Publisher-Subscriber: An Agent System for Notification of Versions in OODBs

K. Palanivel
Computer Centre
Pondicherry University
Puducherry – 605014, India
kpalani.cce@pondiuni.edu.in

V. Amouda
Department of Banking Technology
Pondicherry University
Puducherry – 605014, India
vmouda.dbt@pondiuni.edu.in

S. Kuppuswami
Department of Computer Science
Pondicherry University
Puducherry – 605014, India
skswami.cse@pondiuni.edu.in

***Abstract*: In object-oriented databases, the conceptual structure may not be constant and may vary due to variety of reasons like correcting mistakes, adding/removing features, updating database, etc. Class versioning is one the evolution strategy that addresses the above issues. However, class versioning may be yields some unexpected results to the database clients. The easiest method is to just set a cache to expire periodically depending on the types of data. This solution is approximation and tends to either refresh too often so that the data is unnecessarily sent over the network or not often enough so that the data becomes stale. Query notification is the solution to avoid frequent access to the database and hence proposed to design agent system for monitoring class versioning. Agent Technology has generated lots of excitement in recent years because of its promise as a new paradigm for conceptualizing, designing and implementing software systems. Software agents coordinate the application that run on databases and allows performing the sequence of events like flushing cache, database operations, monitoring, etc. This agent system consists of a publisher agent, subscriber agents and a database update agent. These agents monitor the database for any change that would change data the data in the query or its structure. When it detects such a change, the publisher agent generates a change notification message and subscriber agent subscribes the notification message. To implement this system, a database application was developed with two versions and applied the database change notifications. To promote more reusability and scalability, a publisher-subscriber pattern was integrated with this application.**

Keywords: OODBS, Class Versioning, Instance Adaptation, Aspect-oriented Programming, Database Change Notifications

I. INTRODUCTION

Object-oriented databases (OODBs) evolved from a need to support object-oriented programming. OODBs allow for the storage of complex data structures that cannot be easily stored using conventional database technology and support all the persistence necessary when working with object-oriented languages. OODBs contain active object servers that support not only the distribution of data but also the distribution of work [14]. The conceptual structure of an object-oriented database may not remain constant and may vary to a large extent. The need for these variations (evolution) arises due to a variety of reasons like correcting mistakes in the database design, adding new features during incremental design or changes in the structure of the real world artifacts modeled in the database. This anomalous behavior can arise as a consequence of evolution.

Class versioning is one of the various evolution strategies employed to address the above issues. A new version of a class is created upon any modification. When an object is accessed using another type version (or a common type interface) it is either converted or made to exhibit a compatible interface [15]. This is termed *instance adaptation*. A new flexible instance adaptation approach introduced using update/backdate aspects with selective lazy conversion [28]. This makes it possible to make cost-effective changes to the instance adaptation strategy. Class versioning problems may also yield unexpected results which the agent technology may not be able to repudiate. The new class versioning may be yields some unexpected results to the database clients. When a new version is introduced, it is notified by the designers/ developers of the database applications [27]. Here, the simple method is introducing caching mechanism. Caching is the retention of data, usually in the application, to minimize network traffic flow and/or disk access. The caching mechanism tends to either refresh too often so that the data is unnecessarily sent over the network or not often enough so that the data becomes stale. Query notification is the solution to avoid frequent access to the database.

Recently, software agents coordinate the application that run on databases and allowing that database to perform the sequence of events, like flushing cache, database operations, monitoring, etc. Agent technology has for a long time been the domain of databases. Agents carry out information processes whereas databases supply information to processes. Agent communities seem a natural model for loosely coupled distributed systems, while database technology sees more and more of its mission in the support of highly distributed information systems [3]. In databases, version management is one of the problems and this can be solved using software agents. The main goal is to manage, change and organize class version sets in a distributed environment.

Hence, it is proposed to design an agent system for monitoring versions using database change notifications features. This agent system consists of publisher agent, subscriber agent and database update agent. The publisher agent generates and publishes notification messages. The subscriber agent subscribes the messages that have been published by the publisher agent. The database agent monitors the database and sends the notification messages. The communication between these agents should be *asynchronous*. Hence, it is proposed to use publish-subscribe model for performing asynchronous communication between agents in distributed environments [9].

Publish-subscribe systems suggest that they are highly scalable and are efficient means of communication in large distributed database systems [1]. Ppublisher/subscriber system is more flexible in distributed environment. While developing database application in distributed environment, it encounters a situation where many entities are interested in occurrence of a particular event. This system introduces a strong coupling between the publisher and subscriber of this event change notification. Thus whenever a new entity needs the information, code for the publisher of the information also needs to be modified to accommodate the new request. The Publish-Subscribe pattern [31] solves the tight coupling problem. Here the coupling is removed by the publisher of information supporting a generic interface for subscribers.

The proposed system is called publisher-subscriber agent system. The main function of this agent system is to monitor different versions of the database introduced by the developer/designer. It is a layered architecture and consists of subscriber layer, agent service layer, application layer and storage layer. All subscriber or client agents are exists in the subscriber layer. The agent service layer starts/stops the services like event notification, monitor, logger and timer. The application layer includes various modules as prescribed in section 4 and finally the storage layer stores databases. In order to re-use the designed system, we consider modularity, generality, and flexibility. Currently, this system contains an integrated OODBs and a user-interface capability.

This paper is organized as follows: Chapter 2 introduces the concepts of OODBS and class versioning in OODBS; the relationships between the agents and databases; and the technology publish-subscribe model deployed for distributed environment. Chapter 3 describes the concepts of database change notifications and the solutions of database change notifications. Chapter 4 explains in details of the architecture and the integration of publisher-subscriber pattern and finally concludes the paper.

II. BACKGROUND

Object-oriented databases (OODBs) allow objects that are manipulated by programs to be stored reliably so that they can be used again later and shared with other programs. The database acts as an extension of an object-oriented programming language such as Java, allowing programs access to long-lived objects in a manner analogous to how they manipulate ordinary objects whose lifetime is determined by that of the program. The two different approaches [24] to modifying the conceptual structure of an object database, broadly categorized in, are *schema evolution* and *class versioning*. In the schema evolution approach [12], the database has one logical schema to which modifications of class definitions are applied. Instances are converted (eagerly or lazily, but once and forever) to conform to the latest schema. In the class versioning approach, multiple versions of a schema or class can coexist [2]. Instance adaptation approach was used in many object oriented database systems like ENCORE, ADVANCE [13] and CLOSQL[25]. These database systems address the problems of schema evolution.

A. *Class Versioning*

In OODBs, the classes can be versioned, and the set of versions of one class is called the *version set* of the class. In addition to a version set for each class, there is also a *version set interface* for the class. As new versions of the class are created, the version set interface is extended to add extra attributes [4]. This object versioning is extended to the versioning of class definitions. These systems adopt exception handling to cope with mismatches between the version of the object expected by the query and the actual version of an instance. The exception handlers service the query with values appropriate to the version of the class demanded by the query. Also, a simple schema change such as –*change the name of an attribute*– could in reality mean - replace an attribute with a new attribute with a new name, which means something different to the old attribute, but is loosely related to it in some way [12].

Versioning problem can be solved using aspect-oriented programming techniques [19]. This technique for the separation of crosscutting, customizable features such as instance adaptation (simulated or physical conversion of objects in accordance with schema changes), versioning, links among persistent entities, change propagation and referential integrity semantics. The aspect-oriented approach has been employed to provide cost-effective, localized changes during both schema evolution and customization in the object database evolution system [14]. Here, the actual version to which a particular instance belongs at any one time is irrelevant as far as the end user is concerned because any query will automatically convert the instance to the version implied in the *query*. The version to which instances are converted will normally be the current (latest) version of the class. The class version used is determined by the queries attempting to extract the instances from the system.

Many database applications cache the information to avoid accessing the database on every request. Refreshing the cache too frequently will needlessly burden the backend databases. Refreshing infrequently can lead to application errors due to data inconsistency between the cache and truth stored in the database [22]. Database change notifications [17] allow a cache client to subscribe to changes in a query result set and receive notification when any underlying change in the database would alter results of the query the cache has subscribed to. With database change notifications, the client cache can avoid periodic refresh and can react quickly to change that invalidate the cache [22].

B. *Database Change Notifications*

One of the most useful ways to improve the performance of widely distributed, loosely coupled applications is to cache data. By caching rarely changing and easily refreshed data at a service provider or consumer that is remote from the source database, messages can be lighter weight and use less bandwidth [22]. Such a system needs a way to refresh data when data changes at the source. There are plenty of ways to do this. Probably the easiest method from a programming perspective is to just set a cache to expire periodically, anywhere from a few minutes to a few weeks, depending on the type of data. But such schemes are approximations, and tend to either refresh too often so that data is unnecessarily sent over the network or not often enough so that the data becomes stale.

In many situations, the most efficient method is to simply notify the keeper of each cache that the data has changed, and either push the data out from the source database or pull it from the service program. A service program can use its own logic and choose to ignore the notification, for example, if it determines that the data doesn't need to be refreshed or if the refresh can be deferred to off-peak hours. The general workflow for a query notification is presented in *figure 1*.

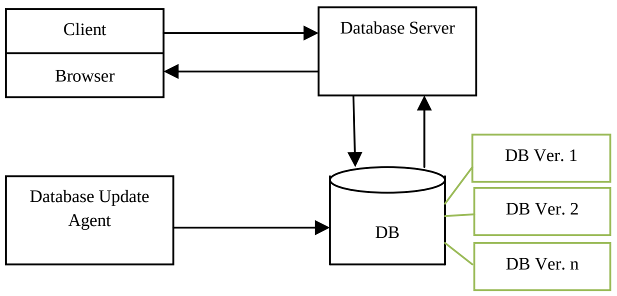

Figure 1: Workflow of Database Change Notification for Versioning

Subscribers register their interest in specific events with the application server. Upon receiving a notification (message) from a publisher, the application server forwards the notification to the relevant client subscribers by comparing the message content with the list of subscriptions it holds. It provides the flexibility to operate in a dynamic environment and is independent of the need to configure information relating to the subscriber of a notification [6]. The notification itself is encapsulated within an object and contains a list of key value pairs. A publisher expresses its interest with a subscription element, which is built with a special subscription language containing simple logical expressions [36]

C. *Agents and Databases*

Developing database applications using autonomous agents have a highly distributed database. However, for the purpose of studying the effects of agents on database technology, abstract agents to just that, local databases and the challenge is how the abstraction from the agent properties is reflected in their databases. A multi-agent system presents us with a distributed database that has distinctive characteristics [4].

A multi-agent system comprised of multiple autonomous agents/components needs to have certain characteristics like each agent has incomplete capabilities to solve a problem; there is no global system control; data is decentralized; and computation is asynchronous. That is, combining multiple agents in a framework presents a useful software engineering paradigm where problem-solving components are described as individual agents pursuing high-level goals. A multi-agent system can be considered as a loosely coupled network of problem solver entities that work together to find answers to problems that are beyond the individual capabilities or knowledge of each entity [10].

In multi-agent systems, the communication between agents should be more powerful because they share similar data without having prior knowledge of each other. A pure, fast, scalable, lightweight and publish-subscribe model [23]with a rich and expressive language for message subscription is useful for disseminating large numbers of short notifications efficiently between agents in a large-scale multi-agent system [5]. The fundamental principles of the general publish-subscribe model and its use for agents' remains, however, the same for other middleware, and so use the name. This architecture is similar to client-server architecture, where server is responsible for managing client connections and transferring messages between publishers and subscribers [21].The architecture of publisher-subscriber model using agents is shown in *figure2*.

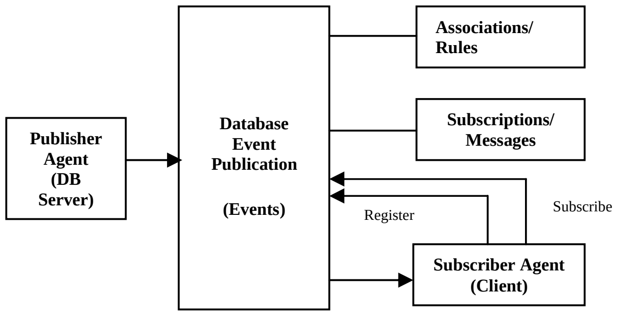
Associations/ Rules
Subscriptions/ Messages
Publisher Agent (DB Server)
Database Event Publication (Events)
Register
Subscribe
Subscriber Agent (Client)

Figure 2: Functionality of Publish-Subscribe Model

It is proposed to integrate existing OODBS with database change notifications, in order to provide scalable change notification within the global environment [30]. This integration will allow client applications to go beyond the traditional lookup/update scheme, where passive queries are issued over past and present data. In this model, local DBMSs advertise details of changes to data that they are willing to notify to clients, and publish events through publish-subscribe system when such changes occur. Clients can subscribe to one or more of these change events, or specify restrictions to them by providing attribute based filters. In consequence, they are informed in a timely manner when particular state changes occur in some database. The publish-subscribe communication model used in our integrated architecture allows database notifications [5] to be disseminated without requiring clients to know the location of the information sources involved.

III. PUBLISHER-SUBSCRIBER AGENT SYSTEM

A versioning management system is described for use in connection with a database management system to facilitate versioning of classes, the system including a class and a version control module. The version management is configured to, in response to a user query related to the classes and related to a version, generate an augmented query for processing by the data base management system, the augmented query relating to the user query and the version control information.

Designing a publisher-subscriber agent system for version management using the publisher/subscriber [7] requires mapping the abstractions of that model like *publisher*, *subscriber*, *message*, etc. Now, the problem consists of making sure that a message is sent to all potential subscribers, i.e., each subscriber has a chance to get all messages it is interested in. The message may be as a multicast message or send a broadcast message. This problem is tackled by maintaining a replicated subscription registry at the publishers, i.e., each publisher knows to which subscribers it has to send a message. All subscribers register at the channel object. Thus, the channel object acts as interface between the publishers and the subscribers.

Publishers and subscribers are internally represented as agents. There is a distinction between an agent and a client. The attributes of a client include the physical process where the client programs run, the node name, and the client application logic. There could be several clients acting on behalf of a single agent. Also, the same client, if authorized, can act on behalf of multiple agents. A database server behaves as a publisher agent and advertises its service to other agents. Other agents may subscribe to the service and receive notification of events published by the publisher agent [8].

The Student Database System with n tier-architecture [14] shown in *figure 3* is broken up into separate logical layers namely subscriber layer, agent service layer, application layer and storage layer. Each layer with a well defined set of functions and interfaces. The subscriber agent exists in the subscriber layer. The subscriber registers/unregisters with application server. The agent service layer offers *event notification, monitor, logger and timer services* [29]. Event notification is a publisher-subscriber kind event type based notification service. Event notification is built on top of event notification system service [11]. Monitor provides polling mechanism to monitor registered resources periodically. Logger provides the application a mechanism for selectively logging to certain activities or events and Timer provides time based notification services.

The application layer consists of version manager, persistence object manager, user manager, event manager and query manager. The *version manager* allows the administrator to view information about the different database version and permits the administrator to upgrade to the latest version from the interface. The *persistence object manager* deals with database objects. This includes compiled versions, modules, types, schemata, arbitrary texts, graphic screens, diagnostic information, system messages, etc. All such objects are stored on equal rights and are accessible by standard mechanisms. The *user manager* allows creating different types of users and maintaining them. The clients/users create a new registration or unregistration for the web site. It replicates all the actions that a new user would go through at the front-end of the web site when registering for an account with limited access controls.

The *event manager* assists the database user in establishing, relating, and managing an unlimited number of events, clients, and activities in a manner which facilitates communication with clients, and the distribution of communications and activity. The *query manager* is a database program that can be used to access data from a relational database. It organizes data sets from multiple studies into a consistent and standardized structure, thereby improving data delivery and ease of interpretation for coastal resource managers. Clients can sort and examine data in a variety of ways by selecting from a menu of preprogrammed queries that evaluate individual contaminants, contaminant groups, comparisons to common toxicological benchmarks, user-defined ranges, and many other filters. These queries sort and analyzed the data to produce summary output table. Various functions of this architecture include *registration/unregistration*, *association*; *attach/detach subscriber*, and *rules.*

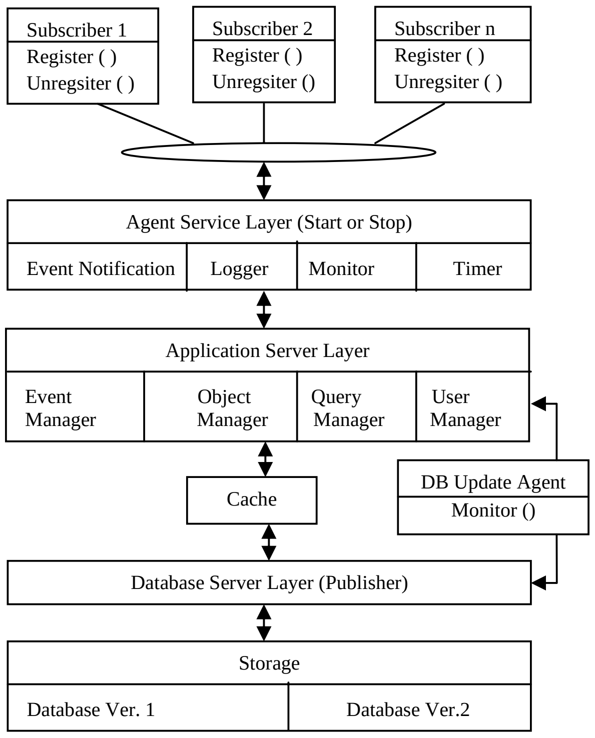

Figure 3: Architecture of Publisher-Subscriber Agent System

A registration is a persistent entity that exists on all nodes. If a subscriber down, then the registration continues to exist and will be notified when the table change. The *register()* method creates a new database change registration in the database server with the given options. It returns a *registration()* object which can then be used to associate a statement with this registration. It also opens a subscriber socket that will be used by the database to send notifications. After, explicitly unregister a registration to delete it from the server and release the resources in the driver. Unregister a registration using a connection different from one that was used for creating it. To unregister a registration, the *Unregister()* method should be defined. After creating a registration or mapped to an existing registration, associate a query with it. Like creating a registration, associating a query with a registration is a one-time process and is done outside of the currently used registration. The query will be associated even if the local transaction is rolled back. Associate a query with registration using the *set()* method and this method takes database change notification object as parameter. *Attach/Detach Subscriber* - to receive database change notifications, attach a subscriber to the registration. When a database change event occurs, the database server notifies the driver. The event contains the object id of the database object that has changed and the type of operation that cause the change. Depending on the registration options, the event may contain row-level detail information. The subscriber code then uses the event to make decision within it the data cache. Now, attach a subscriber to registration using *AddSubscriber()* method.

IV. INTEGRATING WITH PUBLISHER-SUBSCRIBER PATTERN

The publisher-subscriber pattern is applied to notify event handlers or subscribers when some interesting object or publisher changes state. The publisher-subscriber pattern[21] is used for situations in which many clients have to listen for information published by one or more servers. A publisher cannot distinguish between the subscribers which are listening on it; it therefore sends the information to all subscribers. A *Publisher* is an entity that creates *NotificationMessages*, based upon Situation(s) that it is capable of detecting and translating into NotificationMessage artifacts. It does not need to be a Web service. A *Subscriber* is an entity that acts as a service requestor, sending the subscribe request message to a *NotificationPublisherr*. A *NotificationMessage* is an artifact of a Situation containing information about that Situation that some entity wishes to communicate to other entities.

The motivation behind to use design pattern [18] is constructing and maintaining objects to associate changes in the remote objects with cached objects. If data or database in a remote data source changes, database change notifications are used to trigger a dynamic rebuild of associated objects. The updation or change includes storing new version of object or database in the cache or deleting an object from the cache. This publisher-subscriber pattern is parameterized in *figure 4.*

The *Subscriber class* is an abstract base class may specify subscriptions on a queue using a rule. Subscribers are durable and are stored in a catalog. *Database Event Publication* is the database event class represents a significant source for publishing information. It detects and publishes events and allows active delivery of information to end-users in an event-driven manner. *Registration* is the process of associated delivery information by a given client, acting on behalf of an agent. There is an important distinction between the subscription and registration related to the agent/client separation. Subscription indicates an interest in a particular queue by an agent. It does not specify where and how delivery must occur. *Publishing a Message* is a publisher which publishes messages to queues by using the appropriate queuing interfaces. The interfaces may depend on which model the queue is implemented on. *Event* is an abstract base class to generally represent an event. Its source field stores a reference to the object that fired the concrete *Event*. That is usually a concrete *Publisher*, but could also be another event source connected to the concrete *Publisher*. *Publisher* is an abstract base class that frees concrete *Publisher* objects from having to maintain the infrastructure for event notification. *Subscriber* is a subclass that allows registering its interest in database events. Whenever the database events generate an event, it will push the event to subscriber.

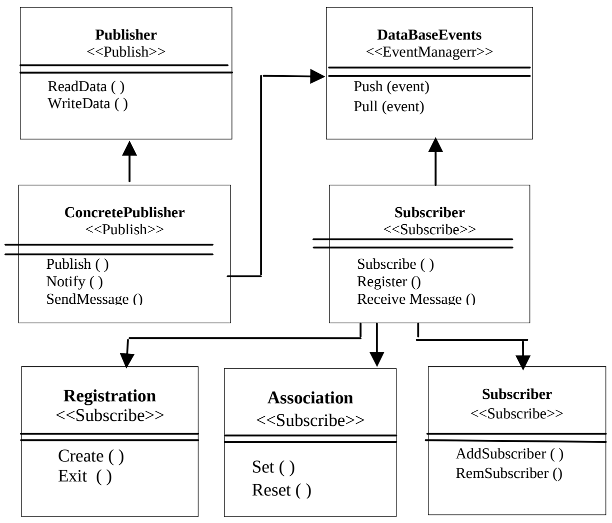

Figure 4: Structure of Publisher/Subscriber Pattern

Implementation using publisher-subscribe pattern that addresses the problems of binding subscribers to publishers, of routing and filtering of messages, as well as reliability, efficiency and latency of message delivery. The advantages of using this pattern are lowered coupling, improved security, improved testability and high degree of scalability [24]. The benefits of publish-subscribe notification for web services are detailed in [32]. Also, the benefits of the publish-subscribe pattern are agents can be varied and reused independently without interfering with their respective subscribers or publishers; state changes in one agent can trigger state changes in other agents without knowing how many agents need to be changed and agents can notify other agents without knowing about the other agents, and avoid tight coupling.

V. CONCLUSION

In distributed environment, database change notification is one of the features of structure query language. Using this feature, the client knows immediately if any changes made in the database. This paper provided a solution for class versioning in OODBs using database change notification. Existing evolution approaches in OODBs are committed to a particular instance adaptation strategy and the flexibility is achieved by encapsulating the instance adaptation code through cross-cutting concerns and update/backdate aspects. Here, software agents are deployed for monitoring and notifying the evolution of class versions. This agent system uses publish/subscribe model that improve robustness and flexibility in distributed environment. This model provides the agents are built to proactively report their context to the event system. Also, a publisher-subscriber design pattern is integrated with this application that promotes more reusability and ease maintenance.

This paper demonstrated the applicability of the ideas by developing an agent system with publishes-subscribe mechanism, and integration of a design pattern with agents system. This system was designed and implemented using agent-oriented architecture and partially with services. To provide better flexibility and more reusability, we may use service-oriented architecture. However, the challenge is to support more sophisticated standing queries and to provide better optimization techniques that handle the vast number of queries and vast data volumes.

REFERENCES

[1]. Berthold Reinwald Hamid Pirahesh Tobias Mayr Jussi and Feng Tian, Myllymaki, *Implementing A Scalable XML Publish/Subscribe System Using Relational Database Systems*, IBM Almaden Research Center 650 Harry Road, San Jose, CA, 95120.

[2]. Zui Li and Zahir Tari, *Class Versioning for the Scheme Evolution*, Department of Computer Science ,Royal Melbourne Institute of Technology, Australia.

[3]. Moira Noira and Mario Magnannelli, *Databases for Agents and Agents for Databases*, Institute of Information Systems, Swiss Federal Institute of Technology, Switzerland.

[4]. Hasan Davulcu, Zoé Lacroix, Kaushal Parekh I. V. Ramakrishnan, Nikeeta Julasana, *Exploiting Agent and Database Technologies for Biological Data Collection*, State University of New York, Stony Brook.

[5]. Amir Padovitz, Arkady Zaslavsky, Seng Wai Loke and Milovan Tosic, *Agent Communication Using Publish-Subscribe Genre: Architecture, Mobility, Scalability and Applications*, Annals of Mathematics, Computing & Teleinformatics, Vol. 1, No. 2, 2004, PP 35-50 35.

[6]. Badrish Chandramouli Junyi Xie Jun Yang, *On the Database/Network Interface in Large-scale Publish/Subscribe Systems*, Department of Computer Science, Duke University, Durham, NC 27708, USA.

[7]. Pablo R. Fillottrani, *The multi-agent system architecture in SEWASIE*, Faculty of Computer Science, Free University of Bozen/Bolzano, JCS&T Vol. 5 No. 4 December 2005.

[8]. Fabio Melo, Ricardo Choren, Renato Cerqueira, Carlos J. P. de Lucena, Marcelo Blois, *An Agent Deployment Model Based on Components*, Faculdade de Informática, PUCRS, Rio Grande do Sul, Brasil, PUC-RioInf.MCC 37/03 October, 2003.

[9]. Roy Danial, Anciax Didier, Monterio Thibaud, Ozuzi Lautifa, *An Agent Deployment Model Based on Components*, University de Metz, France.

[10]. Katia P. Sycara, *Multiagent, Systems*, AI magazine Volume 19, No.2 Intelligent Agents Summer 1998.

[11]. Ozgur Koray Sahingoz And Nadia Erdogan, *Agvent: Agent Events*, Air Force Academy, Computer Engineering Department, Istanbul, Turkey.

[12]. Simon Monk and Ian Sommerville. *Schema Evolution in OODBs Using Class Versioning*, SIGMOD RECORD, Vol. 22, No. 3, 1993.

[13]. Bjrrnerstedt, A. and S. Britts, *AVANCE: An Object Management System*, Proceedings of OOPSLA'88, pp.206-221. 1988. Skarra, A. H. and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database" Proceedings of OOPSLA'86, pp.483-495. 1986.

[14]. Eduardo Casais, "Managing Class Evolution in Object-Oriented Systems", Object Oriented Software Composition, O. Nierstrasz and D. Tsichritzis (Eds.), Prentice Hall, 1995.

[15]. Skarra, A. H. and S. B. Zdonik, *The Management of Changing Types in an Object-Oriented Database,* Proceedings of OOPSLA'86, pp.483-495. 1986.

[16]. Willy Farrell, "Introducing the Java Message Service", IBM, June 2004.

[17]. Don Kiely, "How SQL Server 2005 Enables Service-Oriented Database Architectures", SQL Server Technical Article, November 2005.

[18]. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design patterns Elements of Reusable Object-Oriented Software", Pearson Education, Inc., 2002.

[19]. G. Kiczales, J. Lamping, A. Mendhekar, C. Lopes, J. Loingtier and J. Irwin., *Aspect-Oriented Programming,* Proceedings of European Conference on Object-Oriented Programming Finland. Springer-Verlag, 1997.

[20]. Brain Rush, *Event Management/Logging with Publisher-Subscriber Pattern*, Article, Developer ASP Network, 2008.

[21]. Carlos O"Ryan, *Emprical Evolutions of Design Patterns for Publisher-Subscriber Distributed Systems*, Ph.D. Dessertation, University of California, Sprint 2002.

[22]. Cesar Galindo Legaria, Torsten Grabs, Christian Kleinerman, Florian Waas, *Database Change Notifications: Primitives for Efficient Database Query Result Caching*, Proceedings of the 31st international conference on Very large data bases, PP:1275 - 1278, 2005.

[23]. Berthold Reinwald Hamid Pirahesh Tobias Mayr Jussi Myllymaki, *Implementing A Scalable XML Publish/Subscribe System Using Relational Database Systems*, IBM Almaden Research Center , 650 Harry Road, San Jose, CA, 95120.

[24]. M. Dmitriev, *Safe Class and Data Evolution in Large and Long-Lived Java Applications*, Technical Report TR-2001-98, Sun Microsystems, 2001.

[25]. S. Monk, "The CLOSQL Query Language", Computing Dept. Lancaster University, Lancaster Technical Report No.SE-91-15

[26]. Oracle® Database Application Developer's Guide – "Fundamentals, Developing Applications with Database Change Notification", 10*g* Release 2 (10.2).

[27]. Ratnakant, *Flexible Instance Adaptation for Object-Oriented Databases*, M.Tech. Theses Report, Department of Computer Science, Pondicherry University, Puducherry, India, June 2006.

[28]. S.Kuppuswami, K. Palanivel, V. Amouda,, *Applying Aspect-Oriented Programming for Instance Adaptation in Object-Oriented Databases*, International Conference of Advanced Computing and Communications (ADCOM), Indian Institute of Technology, Guhati, Inidia, Dec, 2007.

[29]. Steve Graham, Peter Niblett, *Publish-Subscribe Notification for Web services*, Copyright Akamai Technologies, Computer Associates International, Inc., and The University of Chicago 2004.

[30]. Vargas, L.; Bacon, J.; Moody, K., "*Integrating databases with publish/subscribe*", Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on Volume, Issue, 6-10 2005 Page(s): 392 – 397.

[31]. Neil Loughran, Awais Rashid, Ruzanna Chitchyan, "A domain Analysis of Key Concerns – Known and New Candidates", AOSD Europe, eu Network of Excellence, Feb 2006.