Relational database support for event-based middleware functionality

David M. Eyers
Computer Laboratory
University of Cambridge
Cambridge, UK
{first.last}@cl.cam.ac.uk

Luis Vargas
Computer Laboratory
University of Cambridge
Cambridge, UK
{first.last}@cl.cam.ac.uk

Jatinder Singh
Computer Laboratory
University of Cambridge
Cambridge, UK
{first.last}@cl.cam.ac.uk

Ken Moody
Computer Laboratory
University of Cambridge
Cambridge, UK
{first.last}@cl.cam.ac.uk

Jean Bacon
Computer Laboratory
University of Cambridge
Cambridge, UK
{first.last}@cl.cam.ac.uk

ABSTRACT

Many popular relational database management systems (RDBMS) provide features for operating in a distributed environment, such as remote table queries and updates, and support for distributed transactions. In practice, however, much application software targets a more minimal set of functionality than is offered by the SQL standards. Independently of the database tier, engineering concepts such as the enterprise service bus and service oriented architecture have led to the development of communication middleware to support distributed applications. For applications that require reliable delivery of messages complex event processing, and integrated archiving of data, impedance mismatches are likely to emerge between the database system and the communications middleware—for example with respect to datatypes, and event filtering that is based on information in the database. This paper describes event-based middleware functionality that is supported directly within the database system. In contrast to previous approaches (e.g. being able to name remote tables in SQL statements), the programming of event-based communication operations within the database is explicit. We present initial performance results that compare an augmented PostgreSQL database system to an environment in which a database and an event-based middleware package are used side-by-side.

Categories and Subject Descriptors

C.2 [**Computer Communication Networks**]: Distributed Systems; H.3 [**Information Storage and Retrieval**]: Systems and Software

General Terms

Design, Experimentation, Performance, Reliability

Keywords

Database, Publish/subscribe, Queues

DEBS'10, July 12-15, 2010, Cambridge, UK

1. INTRODUCTION

Research into the design and implementation of distributed relational database systems gained momentum in the 1980s. As in early middleware research, distribution transparency was soon seen to be an unrealistic goal because of the independent failure modes of components and connections, fundamental to distributed systems. In many organisations, failures within their distributed systems—e.g. communication link failures—are significant events. Communication middleware products are designed to handle such failures.

Above the database layer, the scale of software packages in many organisations necessitates distributed design and operation. There are now communication middleware products to coordinate these heterogeneous software packages, regardless of the extent of their distribution. This middleware can be used to engineer service oriented architectures, e.g. building an enterprise service bus or other forms of software as a service, but these solutions carry overheads.

There is a lot of duplication in the capabilities of the database software and the communications middleware. Both must be able to effect reliable, transactional updates, both need sophisticated data-type, security, audit, serialisation and storage frameworks, and both must incorporate flexible application-specific means to fine-tune operation of the system. Concerns such as archiving and logging data and changes to data, and ensuring reliable operation, are also shared.

For some commercial organisations, such as retail outlets, while their specific stock and logistical data will change massively from place to place, the overall data schemas, and thus software requirements, are constrained to be uniform by the need to interoperate. Instead, we have focused on organisations that operate as a federation of many partially-autonomous administrative domains: for example

those in healthcare, and other government bodies such as transport and police services. In these large-scale distributed applications, both the communications middleware and the database engines must make decisions based on information contained in other software packages. For example, in healthcare, it is likely that the delivery of messages by communications middleware should be filtered and transformed on the basis of information contained within other databases within the healthcare system [19].

We are investigating how best to incorporate *explicit* support for distributed coordination into database systems. We distinguish between database systems that facilitate naming of remote tables (many existing database systems allow this), and database systems that allow a programmer to manage distributed operation, and recovery from failures.

We first discuss how to use active database tables to effect distributed coordination. Our prior work in this area related primarily to structured management of triggers. Secondly, we extend the open source PostgreSQL database system directly. New database features are added to allow clients to interact with the database server as if it were communication middleware, and the SQL syntax is extended to incorporate statements that control such middleware. Database programmers can thus direct the delivery of events within a distributed system composed of database nodes.

Factoring communication middleware functionality into a database simplifies software engineering. There is a potential advantage in terms of performance: there is no need to serialise communication between the database system and the communication middleware. We present initial performance results that demonstrate these efficiency benefits by comparing an augmented PostgreSQL system with an environment in which a database and an event-based middleware package are used side-by-side.

This paper is organised as follows. After introducing some application domains, section 2 provides background in the type of event-based middleware and database software we use, and provides an overview of related work. Section 3 discusses incorporation of communication controls through the overloading of existing database structures. Section 4 takes the next logical step, and incorporates communication controls within the database language directly. We present the design and various implementation details of our augmented version of the PostgreSQL database system in section 5. We evaluate the performance of this software in section 6: a comparison is made between an approach using a separate database and communication middleware, with an approach incorporating both functions in the same database server. Finally, section 7 provides concluding remarks.

1.1 Multi-domain application scenarios

This section discusses two typical multi-domain application scenarios. In the first we consider a large-scale, globally distributed financial services company: a hypothetical deployment is illustrated in figure 1. It is commonplace for regional offices to be largely autonomous, and to focus on interacting with the markets local to each office. In the second, we highlight the different types of domains within a national healthcare system.

The monitoring of stock prices is often cited as a complex event-processing application for which publish/subscribe communication is ideal. Stock quotes are a compact, commercially significant event stream: each quote will have attributes including the identifying symbol for that stock, the price, the volume available at that price, and a time-stamp.

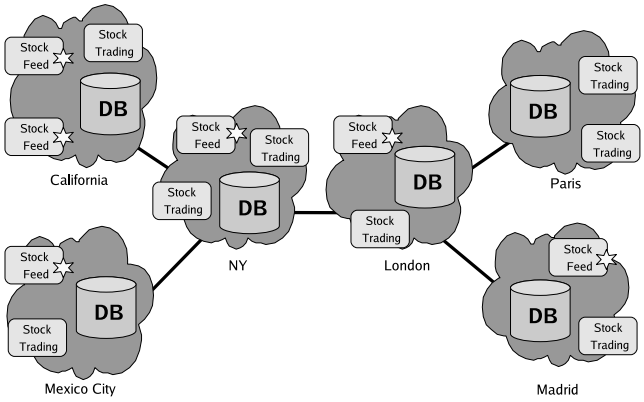

Figure 1: A large-scale financial services firm

Recently, stream processing systems have become the infrastructure of choice for performing analyses of stock quotes, e.g. for algorithmic trading. Our infrastructure provides full online transactional processing (OLTP) facilities, however. Front-line processing of stock streams would not take full advantage of the transactional capabilities on offer in our infrastructure, indeed sometimes it is not even necessary to store the stock price data, and would suffer from processing overheads caused by these extra capabilities.

A related use case that is appropriate, however, is how distributed financial organisations coordinate their reaction to current market data. They will want to store and process the way in which the regional offices make decisions, and use this information to coordinate the control of their algorithmic trading parameters.

Our infrastructure would facilitate the storage and processing of the reactions of the firm to the market, and to synchronise the event streams between different regional offices. This process will benefit from a database infrastructure that incorporates communication middleware capabilities. The specific cross-site links are small in number and are closely managed, so explicit control over distribution is desirable, as opposed to transparent distribution. However, the data-driven interaction offered by publish/subscribe communication is still required.

1.2 Healthcare

The operation of large-scale healthcare environments depends on the collaboration of many specialised but highly distinct organisations. On the one hand, information sharing is key to providing good patient care. On the other, the privacy and security requirements on patient data are stringent [18].

Recent trends in healthcare are to decentralise operation as far as possible, e.g. to perform more healthcare in patients' homes. This further complicates the multi-domain operation of such environments, although it is highly beneficial in terms of patients' recovery. Figure 2 illustrates a typical multi-domain healthcare environment. The 'home' domain on the left is interacting with a 'hospital' domain in the centre. The two other domains are the domain for the medical 'auditor' and the 'pharmacy' domain, which needs less information about patients than the hospital.

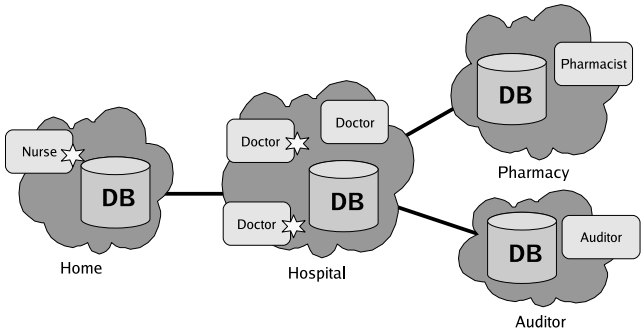

Figure 2: Healthcare environment

We use this domain configuration to illustrate the management of prescriptions. In hospitals, doctors prescribe medication as and when it is necessary. Each prescription for a drug includes the drug identifier and a recommended dosage. The hospital database maintains further information such as the symptoms that led the doctor to make the prescription in the first place. A home-care nurse may record information relating to the patient's health that is relevant to a particular prescription: the hospital doctor should be kept informed of the relevant information obtained in the patient's home.

Only limited information needs to flow to the pharmacy domain. The pharmacy may be required to summarise prescriptions from different hospitals; each prescription record will include the identity of the prescriber and only the minimum necessary personal information on the patient. The auditor domain must maintain a persistent record of controlled drug prescriptions. The analysis of the auditor focuses on prescription details, and does not need to involve the identity of the patient.

Again, this scenario is inherently event-driven, but does not necessarily suit transparent distribution: the edges in figure 2 are few in number, and specific in purpose.

The data protection requirements of the healthcare domain makes this example particularly relevant to the incorporation of communication middleware features directly into the database engine. Each care provider is legally responsible for protecting data, and controlling the ways in which that data is released to other parties. Legislation relating to electronic healthcare records in the UK NHS, for example, requires that patients must be able to exercise fine-grained control over the release of their records. An infrastructure to support particular patient data controls will necessarily need to involve detailed interaction between the event-based communication software, the databases containing patient's healthcare data, and the infrastructure that stores the patients' policy specifications.

2. BACKGROUND

This section gives background on terms and technologies used. We provide a brief overview of the type of communication middleware we require and introduce the PostgreSQL database system used in the performance evaluation in section 6.

2.1 Event-based Middleware systems and the Publish/Subscribe paradigm

We believe that the above applications are well served by the publish/subscribe communication paradigm [8].

The publish/subscribe paradigm encodes self-contained representations of happenings of interest in a system into events that can be delivered from event producers, i.e. publishers, to event consumers, i.e. subscribers. The publish/subscribe paradigm effects asynchronous many-to-many communication. The publishers and subscribers are decoupled, although in some cases it is useful for the event representation to facilitate point-to-point communication or replies [7].

A *client* is an entity that is connected to a publish/subscribe system. Clients may be publishers, subscribers, or both. Publish/subscribe systems are often divided into two broad types: topic-based and content-based. Topic-based publish/subscribe systems use particular attributes of events to determine what content will be delivered from publishers to subscribers. Content-based publish/subscribe systems make event routing decisions based on data contained in the event more generally. It is usually the case that content-based publish/subscribe systems will have more expressive filtering specifications, and the building of spanning trees that dictate event-delivery paths may take advantage of subsumption of event filter expressions.

Many publish/subscribe systems require a publisher to *advertise* events before they can be published. This allows the system to allocate the appropriate resources to support subsequent event delivery.

Wide-area, distributed publish/subscribe systems often deploy an *event-broker* network. Here, broker nodes act as intermediaries, routing events between publishers and subscribers. Thus the spanning trees for event delivery act over edges that are fallible links in the underlying distributed system.

In multi-domain distributed publish/subscribe systems, the level of trust between the domains will determine whether or not it is acceptable to share brokers across the whole distributed system, or whether gateways are required between the different administrative domains [15].

In the above we have used the terminology of the academic middleware community. The event-driven architecture (EDA) work discussed by Luckham [11] and others embodies the same notions for building a distributed system, namely producing, detecting, consuming and reacting to events, albeit using a slightly different terminology.

2.2 PostgreSQL

PostgreSQL [22] is a free and open-source, object-relational database system written in C. It is a mature product: starting in 1985, based on Stonebraker's team's experience with the Ingres commercial database system. It has used SQL as its query language since 1995. Its code-base is still under active development, and new features are frequently released.

PostgreSQL has a high degree of compliance with the SQL standards, and works well in multi-user environments due to its multi-version concurrency control. It offers features one would expect in any serious database product: domain, referential, and transactional integrity, and sophisticated datatype and index support.

PostgreSQL is a good choice for academic work due to its high degree of extensibility. The database uses *system catalogues* to store schema metadata specifying the database structure. In PostgreSQL, catalogues are implemented as regular data tables, supporting the convenient addition of new data types and functions. A variety of procedural lan-

guages are supported for database-server-side programming. Any installation of the core distribution currently includes procedural SQL (PgSQL), Tcl, Perl, and Python, however extensions are available for incorporating Java, PHP, R, Ruby, Scheme and Unix shell scripts. It is apparently reasonably straightforward to integrate other languages into PostgreSQL.

These procedural languages can provide the functionality of active databases through triggers and active rules. Active rules in the PostgreSQL system allow procedural code to make modifications to the syntax tree of a query—their implementation is discussed by Stonebraker et al. [21].

2.3 Related work in database-messaging

It was observed over ten years ago that queues of events can be represented using databases [9]. Thus it is unsurprising that some software vendors have incorporated message queueing functionality of some form into their database systems.

For example, the Microsoft SQL Server Service Broker [3] allows databases to participate in asynchronous dialogue. In contrast to our work, the communication is not publish/subscribe.

The Oracle Streams [12] system supports one-to-many asynchronous replication. The propagation of data is on the basis of specification using content-based rules. However the rules are only managed within the scope of directly-connected systems. The work presented in this paper facilitates the unified management of event propagation over an entire distributed database network.

Distributed stream processing systems [10, 20] have received increasing research attention recently, and may emerge as a new type of large-scale event-based middleware. So far, however, there has not yet been sufficient consensus as to their query languages or data models for their use to become widespread. In contrast to stream processing systems, we focus on environments that require more heavyweight processing per event notification. As a consequence, we can use existing database server software and query languages.

We have previously described features of an integrated publish/subscribe-database infrastructure [23, 24], and some associated security extensions necessary to support distributed healthcare [19, 18, 17]. In this paper we fully describe the motivation for integrating publish/subscribe with relational databases, in addition to presenting implementation and performance specifics.

3. DATABASE-TABLE SIDE-EFFECTS

This section shows how to incorporate event-based communication features into a database system while causing a comparatively small impact on the database software. RDBMSs have always included mechanisms to manage transactional changes to sets of rows within tables. Database systems that have active database features can additionally trigger actions when changes are made to tables. Active database systems can be used to build software that reacts to events. Database engines also provide a type system—a particularly flexible and expressive one in the case of PostgreSQL. It may be possible to re-use the serialisation code within the database to assist with placing a representation of the events on the wire.[1]

In the case of communication middleware, each row in a designated table can be related to an event in transit. A worker process that effects distributed communication can perform confirmed or best-effort delivery of data across a network link, and update the tables appropriately (within a transaction if appropriate). Likewise, the management of event routing can be done within database tables. Doing so has the advantage that modifications to event routing configuration can be done within transactions. The notion of using tables as queues of events is employed (behind the scenes) by the PostgreSQL-PS software discussed in section 4.

The worker processes that effect network communication of events are likely to be the only aspects that require software design outside the database system. Much of the rest of the interaction within the database of a publish/subscribe system built this way can use existing active database features such as trigger functions.

An example of this type of overloading of database tables is given in the 't5' Active Predicate Store [4]. Certain tables in the database were used to control database triggers dynamically. Whenever a row was added to the 't5' tables, an instance of a database trigger would be dynamically created. The fields in the table determined parametrization of each trigger instance.

When managing communication over network links explicitly, communication and link failures must be handled. In the case of a wide-area financial organisation, we suggest that a small number of redundant links should exist between the organisation's sites. If serialising rows from a table over one link fails, the communication software can be designed to serialise information over an alternative link. If no alternative links are available, the event should be queued so that appropriate recovery or compensation actions can be taken.

Another area in which database tables with overloaded semantics can potentially be employed is for event routing within brokers. The database engine allows updates to be made to the tables containing representations of events—i.e. performing event deliveries—within transactions. Likewise, the event delivery transactions can be synchronised with any changes to the routing data contained within the control table.

The approach discussed in this section should make it possible to incorporate messaging facilities into active database engines that are closed-source, or not sufficiently extensible to support the addition of explicit communication directives into the database API. The danger of incorporating communication facilities at the same level of abstraction as regular database applications is that the separate software layers are not made clear—the communication functions are performing a task that should be abstracted and isolated from normal database operations. Connecting elements of the database to elements of communication middleware underlies the work described in the next section. This involves a more comprehensive modification of the database infrastructure to incorporate communication primitives directly.

[1]Note that the PostgreSQL-PS implementation used in this paper employs a fairly minimal type-system. This is to ease the process of maintaining type correspondence between the Java and SQL parts of the software.

4. THE POSTGRESQL-PS DATABASE AND MIDDLEWARE SYSTEM

This section describes the PostgreSQL-PS database system. This is a variant of the PostgreSQL database system, introduced above, that incorporates distributed, content-based publish/subscribe functionality. PostgreSQL-PS extends PostgreSQL by providing the database system with event-broker functionality. This means that each database instance (node) also fulfills the role of an event broker in a distributed publish/subscribe system. We now describe the key aspects of its design, and then discuss its implementation.

4.1 Event Types

The publish/subscribe event types in the PostgreSQL-PS system all have a system-wide unique name and a schema that define them. This builds on our work [14] on managing a wide-area, multi-domain publish/subscribe infrastructure.

The schema for each event type is a set of pairs of attribute name and data type. The data types in our PostgreSQL-PS implementation are from the SQL92 specification [1]. A dedicated table in the system catalogue maintains event type schemata. The schema of an event type is used to check that received events are well-formed. Also, subscription filters can be checked for validity against the event type's attributes. Finally, if the filter contains any functions or operators, these can be verified against what is valid for the attribute type stored in the event type schema.

4.2 Events

Each instance of an event in PostgreSQL-PS is represented as a database tuple structure. Each tuple consists of attribute name and value pairs, corresponding to the event type schema. The database has options that alter the visibility and reliability level for each event. In terms of visibility, an event can be set to be *immediate* in which case it is emitted as soon as possible, without respecting transactional properties (see §5.2). The alternative is *deferred* events, which are not emitted until the transaction in which they are published has committed. In terms of reliability levels, event delivery can either use a *guaranteed* or a *non-guaranteed* approach. In the latter, events are delivered zero or one time. In the former, events are delivered exactly once to each subscriber, and the delivery order matches the publication order for each event producer (see section 5.3).

4.3 Subscriptions

All subscriptions in the PostgreSQL-PS system are given a unique identifier. Each subscription indicates the event-type, and may optionally specify a filter expression. Filter expressions are SQL predicates that are evaluated by the database system's query engine in the context of an event instance. Thus filter predicates, in addition to event content, may also access locally stored data and functions through standard SQL operators.

Each subscription also indicates whether it comes from an *internal* or an *external* source. The former are used by that particular database engine to process events, operating like an active rule. The latter may be received from client software, or from other database-brokers in the distributed system. Subscriptions also indicate their scope: either *local* or *global*. Subscriptions that use local scope will only receive events that are known to that particular database node. A global subscription indicates an interest in events that may be known to other databases in the distributed environment.

In order to survive software and hardware failures and network link failure, each database-broker persists subscriptions pertaining to its direct connections in its local system catalogue.

4.4 Queues

For each event type three event queues are maintained. The *in* queue accumulates events that are produced locally, or received from other network locations. The *out* queue contains events that have matched against subscriptions. These events will either be delivered externally, or used in local processing. If failures occur in the processing of events, they are placed into the *exception* queue.

Queues can be either persistent or non-persistent. Persistent queues are stored on disk using the database engine. They are placed within a special table that does not permit `INSERT`, `UPDATE` or `DELETE` statements. Non-persistent queries are stored in volatile memory only.

Persistent queues support `SELECT` statements over the event type schema of the data in that queue, as well as additional system information such as the enqueue time. Persistent queues can either be *auditable* or *non-auditable*. An event in a non-auditable queue is deleted as soon as it is no longer needed—e.g. it has been successfully (reliably) delivered to another node. Auditable queues are for keeping long-term records or for when fine-grained management of the events in the queue is required.

4.5 Advertisements

The PostgreSQL-PS system stores advertisements in the system catalogue of the database engine. As discussed in section 2, advertisements are required before a publisher is permitted to emit events into the system.

4.6 Links

The connections between databases are stored in the system catalogue as *links*. Each link indicates the necessary connection and authentication information required to deliver events across that link. When a PostgreSQL-PS node starts up, it will connect to all the links for which it has definitions saved in its catalogue. Advertisements, (global) subscriptions and events can be propagated over these links.

The specification of links means that PostgreSQLPS network topology is predefined. Unlike infrastructure aimed at supporting unconstrained peer-to-peer applications, database infrastructure is carefully managed by an organisation. Here, each broker is a database system with specific data-handling responsibilities. Nodes do not anonymously join the network and participate in routing; instead, connections between databases are defined for a reason, i.e. to store and route particular information.

The following sections explores local broker processes and database-brokers connect to form a distributed event-based system.

4.7 Subscription/Advertisement Processing

Figure 3 illustrates the interactions between the components of a database-broker in processing a subscription. On receipt of a subscription the event type and subscription filters (if any) are validated against the type and database schemata through the query planner. If the subscription is

valid, it is activated by being persisted in the subscription catalogue. To ensure reliability and atomicity, this process takes place inside a transaction. After acknowledging the subscription, it must be forwarded to the directly connected brokers advertising the subscription's event type. Such information is determined by querying the advertisement catalog. A copy of the subscription is added to the out queue, in a separate transaction, for transmission to each relevant broker.

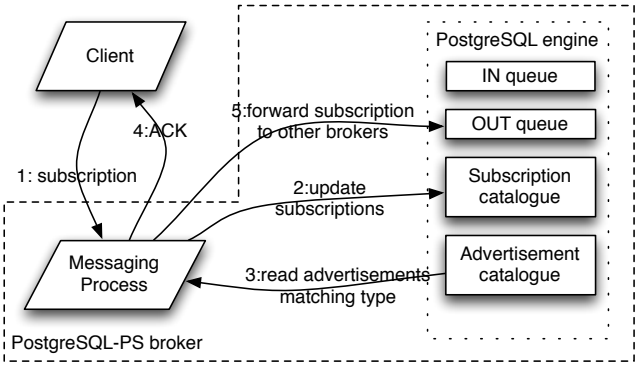

Figure 3: Subscription sequence

The processing of advertisements in similar, except that advertisements are forwarded to all connected brokers that have not already received an advertisement for the event type.

4.8 Event Notification Process

Figure 4 illustrates the steps a database-broker undertakes in processing a publication. The first task concerns event validation. This involves checking that the publication's values and types adhere to the event type and database schemata. The advertisement catalogue is then queried to ensure that the publisher has previously issued an advertisement for the type. If the event is validated, it is persisted in the in queue and the publication is acknowledged. This process occurs in a single transaction.

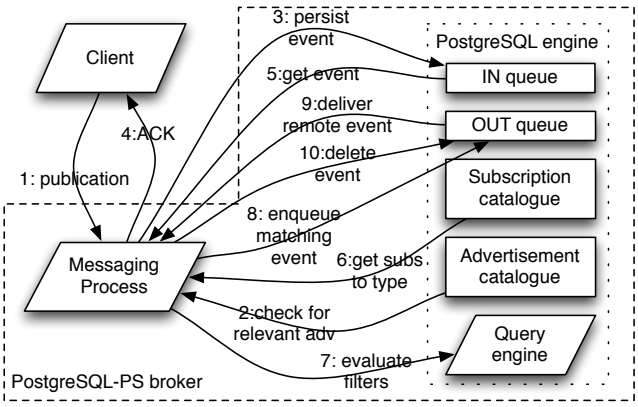

Figure 4: Notification sequence

The next stage of processing concerns delivery. In a separate transaction, an event is taken from the in queue, and the relevant subscriptions to the event type are determined from the subscription catalogue. The filters of each subscription are evaluated by the query engine in the context of the event instance. If the filter matches, a copy of the event is placed on the out queue for delivery to the particular subscriber. Once all subscription filters are evaluated, the event is deleted from the in queue and the transaction commits.

4.9 PostgreSQL-PS Cooperative Event Distribution

This section explores how the interconnected databases (brokers) determine how to distribute events cooperatively. Databases are considered to communicate as peers. When designing the initial distributed system, it is important to consider how to minimise the event delivery latency due to different network environments. It is also desirable to structure the networked databases so that event-processing locality is preserved where possible.

An event dissemination tree is built by propagating advertisements and subscriptions in an advertisement-based forwarding scheme that is similar to the one used in the Siena publish/subscribe system [5]. This requires each broker to maintain state regarding its position in the network, regarding the advertisements and subscriptions of its direct connections. Considering that a broker is running a complete database system, these storage requirements should be non-problematic. *Reverse path forwarding* [6] is employed to ensure that the dissemination tree is free of cycles. The process for building the event dissemination trees is as follows:

1. An advertisement is flooded along each of a database's links for each event type. The receiving database either stores this advertisement, along with information about the source, or discards those advertisements that the receiving database already knows about.

2. Any subscription for an event type propagates along the reverse path of the advertisements for that event type. As for advertisements, each database only stores each particular subscription filter from a given remote database once.

After the receipt of an event, a database determines the active, locally-registered subscriptions relevant to the event type. The filters of each subscription are then evaluated in the context of the event. If a matching subscription is an internal database function, the subscribing function is executed on a copy of the event instance. Otherwise, each subscriber with a matching subscription is forwarded a copy of the event instance. The matching and forwarding process is the same regardless of whether the subscriber is a local client or another linked database. However, if the recipient is a database, the same process repeats—allowing wide-area event distribution.

Figure 5 illustrates the event distribution mechanism. Consider an event type `t`. There are six connected databases labelled $\texttt{DB}_{1-6}$ and a set of applications labelled $\texttt{App}_{1-6}$. Assume that $\texttt{App}_1$ and $\texttt{DB}_2$ produce events of type `t`, and $\texttt{App}_3$ and $\texttt{DB}_6$ are event consumers. When the databases start, $\texttt{App}_1$ will advertise `t` via its local database $\texttt{DB}_1$. This is shown on figure 5 as $\texttt{a}_1$. As described above, $\texttt{a}_1$ will be flooded to, and will be stored by, all of the databases labelled $\texttt{DB}_{2-6}$. Next $\texttt{DB}_2$ creates an advertisement for `t`, labelled $\texttt{a}_2$. This advertisement will only be stored by $\texttt{DB}_3$, since the other

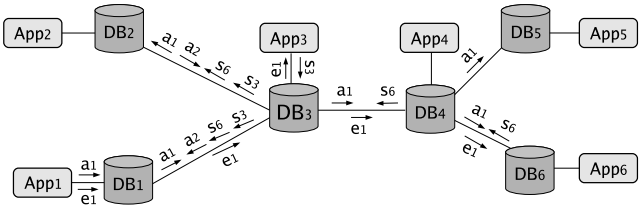

Figure 5: Cooperative event distribution

paths already contain the same filter. Consider $\texttt{DB}_6$ creating a global subscription to `t`, labelled $\texttt{s}_6$. By reversing the paths of the two advertisements $\texttt{a}_1$ and $\texttt{a}_2$, the subscription will be stored by $\texttt{DB}_{1-4}$. If $\texttt{App}_3$, connected to $\texttt{DB}_3$, then creates the global subscription to `t`, $\texttt{s}_3$, it will be stored by $\texttt{DB}_1$ and $\texttt{DB}_2$. This will extend the event dissemination route for `t`. When $\texttt{DB}_1$ receives an instance of `t` from $\texttt{App}_1$ the event labelled $\texttt{e}_1$, it will be propagated to $\texttt{App}_3$ and $\texttt{DB}_6$.

4.10 PostgreSQL-PS Programming Interface

Two independent programming interfaces are provided to the PostgreSQL-PS system. The *database API* is provided for event-processing within the database server. System administration, such as the management of event types, is performed through this interface. In a similar manner to triggers, the database API enables the database system to automatically respond to events as they occur. This allows applications, or parts thereof, to be written to run within the database engine itself. This has the advantage of avoiding potential impedance mismatches between the database and application code—e.g. the database query optimiser can see the application logic directly.

It is not always desirable for an application to directly implement (some) functionality within the database. As such, the *external client API* allows applications to produce and consume events using only the publish/subscribe functionality of PostgreSQL-PS.

4.10.1 Database API

The extension of SQL that supports syntax that is directly related to publish/subscribe is presented in table 1. It can be used from the usual PostgreSQL database console (psql), as well as via language interfaces that support delivery of query strings, such as Java's JDBC interface.

Note that our extension of SQL is not a suggestion that we intend the SQL standards to be extended with communications facilities. Indeed the syntactical requirements of publish/subscribe communication can be largely met using database functions and tables. The extension of the parser relates to conveniences provided by doing so in the PostgreSQL environment specifically. For example, tab-completion operates only on valid communication parts in the command line interface. Also, syntax-checking only allows communications operations to occur using appropriate catalog entries.

First, event types are created using the `CREATE EVENT TYPE` statement. The necessary queues will be created for that event type. The auditable behaviour of a queue can be set with the `ALTER QUEUE` statement.

Before events of a type can be published or subscribed to, the `ADVERTISE` statement must be used. Events can be published using the `PUBLISH` statement, which is parametrised with the event visibility and reliability, and can be used within transactions. The publish statement can also be set as the action of an active rule. When production of events is automated in this way, transition tables `NEW` and `OLD` are available.

Databases subscribe to an event type using the `SUBSCRIBE` statement, which takes the subscription scope as a parameter. A `WHERE` clause on the event type's attributes can be used to specify a subscription filter.

Any subscription created within the database will be treated as *internal* and must specify a database function to receive events. As mentioned in section 2, these functions can be written in any of a number of procedural programming languages. The different languages are passed events in different ways. The C language receives a pointer to an `Event` structure. A `RECORD` variable is provided to PgSQL. Any subscription may choose to specify a priority—this will determine the order of evaluation within the set of subscriptions for the same event type.

Finally, links between databases are created using the `CREATE LINK` statement. The address and port of the remote database's publish/subscribe connection must be specified, and user credentials supplied that are authorised on the remote node.

Because the above statements are incorporated directly into PostgreSQL, its standard privilege management scheme applies to users and roles. Privileges are altered using `GRANT` and `REVOKE` statements.

Also, information about publish/subscribe system objects are available in restricted catalogue views.

4.10.2 External client API

The external client API is shown in table 2. It allows access from outside the database system to the PostgreSQL-PS publish/subscribe features. For the prototype discussed in this paper, a Java implementation of the API was used that provides a `Client` class.

Connecting to the database system is done using the `Client` object's `connect` method. The address and port of the database publish/subscribe service must be provided, along with valid authentication details.

The `EventType` class is used to represent event types. It provides the type name, and a `Map` object for attribute name and type. In this implementation, valid types are `String`, `Date` and all subclasses of the `Number` class. Corresponding SQL92 data types are used.

Events can be instantiated using the `Event` class. This class has a `Map` for attribute name and value. Once an event is created it can be published using the `publish` method. Subscriptions can be registered using the `subscribe` method. A string can be provided to the `subscribe` method that is interpreted as an SQL filter. Note that a subscriber must provide a class that implements the `Callback` interface.

5. IMPLEMENTING POSTGRESQL-PS

The implementation of PostgreSQL-PS discussed here is an extension to version 8.0.3 of the PostgreSQL [22] codebase. In this section we describe the process structure employed, and examine aspects of transactional behaviour, and guaranteed delivery.

5.1 Process Architecture

The PostgreSQL-PS process architecture is shown in figure 6. The list on the left-hand-side of this figure indicates

CREATE EVENT TYPE *event_type* AS (*att1 datatype, att2 datatype, ..*)
ALTER QUEUE *queue_name* SET [NON-AUDITABLE|AUDITABLE]
ADVERTISE *event_type*
PUBLISH [IMMEDIATE|DEFERRED] [NON-GUARANTEED|GUARANTEED] *event_type* (*attvalue1, attvalue2, ..*)
CREATE RULE *rule_name* AS ON {INSERT|UPDATE|DELETE} TO *table* [WHERE *filter*] PUBLISH *event_type*(*attvalue1, attvalue2, ..*)
CREATE [LOCAL|GLOBAL] SUBSCRIPTION *sub_name* ON *event_type* [WHERE *filter*] EXECUTE *func_name*(*args*) [WITH *priority*]
CREATE LINK *link_name* TO *address port* USING *user password*
GRANT [PUBLISH|SUBSCRIBE] ON EVENT TYPE *event_type* TO {*user*|*role*}

Table 1: Database Programming Interface

Client.connect(*address*, *port*, *user*, *password*)
Client.advertise(*EventType*)
Client.publish(*reliability*, *Event*)
Client.subscribe(*sub_name*, *EventType*, *scope*, *filter*, *Callback*)

Table 2: External client API

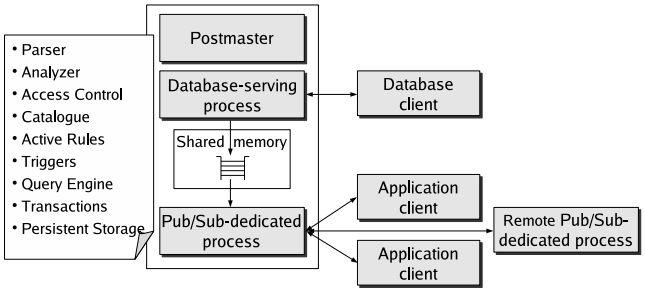

Figure 6: PostgreSQL-PS process architecture

components reused from the PostgreSQL code-base.

The core database daemon remains unchanged: a `postmaster` process listens on a well-known port. Each database client, connecting via TCP, will cause the spawning of a database-serving process. These clients are served using synchronous request-reply. Extensions to the PostgreSQL parser allow these clients to take advantage of the publish/subscribe features.

To handle non-database clients, a separate and additional publish/subscribe process was introduced. In PostgreSQL-PS, the `postmaster` process forks the publish/subscribe process at start-up. In contrast to the way in which PostgreSQL interacts with database-clients, interaction with the publish/subscribe process is message-oriented and asynchronous, using non-blocking sockets. Messages are serialised in XML, to maximise the potential ease of interaction with different client language environments.

The publish/subscribe process also needs to know about publish/subscribe operations issued by the database-serving processes. A shared memory segment is used to coordinate this interaction. Notifications are internally enqueued by the database-serving processes.

In a distributed infrastructure, each broker node will be running both the `postmaster` process, and the publish/subscribe process. However, the overhead in terms of communications latency introduced by this pub/sub process is minimised through the use of non-blocking sockets, and shared memory interactions between the two server processes.

5.2 Transactional Event Management

Transactions are local to a particular broker, to ensure reliable and atomic performance of messaging operations. Providing the choice of immediate or deferred and guaranteed event delivery is expected to cover most application requirements. Note that immediate delivery still uses queues to organise event distribution. As mentioned previously, the queues for immediate event delivery do not enforce any dependencies or transactional effects—events are rapidly copied to the appropriate, non-persistent out queues.

For events with guaranteed delivery, the node's local running transaction includes the enqueueing of a published event. This ensures atomicity of the publication alongside other operations in the transaction. Failure to deliver an event causes the whole transaction to roll-back. For audit purposes, persistent event queues also record the ID of the publication's transaction.

A control queue is maintained that retains the transaction IDs that have successfully committed. This is achieved by incorporating on-commit hooks, in the manner of Paton et al. [13]. For each such transaction ID, the dedicated publish/subscribe process starts a transaction in which the ID is dequeued, copies are created of the committed event for every matching subscriber in the appropriate out queues, and then the event is removed from the persistent in queue.

When multiple functions are provided as callbacks for delivery of a matching event, each is run in its own transaction. The different functions are executed in order, determined by the subscription's priority. If processing of an event in one call-back fails, e.g. because of a violated database constraint, the other event processing callback functions will be isolated from this failure. The failed event processing step will be enqueued in the appropriate exception-queue along with a description of the error. Designers of the distributed system will need to consider the effect of the separate transactions used in event delivery: it is not the case that the transactions indicate that delivery of the event throughout the whole distributed system is atomic. Instead, each separate subscription has atomic delivery. In order to avoid consis-

tency problems, application designers will need to keep this in mind when determining how to recover from the failure of individual subscribers' systems.

Here we have described transactions as local, with respect to protecting the processing of a particular node. For information concerning distributed transactions, see [25] and [16].

5.3 Guaranteed Event Delivery

Guaranteed event delivery concerns the transmission of events. To ensure exactly-once, ordered delivery, the delivery process employs an acknowledgement-based protocol with unbounded sequence numbers, similar to that described in [6]. External producers include a sequence number with their publications, which the (connected) broker returns in an acknowledgement.

Brokers maintain a (persisted) sequence number relevant to their connection for each subscriber/link. These sequence numbers are transmitted as metadata with an event. The recipient acknowledges the sequence number of each event as they are received.

Brokers maintain timeouts for the events they transmit. These were initially set at four seconds. Exponential backoff is employed until a 64 second timeout is reached. Producers can retransmit should their publications remain unacknowledged, in a manner as appropriate to the particular application. Administrators will need to tune the timeout parameters to suit any particular network environment and event workload.

5.4 Scalability considerations

Each database broker maintains state to manage publish/subscribe routing specifics. In order to increase the scalability of PostgreSQL-PS, a broker caches a number of key structures in main memory, rather than relying on its persistent storage engine.

In particular, the publish/subscribe process maintains a hash index of all event types by name. This index stores the event type schemas, the identifiers of its queues, and the subscriptions that are relevant to that time. Note that the PostgreSQL-PS system can reuse the existing indexing features of the underlying database over the persistent storage.

The database will form a query execution plan to compute the result of each subscription's filter expression (if present). In the PostgreSQL-PS system, each broker caches the query plans related to its local subscriptions. This provides a significant performance gain in the cases where data from local tables is incorporated into the decision-making function.

Given that brokers are databases, storage is less of a concern than in implementations where brokers solely route data. For speed, the PostgreSQL database engine does not reclaim space from deletions immediately—records are marked as deleted, and periodic "vacuum" operations can reclaim the space if necessary. Since the event queues reuse the database storage engine, deletion of events from event queues will also maintain this high-speed but space-inefficient mode of operation. Database administrators will need to tune their local systems to perform vacuuming appropriate to the event delivery load.

6. PERFORMANCE RESULTS

In this section we provide some performance results from our PostgreSQL-PS implementation. We compare the performance of an environment that uses a distributed, publish/subscribe middleware, where each node communicates with a local database instance for its storage needs, with a distributed system built from PostgreSQL-PS nodes. Events in many enterprise scenarios require not only transmission, but often storage and reactive processing. The motivation for these experiments is to quantify the performance advantages of a coupled publish/subscribe-storage infrastructure.

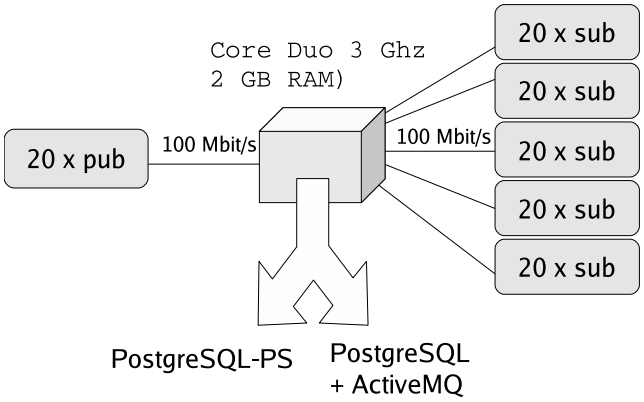

Figure 7: Test environment

ActiveMQ 5.0 [2] is the publish/subscribe messaging middleware that we have used. It is a popular, open-source software package that implements the Java Message Service (JMS) standard. To facilitate reliable delivery of events, an JDBC connection to a database is used to store events and event delivery information.

The test environment consisted of one server node and six client nodes, all of which are interconnected by Fast Ethernet (100Mbit/s). Each node has an Intel Core Duo 3.2GHz CPU and 2GiB of RAM. All were running Fedora Core 6 (kernel version 2.6.22). The environment is illustrated in figure 7.

Five of the client nodes host twenty event subscribers each, and one of the client nodes hosts twenty event publishers. All of the event clients, and the ActiveMQ server, run on version 1.6.0 of the Java runtime environment.

6.1 Experimental design

In this paper we are interested to assess the overall message throughput statistics. That is, the total number of events per second that are successfully received by all of the event subscribers. Although PostgreSQL-PS explicitly facilitates the construction of distributed, event-based middleware, the performance evaluation here focuses on the behaviour of a single node. Our goal is to demonstrate that the performance of PostgreSQL-PS is comparable to that of dedicated event-based middleware systems—particularly when events are being written to persistent storage.

It was determined through experimentation that a load of 20 publishers kept the server in all experiments at close to 100% CPU load. The tools `ntop` and `top` were used to verify that the network and the memory on the system were not bottlenecks in any of the computers used in the experiments.

The event publishers in these experiments publish events as quickly as they can for a period of five minutes. The number of messages that the subscribers receive are counted, although the first ten seconds of each experiment are dis-

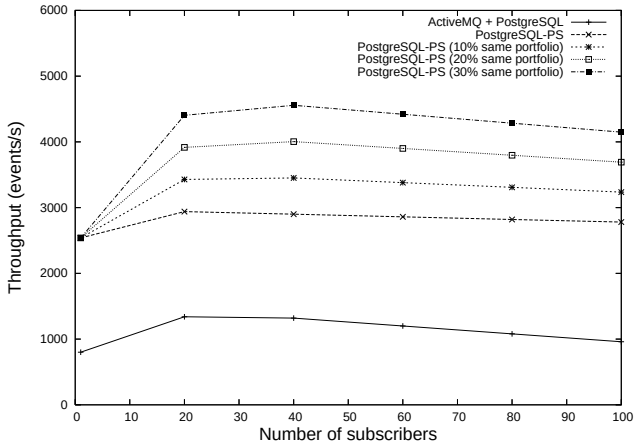

Figure 8: Filters on stored data

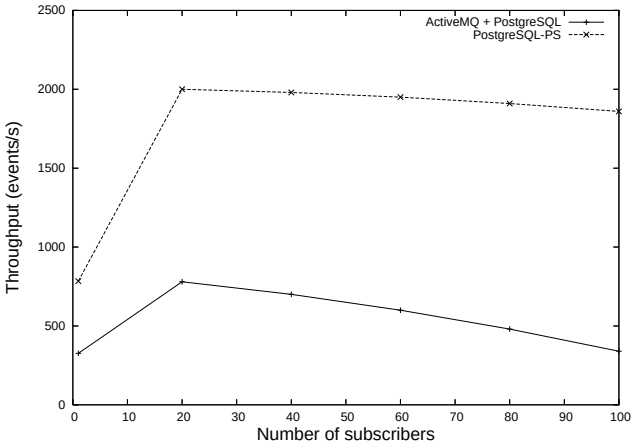

Figure 9: Throughput for guaranteed event delivery

carded as "warm up" time. All experimental measurements are averaged over ten executions.

We use a workload along the lines of the financial services application discussed in section 2. The workload consists of subscribers receiving `StockQuote` events from a random portfolio of stocks. Each subscriber specification is of the form—

```
StockQuote.symbol IN
   ( SELECT symbol FROM Portfolios
                   WHERE portfolioId = [value] )
```

—where `value` is selected randomly from one hundred portfolios.

Because each portfolio contains all of the stock symbols, all of the `StockQuote` events are dispatched to all subscribers.

Figure 8 shows the impact of evaluating complex filters on stored data. In the ActiveMQ + PostgreSQL case, determining whether a symbol in a `StockQuote` is of interest requires ActiveMQ to query the PostgreSQL database. In our experiments, the ActiveMQ software maintained a persistent connection to its database, and the execution plan of the database queries was prepared and cached in advance. Nonetheless, it is clear that the PostgreSQL-PS software provides a significantly higher throughput—by a factor of two in the most general case. Unsurprisingly, the PostgreSQL-PS system benefits from being able to prepare execution plans that incorporate both the publish/subscribe communication aspects, and the querying of persistent data from the database.

In situations where some of the subscribers are interested in exactly the same portfolio, the PostgreSQL-PS system can take advantage of these subscription covering relationships in a way that the separate ActiveMQ system cannot. The other PostgreSQL-PS series in figure 8 indicate that in situations where subscriptions have a common intersection, the throughput of the PostgreSQL-PS system is even higher than in the general case. The tests in this figure set set 10%, 20%, and 30% of the subscribers to be interested in the same portfolio in turn.

In figure 9, we explore the throughput when events must be delivered reliably. Predictably, there is a significant drop in performance, given that a persistent storage system now needs to be synchronised for a large number of `INSERT` and

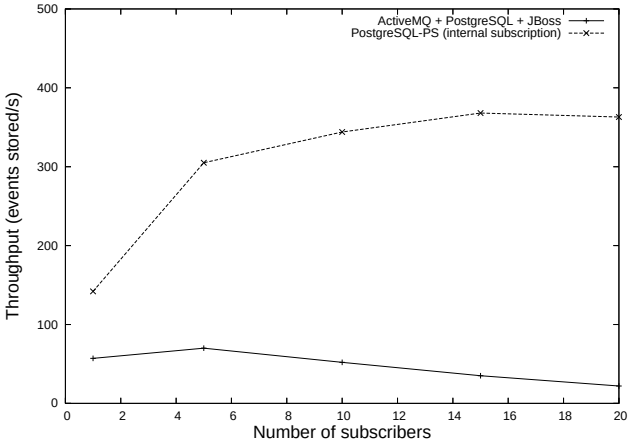

Figure 10: Throughput for transactional event consumption

`DELETE` statements. In the test shown in figure 9, there are no filter expressions, and thus every event is delivered to all of the subscribers.

The ActiveMQ system provides for reliable delivery of events by storing event-subscriber matched pairs in a PostgreSQL database, deleting the records when delivery is completed. A persistent JDBC connection is maintained to the database, but the ActiveMQ system has to send separate `INSERT` requests for each subscriber that is matched with an event. In contrast with the ActiveMQ system, the PostgreSQL-PS system can store all matched pairs at the same time, within the same transaction. It is for this reason that we see a more rapid increase in throughput as the number of subscribers rises to twenty.

Finally, figure 10 shows the throughput of both message delivery approaches when distributed transactions are required for event consumption.

The experiment involves a simple PostgreSQL stored procedure that inserts an event into a table (that belongs to a hypothetical application). Each subscriber uses this function as the callback when they receive events. The number of events stored per second in the hypothetical application's table is measured.

A significant drop in throughput is seen for both environments. In the case of PostgreSQL-PS, there are now a significantly larger number of table operations involved. As expected, for a small number of subscribers, there is a smaller difference in performance between figures 10 and 9 than for large numbers of subscribers: for small numbers of subscribers there are not yet very many application tables to be maintained.

In the case of ActiveMQ, extra infrastructure is required to coordinate a distributed (XA) transaction across the two pieces of software. JBoss 5.0 is a popular, open-source, J2EE application server. It is used here to perform the distributed transaction. However now no less than five disk synchronisations are required per event (two for ActiveMQ and PostgreSQL respectively, and one of JBoss) in order to ensure that the system can recover from failure. In contrast, PostgreSQL-PS can use local transactions (one disk synchronisation) to ensure that the function execution and the removal of an event from the queue is performed atomically. This leads to a order of magnitude difference in throughput in cases that have ten to twenty subscribers.

7. CONCLUSION

In many application domains, data requires both storage and persistence. Relational databases are often an integral component of enterprise infrastructure, recording the state of various processes. Events impact on this state. By building messaging capabilities into relational databases, events can directly effect the appropriate state changes, by updating relevant tables, in addition to being persisted in a form that eases subsequent query and analysis. Such a coupling simplifies administration, through a common interface and type system, enables reliable data processing and delivery and removes the overheads of maintaining separate messaging and storage substrates.

By describing cases of wide-area distributed systems comprising multiple administrative domains, we motivate our case for this integration of function. There will be mutual dependence between the communications middleware (e.g. filters on event delivery), and the data that is stored in distributed applications' databases.

We discuss the way in which certain types of event-based middleware can be built into any relational database system by overloading the functions of tables. The potential risk of conflicts between the communication functions and applications using the database means that, in general, this approach will only suit specialised applications. Instead, we integrate certain middleware features directly into the database engine.

For simple communication tasks, the overhead of using a database engine as communication middleware may be unacceptable. However, for the complex distributed applications discussed in this paper, an approach that augments the database with middleware functionality can actually increase the efficiency of the system. This is because more complex event processing decisions are based on data from both the middleware and the database; the interaction of separate middleware and database software can quickly become a performance bottleneck. Merging the runtime system of the communication middleware with a database engine may seem to be integrating heterogeneous concerns, e.g. with respect to modularity in terms of software engineering. In fact many of the internal functions of communications middleware systems and database engines overlap. Further, only one type system is required, and through shared memory expensive serialisation can be avoided.

We present some performance results that compare our augmented version of the PostgreSQL database system with an environment that uses separate database and event-based middleware software components. The results presented provide promising indicators that decreasing software complexity can provide significant increases in performance.

8. REFERENCES

[1] American National Standards Institute. Standard x3.135-1992, 1992.

[2] Apache Software Foundation. ActiveMQ. `http://activemq.apache.org/`, 2008.

[3] K. Aschenbrenner. *SQL Server 2005 Service Broker*. Apress, 2007.

[4] A. Belokosztolszki. Role-Based Access Control policy administration. Technical Report UCAM-CL-TR-586, University of Cambridge, Computer Laboratory, Mar. 2004.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[6] D. E. Comer. *Internetworking with TCP/IP Vol II. Design, Implementation, and Internals, 3rd edition*. Prentice Hall, 1998.

[7] G. Cugola, M. Migliavacca, and A. Monguzzi. On adding replies to publish-subscribe. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 128–138, New York, NY, USA, 2007. ACM.

[8] P. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[9] J. Gray. Queues are Databases. In *Proceedings of the 7th High Performance Transaction Processing Workshop*, 1995.

[10] J.-H. Hwang, S. Cha, U. Cetintemel, and S. Zdonik. Borealis-r: a replication-transparent stream processing system for wide-area monitoring applications. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1303–1306, New York, NY, USA, 2008. ACM.

[11] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, 2002.

[12] Oracle. Oracle 11g Streams Replication Administrator's Guide, 2007.

[13] N. W. Paton and O. Díaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[14] L. I. W. Pesonen and J. Bacon. Secure Event Types in Content-Based, Multi-domain Publish/Subscribe Systems. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM'05)*, pages 98–105. ACM, 2005.

[15] L. I. W. Pesonen, D. M. Eyers, and J. Bacon. Encryption-Enforced Access Control in Dynamic Multi-Domain Publish/Subscribe Networks. In *Proceedings of the International Conference on*

Distributed Event-Based Systems (DEBS'07), pages 104–115. ACM Press, June 2007.

[16] Y. Shatsky and E. Gudes. TOPS - A New Design for Transactions in Publish/Subscribe Middleware. In *201–210*, pages Proceedings of the 2nd International Conference on Distributed Event–Based Systems (DEBS'08). ACM, 2008.

[17] J. Singh, D. M. Eyers, and J. Bacon. Controlling historical information dissemination in publish/subscribe. In *MidSec '08: Proceedings of the 2008 workshop on Middleware security*, pages 34–39, New York, NY, USA, 2008. ACM.

[18] J. Singh, L. Vargas, and J. Bacon. A Model for Controlling Data Flow in Distributed Healthcare Environments. In *Proceedings of the 2nd International Conference on Pervasive Computing Technologies for Healthcare (Pervasive Health'08)*, pages 188–191. IEEE Computer Society, 2008.

[19] J. Singh, L. Vargas, J. Bacon, and K. Moody. Policy-Based Information Sharing in Publish/Subscribe Middleware. In *Proceedings of the 9th International Workshop on Policies for Distributed Systems and Networks (Policy'08)*, pages 137–144. IEEE Computer Society, 2008.

[20] M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. *Data Engineering, International Conference on*, 0:2–11, 2005.

[21] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. *Readings in Database Systems (2nd edition)*, pages 363–372, 1994.

[22] The PostgreSQL Global Development Group. www.postgresql.org, 2008.

[23] L. Vargas, J. Bacon, and K. Moody. Integrating Databases with Publish/Subscribe. In *Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS'05). In conjunction with the 25th International Conference on Distributed Computing Systems (ICDCS'05)*, pages 392–397. IEEE Computer Society, 2005.

[24] L. Vargas, J. Bacon, and K. Moody. Event-Driven Database Information Sharing. In *Proceedings of the 25th British National Conference on Databases (BNCOD'08)*, volume 5071 of *Lecture Notes in Computer Science (LNCS)*, pages 113–125. Springer, 2008.

[25] L. Vargas, L. I. W. Pesonen, E. Gudes, and J. Bacon. Transactions in Content-Based Publish/Subscribe Middleware. In *Proceedings of the 1st International Workshop on Distributed Event Processing, Systems and Applications (DEPSA'07). In conjunction with the 27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 68–78. IEEE Computer Society, 2007.