any reader. To achieve this, we present a concurrency control mechanism based on continuous query processing that enables the suspended tag operations to be re-executed. The performance study shows that our model improves the number of successful operations considerably in addition to suppressing inconsistent data access completely.

Keywords asynchronous reprocessing, concurrency control, continuous query, RFID, tag access operation

1 Introduction

As radio frequency identification (RFID) technology progresses, low-cost passive RFID tags can store detailed production states in their memory as well as identification information. By storing the ongoing product state of the manufacturing process in the passive RFID tag memory, we can efficiently access the status of tagged items in the field without connecting to an information system[1]. Utilization of tag memory has been becoming widespread in a variety of industries, including supply chain management, asset management, and manufacturing process management[2].

Accessing tag memory requires wireless communication via RF signals from the RFID reader. Since a passive tag does not have internal power, its communication range is limited to less than 10 meters[3]. Moreover, it cannot guarantee 100% accuracy in tag memory even within range, due to interference from various sources such as tag orientation, packing materials, and other obstacles[4-5]. Such characteristics of RF communication have a bad influence on the correct execution of read/write operations. Usually, tags are attached to the product, and are frequently subject to movement out of the range of readers during access to tag memory data. Moreover, a reader can occasionally fail to receive a reply message after sending access command messages because of unreliable RF communication. As a result, the tag data become inconsistent.

To preserve the consistency and correctness of tag data, it is necessary to ensure that any tag operation on tag memory is completely executed. For example, when production information is not completely written to a tag in the assembly line, the operation should be completed the next time the tag appears to any reader. This leads to the concept of an RF transaction that provides atomicity and durability in RFID tag operations. We define an RF transaction as a set of RFID tag operations that retrieve or store data on a tag memory. The goal of RF transaction processing is to guarantee

Regular Paper

This work was supported by the Grant of the Regional Core Research Program/Institute of Logistics Information Technology of Korean Ministry of Education, Science and Technology.

*Corresponding Author

complete execution of tag operations as well as to preserve the consistency of tag data.

Our approach to process RF transactions is to suspend the execution of incomplete operations and to resume them when the tags are re-observed. In this paper, we propose an asynchronous reprocessing model that defines the unprocessed parts of incomplete operations as reprocessing operations. The proposed model guarantees the concurrent execution of the reprocessing operations by using a continuous query scheme. This enables an asynchronous resumption to reprocess the suspended operations when the tag's re-observation is detected by any arbitrary reader.

The main contributions of our work are as follows.

• *Asynchronous Reprocessing Model.* We provide atomicity and durability for tag data reads/writes by internally reprocessing incomplete operations at the middleware level.

• *Proofs for Correctness.* We prove the completeness of the execution via formalization of the model. Our model ensures the complete execution of access operations and always guarantees consistent data access to passive RFID tags, without requiring any consideration of the RF characteristics.

• *Experimentation to Validate Our Proposed Approach.* Through experiments using real data as well as simulated data, we show that our asynchronous reprocessing model ensures completeness in the processing of access operations and also gives high usability of tag data accesses.

The remainder of this paper is organized as follows. Section 2 discusses the accessing mechanism of tag memory data by using an example. We specify our work by analyzing the characteristics of RFID tag access in Section 3. In Section 4, we present an asynchronous reprocessing model and propose a concurrency control mechanism to guarantee correct and consistent data accesses. In Section 5, the efficiency of the proposed model is discussed using experimental studies. Section 6 surveys related work of this paper. Finally, a summary is presented in Section 7.

2 Backgrounds

2.1 RFID Applications

Let us consider an example of RFID applications to clarify the problem of data access on RFID tag memory. There are many industries, such as automotive company, pharmaceutical company, and other manufacturing companies, that utilize tag memory for information accesses[2]. In this paper, we assume a simplified process management system which manages production of automotive components.

A passive RFID tag is attached to each automotive component to maintain production information of the component. The information includes product specifications, production time, person in charge, and assembly line ID to maintain production status of the component. It is recorded or retrieved during the journey of the component through the processes. If a certain component is determined to be defective, the manufacturer can trace the processes and take an immediate action by accessing information in the tag memory.

Let us discuss the example shown in Fig.1. RFID readers are installed at each line and gate to access production information during the production. When a tagged item moves through the lines, the readers $R2$ and $R4$ identify the tag and write manufacturing information, such as line ID, at the end of the line. After then, the components are stored in the warehouse temporarily, and are transported to customers. When the tag passes $R5$ and $R6$, the readers can write production time and release time to the tag, respectively. Although only two assembly lines are illustrated in Fig.1, we can assume there are numerous lines and some items require a series of lines for the production.

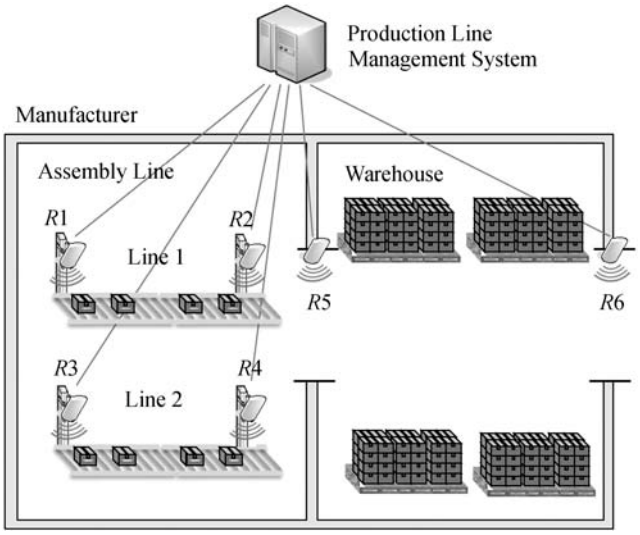

Fig.1. Process management system as an example of RFID applications.

2.2 Accessing Mechanism of Tag Memory Data

To discuss the access mechanism of tag memory data, we need to consider several existing standard specifications proposed by EPCglobal which provides a full set of industry-driven standards for the use of RFID. The standards include the air interface protocol for defining a communication interface between reader and tag[6], and the low-level reader protocol (LLRP) between reader and middleware[7] as well as

RFID middleware (ALE)[8].

Let us discuss the accessing mechanism using the standards. We assume that an example application wants to write a value "LINE1" on tags that pass through Line 1. Once the middleware receives the request from the application, the middleware uses two steps to write data on tag memory as shown in Fig.2. Fig.2(a) shows the first step which is to identify nearby tags in RF field of the reader $R2$. This is done by the reader such that the reader broadcasts inventory operations and generates an inventoried tag list[6-7].

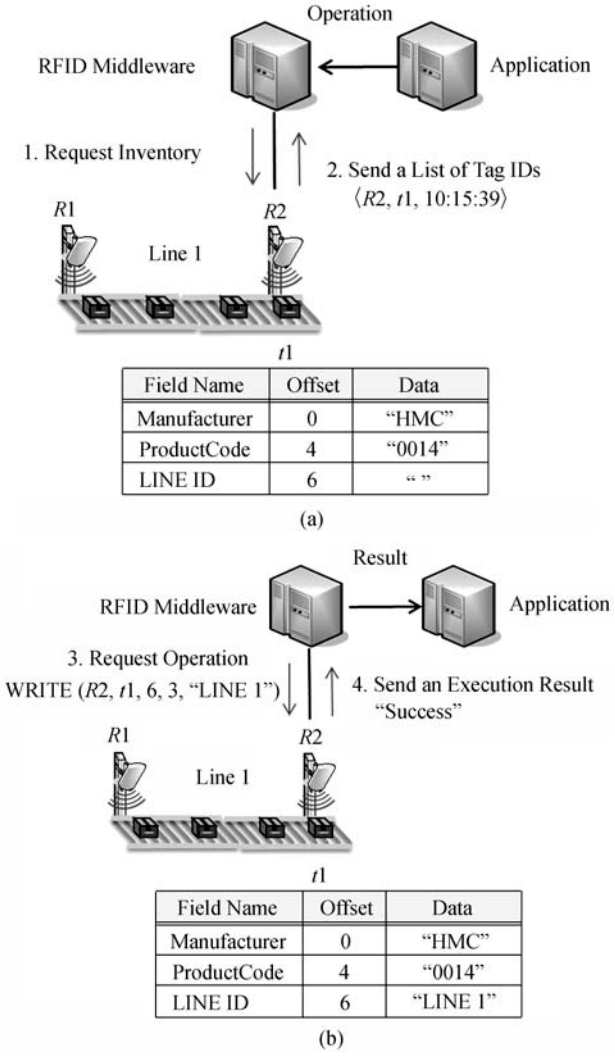

Fig.2. Two-step mechanism for accessing tag memory. (a) Inventory step to obtain tag identifiers. (b) Access step to execute an operation.

If a tag is in the inventory list, we can determine that the tag is accessible via the reader $R2$. Then, the access step follows by sending an access operation to the reader which communicates with the tag using several air commands (see Fig.2(b)). In the figure, "LINE1" is written to $t1$. A reply to the command from the tag is finally returned to the application via the middleware.

3 Problem Analysis

In this section, we first define a formal data access model based on previous section. Then, we discuss the problem data access by analyzing results of the operation execution. Several approaches are also discussed later in this section.

3.1 Tag Data Access Model

From the database point of view, we can consider each data stored in the tag memory as a data item in database. As data item has a value that represents a specific production status, persistency of the data should be preserved. In this paper, we define an RFID database as a collection of tag data items that are spread over dispersed tags.

Tag data items can only be accessible when the tag is identified. To represent an identification of a tag to a reader, we define a tag identification event e_i as a tuple of three attributes $\langle rid, tid, t\rangle$, where $rid \in D_R$ is the identifier of a reader that observes a tag's identification, $tid \in D_T$ is the observed tag's identifier, and t is a timestamp. D_R is the domain of reader identifiers and D_T is the domain of tag identifiers, respectively. As tag events are continuously inventoried at each reader, we can model a tag event stream $ES = (e_1, e_2, e_3, \ldots)$ as a continuous stream of tag identification events generated by the connected readers. Here, ES represents the time-serialized tag events from all of the readers to the middleware.

A tag operation is an access operation, such as read or write, on a specified tag that is to be identified by a specified reader. After a tag operation is requested by the application, the operation is executed when the tag is observed by the specified reader. Definition 1 gives the formal definition of an RFID tag operation based on ALE [8]. Definition 2 gives the formal representation of an execution of a tag operation on a tag event in ES.

Definition 1 (Tag Operation). *A tag operation o_i is defined as a tuple of five attributes ⟨rid, tid, offset, size, data⟩, where $rid \in D_R$ is the identifier of the reader that observes the identification of the tag, $tid \in D_T$ is the identifier of the target tag to perform the operation, offset is the location of the memory area, size is the length of the data item being read/written, and data is the target data to be written.*

Definition 2 (Operation Execution). *Let $Ex(o_i)$ denote an execution of a tag operation o_i. $Ex(o_i)$ is defined as a mapping function $M(o_i, e_j)$ where e_j is a tag identification event and $o_i.rid = e_j.rid$ and $o_i.tid = e_j.tid$.*

Using the above definitions, the accessing mechanism of tag memory data in Fig.2 can be re-illustrated as shown in Fig.3. In the figure, tag identifications from readers are represented using *ES*. Then, executions of tag operations can be represented as a mapping between a tag event and a corresponding tag operation.

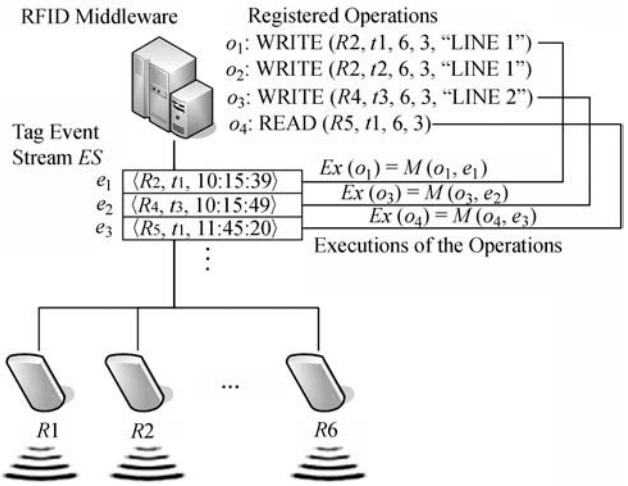

Fig.3. Tag data access model.

Let us discuss an execution result of the tag operation. Definition 3 describes an execution result of the operation based on LLRP[7]. The code attribute of $Ex(o_i)$, denoted by $Ex(o_i).code$, specifies the error condition of the execution of o_i. Table 1 shows the detailed description of the code parameter. The *Failed* error is caused mainly by unreliable communications and, therefore, the operation can be processed next time. The *No Response* error is caused mainly by collisions, surrounding obstacles, and so on. Here, we assume that the code does not include any logical error such as memory overrun or insufficient access privileges, it means that the application always requests logically correct operations.

Table 1. Classification of the Code Parameters of the Execution Result

Error Code	LLRP Result	Description
Successful	Success	The operation has been processed successfully
Failed	Insufficient Power	The operation has not been processed successfully because of environmental reasons
No Response	No response from tag	The reader has failed to receive a reply to a command from a tag

Definition 3 (Result of Execution). *A result of $Ex(o_i)$ is defined as a tuple of three parameters ⟨nwp, code, data⟩, where nwp is the number of words that are read or written, code ∈ {Successful, Failed, No Response} is the result code of the execution, and data is the written or retrieved data following the execution of o_i.*

3.2 Problems in Processing Tag Operations

According to Definition 3, we classify the execution result of a tag operation into four cases, namely *Success*, *Failed*, *Partially Processed*, and *Unsure*, as shown in Tables 2 and 3. The execution of an operation is called incomplete if the result is either *Partially Processed* or *Unsure*. *Partially Processed* means that the tag operation is not fully executed. *Unsure* can also cause incomplete tag operation execution because the reader failed to get any response from the tag in reply to the operation request. If the tag operation is a write operation, it is possible that the tag memory has been written or is still unchanged.

Table 2. Classification of the Results of an n-Word Read Operation

Code	nwp		
	0	$< n$	n
Successful	N/A	N/A	Success
Failed	Failed	Partially Processed	N/A
No Response	Failed	Partially Processed	N/A

Table 3. Classification of the Results of an n-Word Write Operation

Code	nwp		
	0	$< n$	n
Successful	N/A	N/A	Success
Failed	Failed	Partially Processed	N/A
No Response	Unsure	Partially Processed	N/A

For example, let us consider the operation o_1 in Fig.3. The data to be written to $t1$ is composed of three words, "LINE1". Assume that the result of $Ex(o_1)$ (wants to write data in tag $t1$) is ⟨1, "No Response", "LI"⟩. This implies that only one word "LI" is actually written, and we cannot judge whether the remaining words "NE1" are written or not. If "NE1" are not actually written, $Ex(o_4)$ (wants to read the data in $t1$) will return semantically wrong information. This causes inconsistency of tag data item, and therefore, incompletely executed operation should be reprocessed.

3.3 Approaches for Handling Inconsistency

For handling inconsistency, there are two possible ways: simple reprocessing model (SR model) and Locking model (LK model).

In the SR model, a user can let the middleware repeat the execution when the tag is identified again. This approach just follows the policy of standard: ALE[8] returns an error message to the application and

does not provide any reprocessing mechanism. However, this approach cannot solve the inconsistency problem that can be caused by the execution of another tag operation by a different reader. When the tag appears at another reader, other read operations might access partially written data. In the previous example, the tag might be identified at $R5$ or other readers, which would access "LI" stored in $t1$.

In the LK model, we block inconsistent data accesses from unspecified readers. By maintaining a lock table, an access lock is requested before executing any operation. If a tag is locked by another operation, the operation cannot be processed. As a traditional concurrency control mechanism, this approach completely prevents inconsistent data access. However, this may delay execution of other operations while the tag is locked.

Our approach in this paper is to enable reprocessing of the incomplete operation at any arbitrary reader. We discuss a detailed description of our model in next section.

4 Asynchronous Reprocessing Model

In this section, we present an asynchronous reprocessing model (AR model) for finalizing incomplete operations. The objective of this model is to ensure complete execution of tag operations while preserving consistency of tag data by reprocessing *Unsure* operations and *Partially Processed* operations. We start the discussion of our model by defining an execution mechanism of the model.

4.1 Execution Mechanism of the AR Model

To ensure complete execution of a tag operation, we introduce a new type of tag operation called a reprocessing operation. The reprocessing operation represents unresolved parts of the incomplete operation to guarantee complete execution of the operation. It is created when the execution of a tag operation results in incompletion. The following definitions give a formal representation of the AR model.

Definition 4 (Reprocessing Operation). *A reprocessing operation ro_i is defined as a sub-operation of an incomplete operation o_i. An ro_i is represented as a tuple of four attributes (tid, offset, size, data), where tid is the identifier of the target tag performing the operation, offset is location of the memory area, size is the length of the data item to read/write, and data is the target data to be written.*

Definition 5 (Creation of RO). *When an execution of tag operation $Ex(o_i)$ results in incompletion, o_i is suspended and an ro_i is created, such that $ro_i = (o_i.tid, o_i.offset + Ex(o_i).nwp, o_i.size - Ex(o_i).nwp, o_i.data + Ex(o_i).nwp)$.*

The AR model is called asynchronous because the suspended operation o_i can be resumed via ro_i whenever tag $o_i.tid$ is identified by any reader. For example, assume that $M(o_1, e_1)$ results in incompletion, in the previous example in Fig.3. Our model enables the execution of ro_1 on e_3 even if the $o_1.rid$ and $e_3.rid$ are different. This makes sense semantically because ro_i accesses only the data item that should had been accessed by o_i. Definition 6 gives the formalized representation of the execution of the reprocessing operation.

Definition 6 (Execution Rule of RO). *The ro_i can be executed on any incoming tag event e_k such that $ro_i.tid = e_k.tid$. The o_i finishes its execution when the ro_i is executed completely.*

A reprocessing operation can be nested because of the recurrence of incomplete execution of the reprocessing operation. If an execution of ro_i results in incompletion again, it creates another sub-operation to reprocess it. Let ro_{ij} be a j-th reprocessing operation of o_i. The execution result of ro_{ij} is reported automatically to the parent operation. Eventually, the results reach o_i. Then, o_i finishes its execution by reporting the aggregated result to the application.

To execute the reprocessing operation, we need to evaluate every tag identification event in *ES* to check whether the condition matches. Monitoring of *ES* is similar to the continuous query processing[9]. By defining attributes of the reprocessing operation as a continuous query, each tag identification event can be compared to the continuous queries whether a matching reprocessing operation exists or not. A formal representation of a continuous query for the reprocessing operation is provided in Definition 7.

Definition 7 (Continuous Query for RO). *A q_i is a continuous query with a single attribute tid for the reprocessing operation ro_i where $q_i.tid = ro_i.tid$.*

The execution of the continuous query is done by the following manner. Let Q denote a set of q_i. For each tag event e_j in *ES* which is serially flowed to the middleware, e_j is compared with queries in Q whether matched $q_i \in Q$ exists or not; if exists, a reprocessing operation ro_i related to q_i can be executed on e_j.

Let us discuss a detailed execution mechanism for the AR model. To do this, we need to define an internal message protocol which detects a tag event and executes a corresponding reprocessing operation. Here, we assume a continuous query manager (CQM) which maintains the queries and continuously evaluates them with tag events in the stream. To enable efficient searching for a matched continuous query for each event, the CQM may construct a continuous query index as a one-dimensional index using $q_i.tid$, such as hashing. It enables fast processing of continuous queries

even if the volume of input tag stream is huge.

A message protocol between the reprocessing operation and the CQM is described in Table 4, and Fig.4 illustrates an example of the protocol. After $M(o_1, e_1)$ results in incompletion, ro_1 is created and a REGISTER message is sent to the CQM (steps 1∼3). Then, the CQM creates q_1 for ro_1 and continuously examines every tag event e in *ES*. When the CQM evaluates e_3, which matches the condition $e_3.tid = t1$, the CQM sends an OBSERVE message to the handle of the ro_1. Then, ro_1 is executed on e_3 (steps 4∼6).

Table 4. Message Protocol Between the Reprocessing Operation and the CQM

Message	Direction	Description
REGISTER	RO → CQM	Add a continuous query for the reprocessing operation
OBSERVE	CQM → RO	Execute the reprocessing operation
UNREGISTER	RO → CQM	Remove a continuous query for the reprocessing operation
SUSPEND	RO → CQM	Continue processing of a continuous query for the reprocessing operation
NESTED	RO → CQM	Modify handle of the continuous query to the nested reprocessing operation

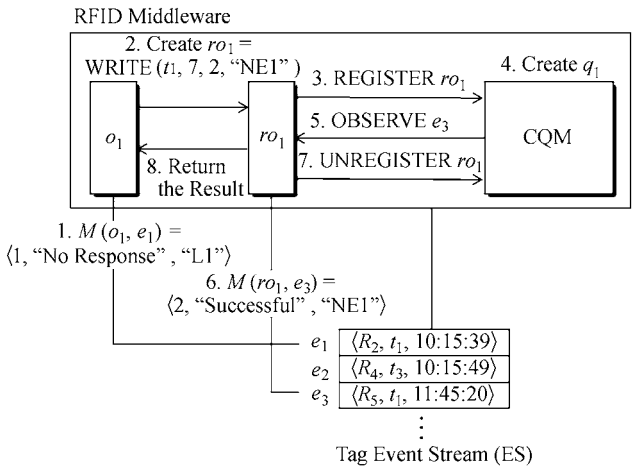

Fig.4. Example of the protocol that executes o_1 in Fig.3.

After the execution of ro_1, the reply message will be one of three messages: UNREGISTER, SUSPEND, and NESTED. If the execution is successful, ro_i sends the UNREGISTER message and q_i can be removed (steps 7∼8). If the message is SUSPEND, the q_i needs to be continuously evaluated again. The purpose of the NESTED message is to notify the nested creation of the reprocessing operation.

An overall algorithm for the reprocessing of an incomplete operation is shown in Algorithm 1. The operation is resumed when it receives the OBSERVE message from the CQM (lines 26∼28). After the execution of ro_i, the algorithm sends a proper message to the CQM by analyzing the execution result of ro_i. If ro_i is partially processed, it calls *Reprocess*() recursively, resulting in creation of a nested reprocessing operation.

Algorithm 1. Reprocessing of an Incomplete Tag Operation

```
1   Algorithm Reprocess (o, result, isNested)
2   o is an incomplete operation
3   result is a result of the execution of o
4   isNested specifies whether o is a ro
5   Begin
6     ro = create_RO (o, result);
7     if (isNested)
8       send_message_to_CQM (NESTED, ro);
9     else
10      send_message_to_CQM (REGISTER, ro);
11    end if
12
13    while (TRUE)
14      //suspend until the tag is re-observed
15      msg = receive_message_from_CQM ();
16      if msg.message != OBSERVE then
17        if o.timeout > current_time then
18          send_message_to_CQM (UNREGISTER, ro);
19          result.res = Failed;
20          return result;
21        else
22          continue;
23      end if
24
25      //execute the access operation
26      reply = execute_operation(msg.event.rid,
27            ro.tid, ro.offset,
28            ro.size, ro.data);
29
30      if reply.nwp = ro.size then
31        send_message_to_CQM (UNREGISTER, ro);
32        return reply;
33      else if reply.nwp > 0 then
34        //create nested reprocessing operation
35        reply2 = Reprocess (ro, reply, TRUE);
36        return combine_result(reply, reply2);
37      else
38        send_message_to_CQM (SUSPEND, ro);
39      end if
40    end while
41  End
```

The algorithm also includes a timeout value for each operation (refer to line 11). In the real environment, it is possible that a certain tag would never be observed again for the reasons such as malfunction of the tag, physical corruption, or being moved to other sites. We call this situation as starvation of the re-observation of an incomplete tag. To avoid waiting indefinitely for the tags that are not observed over a long period, the

middleware can terminate the execution of the operation if its timeout period expires. This imposes the condition that the tag will be unobserved permanently after expiration of the timeout period.

4.2 Autonomous Concurrency Control Using a Continuous Query Scheme

A key matter of consideration in the model is how to control the concurrent execution of the tag operations for preserving consistency of tag data. When multiple tag operations are registered in the middleware, it is possible that two or more operations access the same tag memory, either via the same reader or via different readers. In the RF transaction, the concurrency control problem can be restated as a selection problem of the operations for a given tag event in *ES*.

As an example, assume that $Ex(o_1)$ has resulted in incompletion and ro_1 is created in Fig.3. When e_3 arrives, a conflict would occur between the two executions, $M(ro_1, e_3)$ and $M(o_4, e_3)$. In this case, we call o_4 a conflict operation of ro_1 on e_3 because o_4 should not be executed prior to the execution of ro_1. This brings out an isolation rule for conflict operations as following definitions. In the definition, we denote the set of user operations by O_U and the set of reprocessing operations by O_R.

Definition 8 (Conflict Operation). *A reprocessing operation ro_i conflicts with o_j on e_k if $ro_i.tid = o_j.tid = e_k.tid$ and $o_j.rid = e_k.rid$.*

Definition 9 (Isolation Rule). *For all user operations $o_i \in O_U$ that conflict with $ro_j \in O_R$ on e_k, $M(o_i, e_k)$ will always be preceded by $M(ro_j, e_k)$.*

It is also possible that a tag is observed at multiple readers simultaneously. Therefore, blocking of multiple execution of a reprocessing operation is required. This is achieved by maintaining locking information which locks the ro_i while it is being executed. When a reprocessing operation is blocked, no operation can access the incomplete tag until the operation finishes its execution. Definition 10 describes a locking rule for the reprocessing operation.

Definition 10 (Locking Rule). *For a tag identification event e_i in ES, if an $ro_j \in O_R$ for the e_i such that $e_i.tid = ro_j.tid$ exists, locking on ro_j precedes sending an OBSERVE message to ro_j. While ro_j is blocked, ro_j cannot be executed on e_k where $e_i.t < e_k.t$ even if $e_k.tid = ro_j.tid$.*

Algorithm 2 gives the concurrency control algorithms for the CQM. If a reprocessing operation is not found, it searches the user operation table, which stores records of operations registered by applications. Consistency control of the conflict operations can be achieved by maintaining continuous queries for the reprocessing operations in a separated table. The status variable (line 17) is used for locking the record to prevent simultaneous executions of a reprocessing operation.

Algorithm 2. Concurrency Control Algorithm Using Continuous Query Processing

```
1   Algorithm CQ_Processing (ES)
2   ES is a tag identification event stream
3   Begin
4     while (TRUE)
5       e = ES.GetEvent();
6
7       record = SearchReprocessingOperation (
8                  e.tid);
9       if record = NULL then
10        record = SearchUserOperation(
11                   e.tid, e.rid);
12        if record = NULL then
13          continue;
14        end if
15      end if
16
17      if record.status = ACTIVE then
18        continue;
19      else
20        record.status = ACTIVE;
21        Send_message_to_RO (OBSERVE,
22              record.oid, e)
23      end if
24    end while
25  End
```

4.3 Correctness of the Reprocessing Operation

The main objective of the reprocessing model with the protocol is to guarantee complete execution of incomplete tag operations. The protocol is considered correct only if inconsistent data access does not occur during the reprocessing of the operation. To show how complete execution is achieved, we prove our model and the protocol via some theorems.

Theorem 1. *For any two reprocessing operations, ro_i and ro_j, $ro_i.tid$ and $ro_j.tid$ always differ if $i \neq j$.*

Proof. Assume $ro_i \in O_R$, $o_j \in O_U$ and $ro_i.tid = o_j.tid$. Then, the execution order of the two operations will be $ro_i \rightarrow o_j$, by Definition 9. It follows that o_j cannot be processed before ro_i is processed completely. Therefore, o_j cannot become an incomplete operation before ro_i is removed from O_R. □

Theorem 2. *The reprocessing protocol does not allow inconsistent data access during the reprocessing of an incomplete operation.*

Proof. More formally, this can be restated as follows. When $M(o_i, e_j)$ results in incompletion, any tag

identification event e_k such that $e_k.tid = o_i.tid$ will execute ro_i until ro_i is completely processed. This is achieved by Definition 9. Even if the tag is observed by multiple readers simultaneously, the execution of the reprocessing operation is granted to only one tag identification event because the first event places a lock on the tag by Definition 10. □

Theorem 3. *The AR model guarantees complete and correct execution of any tag operation o_i when $o_i.tid$ is identified.*

Proof. Complete and correct execution of o_i requires two conditions. One is that o_i should be executed when $o_i.tid$ is identified, and the other is that o_i should not access inconsistent data. The first condition is achieved as follows. When an execution of o_i results in incompletion, execution of o_i is via ro_i, which executes the unprocessed part when $o_i.tid$ is identified again. This continues recursively until the operation is completely processed. The second condition is explained above in Theorem 2. □

4.4 Combined Execution of Reprocessing Operations and Conflict Operations

As an execution of the reprocessing operation always precedes the executions of conflict operations as defined in Definition 9, conflict operations may lose its chance to be executed properly. For example in Fig.3, o_4 would be executed on e_3 if o_1 is executed successfully. However, because of ro_1, o_4 may not be executed on e_3. It is possible that the tag disappears after $M(ro_1, e_3)$, resulting that o_4 cannot be executed.

Let us consider the detailed relationships between the reprocessing operation and the conflict operation case by case. If a reprocessing operation is a read operation and a conflict operation is a write operation, the reprocessing operation should be executed before an execution of the conflict operation to preserve consistency. However, in other cases, we can combine two operations to reduce repeated execution of the operations. Following definitions provide rules for the combined execution of conflict operations.

Definition 11 (Forced Completion). *Let ro_i and o_j be conflict on e_k. If both operations are writes, then ro_i immediately finishes its execution by returning "Successful" and o_j is executed on e_k.*

Definition 12 (False Execution). *Let ro_i and o_j be conflict on e_k. If ro_i is a write operation and o_j is a read operation, then o_j finishes its execution by returning $\langle o_i.size$, "Successful", $o_i.data\rangle$ and ro_i is executed on e_k.*

Definition 13 (Operation Merging). *Let ro_i and o_j be conflict on e_k. If both operations are reads, then o_j is suspended and ro_i is executed on e_k. o_j finishes its execution when $Ex(ro_i)$ is completed.*

For example, in Fig.3, there is a conflict between o_4 and ro_1 on e_3. As o_4 is a read operation, its fate is to read the tag data that are completely written by ro_1. As the AR model guarantees the complete execution of ro_1, o_4 can return the $o_1.data$ immediately without waiting for the execution of ro_1 as described in Definition 12. This also preserves an execution order $o_1 \rightarrow o_4$ and does not allow any inconsistency. Other two rules can also be easily justified. This implies that the model has a great potential to be easily extended to enhance performance of data accesses as well as completeness and consistency.

5 Experiments

In this section, we aim to evaluate the completeness of the proposed model and compare it with other processing models. We first build an experimental environment using real RFID devices, and analyze the feasibility of our model by investigating the results of experiments. We also discuss the simulation results to verify usability of the model in the large-scaled environment.

5.1 Experimental Setup

We first implemented the proposed model in an RFID middleware using Java. It continuously receives a tag event stream from RFID readers, and reports the execution results of registered operations. To compare the feasibility and performance of the proposed model, we also built two other reprocessing models, the SR model and the LK models, for a comparison purpose (see Subsection 3.3).

To measure the performance of each model more precisely, we use a belt conveyor with approximately 12 meters round as shown in Fig.5. While the tagged items move on the conveyor, two Alien 9900 RFID readers detect tag's identification and execute a requested access operation. Although the configuration is quite simple, this provides fully-controlled movements of the tags. Therefore, we can expect more accurate comparisons of the performances among three models.

A detailed scenario of the experiments is as follows. We first register 20 access operations in the middleware: 10 write operations for $R1$ and 10 read operations for $R2$, respectively. The operations read or write 32 words data from/to the memory of the tag. When the scenario starts, 10 passive tags are loaded on the conveyor one by one. They pass readers $R1$ and $R2$, successively. If all of the operations are processed successfully without any failure, the middleware will return memory data previously written by $R1$ when a tag passes $R2$.

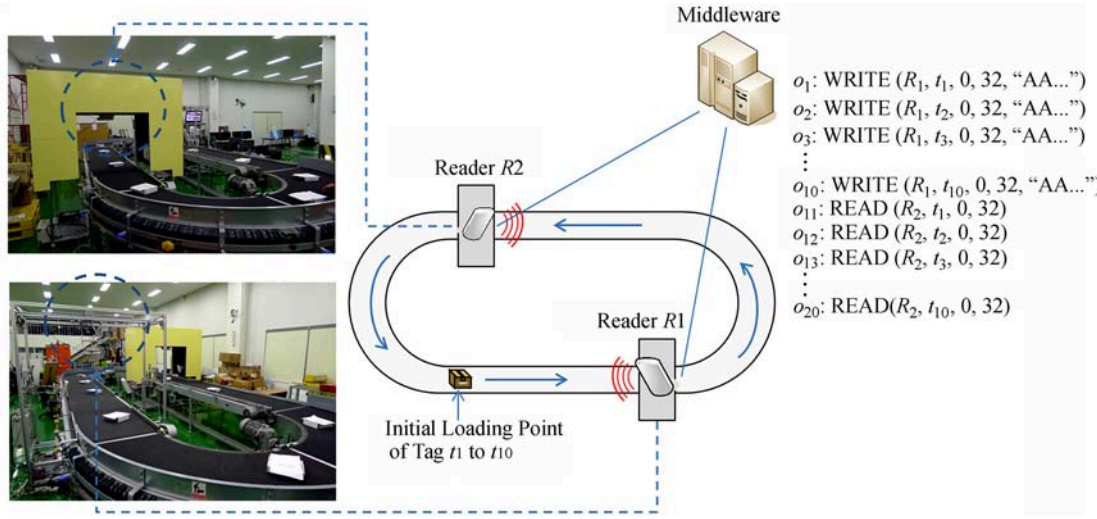

Fig.5. Configuration of the experiments.

5.2 Experimental Results

Firstly, we examine the number of accesses of inconsistent data after the execution of 10 read operations. If a write operation is partially processed at $R1$, a read operation may access partially written data at $R2$. Fig.6 shows the result when the SR model is applied. If RF power at reader's antenna reduces, which emulates that the environment becomes harsh, more inconsistent data accesses occur. Note that two others did not show any inconsistent data access.

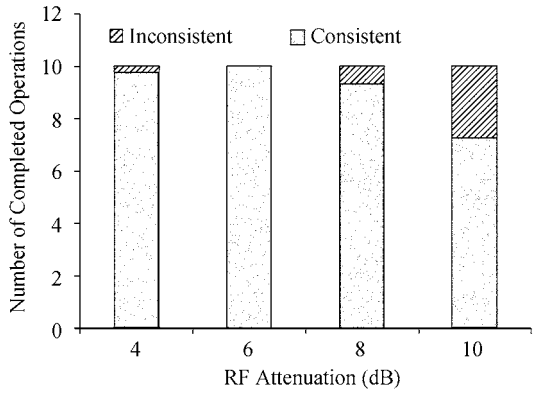

Fig.6. Number of accesses of inconsistent data.

Fig.7 shows the average number of executions until all operations are completely processed. The LK model preserves consistency of tag data with the similar number of executions compared to the SR model. AR model processes write operations at both readers. We can expect that AR model can process write operation more quickly because incomplete write operations are processed by both readers. The result also shows that write operations require more executions compared with read operations. The reason is that the write operation requires more energy consumption than the read operation.

Next, we measure the processing time of access operations. Fig.8 shows a comparison result of the average processing time of each model. In the case of the LK model, the average processing time of each read operation is higher than that of other models. The reason is that some read operations are blocked by incompletely

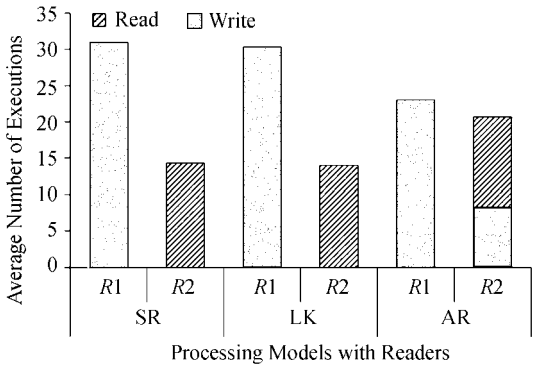

Fig.7. Average number of operation executions.

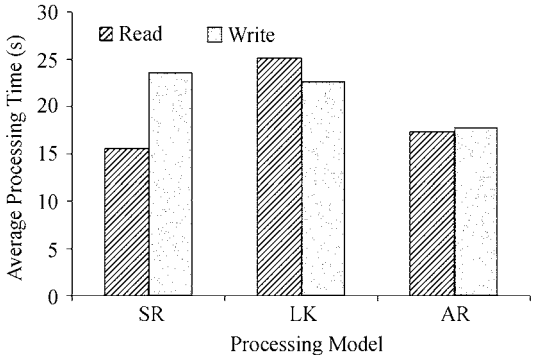

Fig.8. Average processing time of read/write operations.

executed write operations. As a result, the LK model shows even worse performance than the SR model. The AR model shows the best performance among the three models. This suggests that the AR model surely reduces the processing time for incomplete operations.

Fig.9 shows the total elapsed time for executing all access operations. Although the SR model and the LK model require approximately 60 seconds for executing all write operations, the AR model greatly reduces its execution time by 40 seconds. The AR model has completed all operations even faster than the SR model. Although the execution of read operations in the SR model is very fast, the read operations may access inconsistent data as discussed in Fig.6. This suggests that the AR model can be an efficient model for accessing RFID tag memory in real RFID environments.

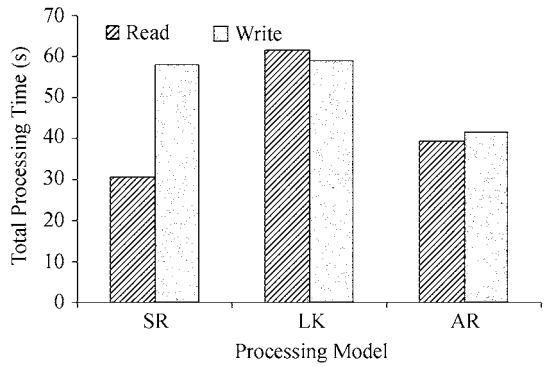

Fig.9. Elapsed time for processing all read/write operations.

5.3 Simulation Results

To verify performance of the proposed model, it is also required to measure the performance in a large-scaled environment. However, it is a costly process because deploying a lot of real RFID devices requires huge resources. As an alternative, we developed a simulated environment using 100 virtual readers and 10 k virtual tags. We synthesized a write operation set using one million operations with a uniform distribution. We also randomly generated 100 k tag events for the experiment. To specify the incompletion of operations, we configured a variable which denotes the probability of incompletion (p_i) after an execution of an operation on a tag.

Table 5 shows the results of the comparison where p_i is 10%. For the comparison, we added the Non-Reprocessing model (NR model) which does not reprocess any incomplete operation. Although above two models suffer from inconsistency, later two models completely suppress inconsistent data accesses. However, the number of successful executions is decreased in the LK model because locking of the incomplete tags blocks the execution of other operations. The AR model increases the rate of complete execution of the incomplete operations to almost 90%. Remaining operations would also be processed completely if the tag events were streamed indefinitely.

Table 5. Simulation Results of the Access Operations

Model	Result			
	Success	Incompletion	Reprocessed	Inconsistent Access
NR	81 681	9 054	0	6 814
SR	82 043	9 081	102	6 745
LK	54 093	6 010	302	–
AR	80 995	8 988	8 043	–

Fig.10 shows the reprocessing ratio of unsuccessful operations as the number of readers varies. Although the performances of the SR model and LK model rapidly decrease, the AR model shows a constant performance even if the number of reader increases. This suggests that the AR model shows better performance especially when the number of readers is large, which is not captured by the previous experiments when some real devices are used.

Finally, we measured the total amount of successful executions as p_i varies. When p_i increases, the result for the LK model becomes poor as shown in Fig.11. The AR model outperforms other two models even though the SR model possesses inconsistency. We can conclude that the proposed model increases the availability of tag

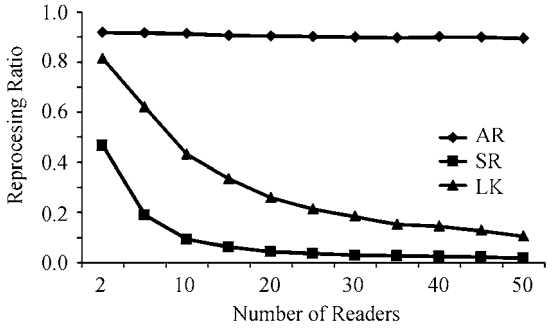

Fig.10. Reprocessing ratio for varying number of readers.

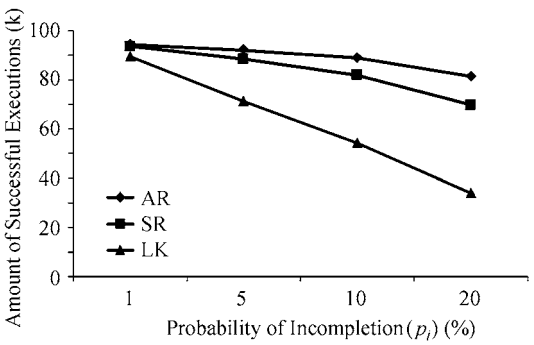

Fig.11. Total amount of successful executions for varying p_i.

data accesses by increasing the complete executions of access operations.

6 Related Work

To the best of our knowledge, there has been no work on processing RF transactions. We first look at previous studies of wireless transactions and discuss other related studies in RF environment.

Let us consider wireless transaction processing schemes for mobile devices. A mobile computing system is a special type of distributed system in which the network between nodes is changed dynamically. Because of high communication costs, limited battery power, long periods of disconnection, and other constraints, traditional commit protocol techniques such as two-phase commit protocol[10] cannot be applied directly to the mobile environment. The network is easily disconnected because of communication latency, hand-off, and power saving, in addition to cost reduction. Several techniques are proposed for recovering the transaction after reconnection, cache-based schemes[11-13] and replication-based schemes[14-15].

It is clear that applying mobile transaction models to RF transactions is unrealistic. In the mobile environment, disconnection of the network is mostly predictable[16]. Even if the connection is lost unexpectedly, the mobile hosts, by themselves, can reconnect and resume the transaction. Mobile hosts have sufficient battery power to resume suspended transactions even if the network is totally disconnected. By contrast, the RFID tag can be activated only when it is in the interrogation area of an RFID reader and unexpected disconnection of RF transactions frequently occurs. Unlike mobile hosts, an RFID tag cannot reconnect and reprocess the suspended operations without help from the RF infrastructure such as the reader and the middleware.

Transaction management in sensor networks was first discussed in [17]. This paper also defined an update transaction of sensor database which modifies attributes of the sensors, such as the name, id, and sampling rate of temperature sensors. However, this paper did not touch on the problem of disconnection between the sensors and the host because of basic assumptions about the sensor network. At all times, each sensor node is completely connected either to the base stations or to other sensor nodes by ad-hoc networking[18].

The disconnection problems of RFID tags are discussed in [19]. To deal with uncertainty in the execution of access operations, this work has proposed a virtual tag memory service, which is an additional service system in the distributed network infrastructure to provide transparency of tag memory accesses. However, this requires huge cost for maintaining backup copy of each tag memory and each operation's result status. The preliminary work for this paper is presented in [20], where we defined the problem of processing RF transactions and introduced a reprocessing model. However, the analysis and verification of the solution were insufficient. This study extends the reprocessing model by including a concurrency control protocol. In addition to the extension, it presents a formalization and validation of the model via experiments, adding to the utility of the research.

7 Conclusions

This paper has introduced the problem of incompletion in read/write operation execution caused by the volatile and uncertain characteristics of RF communication. Because of the unpredictable disconnection of the RF connection, an access operation might be unprocessed or partially processed. To eliminate tag data inconsistency, we have proposed a reprocessing model which re-executes incomplete operations by defining a sub-operation of the incomplete operation. We also present a concurrency control mechanism based on a continuous query scheme that asynchronously detects re-observation of incomplete tags by an arbitrary reader and removes inconstant data accesses. The main contribution of our work is the provision of a reprocessing model with a protocol that guarantees atomicity and durability for tag data accesses. Our work also provides proofs of the usability of the model via a set of experiments using both real data and simulated data.

As the proposed model provides a reprocessing mechanism for incomplete tags in an RFID middleware, we need to further consider the outgoing tags from the coverage of the middleware. As a future work, the reprocessing model needs to be extended to distributed middleware environments to handle the flow of incomplete tag among multiple sites.

References

[1] Want R. An introduction to RFID technology. *IEEE Pervasive Comput.*, 2006, 5(1): 25-33.

[2] Banks J, Hanny D, Pachano M, Thompson L. RFID Applied. Wiley, Chichester, March 2007, pp.328-329.

[3] Weinstein R. RFID: A technical overview and its application to the enterprise. *IT Professional*, 2005, 7(3): 27-33.

[4] Fishkin K P, Jiang B, Philipose M, Roy S. I sense a disturbance in the field: Unobtrusive detection of interactions with RFID-tagged object. In *Proc. the 6th Int. Conf. Ubiquitous Computing*, Sept. 2004, pp.268-282.

[5] Jung S, Cho J, Kim S. FQTR: Novel hybrid tag anti-collision protocols in RFID system. *Journal of KIISE: Software and Applications*, 2009, 36(7): 560-570.

[6] EPCglobal Inc. Class 1 generation 2 UHF air interface protocol standard "Gen 2", http://www.gs1.org/gsmp/kc/epcglobal/uhfc1g2.

[7] EPCglobal Inc. Low level reader protocol, http://www.gs1.org/gsmp/kc/epcglobal/llrp.
[8] EPCglobal Inc. Application level events (ALE) standard, http://www.gs1.org/gsmp/kc/epcglobal/ale.
[9] Golab L, Özsu M T. Issues in data stream management. *SIGMOD Record*, 2003, 32(2): 5-14.
[10] Silberschatz A, Korth H F, Sudarshan S. Database System Concepts, 4th edition, New York: McGraw-Hill, 2002, pp.709-722.
[11] Kisler J, Satyanarayanan M. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems*, 1992, 10(1): 3-25.
[12] Wu K, Yu P S, Chen M. Energy efficient caching for wireless mobile computing. In *Proc. the 12th Int. Conference on Data Engineering*, Feb. 26-Mar. 1, 1996, pp.336-343.
[13] Madria S K, Bhargava B. A transaction model for mobile computing. In *Proc. Int. Database Engineering and Application Symposium*, July 1998, pp.92-102.
[14] Rasheed A, Zaslavsky A. Ensuring database availability in dynamically changing mobile computing environments. In *Proc. the 7th Australian Database Conference*, Melbourne, Australia, Jan. 1996, pp.100-108.
[15] Ding Z, Meng X, Wang S. A transactional asynchronous replication scheme for mobile database systems. *Journal of Computer Sci. and Tech.*, 2002, 17(4): 389-396.
[16] Madria S K, Mohania M, Bhowmick S S, Bhargava B. Mobile data and transaction management. *Information Sciences*, 2002, 141(3-4): 279-309.
[17] Gürgen L, Roncancio C, Labbé C, Olive V. Transactional issues in sensor data management. In *Proc. the 3rd International Workshop on Data Management for Sensor Networks*, Sept. 2006, pp.27-32.
[18] Wang B, Yang X, Wang G, Yu G. Continuous approximate window queries in wireless sensor networks. In *Lecture Notes in Computer Science 4505*, Dong G *et al.* (eds.), Springer-Verlag, 2007, pp.407-418.
[19] Floerkemeier C, Roduner C, Lampe M. RFID application development with the Accada middleware platform. *IEEE Systems Journal*, 2007, 1(2): 82-94.
[20] Ryu W, Hong B. A reprocessing model based on continuous queries for writing data to RFID tag memory. In *Proc. the 14th International Conf. Database Systems for Advanced Applications*, April 2009, pp.201-214.

Wooseok Ryu received the B.S. and M.S. degrees in computer engineering from Pusan National University (PNU), Busan, Korea, in 1997 and 1999, respectively. He is currently pursuing his Ph.D. degree in computer engineering at PNU. He has developed RFID middleware with Institute of Logistics Information Technology. His research focuses on RFID middleware, tag data model, RF transaction, stream databases, spatial databases and moving object databases.

Bonghee Hong received the M.S. and Ph.D. degrees in computer engineering from Seoul National University, Seoul, Korea, in 1984 and 1988. In 1987, he joined the faculty of Computer Engineering of the Pusan National University (PNU). He is working as a professor of database in the Department of Computer Engineering at the PNU. He is a director of Institute of Logistics Information Technology. He is also a steering committee member of DASFAA. His research interests include theory of database systems, moving object databases, spatial databases, RFID system and RFID/RTLS/Sensor middleware.

Joonho Kwon received his Ph.D., M.S. and B.S. degrees in the School of Electrical Engineering and Computer Engineering from Seoul National University, Seoul, Korea, in 2009, 2001 and 1999, respectively. He is an assistant professor of Institute of Logistics Information Technology at Pusan National University, Korea. His current research interests include XML filtering, XML indexing and query processing, Web services, RFID data management, serious game and multimedia databases.

Ge Yu received his B.E. and M.E. degrees in computer science from Northeastern University of China in 1982 and 1986, respectively, Ph.D. degree in computer science from Kyushu University of Japan in 1996. He has been a professor at Northeastern University of China since 1996. He is a member of ACM, IEEE, and a senior member of CCF. His research interests include database theory and technology, distributed and parallel systems, embedded software, and network information security.