Integrating Databases with Publish/Subscribe

Luis Vargas Jean Bacon Ken Moody
Computer Laboratory, University of Cambridge
{firstname.lastname}@cl.cam.ac.uk

Abstract

Publish/subscribe is emerging as an appropriate communication paradigm for large-scale, widely-distributed systems. In this paper, we describe our work on integrating active databases with publish/subscribe, using PostgreSQL and Hermes as the experimental context. In the proposed architecture, each database manager defines and advertises change events, in contrast with a continuous query model. Advertised events, which may span a number of physical relations, correspond to the virtual relations of a security view. Clients subscribe to events of interest, and can refine their subscriptions through content-based filter expressions. An event is published whenever a database change, detected via a dynamic triggering layer, matches some active subscription. Security and routing of database events are handled in the same way as for conventional Hermes events.

Key Words

active databases, publish-subscribe, event-based systems

1. Introduction

The proliferation of networked environments and the use of database systems across organizations in recent years have led to the creation of innumerable information systems that follow the Client-Server communication model. In these systems, a server executes actions, or delivers data to clients, only when explicitly requested to, by means of queries or transactions issued in a passive, client-pull-based approach. Most online transaction processing (OLTP) systems today follow this mode of operation. It has proved to work well in the traditional context of local area networks (LANs), used with a moderate number of clients and servers in corporate environments. This relatively small-scale use has avoided some of the limitations imposed by the latency of the request/reply paradigm, but these limitations are inherent in the Client-Server model when used for conventional query/update systems.

The massive growth of the Internet and the exponential increase in the number of interconnected parties, both static and mobile, has called this style of use into question. What were once closely controlled environments have evolved into large scale collaboration spaces that are both highly distributed and dynamic. This evolution has led to new requirements, as well as to exciting and novel application domains for distributed information systems. Examples arise in situation assessment, business process (workflow) control, network management, and access control policy evolution, to name but a few areas. These areas share the same basic requirement of tracking changes at a collection of information sources in order to detect situations of interest; what is new in today's world is that this set of information sources may be globally distributed and dynamic. The information system must identify relevant changes and notify them to users or client applications, without requiring them to know a priori where and when to look for data. Thus, the challenge for these change management systems is to provide the necessary mechanisms and strategies for the scalable, reliable, and secure monitoring, processing, and notification of changes at the relevant information sources.

The Publish/Subscribe communication paradigm [1] can help to meet the challenge by providing efficient, scalable, many-to-many, push-based delivery of messages between the various parties in a system. We are therefore creating a secure, attribute-based pub/sub system in our EDSAC21 project (Event-Driven, Secure Application Control for the 21st Century). Given the large number of deployed database systems, we are exploring how to integrate existing inter-domain databases into this framework, in order to provide scalable change notification within the global environment.

This integration will allow client applications to go beyond the traditional lookup/update scheme, where passive queries are issued over past and present data, such as "tell me the price of all automatic cars currently on sale in Cambridge". In our model, local DBMSs advertise details of changes to data that they are willing to notify to clients, and publish events through the 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, e.g. "notify me of the price of any new automatic that comes up for sale in my town". The scope of queries may thus be future as well as existing data, and span a widely distributed, dynamic collection of databases. The publish/subscribe communication model used in our integrated architecture allows database notifications to be disseminated without requiring clients to know the location of the information sources involved.

In this paper we describe the approach that we are taking to achieve this integration. We are extending the built-in active facilities provided by the object-relational database management system (ORDBMS) PostgreSQL [2], which are similar to those available in most commercial RDBMS today.

Our system supports a set of fine-grained active predicates that can be integrated into our publish/subscribe communication system, Hermes [3], in the form of generated, typed events.

Sections 2 and 3 set up the background in active databases, the ORDBMS PostgreSQL, and the publish/subscribe paradigm, with a particular focus on Hermes. Section 4 describes our integrated architecture as well as its implementation and execution model. Section 5 outlines related work, and Section 6 mentions some future plans, before the paper is concluded in Section 7.

2. Active Databases

In traditional database systems, data is created, retrieved, modified and deleted, in response to requests issued by users or applications. This passive pattern cannot be used effectively to model requirements that involve monitoring and reacting to particular situations in the database, such as the automated creation of purchase orders for items that go below a low inventory threshold. Supporting such a requirement on a passive database system would need a polling mechanism to check the number of product units in stock periodically. The (complex) estimation of an appropriate polling frequency is of central importance. While a high value introduces costly overheads into the system, too low a value may result in late detection of, and reaction to, a critical situation.

Active DBMSs [4] extend "passive" DBMSs with the possibility of specifying reactive behaviour. Such behaviour, implemented by triggers, allows monitoring for and reacting to specific database circumstances at the DBMS itself.

An active database built on top of an active DBMS must specify a knowledge model and an execution model. The knowledge model defines the database reactive behaviour, generally in terms of event-condition-action (ECA) rules. The event part of a rule defines the situation that triggers the rule, the condition evaluates the context in which the event takes place, and the action formulates the task to execute after the rule has been triggered and its condition validated.

The execution model specifies how to treat a set of rules at run-time. It defines, for example, the rules' coupling mode and prioritization strategy. The coupling mode of a rule defines when its condition and action are evaluated and executed, relative to the event that triggers the rule. The prioritization strategy determines the order in which the system triggers multiple rules associated with the same event.

2.1. PostgreSQL

PostgreSQL is an open-source object-relational database management system supporting the ANSI SQL92 and SQL99 standards. Domain, referential, and transactional integrity, as well as multi-version concurrency control, are offered as part of its features. In PostgreSQL, active database functionality is provided in the form of triggers. A trigger, associated with a function (possibly parameterized) that implements it, can be defined to execute before, after, or instead of a data manipulation operation. This definition can be set either at a tuple-level, executing the trigger-associated function once per affected row, or at a statement-level, performing the function over the entire affected tuple set.

Because its operation is catalogue-driven, PostgreSQL can be extended in many ways, for example by adding new data types, functions, operators, or procedural languages.

3. Publish / Subscribe Systems

The publish/subscribe communication paradigm is well-adapted to the loosely coupled nature of large-scale distributed systems. This paradigm builds on the notion of an event; that is, a particular happening of relevance in the system. In a publish/subscribe system, subscribers express their interest in an event or a pattern of events, and are asynchronously notified when publishers produce them. This many-to-many data dissemination model removes for clients (publishers and subscribers) the need to know each other. All that is needed is that the definition of events of potential interest is advertised before such events are published in the system.

Publish/subscribe systems are classified as either (type-) topic-based or (attribute-) content-based. In topic-based systems, publishers generate events with respect to a topic or subject. Subscribers then specify their interest in a particular topic, and receive all events published on that topic. Defining events in terms of topic names only is inflexible and requires subscribers to filter events belonging to general topics. Content-based systems solve this problem by introducing a subscription scheme based on the contents of events. Siena [5], Gryphon [6], Rebecca [7], and Hermes are examples of scalable content-based publish/subscribe systems.

3.1. Hermes

Hermes is a distributed, content-based publish/subscribe architecture with an integrated programming model and strong message typing. It is built on a peer-to-peer routing substrate to provide scalable event dissemination and fault-tolerance. A distributed event-based system implemented on top of Hermes consists of two kinds of components: event clients and event brokers. Event clients (event *publishers* or event *subscribers*) use the services provided by the architecture to communicate using events. Event brokers form the application-level overlay network that performs event propagation using a novel type- and attribute-based routing algorithm.

Hermes enforces the strong typing of events; thus every published event (publication) in Hermes is an instance of an event type. Publishers define event types using a name and a list of typed attributes according to their data notification needs. These definitions are used for type-checking publications and subscriptions at runtime. Before publishing an event instance, a publisher must advertise the associated event type by sending an advertisement message to its local broker. This message is then forwarded to a special node, known as a rendezvous node, whose address is computed from the type-name using a distributed hash table (DHT) algorithm. Routing state for the advertised event type is set up in appropriate event brokers as a result of this action.

In our integrated database – publish/subscribe architecture, we have created an *adapter* component (described in Section 4.3) to connect the active layer of a database with Hermes.

4. Pub/Sub with Database integration

In this section we present our approach to integrating a number of PostgreSQL databases with Hermes, our publish/subscribe communication system, to form one global event-based system. Figure 1 gives an architectural perspective of the various components involved in the integration.

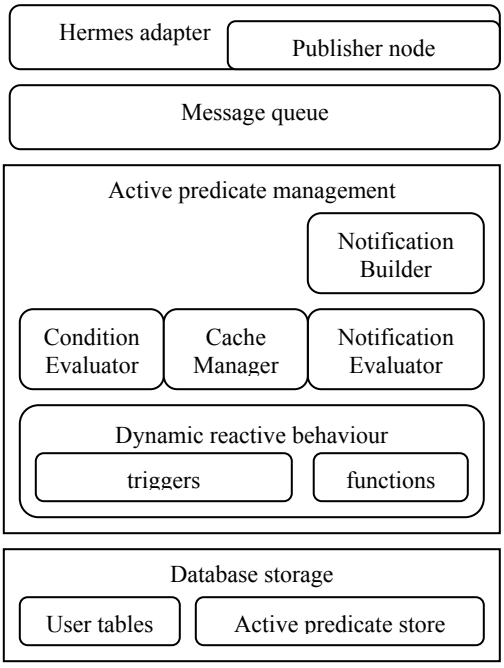

Figure 1. Integrated Database and Publish/Subscribe System Architecture

4.1. The Active Predicate Store

At the lowest level of the architecture is the database storage layer, where user tables reside and the active predicate store component is maintained. This component builds on a number of interrelated relational tables; it provides the required infrastructure for storing the definitions of the relevant conditions to monitor, the optional actions to be executed, and the notifications to be published on their occurrence.

The situations to be monitored, as well as optional actions to be executed on their detection, are expressed in the form of Event-Condition-Action (ECA) rules. These rules are based on the notion of transitions, where a transition is a database state change resulting from the execution of a data manipulation operation. A rule in the active predicate store is triggered by a given transition when its transition predicate, expressed in terms of the rule's event and condition definition, holds with respect to that transition.

The event part of a rule specifies a primitive SQL operation, such as "INSERT", "UPDATE", and "DELETE", on a particular relation, and a set of optional attributes. This specification works as a first filter for the rule, and only after a matching occurrence will the condition part of the rule be considered for evaluation. In order to provide the system with greater flexibility, we allow a rule to be checked before or after the signalling of an event.

The condition part of the rule is an optional, arbitrary predicate expressed as a standard Boolean SQL operation. It may involve the database state before or after the transition (depending on the definition), and the values in the state transition tables, which reflect the changes that occurred.

Finally, the action part of the rule defines an optional operation to be executed, after the rule's condition has been validated, and before (or after - as specified) any notification associated with the rule is generated. There are various predefined actions to choose from; for example, to disable a rule after it is triggered a number of times. Also, user defined actions written in any of the four procedural languages supported by the PostgreSQL server interface (pgSQL, Tcl, Perl, and Python) can be declared at any time, in the active predicate store, to be used as part of a rule.

An example of a simple rule, expressing interest in products that fall below their established threshold, is presented below.

```
name        R1
relation    Products
event       AFTER UPDATE (threshold, unitsInStock)
condition   SELECT OLD.unitsInStock >= threshold
            AND NEW.unitsInStock < threshold
action      NULL
```

Besides the different rules used to specify the many situations of interest in the database, the active predicate store allows the definition of notification types. These each contain a name and a schema that describes the type, and serve as the basic templates for any notification that is published from the database. The schema of a notification type is defined as a set of attribute-name/datatype pairs called notification attributes.

Creating a new notification type in the active predicate store causes the sending of an advertisement message to the Hermes adapter through the message queue. This XML-based message, which specifies the schema of the notification type (see example below), indicates the intention of the database to publish notifications of that type. A rendezvous node and a set of dissemination paths routed towards it from the Hermes adapter are set up by issuing an advertisement message in the Hermes event system. A corresponding unadvertisement message is used to undo a previous advertisement when a notification type is removed from the predicate store.

```
<hermes>
   <advertise type="ProductBelowThreshold">
         <att name="productId" type="varchar"/>
         <att name="unitsInStock" type="int"/>
         <att name="dateTime" type="timestamp"/>
   </advertise>
</hermes>
```

The active predicate store also maintains sets of associations between the different notification types and rules defined in the system. A stored association states that when a particular rule is triggered, a notification should be derived from the related type. Such associations are rendered in the standard manner for *n*-to-*n* relations as in:

```
rules_notificationTypes('R1',
                        'ProductBelowThreshold',
                        'SELECT NEW.productId,
                                NEW.unitsInStock,
                                now')
```

An extra field, known as the notification type source, has been attached to such associations. This field, expressed as a standard SQL SELECT operation, specifies where the data used to fill the attributes in a notification type should come from: the state of the database before or after the transition; the values of the state transition tables that reflect the changes on the transition; or both. A notification type's source thus represents an active virtual relation in some view of the database at a particular time. Database administrators can therefore normalise the underlying physical database model as they require, without affecting the definition of the notification type sources in the predicate store.

We overload the definition of the active predicate store tables, using PostgreSQL's rule base and built-in triggering, so that any change to the definition of a rule results in the automatic (re-)generation of both the extended triggering behaviour and the metadata used by the active predicate management layer for monitoring the selected relations.

Storing the definition of relevant database conditions in this way has many advantages over explicitly setting up triggers on each of the relations containing data to be monitored. Firstly, by making the configuration of the reactive behaviour on these relations automatic, the maintenance of the ECA rule system across the database is significantly simplified. Also, from a user perspective, the tables in the active predicate store behave exactly the same as other tables in the DBMS. This makes it simple to make the features of the store available through any SQL-compliant client console. To further assist the user we have devised a graphical interface for access to the store.

Because the semantics of trigger conditions, and the schema and source of system notifications, are defined and validated by the time they are introduced into the active predicate store, a number of optimization strategies can be considered at the active predicate management layer (see Section 4.2), e.g.

a) By promptly determining if a particular change in a relation should be considered as relevant, based on the detailed description in the active predicate store (expressed in terms of relation, event, affected columns, and time interval).

b) By determining whether or not the condition of a rule covering a relevant change, or the data used to build a notification associated with that change, can be evaluated by simple inspection of the state transition values of the tuples involved in the change.

c) Through a caching strategy that allows the execution plan of frequently referenced rule conditions and notification sources to be readily available in a shared buffer cache.

4.2. Active Predicate Management

Above the lowest database storage level, where the active predicate store is found, we have created an active predicate management layer. This layer extends the PostgreSQL base triggering system using a number of dynamically loadable C functions (also called shared libraries) to extend the relations in the underlying database storage level with dynamic reactive behaviour. In addition, it hosts a number of components for evaluating rule conditions and notification sources, and for creating the associated notifications and passing them to the message queue for later delivery.

Having established the required extended triggering mechanisms on the relations of interest, the predicates defined on these relations are monitored. Once a situation of interest is detected, the dynamic reactive behaviour comes into place, first determining the set of relevant rules, then passing them to the condition evaluator component.

The condition evaluator makes use of the existing PostgreSQL query processing architecture to evaluate the condition part of a rule. Rule conditions are parsed and converted into internal PostgreSQL query tree structures in the usual way. At this stage, however, a new component, known as the cache manager, has been introduced. The task of the DBMS planner/optimizer is that of creating an optimal execution plan for a given query. This operation can be lengthy, especially in the presence of joins. It can be improved by maintaining a copy (in the shared buffer cache) of the optimal execution plan for those queries associated with frequently referenced rule conditions. A cache manager component has therefore been incorporated in the architecture. There are a number of trade-offs and decisions involved in designing the caching strategy to follow, for example, when should an execution plan be stored, removed or recreated?
At this early stage, we have started to experiment with various approaches. We are certain that an efficient caching strategy will result in considerable optimization gains during the evaluation phases of the integration architecture.

Once the condition part of a rule has been validated, the next task for the active predicate management layer is to fetch the notification sources of those notification types associated with the rule, and send them to the notification evaluator.

The notification evaluator follows the same principle as the condition evaluator for assessing the predicate contained in a notification type source. The evaluation of such a predicate will provide as result a virtual relation on the state of the database at that time, and/or on the values of any state transition tables associated with the current database event. Any tuple contained in the virtual relation returned by the evaluator is then passed to the notification builder component for the creation of the associated database notification.

The main task of the notification builder is to map every tuple coming from the notification evaluator into a valid database notification, according to the schema of the notification type defined in the active predicate store. This involves the assignment (and possibly casting) of the values of tuple attributes to their corresponding notification attributes. The notification generated, which contains a notification type name and a set of attribute name/value pairs, is encoded as an XML notification message (exemplified below) and sent to the message queue for its later delivery to the Hermes adapter.

```
<hermes>
   <notify type="ProductBelowThreshold">
      <productId>NES206</productId>
      <unitsInStock>342</unitsInStock>
      <dateTime>2005-01-14 11:48:23</dateTime>
   </notify>
</hermes>
```

4.3. Message Queue and Hermes Adapter

A message queue, implemented in Java, has been located between the active predicate management layer and the Hermes adapter to provide the architecture with reliable publication of notification messages originating at the database. By writing every notification message to disk, and making use of a set of acknowledgements and timeouts, exactly-once message delivery from the dynamic reactive layer of the database to the Hermes adapter (and its related publisher node) has been ensured.

The Hermes adapter is a multi-threaded Java application in charge of translating advertisement and notification messages coming from the database into their corresponding event advertisements and publications in Hermes. It makes use of a configurable data type mapping table and the Hermes Publisher Application Programming Interface (API) to transform SQL-typed attribute-based messages received from the active predicate management layer into their equivalent Hermes internal representation. Such a representation, class-based in our current Java implementation, is exemplified by Figure 2.

java.lang.String eventType	"ProductBelowThreshold"
Java.util.HashMap eventAttributes	
java.lang.String productId	"NES206"
java.lang.Integer unitsInStock	342
java.util.Date dateTime	"2005-01-14 11:48:23"

Figure 2. Internal representation of a Hermes event publication

As result of this transformation, database notifications are defined and managed exactly the same as any other events in the event-based system. This allows subscriptions to events originating in the database to have the same syntax and semantics as other advertised events, as exemplified below.

```
EventFilter eventFilter = new EventFilter();
eventFilter.addEqualityFilter("productId", "NES206");
eventFilter.addGreaterThanFilter("unitsInStock", 200);
eventSubscriber.subscribe(eventType, eventFilter, this);
```

The Hermes adapter maintains, at all times, an instance of a Hermes publisher node and a list of one or more broker nodes that it connects to when started. These broker nodes, known as publisher-hosting-brokers, form part of the Hermes overlay delivery routing network, and constitute the local entry points to the publish/subscribe system for database notifications.

5. Related Work

To our knowledge, there is no published work on integrating active databases with a publish/subscribe service. However, considerable research has been done in monitoring data changes at information sources. Most of this research has been based on the concept of continuous queries [8].

In continuous query (CQ) systems, clients set up standing queries at each information source that they are interested in. These queries specify the data they wish to monitor for changes. Whenever any of these changes is detected, the system delivers the requested information (defined in the query) to the client who installed the query at the information source. This approach differs significantly from the one proposed in our integrated database - publish/subscribe architecture, where change notifications are defined and advertised at the database using a structured event type model.

In CQ systems, queries on the state of a database originate from clients. Installing a continuous query at a database system thus requires that a client know its location in advance. This is unlikely in open distributed environments with possibly hundreds of information sources. Moreover, the installation of a continuous query requires that the associated triggering behaviour at the database is set up remotely. Serious security concerns arise if clients are allowed to execute data definition statements on the database remotely. Because of the decoupled nature of publish/subscribe, our approach removes from clients the need to know the location of databases. Moreover, the decision of which data is publicly accessible from the database is taken by the database administrators when defining their space of events, each corresponding to the virtual relation of a security view.

In CQ systems, all queries installed by clients are stored at the information source. However, for reasons of efficiency, these systems hypothesize that a large percentage of user queries will tend to be similar, and base their optimization techniques on grouping similar queries together. Our system avoids this redundancy (and its associated performance penalties) by permitting database managers to define and advertise their own space of relevant situations using structured event types. Also, because published events and client subscriptions are efficiently matched and routed by the publish/subscribe system, no subscription data is required to be maintained at the database side.

In CQ systems, once a query is triggered, the notification data associated with the query is delivered to the issuer who installed it, using some notification channel (e.g. SMTP). Our architecture, on the other hand, allows any number of clients to subscribe to an advertised event type (or a subset of it) by indicating filter expressions. As a result, event publications are delivered efficiently to clients via the publish/subscribe system. This is a more flexible approach that, as the number of clients grows, scales better than the one-to-one communication model used by continuous query systems.

6. Future Work

A prototype version, implementing the various components of our integrated database - publish/subscribe architecture, has been developed and tested. From this, a number of interesting research opportunities have emerged.

Regarding system optimization, we intend to investigate how a number of techniques can be used for efficiently evaluating conditions, and the clients to be notified of them, at the publishing databases. The first technique, incremental query evaluation, consists in the incremental maintenance of materialized views that can be used to evaluate rule conditions and notifications. Different incremental update algorithms have been proposed in the database literature. Deciding whether using these will carry lower execution costs than our current approach (based on virtual relations and cached execution plans) will require careful controlled experiments under different query load scenarios. The second technique, parallel condition evaluation, makes use of various degrees of concurrency to achieve scalable condition and notification evaluation.

We are also interested in extending our current database event model, in order to support the definition of composite events. By allowing more complex event patterns to be defined and detected at the database, a considerably larger set of situations of interest could be expressed. Different approaches for defining the semantics and execution model of such event patterns in active databases will be evaluated.

In today's large-scale collaboration environments, database systems may span different cultural, institutional, and geographical spaces. Since publish/subscribe decouples event publishers and subscribers, an event, encapsulating data about a happening of interest at one particular database, will only be properly interpreted by clients when sufficient context information is known. This requires publishers and subscribers to share a common understanding in order to express their mutual interests. We are thus interested in studying how ontologies and other semantic mechanisms can be used to make events in the system more "context-aware".

In terms of security, we are investigating how controlled access to Hermes events can be achieved in large-scale multi-domain distributed systems, using role-based, policy-driven access control. We envisage independently administered but related domains, each providing a context for defining a set of events (in our case database change notifications), and a set of principals and roles expressing and enforcing a security policy over these events and their associated information sources. We believe that OASIS [9] (Open Architecture for Securely Interworking Services), a parametrised RBAC system developed at the Computer Laboratory, fits naturally with our architecture. It can be used to secure the communication channel within a domain, controlling the roles that can advertise, publish, or subscribe to an event type (e.g. a particular change in a database). Moreover, secure inter-domain communication can be achieved through negotiated access control policies defining which role(s) of one domain may receive (which attributes of) which event message(s) of another.

7. Conclusion

We have proposed a novel approach to handling database change notifications in large-scale distributed environments. It is based on the integration of active databases and the publish/subscribe communication model to form a global event-based system: databases define and advertise change events, and clients subscribe to events of interest, and can refine their subscriptions through content-based filter expressions. An integrated architecture was presented, and its different components were explained, using our experimental context, based on the ORDBMS PostgreSQL and Hermes, our content-based publish/subscribe system. Our approach differs from the continuous query model, being more flexible, scalable and secure. It constitutes a good basis for future research in scalable and secure inter-domain data notification.

Acknowledgements

Luis Vargas is supported by the National Council of Science and Technology of Mexico (CONACYT). The many contributions of members of the Opera research group are acknowledged.

References

[1] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec, "The Many Faces of Publish/Subscribe", *ACM Computing Surveys 35*, pages 114-131, 2003.

[2] T.P.G.D. Group, "*PostgreSQL 7.4 Programmer´s Guide"*, http://www.postgresql.org.

[3] P.R. Pietzuch, "Hermes: A scalable event-based middleware", *University of Cambridge PhD Thesis and TR590,* 2004.

[4] ACT-NET Consortium, "The Active Database Management System Manifesto: ADBMS Features", *ACM SIGMOID Record 25 (3)*, pages 40-49, 1996.

[5] A. Carzaniga, D.S. Rosenblum, A.L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service", *ACM Transactions on Computer Systems* 19, pages 332—383, 2001.

[6] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, M. Ward, "Gryphon: An Information Flow Based Approach to Message Brokering", *IBM TJ Watson Research Center*, 1998.

[7] G. Muhl, L. Fiege, F. Gartner, A. Buchmann, "Evaluating Advanced Algorithms for Content-Based Publish/Subscribe Systems", *In Proceedings of the 10th International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 167-176, 2002

[8] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous Queries over Append-Only Databases", *In Proceedings of the 1992 ACM-SIGMOD International Conference on Management of Data*, pages 321-330, San Diego, CA, January 1992.

[9] J. Bacon, K. Moody, W. Yao, "Access Control and Trust in the Use of Widely Distributed Services", *Middleware 2001, Volume LNCS* pages 300-315, 2001*2218, Springer-Verlag*,