통합 RFID 미들웨어의 응답시간 개선을 위한 효과적인 캐쉬 구조 설계

김 정 길[†] · 이 준 환[††] · 박 경 랑[†††] · 김 신 덕[††††]

요 약

본 논문에서는 WSN(wireless sensor networks)과 RFID(radio frequency identification) 시스템을 통합하여 이용할 수 있는 통합 RFID 미들웨어에서의 효과적인 캐싱 기법을 제시한다. 통합 RFID 미들웨어가 운영되는 환경은 연결된 RFID리더로부터 대규모의 데이터가 입력되고, 다수의 무선 센서로부터 끊임없이 데이터가 입력되는 상황을 가정하고 있으며 또한 특정 목적을 위해 과거에 센서로부터 입력되어 분산 저장되어 있는 히스토리 데이터도 활용될 수 있음을 가정하고 있다. 따라서 캐싱 기능을 구비한 특정 미들웨어 레이어에서 센서 노드로부터 수신되는 연속 데이터와 분산 저장되어 있는 히스토리 데이터에 대한 신속한 질의 및 응답을 위한 효율적 데이터 처리가 절실히 요구된다. 이를 위하여 본 논문에서 제안되는 캐싱 기법은 기존의 캐싱 기법 기반으로 통합 RFID 미들웨어에 특화하여 데이터 처리의 효율을 높이기 위하여 두가지 방법을 제시하고 있으며, 이는 처리 데이터의 유형에 따라 DSC(data stream cache)와 HDC(history data cache) 로 구분된다. 제안된 캐싱 기법은 다양한 파라미터를 이용한 실험을 통하여 신속한 질의 및 응답이 이루어짐을 보여주고 있다.

키워드 : RFID, WSN, 미들웨어, 캐쉬 기법, 응답시간, 연속 데이터

An Efficient Cache Mechanism for Improving Response Times in Integrated RFID Middleware

Cheong-Ghil Kim[†] · Jun-Hwan Lee[††] · Kyung Lang Park[†††] · Shin-Dug Kim[††††]

ABSTRACT

This paper proposes an efficient caching mechanism appropriate for the integrated RFID middleware which can integrate wireless sensor networks (WSNs) and RFID (radio frequency identification) systems. The operating environment of the integrated RFID middleware is expected to face the situations of a significant amount of data reading from RFID readers, constant stream data input from large numbers of autonomous sensor nodes, and queries from various applications to history data sensed before and stored in distributed storages. Consequently, an efficient middleware layer equipping with caching mechanism is inevitably necessary for low latency of request-response while processing both data stream from sensor networks and history data from distributed database. For this purpose, the proposed caching mechanism includes two optimization methods to reduce the overhead of data processing in RFID middleware based on the classical cache implementation polices. One is data stream cache (DSC) and the other is history data cache (HDC), according to the structure of data request. We conduct a number of simulation experiments under different parameters and the results show that the proposed caching mechanism contributes considerably to fast request-response times.

Keyword : RFID, WSN, Middleware, Cache, Response Time, Data Stream

1. Introduction

The rapid advances in technology today, especially in the areas of data acquisition through sensors and communications, have opened the era of Ubiquitous Computing in which computers are so deeply integrated into our lives and communicate wirelessly with network identities. This environment has been expected as "Sensor Internet" or "Internet of things" in which "everything is alive" [1, 2]. We, considering the situation that tiny and battery powered wireless sensors with low prices are spread over around us, can easily expect to see large numbers of autonomous sensor networks and need to collect a significant amount of interesting data about the

† 정 회 원 : 연세대학교 컴퓨터과학과 BK21 연구교수
†† 준 회 원 : 연세대학교 컴퓨터과학과 석사과정
††† 준 회 원 : 연세대학교 컴퓨터과학과 박사과정
†††† 정 회 원 : 연세대학교 컴퓨터과학과 교수
논문접수 : 2007년 10월 17일, 심사완료 : 2007년 12월 17일

DOI: 10.3745/KIPSTA.2008.15-A.1.17

world [1]. In such a sensor abundant environment, one of major challenges would be to provide the request-response with low latency upon every requests form multiple applications.

Until now, most researches in the sensor network domain have focused on routing, data aggregation, and energy conservation inside a single sensor network [3]. However, in recent years, WSNs become a significant technology which begins to attract considerable research attention. At the same time, they are being developed for a wide range of applications. Generally, a WSN consists of large numbers of tiny sensor nodes that communicate over wireless channels and performs distributed sensing and collaborative data processing. The major functionality of it is simple data gathering style applications, and in most cases, supports one application per network [4].

An RFID system consists of RFID readers, tags, and middleware. Here, the middleware collects the tag data identified from the readers; preprocesses and converts them into meaningful representations. Being compared with WSNs, RFID systems are considered to be large-scale networks such as supply chain management. However, each system has its strength and weakness for organizing ubiquitous services. For example, RFID systems use the tag data identified from the readers located at static locations with limited numbers. In this case, applications may not know about the real-time information on an object and environment when objects move through certain routes. On the other hand, WSNs can collect real-time information of objects. Therefore, the integration of WSNs and RFID systems can amplify the performance of senor networks by taking advantage of the merits from both and complementing demerits each other. In order to do that, a dedicated layer in the middleware is required to provide (1) transparent interface for diverse applications to access sensors or sensed data and (2) common data model and efficient mechanism to manage sensed data from both RFID systems and WSNs, Under these circumstances, a cache mechanism in RFID middleware becomes a stringent performance constraint because there are a significant amount of data processing requested from multiple applications and large numbers of tiny sensor nodes.

This paper focuses on an efficient caching mechanism appropriate for RFID middleware which can integrate WSNs and RFID systems. Typically, a caching mechanism which temporarily stores data for later retrieval has been an effective method of improving the performance of request-response exchanges and reducing recurring computation in distributed systems. Based on this idea, the proposed caching mechanism consists of two modules, data stream cache (DSC) and history data cache (HDC) according to the structure of data request for the purpose of the low latency of request-response. They are implemented on the middleware, Ubiquitous Information Middleware (UIM).

The remainder of the paper is organized as follows. Section 2 includes related works. Section 3 introduces the environments relevant to the proposed cache systems. In Section 4, we describe the proposed caching mechanisms. Experimental results are provided in Section 5. Finally, Section 6 concludes this paper.

2. Related Work

While there have been several researches on caching mechanisms for Web services [5] such as remote object caching for distributed systems including COBRA [6] and Java RMI [7], web page caching (web proxy) [8], and caching DB query result [9, 10], the cache issues in RFID middleware for optimal realization of sensor technologies and applications have not been studied carefully. Furthermore, only few researches have been introduced in the area of integrating WSNs and RFID systems. However, neither of them tackles the issues of caching mechanisms in detail. Followings include the introduction of several researches regarding the integration of WSNs and RFID systems.

The HiFi approach [11] and Global Sensor Networks middleware [3] are representatives of efforts to provide the integration of WSNs. The former, being researched in UC Berkeley Database Research, is an integrated solution for managing distributed receptor data. It provides hierarchical data stream query processing to acquire, filter, and aggregate data from multiple devices in a static environment. It aggregates distributed data using the hierarchical structure of views combined with the others to form a new one. The latter is similar to the HiFi approach but it takes tables instead of views. It uses the peer-to-peer technology to support dynamic environment. Both approaches use the SQL to aggregate the data. However, SQL based systems have a problem that an application always has to know about information of deployed sensor environments such as database schema. Somewhat they are useful for pre-configuration

applications, but require many efforts to make a new application.

Ubicore (Ubiquitous Core) [12], being researched in ETRI, is an XML based RFID middleware system. It focuses on large sensor data from many sensor devices. It uses its own query language called XQueryStream (XQuery for Stream data) which is originated from XQuery [13]. Using this language, it processes the data before query evaluation using the pre-filtering method. It reuses the intermediate results of previous evaluations to improve the processing of RFID tag data streams. Although this system supports only EPC-based sensor devices, it takes a growing interest in decreasing data for high throughput. In [14], an adaptive middleware framework is proposed to explore the resource/quality tradeoffs during information collection. The main idea is to reduce the communication frequency at sensor nodes by lowering the sampling frequency without compromising the accuracy of the results.

3. Cache Environments

In this section, we first look over the proposed middleware architecture. This procedure motivates the necessity of developing an efficient cache mechanism in RFID middleware. (Fig. 1) shows the overall architecture of the designed middleware called UIM which consists of four layers: device connection layer, data translation layer, data management layer, and data service layer. The major role of UIM is gathering sensor data and converting them into meaningful representations for various applications.

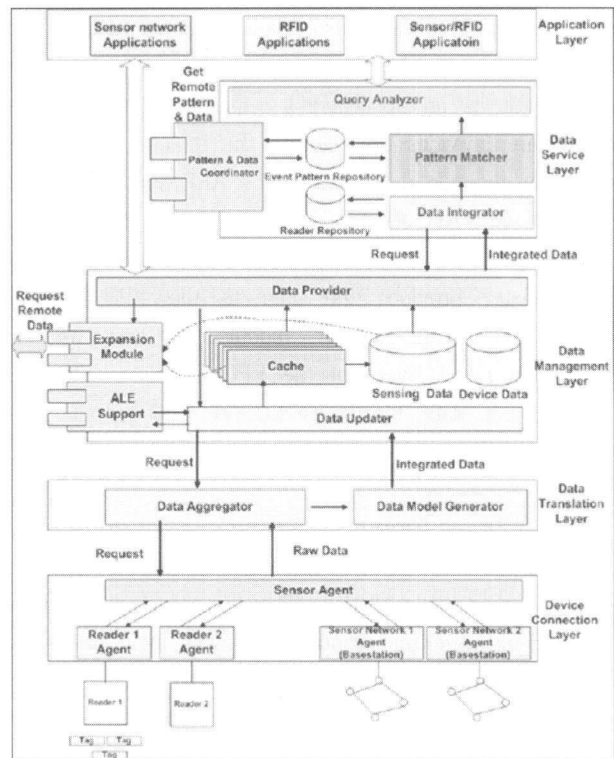
3.1 Device Connection Layer

The device connection layer has the responsibility of communicating with WSNs and RFID systems. This layer provides compatibility and extensibility to communicate between heterogeneous networks. Also this layer equips with a common interface to communicate with upper layer. Each WSN has a base station, a representative of the network which acts as a local sensor agent to communicate with UIM. Through this abstraction, a WSN can be recognized as a singe device. Therefore, from the point of UIM view, it may have a global sensor-agent in which each WSN or RFID connection could be implemented with its own protocol.

This layer periodically subscribes data from sensor nodes and aggregates them through a base station. And then it delivers raw data to the upper layer when a base station receives the data request initiated by applications from upper layer. The base station interface is implemented by using the Java RMI. Responses of a base station are raw data, called a *sensorDataType*. This is a pre-defined raw data type such as EPC, temperature, humidity, etc.

3.2 Data Translation Layer

The data translation layer aggregates the data from base stations upon application requests; converts them into integrated data models. Those tasks are processed by two components: data aggregator and data model generator. When the data model generator generates the integrated data model, it utilizes the meta-data of the network and provides them to the data management layer.



(Fig. 1) Architectural overview of UIM.

3.3 Data Management Layer

The data management layer consists of two components: the data updater and data provider. The data updater, upon requests from applications, receives sensing data from the lower layer; delivers them to the data provider as well as ALE support module; if necessary, stores them to local storages. This kind of request is

initiated by the data provider and the data provider receives it from the data service layer. At the same time, the data provider has the functionality of returning the requested data to the upper layer. Here, there might be two different data types: either real-time or history data. That is, this layer is in charge of delivering the data to a specific object relevant with low latency. As a result, this layer is a central station in which all request-response data branch off; such that the proposed cache mechanism is devised in this layer to reduce the response times. More details follow in Section 4.

Another data flow that we may think of is as following. To build a global network, a number of middleware should be deployed. For exchanging the local sensor data, one middleware has to communicate with other middleware. It is the role of the extended module. To communicate with the other middleware, the local device information should be stored into database and updated periodically. The middleware uses the device information to request sensor data to other middleware. The middleware, which received the request, processes the request as an application's request. After then, the middleware provides the data to the remote middleware using the common data model. The ALE support module connects with EPC Global network [15, 16, 17]. If only EPC based data is requested by the RFID application, the data updater provides the data to the EPC-IS [17] through the ALE support module. It acts as an Application Level Event of the EPC Global architecture [16] - Input data format is ECSpec and output data format is ECReport.

3.4 Data Service Layer

The data service layer returns data or high-level event messages to the application. The high-level event is a semantic message that contains business logic based on Event-Condition-Action based query [16, 17, 18]. The application can use events to improve the performance without complicated data processing phase done by the middleware. This layer is optional; that is, if an application does not require the high-level event, the data may be delivered directly to application not via this layer. In this paper, we only introduce basic concept. The details remain open for the future work.

3.5 Common Data Model

The raw data may be aggregated from the heterogeneous devices which have their own data formats. Therefore, a common data model is required to process those different data in UIM and communicate with remote middleware. For this extensibility, the designed common data model is composed of three parts: device schema, value schema, and option schema. The first two schemas are commonly used for all of data; the last one is added for multiple types.

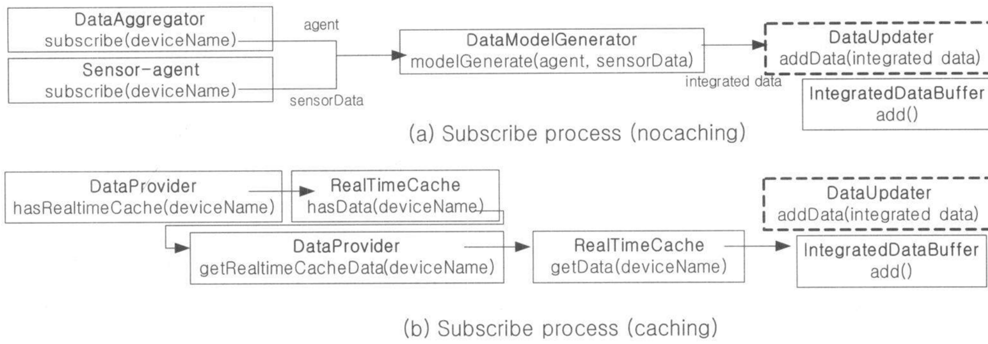
The device schema has 4 fields: *deviceName*, *networkID*, *sensorID*, and *deviceType*. They contain the information about network and device identification. The value schema has 4 fields: *format*, *value*, *type*, and *timestamp*. It represents the characteristics and value of a raw data. The option schema includes the optional information such as the storage flag, extension mode parameter, and the device meta-data; it enables to process a variety of message types. This data integration is made by the data model generator in the data translation layer. Finally, all of these data are delivered to the application layer with the form of XML document in our research.

4. Proposed Caching Mechanism

Given the architecture and the execution model of UIM, most of core data processing related with request-response is made by the data provider in the data management layer. Therefore, the proposed caches, DSC and HDC, are devised into this layer to manage the data efficiently and improve the middleware performance of data processing. The proposed caching algorithms have similar structures with conventional caching mechanisms, which can be summarized as follows: (1) maintaining data access history information, (2) establishing data priorities from past statistics, and (3) replacement decisions based on this priority scheme when the cache does not have enough space to accommodate new data.

4.1 Data Stream Cache

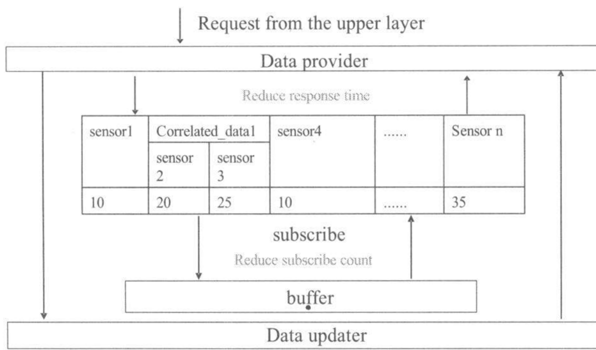
(Fig. 2) depicts the model of data stream accesses with and without caching mechanism. According to the execution flow, the data management layer ought to subscribe to the local sensor-agents whenever there is a request from applications. Suppose many applications make requests data simultaneously through UIM, the response time of each request will be delayed. To solve this problem, we propose a caching mechanism, called data stream cache (DSC) which is placed in the data management layer. (Fig. 3) shows the structure of DSC.



(a) Subscribe process (nocaching)

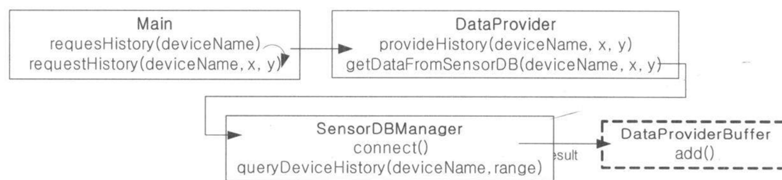
(b) Subscribe process (caching)

(Fig. 2) Model of data stream accesses

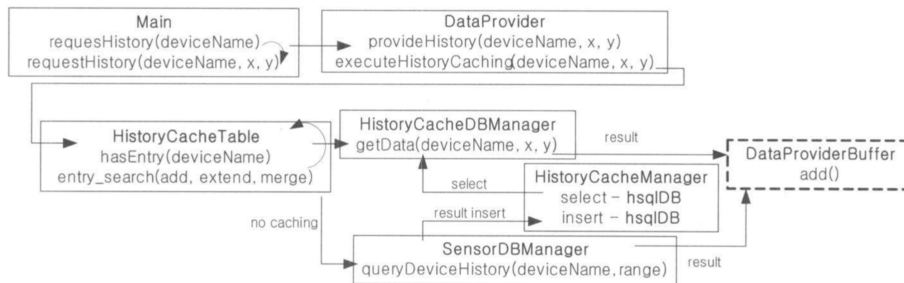


(Fig. 3) Data stream cache architecture

The design policy of DSC is to store data frequently requested from UIM. At this time, we take advantage of the value of *valid_time* which comes from an application according to the characteristics of the network. This value assigns the time interval of subscribing to sensor data. For example, there is a temperature monitoring system which issues the sensor data periodically about 10 seconds. In this case, the sensor data has 10 seconds of *valid_time*. If the application requests to the data repeatedly from the same device within a *valid_time* interval, the data provider fetches the data from the cache rather than local sensor agents because the result of the subscribing works is same as with the value in *valid_time*. Therefore, the job of subscribing to local sensor agents is not necessarily required. These data should be managed in DSC in such a way that when the data is requested by the application, it stores to the cache; DSC is updated periodically at *valid_time* interval. If the application requests the data which is managed in DSC, the data provider does not need to subscribe a sensor-agent. The data provider can find those data in DSC. As a result, the application can reduce the data response time. In addition, the middleware can reduce the workload by reducing the number of subscribing works because it does not have to issue subscribing to local sensor agents on every request.

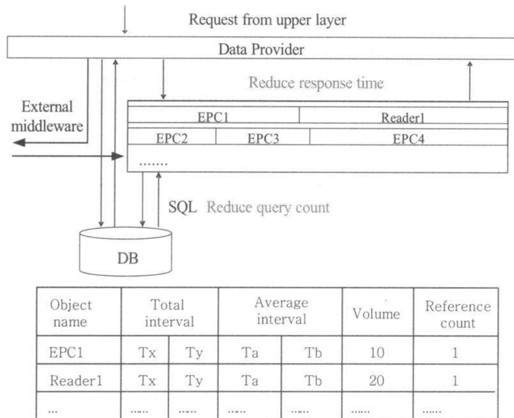


(a) History data request (noncaching)



(b) History data request (caching)

(Fig. 4) Model of history data accesses



(Fig. 5) History data cache architecture.

As web caching and conventional memory caching, DSC has to establish a replacement policy for the situation that it does not have enough space to accommodate a new entry. For this purpose, DSC calculates data priorities from past statistics and maintains the priorities order among data entries. The data priority, $P_{d,}$ is calculated using the bellowing equation:

$$P_d = \#\ of\ references \diamond valid_time\ /\ remain_time \qquad (1)$$

Here, *remain_time* is the time when an entry stays in DSC. Because the data in the cache is updated at *valid_time* interval, if P_d is less than 1, it means that this entry is no longer worthwhile being managed in DSC; such that DSC checks those kinds of data periodically and removes them.

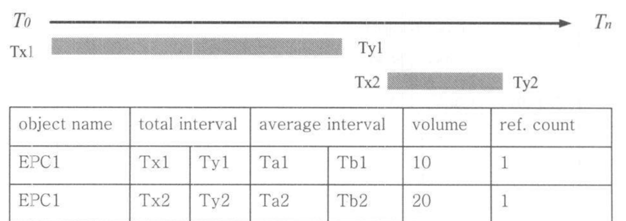
4.2 History Data Cache

(Fig. 6) depicts another model of data accesses with and without caching mechanism, which may happen in UIM. This is the case of accessing to database for the history data of a specific device or an object. In this case, the response time may require much longer latency than the case of data stream reference. Sometime it becomes worse when the history data does not exist in local middleware because UIM has to access to other middleware. (Fig. 5) shows the proposed cache structure for the history data, called history data cache (HDC).

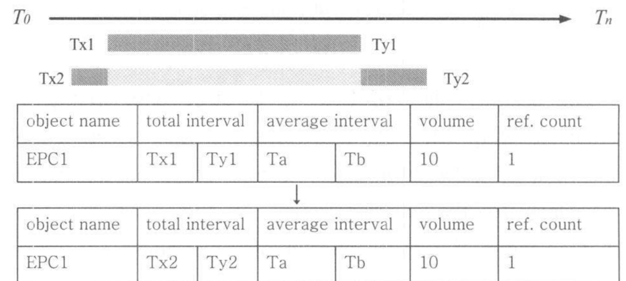
History data may have different sizes according to each request; such that HDC maintains an additional table, called a history data table (HDT), which includes total interval, average interval, volume, and reference count. Here, the total interval represents range of the data in HDC; the average interval does the average value of request which is referenced the entry; the volume field does count of the data. The order of entries in HDT is derived from the timestamp range of an object. If a specific device or an object is referenced by the history request, the result of the request is stored into HDC.

At this moment, the UIM executes the history request preparing algorithm. In case of the history request, when a requested pattern is found in HDT, the result is prepared from HDC. When a new data is stored into HDC, the entries of HDT have to be updated using the range. The updating is conducted with four different modes according the range.

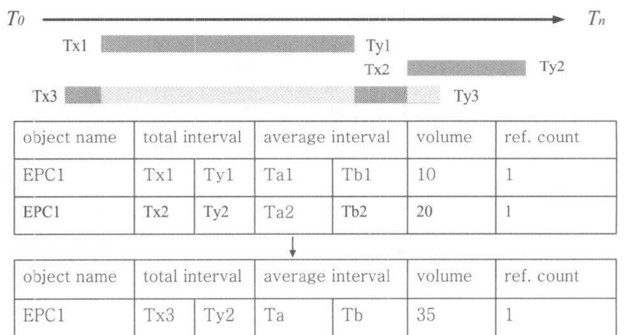
If the range of the new data does not overlap with the range of the existing entries correspondent to the same



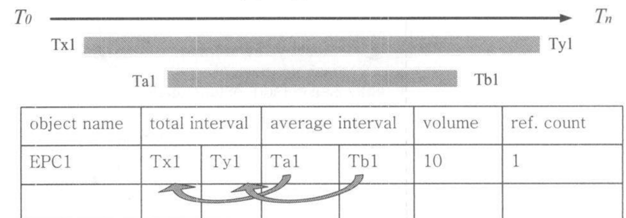
(a) Add mode



(b) Expanded mode



(c) Merged mode



(d) Reduced mode

(Fig. 6) Four different update modes.

device, the new data is added as a new entry, called an *add_mode*. (Fig. 6(a)) shows the example of it. The HDT has the entry which has the range from T_{x1} to T_{y1}. If a request has the range from T_{x2} to T_{y2}, it does not overlap. As a result, a new entry is added into HDT.

If the range of the new data overlaps with that of current entry and includes it, the range of current entry is replaced with that of the new data, called *expanded_mode*. (Fig. 6(b)) shows the example of it. If a new request has the range from T_{x2} to T_{y2}, and that is longer than the current range from T_{x1} to T_{y1}, the range of a new request is replaced with that of current entry.

In *expanded_mode*, we have to consider another situation that a new range is overlapped with more than one entry. In this case, all of the entries should be updated, called *merged_mode*. The new entry after merging operation updates the range using the minimum value of T_x and maximum value of T_y. (Fig. 6(c)) shows the example of it. If a new request has the range from T_{x3} to T_{y3}, all entries are merged into the one with the new range from T_{x3} to T_{y2}. In this way, it is always possible to trace temporal history of the object.

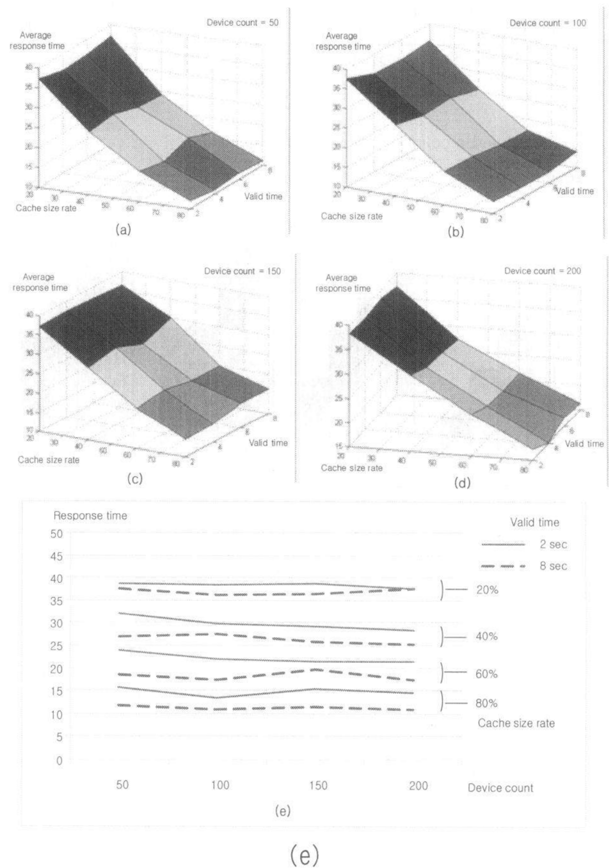
The final mode is applying a replacement policy when HDC does not have enough space to accommodate new entries. In this case, HDC should execute the swapping job. Before swapping out a certain data, HDC increases the space by reducing the range of the entries. The new range value is updated with the current average value stored in HDT. This average value is also updated at every referencing. This process is called *reduced_mode*, as shown in (Fig. 6(d)). A certain data to be abandoned is selected when data has a low degree of utilization statistics calculated by the product of volume and reference count from HDT. That is, the data which has the lowest value of product result is taken out.

5. Experimental Result

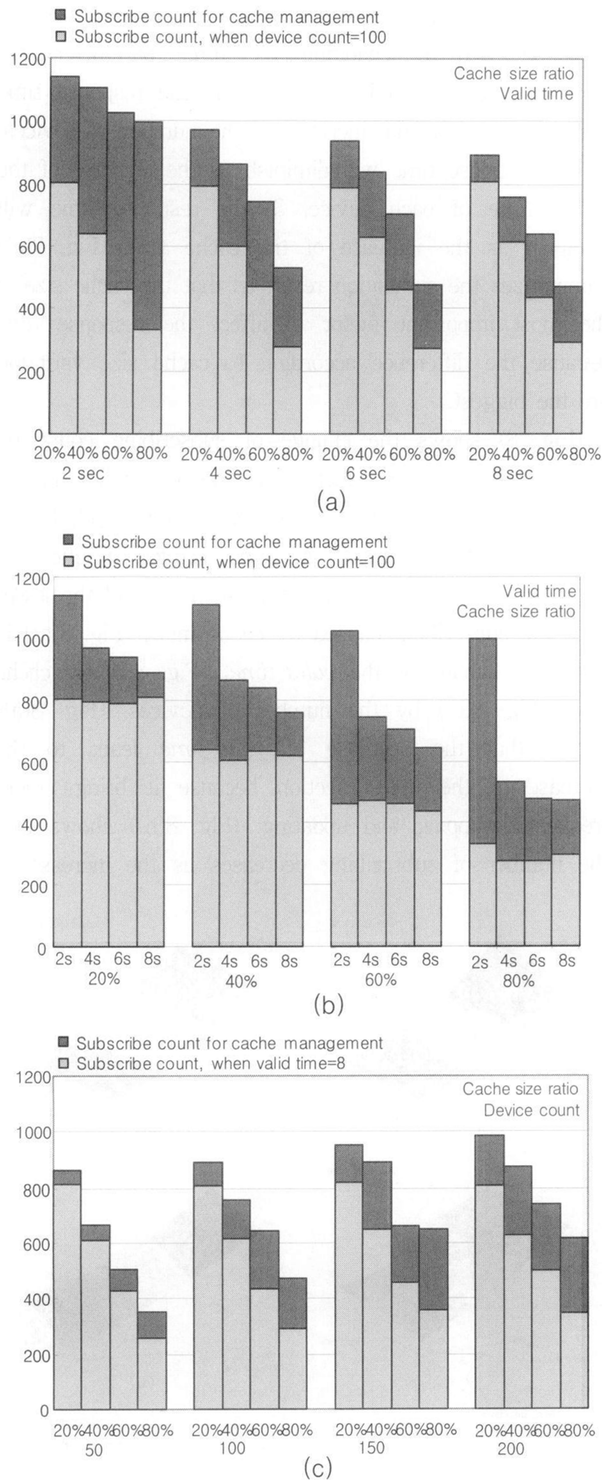
We implemented the prototype of UIM by using JDK 5.0. To store the trace data for tracking request, we use the MySQL. The data for HDC is managed in memory by using in-memory database. For the simulations, we selected three parameters such as the number of devices, cache size, and the *valid_time* of each device. We collected 1,000 of real-time data from random devices for each parameter. In order to estimate the performance improvement, we counted the frequencies of subscribing and calculate the average response time.

(Figs. 7(a)-7(d)) show the variation of the average response time by changing the three parameters. In general, it is expected as follows: 1) the response time will take more as the increase of the number of devices; 2) the response time will diminish as the increase of the *valid_ time* of each device; 3) the response time will diminish as the increase of the cache size. (Fig. 7(e)) summarizes the simulation result as that the cache size is the most important factor to affect the response time because the difference according to cache size variation are the biggest.

(Fig. 8) shows the change of subscribing count by changing the three parameters. In this figure, the names of x-axis appear at the top of right corner. Each bar is divided into two sections; the upper section signifies the overhead caused by the cache management and the lower does the subscribing caused by cache miss. (Fig. 8(a)) is plotted by changing the *valid_time*; (Fig. 8(b) by cache size; (Fig. 8(c)) by the number of devices. (Fig. 8(a)) shows that the decrease of *valid_time* leads to the increase of the upper section because it brings more frequent swapping and updating. (Fig. 8(b)) shows that the number of subscribing decreases as the increase of



(e)
(Fig. 7) Average response times



(Fig. 8) The number of subscribing

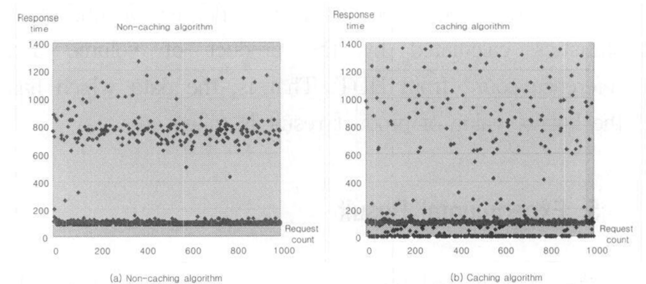
cache size. Also, we can recognize that the variation of *valid_time* does not affect the cache hit ratio, but the increase of cache size enlarges the upper section; this means that bigger caches require more handling overheads. (Fig. 8(c)) shows that the increase of devices leads to the increase of cache overhead. This result may change as the number of devices increase rapidly. Under these limited simulation parameter ranges, it is hard to attain correct overhead ratio.

(Fig. 9) shows the performance result of history request processing. To estimate the caching algorithm, we stored 101,000 history data. That is configured with 100 box packages; each package contains 1,000 objects; the event period is assumed as 10 days of supply chain scenario. We measured the average response time of 248.92ms, as shown in left side graph without caching. With the caching algorithm, we measured 198.88ms of the average response time. As a result, we get the requests of 67.1% have the less response time than accessing the database. We found that there is more room to be optimized with enhanced caching mechanism.

Through the experiments, the effectiveness of the caching algorithm is examined on the collected data from heterogeneous system, accounting for respective validity of data. The caching mechanism aims to reduce the response time on processing realtime queries in integrated RFID middleware. The superiority of the proposed scheme can be verified the following equation:

$$\text{Average Response Time} = (\text{Hit Ratio} * \text{Cache Search}) + (1 - \text{Hit Ratio})*(\text{Cache Search}+\text{DB Search}). \quad (2)$$

Here, the response time has gains in proportion as the increase of hit ratio and degradations; on the contrary, it degrades as increasing of miss ratio. Therefore, the result that gain becomes larger than degradation when taking the caching scheme shows its superiority.



(Fig. 9) Response times for history request

6. Conclusions

In this paper, we introduced an efficient caching mechanism appropriate for RFID middleware which can integrate WSNs and RFID systems. Although RFID middleware sitting between applications and low-level sensor nodes have been a hot research area, only few shows the interest of coupling two important components

of sensor networks. Even worse is that no research has tackled the issues of caching mechanism in RFID middleware. In future sensor networks, we can easily expect that a significant amount of data processing is required in RFID middleware because of large numbers of sensors and complex and multiple applications. Consequently, an efficient middleware layer equipping with caching mechanism is inevitably necessary for low latency of request-response while processing data from sensors and database. For this purpose, the proposed caching mechanism includes two optimization methods to reduce the overhead of data processing in RFID middleware based on the classical cache implementation polices. We conduct a number of simulation experiments under different parameters. The simulation results show that the proposed caching mechanism contributes considerably to fast request-response times.

Acknowledgement

This work has been partly supported by the BK21 Research Center for Intelligent Mobile Software at Yonsei University in Korea.

References

[1] D. Abadi, W. Lindner, S. Madden, and J. Schuler. "An Integration Framework for Sensor Networks and Data Stream Management Systems," Proceedings of VLDB, Toronto, 2004.

[2] J.E. Hoag and C.W. Thompson, "Architecting RFID Middleware," IEEE Internet Computing, Vol. 10, Issue 5, pp.88-92, Sept.-Oct., 2006.

[3] K. Aberer, M. Hauswirth and A. Salehi, "The Global Sensor Netoworks middleware for efficient and flexible deployment and interconnection of sensor networks," Technical report LSIR-REPORT-2006-006. Lausanne, Switzerland, 2006.

[4] Y. Yu, B. Krishnamachari, and V.K. Prasanna, "Issues in Designing Middleware for Wireless Sensor Networks," IEEE Network, Vol. 18, Issue 1, pp.15-21, Jan.-Feb., 2004.

[5] Y. Jin, W. Qu, and K. Li, "A Survey of Cache/Proxy for Transparent Data Replication," In Proc. Second International Conference on Semantics, Knowledge, and Grid, pp.35-35, Nov., 2006.

[6] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. "Implementing a Caching Service for Distributed CORBA Objects," In Proc. of Middleware 2000, pp.1-23, April, 2000.

[7] J. Eberhard, and A. Tripathi. "Efficient Object Caching for Distributed Java RMI Applications". In Proc. Middleware 2001, LNCS 2218, pp.15-35, 2001.

[8] R. Tewari, M. Dahlin, H. Vin, and J. Kay. "Design considerations for distributed caching on the Internet". In Proc. IEEE 19th Int. Conf. on Distributed Computing Systems, pp.273-284, 1999.

[9] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. "A Middleware System Which Intelligently Caches Query Results," In Proc. of Middleware 2000, pp.24-44, April, 2000.

[10] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. "Adaptive Push-Pull: Disseminating Dynamic Web Data," In Proc. of the 10th Int. WWW Conf., Hong Kong, China, pp.265-274, May, 2001.

[11] M. Franklin, S. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong, "Design Considerations for High Fan-in Systems: the HiFi Approach," Proc. of the 2nd CIDR Conference. Asilomar, California, U.S.A., 2005.

[12] H.S. Lee, H.H. Choi, B.S. Kim, M.C. Lee, J.H. Park, M.Y. Lee, M.J. Kim, and S.I. Jin, "UbiCore: An Effective XML-based RFID Middleware System," Journal of KISS: Database, Vol. 33, No. 06, pp.578-589, Korea, 2006.

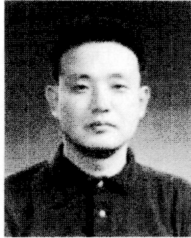
[13] D. Draper, P. Fankhauser, M.F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler, XQuery 1.0 and XPath 2.0 Formal Semantics, W3C Work Draft, 2004.

[14] X. Yu, K. Niyogi, S. Mehrotra, and N. Venkatasubramanian, "Adaptive middleware for distributed sensor networks," IEEE Distributed Systems Online, May, 2003.

[15] EPC Global., The Application Level Events (ALE) Specification, Version 1.0, http://www.epcglobalinc.org, 2005.

[16] EPC Global, EPC Information Services (EPCIS) Version 1.0, http://www.epcglobalinc.org, 2005.

[17] Verisign, The EPC Network: Enhancing the Supply Chain, Whitepaper, 2004.



김 정 길

e-mail : tetons@yonsei.ac.kr

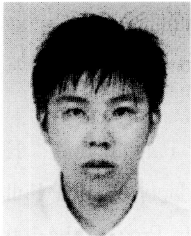
2003년 연세대학교 컴퓨터과학과 (공학석사)

2006년 연세대학교 컴퓨터과학과 (공학박사)

2006년~2007년 연세대학교 컴퓨터과학과 BK21 박사후 연구원

2007년~현 재 연세대학교 컴퓨터과학과 BK21 연구교수

관심분야 : 컴퓨터구조, 멀티미디어 내장형 시스템, 병렬처리, RFID 미들웨어



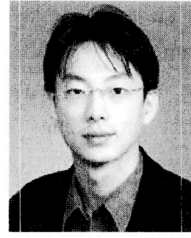
이 준 환

e-mail : jhlee@parallel.yonsei.ac.kr

2006년 중앙대학교 컴퓨터공학과(학사)

2007년~현 재 연세대학교 컴퓨터과학과 석사과정

관심분야 : 유비쿼터스 컴퓨팅, WSN, RFID 미들웨어



박 경 랑

e-mail : lanx@yonsei.ac.kr

2002년 광운대학교 컴퓨터공학과(학사)

2004년 연세대학교 컴퓨터과학과 (공학석사)

2007년~현 재 연세대학교 컴퓨터과학과 박사과정

관심분야 : 분산 시스템, 그리드 컴퓨팅, 유비쿼터스 컴퓨팅

김 신 덕

e-mail : sdkim@cs.yonsei.ac.kr

1982년 연세대학교 전자공학과(학사)

1987년 University of Oklahoma 전기공학과(공학석사)

1991년 Purdue University 전기공학과 (공학박사)

1993년~1995년 광운대학교 컴퓨터공학과 조교수

1995년~현 재 연세대학교 컴퓨터과학과 교수

관심분야 : 고성능 컴퓨터 구조, 그리드 컴퓨팅, 유비쿼터스 컴퓨팅 등