

# A Publish/Subscribe CORBA Persistent State Service Prototype

C. Liebig, M. Cilia<sup>†</sup>, M. Betz, and A. Buchmann

Database Research Group - Department of Computer Science  
Darmstadt University of Technology - Darmstadt, Germany  
{chris,cilia,betz,buchmann}@dvs1.informatik.tu-darmstadt.de

**Abstract.** An important class of information dissemination applications requires 1:n communication and access to persistent datastores. CORBA's new Persistent State Service combined with messaging capabilities offer the possibility of efficiently realizing information brokers between data sources and CORBA clients. In this paper we present a prototype implementation of the PSS that exploits the reliable multicast capabilities of an existing middleware platform. This publish/subscribe architecture makes it possible to implement an efficient update propagation mechanism and snooping caches as a generic service for information dissemination applications. The implementation is presented in some detail and implications of the design are discussed. We illustrate the use of a publish/subscribe PSS by applying it to an auction scenario.

## 1 Introduction

The deployment of large scale information dissemination systems like Intranet and Extranet information systems, e-commerce applications, and workflow management and groupware systems, is key to the success of companies competing in a global marketplace and operating in a networked world. Applications like warehouse monitoring, auctions, reservation systems, traffic information systems, flight status tracking, logistics systems, etc. consist of a potentially large number of clients spread all over the world demanding timely information delivery. Many of these applications span organizational boundaries and are centered around a variety of data sources, like relational databases or legacy systems that maintain business data. The business logic may be spread over separate modules and the entire system is expected to undergo continuous extension and adaptation to provide new functionality.

Common approaches in terms of systems architecture can be classified into traditional 2-tier client/server, 3-tier TP-heavy using TP monitors and n-tier Object-Web systems.

In 2-tier client/server the client part implements the presentation logic together with application logic and data access. This approach depends primarily on RPC-like communication and scales well only if client and server are close together in terms of

---

<sup>†</sup> Also ISISTAN, Faculty of Sciences, UNICEN, Tandil, Argentina.

network bandwidth and access latency. However, it does not scale in the face of wide-area distribution. Moreover, the fat-client approach renders the client software dependent on the data model and API of the backend.

In a 3-tier architecture a middle-tier – typically based on a TP monitor - is introduced to encapsulate the business logic and to hide the data source specifics. TP monitors provide scalability in terms of resource management, i.e. pooling of connections, allocating processes/threads to services and load balancing. The communication mechanisms used in 3-tier architectures range from peer-to-peer messaging and transactional queues to RPC and RMI. TP monitor based approaches assume that the middle-tier has a performant connection to the backend data sources, because database access protocols for relational systems are request/response and based on “query shipping”. In order to reduce access latency and to keep the load of the data source reasonably low, the application programmers are urged to implement their own caching functionality in the middle-tier. A well known example of such an architecture is the SAP system [21].

In n-tier Object-Web systems the clear distinction between clients and servers gets blurred. The monolithic middle-tier is split up into a set of objects. Middleware technology, such as CORBA, provides the glue for constructing applications in distributed and heterogeneous environments in a component-oriented manner. CORBA leverages a set of standard services [22] like Naming Service, Event and Notification Service, Security Service, Object Transaction Service, and Concurrency Control Service. CORBA has not been able to live up to expectations of scalability, particularly in the information dissemination domain, because of a limiting (synchronous) 1:1 communication structure and the lack of a proper persistence service. The new CORBA Messaging standard [23] will provide true asynchronous communication including time independent invocations. We argue, that the recently proposed Persistent State Service [14], which replaces the ill-fated Persistent Object Service, will not only play a key role as integration mechanism but also provides the opportunity to introduce efficient data distribution and caching mechanisms.

A straightforward implementation of the PSS relying on relational database technology is based on query shipping. The PSS must open a datastore connection to the server, then ships a query that is executed at the server side and the result set is returned in response. Such a PSS implementation realizes storage objects as stateless incarnations on the CORBA side, that act as proxies to the persistent object instance in the datastore. Operations that manipulate the state of objects managed by the PSS are described in datastore terms. This approach generates a potential bottleneck at the datastore side, because each operation request on an instance will result in a SQL query. Furthermore, for information dissemination systems, where the user wants to continuously monitor the data of interest, polling must be introduced which results in a high load at the backend, wasting resources and possibly delivering low quality of data freshness.

For information dissemination systems an alternate approach based on server-initiated communication is more desirable. Techniques ranging from cache consistency mechanisms in (OO)DBMSs [33,5] and triggers/active database rules [10] to broadcast disks [1] can be used to push data of interest to clients. In the context of the PSS a new publish/subscribe session is needed. A publish/subscribe session represents the scope of the objects an application is interested in, i.e. subscribes to. For those

objects in a publish/subscribe session the cache is loaded and updated automatically. Additionally, this session provides notifications about insert, modify and delete events to the application. While publish/subscribe sessions currently are not part of the PSS specification they are definitely not precluded by it and would represent a useful extension to the spec.

In this paper we present an implementation of a PSS prototype that provides an intelligent caching mechanism and active functionality in conjunction with message oriented middleware (MOM) that is capable of 1:n communication. By removing two crucial bottlenecks from the CORBA platform we claim that highly scalable Object-Web systems become feasible.

In our PSS prototype<sup>1</sup> we take advantage of commercial publish/subscribe middleware that provides the paradigm of subject based addressing and 1-to-many reliable multicast message delivery. We show how a snoopy cache can be implemented for multi-node PSS deployment. We make use of a prototype of a database adapter for object-relational databases (Informix IUS, in particular) that was partially developed and extended in the scope of this project. The database adapter allows to use publish/subscribe functionality in the database and to push data to the PSS caches when update transactions are issued against the data base backend or when new data objects are created.

This paper concentrates on the basic infrastructure needed to provide scalability with respect to dissemination of information from multiple data sources. We explicitly exclude from the scope of this paper federated database and schema integration issues.

The remainder of this paper is organized as follows: Section 2 briefly introduces key concepts of the PSS specification and the multicast-enabled message oriented middleware; Section 3 provides an overview of the architecture of our prototype implementation of the PSS and identifies the main advantages of integrating the reliable multicast functionality of the TIBCO platform; Section 4 describes the implementation; Section 5 introduces auctions as a typical scenario for middleware-based Web-applications and Section 6 presents conclusions and identifies areas of ongoing research.

## 2 CORBA PSS and Messaging Middleware

### 2.1 CORBA Persistent State Service

The need for a persistence service for CORBA was recognized early on. In 1995, the Persistent Object Service was accepted but failed because of major flaws: the specification was not precise, persistence was exposed to CORBA clients, transactional access to persistent data was not covered, and the service lacks integration with other CORBA services. Recently, the Persistent State Service (PSS) was proposed to overcome those flaws. The goals of the PSS specification [14] are to make the state of the servant persistent, to be datastore neutral and implementable with any datastore, to be CORBA friendly, consistent with other OMG specifications

---

<sup>1</sup> The work of this project is partially funded by TIBCO Software Inc., Palo Alto.

(Transactions, POA, Components, etc.) and also with other standards like SQL3 [18] and ODMG [7].

The PSS provides a single interface for storing objects' state persistently on a variety of datastores like OO-, OR-, R-DBMS, and simple files. The PSS provides a service to programmers who develop object implementations, to save and restore the state of their objects and is totally transparent to the client. Persistence is an implementation concern, and a client should not be aware of the persistence mechanisms. Therefore, the PSS specification does not deal with the external interface (provided by a CORBA server) but with an internal interface between the CORBA-domain and the datastore-domain.

Due to numerous problems with IDL valuetypes - used in previous proposals as requirement imposed by the RFP - the IDL was extended with new constructs to define storage objects and storage home objects. The extended IDL is known as Persistent State Definition Language (PSDL). Storage objects are stored in storage homes, which are themselves stored in datastores. In order to manipulate a storage object, the programmer uses a representative programming-language entity, called storage object *instance*. A storage object instance may be connected to a storage object in the datastore, providing direct access to the state of this storage object. Such a connected instance is called storage object *incarnation*. To access a storage object, a logical connection between the process and the datastore is needed. Such a connection is known as *session*.

There is also a distinction between abstract storage type specification and concrete storage type implementation. The abstract storage type spec defines everything a servant programmer needs to know about a storage object, while an implementation construct defines what a code generator needs to know in order to generate code for it. A given abstract specification can have more than one implementation and it is possible to update an implementation without affecting the storage objects' clients. So, the implementation of storage types and storage homes lies mainly in the responsibility of the PSS. An overview of these concepts is depicted in Figure 1.

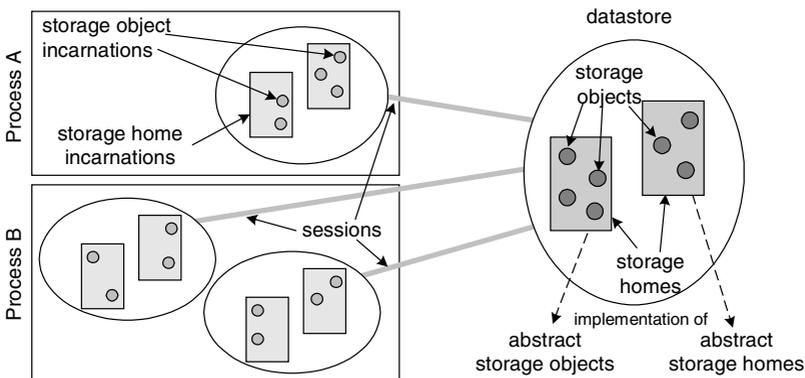


Fig. 1. PSS concepts [14]

A storage object can have both state and behavior, defined by the storage type : its state is described by attributes (also called state members) and its behavior is described by operations. State members are manipulated through equally named pairs of accessor functions. Operations on storage objects are specified in the same manner as with IDL. In addition to IDL parameter types, storage types defined in PSDL may be used as parameters. In contrast to CORBA objects, operations on storage objects are locally implemented and not remotely accessible.

A *storage home* does not have its own state, but it can have behavior, which is described by operations in the abstract storage home. A storage home can ensure that a list of attributes of its storage type forms a unique identifier for the storage objects it manages. Such a list is called a *key*. A storage home can define any number of keys. Each key declaration implicitly declares associated finder operations in the language mapping. To create or locate a storage object, a CORBA server implementor calls `create(<parameters>)` or `find_by_<some key>(<parameters>)` operations on the storage home of the storage type and in return will receive the according storage object instance.

The inheritance rules for storage objects are similar to the rules for interface inheritance in IDL. Storage homes also support multiple inheritance. However, it is not possible to inherit two operations with the same name; as well as to inherit two keys with the same name.

In the PSS spec the mapping of PSDL constructs to several programming languages is also specified. A compliant PSS tool must generate a default implementation for storage homes and storage types based on the given PSDL definition.

For the case that the underlying datastore is a database system, the PSS introduces a *transactional session* orchestrated by OTS through the use of the X/Open XA interface [34] of the datastore. Access to storage objects within a transactional session produces executions that comply with the selected isolation level i.e. *read uncommitted*, *read committed*. Note that stronger isolation levels like *repeatable read* and *serializable* are not specified.

## 2.2 Multicast-Enabled MOM

We use COTS MOM [31] to build the PSS prototype, namely TIB/Rendezvous and TIB/ObjectBus products. TIB/Rendezvous is based upon the notion of the *Information Bus* [26] (interchangeable with the wording “message bus” in the following) and realizes the concept of *subject based addressing*, which is related to the idea of a *tuple space*, first introduced in LINDA [6]. Instead of addressing a sender or recipient for a message by its identifier, which in the end comes down to a network address, messages are published under a subject name on the *message bus*. The subject name is supposed to characterize the contents - i.e. the type - of a message. If a participant, who is connected to the *message bus*, is interested in some specific message types, she will subscribe for the subjects of interest and in turn be notified of messages published under the selected subject names. The subject name space is hierarchical and subscribers may use subject name patterns to denote a set of types to which they want to subscribe.

Messages are constructed from typed fields and can be recursively nested. Furthermore, messages are self-describing: a recipient of a message can inquire about the structure and type of message content. The abstraction of a bus inherently carries

the semantic of many-to-many communications as there can be multiple publishers and subscribers for the same subject. The implementation of TIB/Rendezvous uses a lightweight multicast communication layer to distribute messages to all potential subscribers. On each machine, a daemon manages local subscribers, filters out relevant messages according to subject information and notifies individual subscribers. The programming style for listening applications is event-driven; i.e. eventually the program must transfer control to the TIB/Rendezvous library which runs an event-loop. Following the Reactor-Pattern [29] the `onData()` method of an initially registered callback object will be invoked by the TIB/Rendezvous library when a message arrives with a subject that the subscriber is listening to.

Message propagation can be configured to use IP multicast or UDP broadcast. In the latter case, a special message routing daemon must be set up in each subnet in order to span LAN (broadcast) boundaries. Optionally, TIB/Rendezvous can make use of PGM, a reliable multicast transport on top of IP multicast, which has been developed by Cisco Systems in cooperation with TIBCO and proposed to the IETF [30].

Two quality of service levels are supported by TIB/Rendezvous: reliable and guaranteed. In both modes, messages are delivered in FIFO order with respect to the publisher. There is no total ordering in case of multiple publishers on the same subject. Reliable delivery uses receiver-side NACKs and a sender-side in-memory ledger that buffers messages for some amount of time in case of retransmission requests. With guaranteed delivery, a subscriber may register with the publisher for a *certified session* or the publisher preregisters dedicated subscribers.

Strict group membership semantics must be realized at the application level if so required. However, atomic message delivery is not provided. The TIB/Rendezvous library uses a persistent ledger in order to provide guaranteed delivery. Messages may be discarded from the persistent ledger as soon as all subscribers have explicitly acknowledged the receipt. In both variants, the retransmission of messages is receiver-initiated by sending NACKs.

The diagram in Figure 2 depicts, how the multicast messaging middleware is introduced to CORBA in ObjectBus, a CORBA 2.0 compliant ORB implementation.

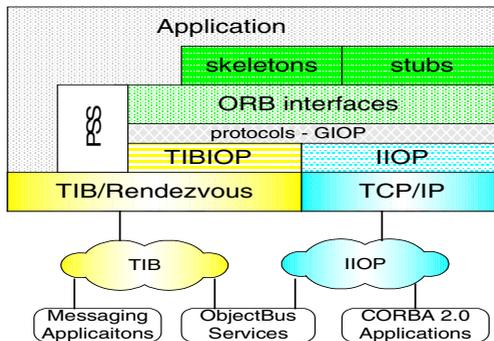


Fig. 2. ObjectBus Architecture

The General Inter-ORB Protocol (GIOP) is implemented both by a standard Internet Inter-ORB Protocol (IIOP) layer and a TIBCO specific layer (TIBIOP). When using

TIBIOP, the GIOP messages are marshaled into TIB/Rendezvous messages and published on the message bus on behalf of a specific subject. The CORBA (server) object may be registered with the ORB presenting an application specific subject name. In that case the returned Interoperable Object Reference (IOR) carries the subject name on behalf of the TIBIOP addressing profile. In order to preserve interoperability, server objects may be registered with both, TIBIOP and IIOPI profiles at the same time. Additionally, CORBA applications may access the TIB/Rendezvous API directly to register listeners and publish messages on behalf of some subject. The PSS prototype implementation is mainly based on this TIB/Rendezvous messaging API.

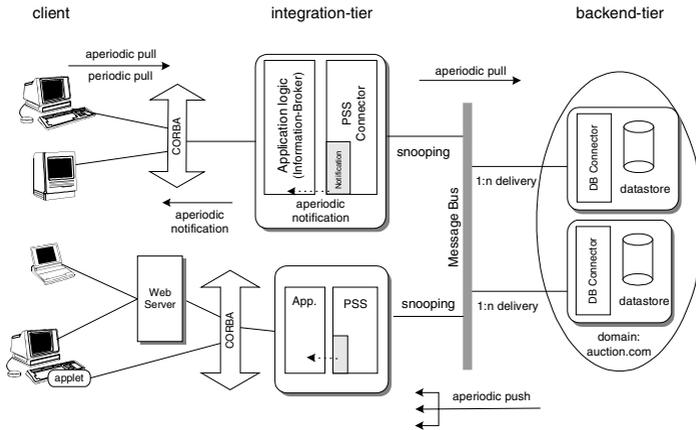
### 3 Overview of the Prototype Architecture

In [13], the nodes in a general distributed information system are classified into: i) *data sources* which provide the base data that is to be disseminated, ii) *clients* which are net consumers of information and iii) *information brokers* (agents, mediators) that acquire information from data sources and provide the information to the clients. Data delivery mechanisms are distinguished along three main dimensions: push vs. pull, periodic vs. aperiodic and 1:1 vs. 1:n.

An analysis of the large, scalable, distributed applications that we are addressing reveals that they are best built using multi-tier architectures. The diagram in Figure 3 below shows this: clients can interact with an application either directly through an ORB or via a Web-server (optionally using an applet). Both periodic and aperiodic pull may be used to begin an interaction, while aperiodic notification and polling are required to propagate change to the users. At the integration-tier the application logic is realized through CORBA objects and services.

The interaction between the integration-tier and the backend-tier requires both pull and push communication to initiate individual requests and to update the caches, respectively. Further, aperiodic event-driven interaction is required and 1:n communication capabilities are essential for effective dissemination of updates and for snooping of load reply and *creation/deletion* events. Under these conditions, the PSS provides the means to efficiently realize CORBA objects as information brokers between data sources and CORBA clients.

In our prototype architecture of a publish/subscribe based PSS, we include a PSS Connector on the side of the integration tier and its counterpart, the DB Connector on the datastore. In terms of Object Oriented Database Systems architecture, the DB Connector plays the role of an object server, leveraging extended relational data base technology and the PSS Connector acts as the object manager.



**Fig. 3.** Multi-tier Architecture for Information Dissemination Systems

We unbundle object caching and object-relational mappings and benefit from the reliable multicast messaging services provided by publish/subscribe MOM:

1. The PSS Connector at the CORBA side interacts with the data sources at the backend in *aperiodic pull* combined with *1:n* delivery. A storage object lookup request is initiated by some PSS Connector on application demand. The response is *published* by the DB Connector under an associated *subject* and all PSS Connector instances that have *subscribed* to that kind of object will snoop the resulting messages and possibly refresh or add a new incarnation to their object cache.
2. Updates to storage object instances result in *publishing* update notifications under an associated subject including the new state of the object, i.e. *aperiodic push* combined with *1:n* delivery. Again, the PSS Connector instances may snoop the update notifications to update the cached incarnation of the object and notify the application of the update.
3. In addition to update notifications, creation and deletion events can be signaled to the application by letting the PSS snoop the respective messages. The application is thus relieved from polling and may extend the chain of notification to the client-tier in order to facilitate timely information delivery.
4. The implementation of the PSS uses a hierarchy of subject names to address objects. Instead of addressing by location (i.e. IP number, DB listener socket address), *publish/subscribe* interactions use the paradigm of addressing content (i.e. *subject based addressing*). Thereby several data sources may be federated in a single data source domain. Additionally, a labeling mechanism can be introduced to support subscription to a collection of storage objects and simple subject-based queries.

Given the potential distribution of clients and caches we expect to benefit from reference locality not only in the scope of a single PSS instance but because of the snooping of load replies and update notifications we benefit from reference locality throughout the datastore domain across different PSS nodes.

## 4 Prototype Design & Implementation

The implementation consists of the realization of the PSS Connector and the DB Connector including the definition of the corresponding formats and protocols (FAP), provision of snoopy caching and active functionality, the mechanisms to adapt the database to the TIB/Rendezvous message bus, the mapping between PSDL and the (object-) relational data model, and last but not least the transactional semantics and correctness criteria that can be enforced.

### 4.1 Formats and Protocols between Connectors

In defining the FAPs we must specify the basic functionality to create, lookup, load, change/save and delete objects. More advanced features are snooping load replies, generating and snooping update notifications, and generating and snooping create/delete events. Most important for the implementation of the advanced features on top of publish/subscribe messaging middleware is the definition of the subject namespace, i.e. the categories an application can subscribe to and under which to publish. Subjects must be chosen in a way that enables snooping load and update payload data, as well as detecting create/update/delete events and signaling them to the application. Appendix A presents the subject name space with respect to the FAP. Figure 4 below shows the basic functional units of the PSS prototype.

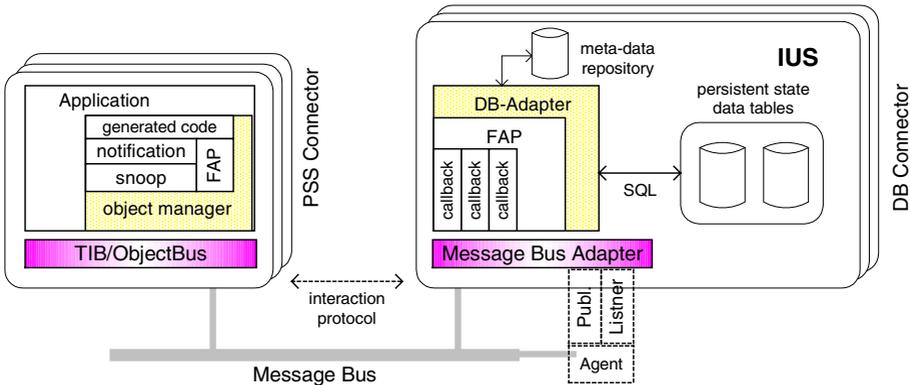


Fig. 4. PSS Prototype Components

The FAP is materialized by type-specific generated storage object (home) implementation on top of a general object manager library at the PSS Connector. At the DB Connector the FAP is implemented using callback handlers (external SQL UDR, see also 4.2). Additionally we must provide a DB Adapter that maps the payload data to the constructs of the datastore as reflected in the metadata repository.

#### 4.1.1 Loading a storage object in a publish/subscribe session

An application gets access to one or more storage object incarnations through its respective storage home. Storage homes are managed by a publish/subscribe session,

which also defines the scope of the object cache. Before actually accessing a storage object, the application must retrieve a handle to the object incarnation using `find_by_pid()` or some other key-based finder operation, or using `find_by_label()`. In the first case, the application is returned a handle to the storage object incarnation. In the second case the storage home will return a sequence of handles (see also `ItemHome` in Appendix C).

As the prototype is restricted to C++, the handle is realized by a C++ pointer. The actual implementation of state and of the corresponding accessor functions is delegated to a “data container” object. Thus the handle represents a smart-pointer [12] to the actual storage object incarnation. This approach is somewhat similar to Persistence [28] and other OODB systems.

Although the handle is returned by the finder operation after the object lookup returned successfully, the data container part is not populated by pulling the datastore immediately. Instead, the respective delegate data container object subscribes to the storage object’s subject name and snoops the message bus for `LOADREPLY` and `UPDATENOTIFY` messages.

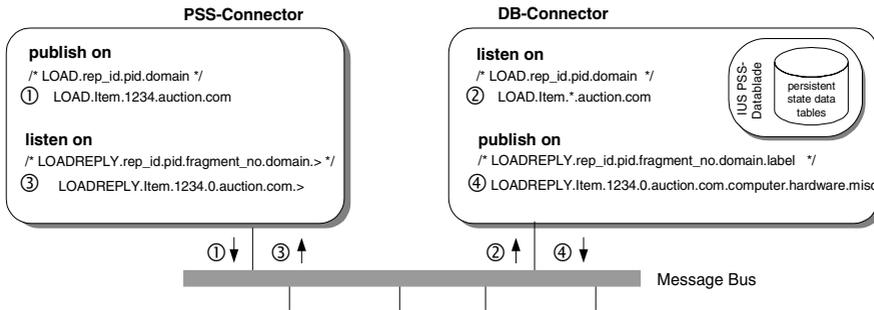


Fig. 5. Object load with publish/subscribe

At the time the application accesses the storage object incarnation - by calling an accessor method - we either have snooped the state in the meantime and can save pulling the object from the data store, or we run into an object fault and initiate a synchronous load request. Figure 5 depicts the object fault situation for a storage object of type `Item` with identifier `1234` in the data store domain `auction.com`. Other nodes running a publish/subscribe session may benefit from snooping message number 4 – an example scenario is presented later in Section 5.

The proposed mechanism is realized by the object manager in the PSS Connector and is transparent to the user. The proposed object faulting technique extends lazy swizzling to the accessed root object, compared to lazy swizzling restricted to contained references [20]. Fetching through collections of objects and navigating through an object tree are typical scenarios where lookup and access are separated in time and thus benefit most from the lazy swizzling with snooping.

As mentioned above, the publish/subscribe PSS provides a supplementary finder operation `find_by_label()` which returns a collection of handles to storage object incarnations. Storage object instances can be assigned a label, which will

become a postfix of the subject name in DB Connector reply messages as depicted in Appendix A. The labeling mechanism presents the subject-based addressing paradigm to the server implementor to explicitly take additional advantage of publish/subscribe capabilities of the PSS implementation. By labeling a collection of related objects, the application can issue subject-based queries to find all storage objects with a given label. In contrast to traditional object server approaches, the result returned by the DB Connector is a set of subject names merely representing the storage object instances. The data container part of the incarnations is eventually filled by snooping on the message bus. As labels can be hierarchically structured, storage objects can be hierarchically categorized. The simple subject-based query mechanism is not supposed to replace a full fledged query service, but comes with our prototype implementation for no additional cost.

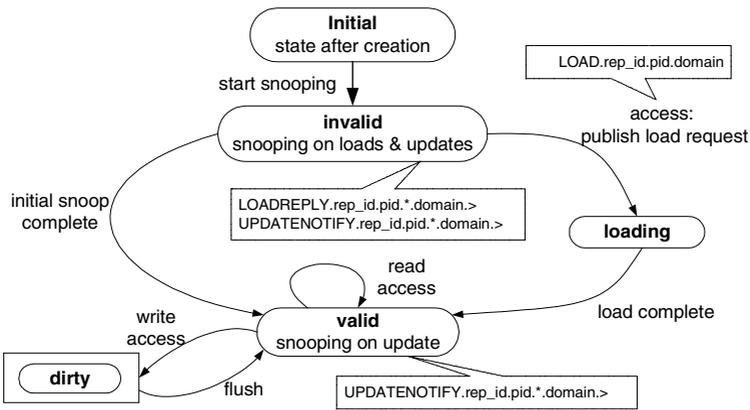
#### 4.1.2 Snooping and state reassembling

As mentioned before, the data container of a storage object incarnation implements the snooping algorithm. In order to collect the state of an storage object the data container may subscribe to `LOADREPLY` as well as to `UPDATENOTIFY` messages. Depending on the storage type definition, the storage object state may be mapped to different tables in the data store (see 4.3) and published on the message bus in different fragments per mapped table respectively. The data container reassembles the fragments according to a common *request\_id* which identifies a particular request/reply interaction and which is enclosed in the message payload data (see Appendix A).

Given a specific incarnation, the data container object subscribes to the message bus using an appropriate *subject mask*. For example, to snoop for update notifications on storage object of type `Item` with identifier `1234` in data store domain `auction.com` the subject mask to use is “`UPDATENOTIFY.Item.1234.*.auction.com.>`”. The subject mask for snooping load replies for the same storage object instance is “`LOADREPLY.Item.1234.*.auction.com.>`”.

Figure 6 summarizes the swizzling with snooping mechanism implemented by any data container in the object manager. Initially the handle to the incarnation is unswizzled and snooping to loads and updates is initiated. Eventually, snooping the collection of fragments is completed and the incarnation is switched to the valid state or an accessor is called beforehand. In the former case, the storage object state members can be accessed without going back to the data store. In the latter case, a blocking load request is published – in turn, replies to this request may be snooped by other PSS nodes. Once in a valid state, the storage object incarnation continuously tracks updates by snooping `UPDATENOTIFY` fragment messages.

The construction of a valid state is possible only if the collection of incoming fragments is complete and all fragments are *compatible*. We say, that two fragments of a storage object instance are compatible if they carry the same *request\_id* and thus are published on behalf of the same interaction. Thereby we assure that we assemble the object state belonging to the same snapshot of the object. The fragment buffer needs only one slot for each fragment, as we are only interested in one version of the object, i.e. the one that represents the latest snapshot.



**Fig. 6.** Snooping states of the data container

As the snooping functionality is executed in an asynchronous thread with respect to the application, we have to synchronize the application access on storage objects with the snooping handler. In order to guarantee snapshot consistency (see also Section 4.4), even if the fragment buffer is complete, we may not unconditionally switch to a new snapshot of the object state in some situations:

- the incarnation is marked as *pinned*: do not switch to a new state until the object is unpinned; continue snooping in the background.
- a read accessor function is currently being executed: switch to the new state on return of the accessor.
- the incarnation is marked as *dirty*: do not switch, until
- the incarnation has been updated and flushed: switch not before a corresponding SAVEREPLY message is received.

#### 4.1.3 Active functionality in PSS

So far, snooping and updating storage objects in the cache has been transparent to the user. To enable the application to reactively monitor significant events like *create*, *update*, *delete*, we extend the PSS API with a notification-channel-like interface.

Each storage home implementation acts as a push-style supplier. It exports the `ProxyPushSupplier` [24] interface, extended with label-based filtering (see `ItemHomeImpl` in Appendix C). An application may register a `PushConsumer` object (IOR) with a storage home, to receive *create* (*update*, *delete*) events, when they occur on any managed storage object whose label matches the given subject predicate. The event parameter of the `push(Any e)` notification carries the type of event (*create*, *update*, *delete*) and the identifier of the affected object. The notification may then trigger appropriate reactions of the application. The addition of a push channel to the PSS interfaces really enables to build CORBA based information brokers in information dissemination systems.

For example, the application could proactively collect instances of objects of some category - identified by label. To do so, the application registers a `PushConsumer` with an appropriate label predicate. Each time a new instance appears the event is

pushed to the application. As a reaction, the application could then issue a `find_by_pid()` using the event payload data and thus proactively start snooping on the recently created object's state. Optionally, the application may itself act as a push-style supplier on behalf of an external notification channel which is connected to the front-end application (e.g. implemented as Java applet).

## 4.2 Message Bus Adapter

The Message Bus Adapter provides the means to connect a relational database to the Rendezvous message bus and thereby to provide transactional publish/subscribe functionality in the database. We use a prototype for Informix Universal Server (IUS) that initially was developed by TIBCO [8] and has been modified and extended by the authors to suit the needs of the project. At the time of this writing, TIB/Adapter ActiveDatabase [32] has been announced. This product shares many features with our prototype of the Message Bus Adapter (see Figure 4).

The API is provided through SQL User Defined Routines which are implemented as external routines in a datablade [15,16]. It is possible to publish row-type data using `EVBSendRow()` as well as results of (restricted) queries using `EVBSendSQL()` on a specific predefined subject.

Publishing is executed on behalf of a database transaction. The data is effectively published iff the publishing transaction commits. If the transaction commits, the published messages are guaranteed to be delivered to all (certified) subscribers eventually. In *certified* mode, the implementation uses *event-tables* to intermediately queue published messages which will be selected and sent on the message bus by a dedicated publication agent process, which runs outside of the server. We added functionality to publish in *reliable* mode directly out of the in-blade UDR, without the overhead of persistently queuing events and switching to the agent process. The reliable delivery multicast is more "lightweight" than the guaranteed delivery multicast on the message bus [9]. In fact, nearly all DB Connector messages are published using *reliable* mode, *certified* mode is used for incoming SAVE messages that contain the state of updated storage objects - SAVE messages are also used when creating an object to initialize its state.

As it is not possible to start a foreign event-loop in a datablade, there is the need for a listener agent process, that subscribes to the message bus on behalf of listeners and the associated subjects in the database. Our Message Bus Adapter provides support to register callback handlers, i.e. SQL UDRs, that will be executed when the subscribed subject is encountered for an incoming message. To implement the logic that is needed to drive the data protocol (i.e. create, find, load, save, delete), we register dedicated handlers for the respective subjects. The handlers are themselves implemented as external UDR in a separate datablade.

A particular problem in the implementation of the DB Connector for the PSS prototype is the need for extended trigger functionality in the database. In order to publish UPDATENOTIFY messages, containing the new state of an object - be it, because it was saved out of a PSS Connector or because the tables were modified directly through SQL - we would like to implement so called *sequential causal dependent* coupling [4] between the triggering transaction and the update notification transaction. This coupling between transactions would assure that the notification is only sent out if the update transaction is successfully committed and that no other

update transaction can modify the data before having sent out the new state. Such a coupling mode is not supported for SQL3 triggers [18] (as well as Informix SQL [17] and SQL92 [11]). There are different ways to tackle this problem. One is to make use of database server extensions that allow to register callback handlers for transaction state changes in database extension modules like datablades in Informix [16]. This way, the DB Connector is able to detect the commit of an update transaction and act accordingly. The IUS 9.14 version, however, does not allow to pass closure data to a transaction state change handler and it is thus hardly feasible to know for which object to send out an update notification. A working (and more portable) solution for us is to let the listener agent run another callback handler after dispatching a save handler. Doing so, situations might occur, when an older update is overwritten before the next update on a storage object and only the latest state of the object will be notified to the PSS connectors. As an effect, a writer may not be able to read its own update, but already receives a more recent version of the storage object.

### 4.3 Mapper

This task is well understood in the database community [3,19]. Automated mapping from PSDL to object-relational therefore is straightforward. However, the derivation of an OO model from a relational schema may need user intervention. In our prototype we implemented the PSDL-to-relational mapping at first and made sure, that the meta-data and the algorithms allow to cope with relational-to-PSDL mappings. The mapping mechanism consists basically of two phases: configuration at compile-time and the mapping process at run-time. The first one is carried out by the PSDL compiler, which maps the definition of storage objects into tabular structures (here, we describe only the PSDL-to-relational mapping). This mapping process involves the following issues:

- inheritance hierarchy: each storage type is mapped into a separate table that contains only the specialized (new) state members, not the inherited ones;
- recursive storage object types: flattened into one table using an attribute as self reference;
- constructed types (array, sequence, enum, union, etc.): mapped into separate tables carrying a reference (primary key) of the related object;
- primitive types: mapped based on a predefined correspondence description between PSDL primitive types and basic SQL types;
- complex SQL types (date, interval, blob etc.): predefined library of storage types (homes) and their implementation.

The result of the configuration process populates the meta-data repository in the data store, which contains all necessary information to transform an object into a tabular structure and vice versa. Based on this meta-data repository the relational schema is generated. Additionally, the mapper creates the messages types, subjects, senders and listeners needed for the FAP in the Message Bus Adapter. One important consequence of the mapping in conjunction with the way, the message bus adapter defines message types is, that the state of a storage object might be fragmented into several messages which have to be published separately. This is the case for derived storage object types and storage object types containing sequences or unions. We are investigating,

in how far we can benefit from object-relational mappings to increase efficiency of data shipping.

Once the configuration is established, the second phase is carried out at run-time when the mapping algorithm is executed in the database adapter i.e. called by some callback handler. We have to unmarshal the payload data of the incoming message, e.g. a fragment of a save request, and update the storage object's tabular representation accordingly.

#### 4.4 Transaction Properties

Accessing a state member of a storage object incarnation is guaranteed to return a consistent value. The implementation of an accessor method is realized as atomic unit of access isolated from concurrent updates on the same storage object in that same publish/subscribe session and isolated from update notifications snooped from the message bus, as described in Section 4.1.2. Using the `pin()` operation on a storage object incarnation allows to extend the unit of isolation with respect to update notifications until the `unpin()` operation is called. Thereby it is possible to bracket several read accessor calls to the same storage object incarnation in order to read a consistent snapshot of the object [2].

Note, that we do not use a lock based implementation of isolation as it would require interaction with the DB Connector. As one consequence, the prototype does not support bracketing access to more than one storage object incarnation for snapshot consistency. Without locking, this would require the PSS Connector to assure a quiescent state [27] of the objects in the readset. In fact, deciding quiescence depends on a bounded transmission delay of update notifications [27], which we think is not realistic in the envisaged scenarios. As another consequence, concurrent updates to the same storage object from different sessions are possible. Updates to the same storage object will be serialized by the DB Connector in reception order. Updates are propagated to the datastore on session `flush()` using certified message delivery. The save handler in the DB Connector updates all the state fragments of the storage object in one database transaction and thus assures the atomicity of writes. Again, this is restricted to single storage objects, there is no transaction bracketing provided. In Section 4.2, we discussed a restriction imposed by the particular message bus adapter: it is not guaranteed that each update on a storage object will result in the publication of an update notification but two updates might be collapsed into one update notification. Nevertheless, snapshot consistency is preserved, as the application will never see an older update after a newer one.

The PSS prototype described in this paper does not support the notion of transactions as proposed for *transactional sessions* in the PSS specification, which is targeted to integration with OTS [25] and X/Open DTP XA compliant data bases [34]. Instead we define a *publish/subscribe session*. We argue that for many applications the object instance is a sufficient granularity of isolation, especially for long-running read-only applications with monitoring semantics. In that respect we trade serializability of computations for timely delivery and freshness of data.

## 5 Putting It all Together: An Auction Application Scenario

A worldwide person-to-person auction system (à la eBay) is the basic scenario. We assume the reader is familiar with the basic auction process. Figure 7 depicts the infrastructure of this application.

Beginning from the back-end, the DB Connector interacts with the datastore(s), where all auction entities are stored. The middle-tier encapsulates the auction business logic (see Appendix D) and the access to persistent auction entity instances through use of a publish/subscribe PSS (see Appendix B). In this scenario, there are multiple middle-tiers organized by region, providing similar functionality, and more important, accessing common storage objects of the `auktion.com` datastore domain. At the front-end, clients use a browser interface, where auction applets are running that in turn are connected to the “nearest” application through an ORB. To refresh the data in the front-end, the user can configure the applet to automatically refresh the data when it receives the corresponding change/update notifications (aperiodic push/pull combination); the user can schedule a periodic refresh, e.g. every 5 minutes (periodic polling) or can refresh information on mouse-click (aperiodic polling).

We present a few characteristic interactions that illustrate the operation of the PSS in the context of the auction application. Space limitations preclude a more detailed analysis.

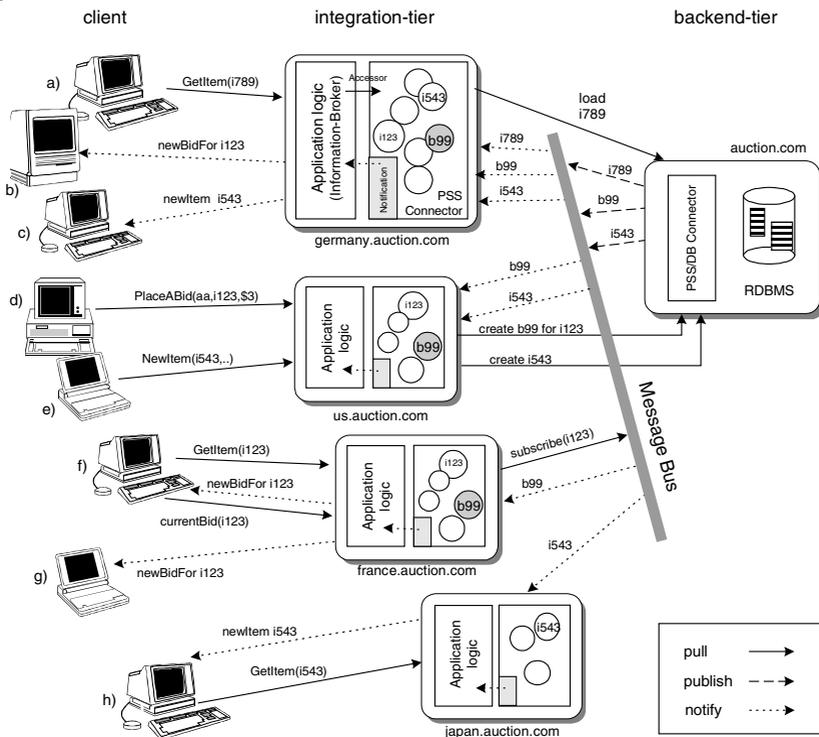


Fig. 7. Auction application scenario

- A German user on client (a) is interested on item i789, the applet calls the operation `GetItem(i789)` through the ORB on the application, which itself issues a `find_by_item_id()` on `ItemHome` (see Appendix C). Given that the object is not in the cache, the application will run into an object fault when reading the state of item i789, causing a load request to be published. The responsible DB Connector looks up the storage object instance and selects corresponding tabular data and publishes the fragments on the message bus. As no sessions exist on other nodes that hold an incarnation for i789, the item i789 is only loaded by the German middle-tier.
- The user on client (d), playing the role of a bidder, places a bid issuing a `PutABid(aa, i123, $3)` operation call on the application. This operation involves the creation of the bid instance b99 with label “i123” for the item i123. The `BidHome` creates this instance on the datastore by publishing a `CREATE` request. The responsible DB Connector of the particular datastore domain, maps the corresponding `Bid` storage object instance into its tabular structure, creates the required types and subjects in the Message Bus Adapter, and publishes a `CREATEREPLY` as well as the object state (after commit) on the message bus. All `BidHome` instances in PSS Connectors for which the application has registered a *PushConsumer* for create events with label “i123”, snoop the message bus and signal the creation of bid b99 to the application. The respective storage homes will proactively create a storage object incarnation in the cache, which instantaneously snoops the new state. In the scenario, the PSS Connectors in Germany, U.S.A. and France have subscribed to “i123” labeled bids. On notification, the corresponding applications in turn read the new b99 storage object - which should already be in the cache – and send a *newBidFor i123* notification to clients (b), (f), and (g).
- An American seller, in front of client (e), wants to sell an item. She requests an item number and fills out all the required information (title, category, description, duration, first bid, etc.) and when completed the `NewItem` operation is called on the application. This method creates a new item instance (through `ItemHome`) identifying it with i543 and category label “comp.misc”, as explained before. All applications that registered with a respective `ItemHome` for *create* events with label “comp.misc” again receive a *create* notification from the PSS, while the item incarnation is already snooped and placed in the cache. Since bidders can specify their interest in categories to the application, all new items under the selected categories can be notified to them. That is the case of `germany.auction.com` and `japan.auction.com`, where notifications are sent to the clients (c) and (h). The latter one pulls i543 item description calling `GetItem` on the application.

## 6 Summary and Future Work

We discussed the need for supporting information dissemination applications by integrating relational databases in a fully distributed Object-Web system. We have shown that this can easily be done in a CORBA framework through the use of a publish/subscribe Persistent State Service. In this paper we introduced an architecture that combines the reliable multicast capabilities of COTS middleware with the requirements of the new PSS specification. We presented implementation details of a

prototype implementation based on Informix IUS 9.14 and TIBCO's TIB/Rendezvous and discussed the main implementation issues: design of PSS and DB Connectors and the associated formats and protocols, realization of an object manager that provides lazy swizzling with snooping techniques, the mechanisms for adapting a DBMS to interact with the messaging middleware and how active event signaling capabilities were built into the DBMS extensions, the mapping procedures between PSDL and the relational model and the required session and transaction semantics. We illustrated the use of the publish/subscribe PSS through an auction scenario and a few characteristic user interactions and the dissemination of the pertinent information.

The main advantage of our publish/subscribe PSS is the ability to support both client- and server-initiated interactions in contrast to typical client-initiated approaches based on query shipping. By exploiting the reliable multicast capabilities of the middleware we provide a scalable generic caching mechanism that enables the application developer to concentrate on application development rather than on the reimplementing of basic functionality. The prototype of our CORBA Persistent State Service is well suited to build n-tier information dissemination systems that require timely delivery of data and exhibit access patterns that are typical of monitoring applications. The introduction of a push channel in the PSS interface makes it possible to notify applications whenever an event of interest occurs. Through the use of lazy swizzling combined with message-bus snooping and subject-based addressing we provide the means to achieve efficient data staging across data stores.

The current PSS implementation is a good platform for further experimentation. On the one hand, future research includes the deployment in a realistic testbed for performance evaluation and the use of this platform in large-scale e-commerce application scenarios. On the other hand, PSS capabilities must be expanded and the interactions with other CORBA services must be tested. Specific issues to be resolved in our ongoing research include the use of timestamp-ordering consistency protocols, the extension of the caching mechanism to do proactive caching, replication among data stores, the integration with query, notification, and transaction services, and tool support for handling heterogeneous data stores.

### Acknowledgements

We would like to thank Arvola Chan for his support and constructive feedback.

### References

1. S. Acharya, R. Alonso, M. Franklin and S. Zdonik, *Broadcast Disks: Data Management for Asymmetric Communications Environments*. In Proceedings of the International Conference on Management of Data (SIGMOD 95), pp. 199-210, San Jose, June 1995.
2. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil, *A critique of ANSI SQL Isolation Levels*. In Proceedings of the International Conference on Management of Data (SIGMOD 95), pp. 1-10, San Jose, June 1995.
3. M. Blaha, W. Premerlani and H. Shen, *Converting OO Models into RDBMS Schema*. IEEE Software, Vol. 11, No. 3, pp. 28-39, May 1994.
4. H. Branding, A. Buchmann, T. Kurdass and J. Zimmermann, *Rules in an Open System: The REACH Rule System*. In Proceedings of Intl. Workshop on Rules in Database Systems (RIDS 93), pp. 111-126, Edinburgh, Scotland, September 1993.

5. M. Carey, M. Franklin, M.Livny and E. Shekita, *Data Caching Tradeoffs in Client-Server DBMS Architectures*. In Proceedings of the International Conference on Management of Data (SIGMOD 91), pp. 357-366, Denver, May 1991.
6. N.Carriero and D. Gelernter. *Linda in Context*. Communications of the ACM, Vol. 32, No. 4, April 1989.
7. R.G.G. Cattell et al (Editors). *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, 1997.
8. A. Chan. *Transactional Publish / Subscribe: The Proactive Multicast of Database Changes*. In Proceedings of the International Conference on Management of Data (SIGMOD 98), pp. 520, Seattle, Washington, 1998.
9. D.R. Cheriton and D. Skeen. *Understanding the Limitations of Causally and Totally Ordered Communication*. In 14th ACM Symposium on Operating System Principles, Asheville, NC, December 1993.
10. U. Dayal and A. Buchmann and D. McCarthy. *Rules are Objects too: a knowledge model for an active, object-oriented database system*. In Proceedings of the 2nd Intl. Workshop on Object-Oriented Database Systems, Lecture Notes in Computer Science 334, Springer, 1988.
11. C. Date and H. Darwen, *The Guide of SQL Standard*, 4th edition, Addison-Wesley, 1997.
12. D. Edelson, *Smart Pointers: They're Smart, but not They're Not Pointers*. Technical Report UCSC-CRL-92- 27, Baskin Center of Computer Engineering & Information Science, University of California, Santa Cruz, 1992.
13. M. Franklin and S. Zdonik, "Data in your Face": *Push Technology in Perspective*. In Proceedings of the International Conference on Management of Data (SIGMOD 98), pp. 516-519, Seattle, June 1998.
14. Fujitsu, Inprise, IONA Technologies, Objectivity, Oracle, Persistence Software, Secant Technologies, Sun Microsystems and TIBCO, *Persistent State Service 2.0, Joint Revised Submission*. OMG Document orbos/ 99-07-07, <ftp://www.omg.org/pub/docs/orbos/99-07-07.pdf>, August 1999.
15. Informix Inc., *Extending Informix Universal Server, User-Defined Routines*, 1997.
16. Informix Inc., *DataBlade API Programmers Manual*, 1997.
17. Informix Inc., *Informix Guide to SQL:Syntax*, Version 9.1, 1997.
18. ISO-ANSI, *Working Draft Database Language SQL (SQL / Foundation SQL3)*. Part 2, X3H2-94-080 and SOU-003, 1995.
19. A. Keller, R. Jensen and S. Agrawal, *Persistence Software: Bridging Object-Oriented Programming and Relational Databases*. In Proceedings of the International Conference on Management of Data (SIGMOD 93), pp. 523-528, Washington, May 1993.
20. A. Kemper and G. Moerkotte, *Object-Oriented Database Management: Applications in Engineering and Computer Science*, Prentice Hall, 1994.
21. R. Munz, *Usage Scenarios for DBMS*. Keynote, International Conference on Very Large Data Bases (VLDB 99), [www.dcs.napier.ac.uk/~vlbd99/IndustrialSpeakersSlides/SAPVLDB.pdf](http://www.dcs.napier.ac.uk/~vlbd99/IndustrialSpeakersSlides/SAPVLDB.pdf), Edinburgh, September 1999.
22. Object Management Group (OMG), *CORBA Services Specification*. OMG Document formal/98-12-09, <ftp://www.omg.org/pub/docs/formal/98-12-09.pdf>, Famingham, MA, December 1998.
23. Object Management Group (OMG), *CORBA Messaging*, OMG Document orbos/98-05-05, <ftp://www.omg.org/pub/docs/orbos/98-05-05.pdf>, Famingham, MA, May 1998.
24. Object Management Group (OMG), *CORBA Notification Service*, OMG TC Document telecom/99-07-01, <ftp://www.omg.org/pub/docs/telecom/99-07-01.pdf>, Famingham, MA, August 1999.
25. Object Management Group (OMG), *Transaction Service Specification*, in *CORBA Services Specification*, Chapter 10, Famingham, MA, May 1998.

26. B. Oki, M. Pfluegl, A. Siegel and D. Skeen, *The Information Bus - An Architecture for Extensible Distributed Systems*. In Proceedings of SIGOPS 93, pp. 58-68, December 1993.
27. E. Pacitti, P. Minet and E. Simon. *Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases*. In Proceedings of the Intl. Conference on Very Large Data Bases (VLDB99), pp. 126- 137, Edinburgh, UK, September 1999.
28. Persistence Software, *Persistence PowerTier: A Technical Overview*. White Paper, [www.persistence.com/Sources/Download/WP\\_Technical.pdf](http://www.persistence.com/Sources/Download/WP_Technical.pdf).
29. D.C. Schmidt. *Reactor -- An Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching*. Proceedings of the First Pattern Languages of Programs Conference in Monticello, Illinois, August, 1994.
30. T. Speakman, D. Farinacci, S. Lin and A. Tweedly. *PGM Reliable Transport Protocol Specification*. Internet Draft <draft-speakman-pgm-spec-02.txt>, Cisco Systems, August 1998.
31. TIBCO Software Inc. *TIB/Active Enterprise*.  
[http://www.tibco.com/products/active\\_enterprise/index.html](http://www.tibco.com/products/active_enterprise/index.html), TIBCO Software Inc., Palo Alto, USA.
32. TIBCO Software Inc. *TIB/Adapter for ActiveDatabase*.  
[http://www.tibco.com/products/adapter\\_adb/whitepaper.html](http://www.tibco.com/products/adapter_adb/whitepaper.html), TIBCO Software Inc., Palo Alto, USA.
33. W. Wilkinson and M. Neimat, *Maintaining Consistency of Client-Cached Data*. In Proceedings of the Intl. Conference on Very Large Data Bases (VLDB 90), pp. 122-133, Brisbane, Australia, August 1990.
34. X/Open DTP, *Distributed Transaction Processing: Reference Model, The XA Specification*, Reading, Berkshire, England, X/Open Ltd., 1991.

## Appendix A. Subject Namespace

**Table 1.** Subject Namespace.

<b>Task</b>	<b>Subject</b>	<b>participant</b>	<b>*</b>	<b>mask</b>	<b>content**</b>
<b>Create</b>	CREATE.rep_id.domain	storagehome	P	CREATE.rep_id.domain	rep_id, [pid]{{(key,value)}*} label, request_id
	CREATEREPLY.rep_id.domain.label	DB-Connector DB-Connector	S P	CREATE.*.domain CREATEREPLY.rep_id.domain.label	rep_id, pid, request_id,label, result
" snoop		storagehome	S	CREATEREPLY.rep_id.domain.>	
		storagehome	S	CREATEREPLY.rep_id.domain.label	
<b>Delete</b>	DELETE.rep_id.domain	storage_type	P	DELETE.rep_id.domain	rep_id, pid, request_id
	DELETEREPLY.rep_id.pid.domain.label	DB-Connector DB-Connector	S P	DELETE.*.domain DELETEREPLY.rep_id.pid.domain.label	rep_id, pid, request_id,label, result
" snoop		storage_type	S	DELETEREPLY.rep_id.pid.domain.label	
		storagehome	S	DELETEREPLY.rep_id.*.domain.label	
<b>Find</b>	FIND.rep_id.domain	storagehome	P	FIND.rep_id.domain	rep_id, pid, request_id, {{(key,value)}*}, label
	FINDREPLY.rep_id.Domain	DB-Connector DB-Connector	S P	FIND.*.domain FINDREPLY.rep_id.domain	(rep_id, pid, label) collection, request_id
<b>Load</b>	LOAD.rep_id.Pid.domain	storagehome	S	FINDREPLY.rep_id.domain	
	LOAD.rep_id.Pid.domain	storage_type	P	LOAD.rep_id.pid.domain	rep_id, pid, request_id
" snoop		DB-Connector	S	LOAD.*.*.domain	
	LOADREPLY.rep_id.pid.fragment_no.domain.label	DB-Connector	P	LOADREPLY.rep_id.pid.fragment_no.domain.label	fragment data
		storage_type	S	LOADREPLY.rep_id.pid.*.domain.>	
<b>Update notification</b>	UPDATENOTIFY.rep_id.pid.fragment_no.domain.label	storage_type	S	LOADREPLY.rep_id.pid.*.domain.>	
	UPDATENOTIFY.rep_id.pid.fragment_no.domain.label	DB-Connector	P	UPDATENOTIFY.rep_id.pid.fragment_no.domain.label	fragment data
" snoop		storage_type	S	UPDATENOTIFY.rep_id.pid.*.domain.label	
<b>Save</b>	SAVE.rep_id.pid.domain	storage_type	P	SAVE.rep_id.pid.domain	Fragment data, request_id
		DB-Connector	S	SAVE.*.*.domain	
	SAVECOMPLETE.rep_id.pid.domain	storage_type	P	SAVECOMPLETE.rep_id.pid.domain	rep_id, pid, request_id
		DB-Connector	S	SAVECOMPLETE.*.*.domain	
	SAVEREPLY.rep_id.pid.domain	DB-Connector	P	SAVEREPLY.rep_id.pid.domain	rep_id, pid, request_id, result
	storage_type	S	SAVEREPLY.rep_id.pid.domain		

\* P:publisher, S:subscriber

\*\* rep\_id: repository identification; pid:persistent state object identification.

**Appendix B. PSDL (Auction Example)**

```

abstract storagetype User {
  //...
};

abstract storagetype Category {
  //...
};

abstract storagetype Item;
typedef sequence<ref<Item>> item_seq;

abstract storagetype Bid{
  readonly state long bid_no;
  state ref<User> bidder;
  state ref<Item> item;
  state date when;
  state money amount;
}

typedef sequence<ref<Bid>> bid_seq;
enum item_state {sold, cancelled, active, inactive};

abstract storagetype Item {
  readonly state long item_no;
  state string title;
  state string description;
  state ref<ItemDetails> details;
  state ref<Category> cat_no;
  state date from;
  state date until;
  state string location;
  state ref<User> seller;
  state item_state thestate;
};

abstract storagetype ItemDetails {
  state ref<Item> item;
  state string longdescription;
  state blob image;
};

storagetype ItemImpl implements ItemDescription{};
storagetype ItemDetailsImpl implements ItemDetails{};
storagetype BidImpl implements Bid{};
// ...

abstract storagehome ItemHome of Item {
  key item_no;
};
abstract storagehome BidHome of Bid {
  key bid_no;
};
abstract storagehome ItemDetailsHome of ItemDetails {};

storagehome BidHomeImpl of BidImpl implements BidHome {};
};
storagehome ItemHomeImpl of ItemImpl implements ItemHome {};
};
storagehome ItemDetailsHomeImpl of ItemDetailsImpl implements
  ItemDetailsHome { };

```

## Appendix C. PSS derived code (Auction Example)

```

// language mappings for special storagetypes date, money, blob
#include "sqltypes.h"

class Item : public virtual CosPersistentState::StorageObjectBase {
public:
    virtual CORBA_Long item_no() = 0;
    virtual const char* title() const = 0;
    virtual void title( const char* s ) = 0;
    virtual void title( char* s ) = 0;
    virtual void title( CORBA::String_var& s ) = 0;
    virtual const char* description() const = 0;
    virtual void description( const char* s ) = 0;
    virtual void description( char* s ) = 0;
    virtual void description( CORBA::String_var& s ) = 0;
    virtual ItemDetails* details() const = 0;
    virtual const ItemDetailsRef
        details(CosPersistentState::YieldRef yr) const = 0;
    virtual void details(ItemDetails* id) = 0;
    virtual void details(const ItemDetailsRef id) = 0;
    //...
}

class ItemHome : public virtual CosPersistentState::StorageHomeBase {
public:
    virtual Item* create(CORBA_Long item_no, const char* title,
        const char* description, const ItemDetailsRef& idr,
        const CategoryRef& cr, const date& from, const date& until,
        const char* location, const UserRef& seller, item_state is,
        const char* label)=0;
    virtual ItemRef create(CORBA_Long item_no, const char* title,
        const char* description, const ItemDetailsRef& idr,
        const CategoryRef& cr, const date& from, const date& until,
        const char* location, const UserRef& seller, item_state is,
        CosPersistentState::YieldRef yr, const char* label,)=0;
    // suppl. subscription based finder methods:
    virtual item_seq* find_by_label(in string label)=0;
    // finder methods for keys
    virtual Item* find_by_pid( const CORBA_OctetSeq& pid ) = 0;
    virtual Item* find_by_item_no( CORBA_Long item_no ) = 0;
    virtual ItemRef find_by_pid( const CORBA_OctetSeq& pid,
        CosPersistentState::YieldRef yr ) = 0;
    virtual ItemRef find_by_item_no( CORBA_Long item_no,
        CosPersistentState::YieldRef yr ) = 0;
}

class ItemImpl : public virtual Item {
public:
    // accessors
    CORBA_Long item_no();
    const char* title() const;
    void title( const char* s );
    void title( char* s );
    void title( CORBA::String_var& s );
    const char* description() const;
    void description( const char* s );
    void description( char* s );
    void description( CORBA::String_var& s );
    ItemDetails* details() const;
    const ItemDetailsRef
        details(CosPersistentState::YieldRef yr) const;
    void details(ItemDetails* id);
    void details(const ItemDetailsRef id);
    // ...
}

```

```

    // methods inherited from StorageObjectBase:
    void _add_ref();
    void _remove_ref();
    void _destroy_object();
    CORBA_Boolean object_exists();
    CORBA_OctetSeq* get_pid();
    CORBA_OctetSeq* get_short_pid();
    StorageHomeBase_ptr get_storage_home();
    void pin();
    void unpin();
private:
    ItemImpl() {};
    ItemImpl( StorageHomeBase_ptr home, const CORBA_OctetSeq& pid,
              OBCM_Session *obrvc_session );
    // ...
    StorageHomeBase_ptr _home;
    ItemImplData* _ptr;
}

class ItemHomeImpl : public virtual ItemHome {
    Item* create(CORBA_Long item_no, const char* title,
                const char* description, const ItemDetailsRef& idr,
                const CategoryRef& cr, const date& from, const date& until,
                const char* location, const UserRef& seller, item_state is,
                const char* label);
    virtual ItemRef create(CORBA_Long item_no, const char* title,
                          const char* description, const ItemDetailsRef& idr,
                          const CategoryRef& cr, const date& from, const date& until,
                          const char* location, const UserRef& seller, item_state is,
                          CosPersistentState::YieldRef yr, const char* label);
    // suppl. notification channel interface, inherited from
    // StorageHomeBase:
    virtual void connect_any_push_consumer(notification_type nt,
                                           const char* label, const PushConsumer& pc);
    virtual void disconnect_any_push_consumer(notification_type nt,
                                              const char* label, const PushConsumer& pc);
};

```

## Appendix D. IDL Interfaces (Auction Example)

```

// data structures for user, bid, item etc.
struct bid {...};
typedef sequence<bid> bid_seq;
struct item {...};
// ...

interface User
{
    string NewUser(in string name, in string email,
                  in string password, in string address);
    boolean LogIn(in string UserId, in string password);
    boolean LogOut(in string UserId);
    void ItemOfInterest(in string UserId, in string ItemNo);
    void CategoryOfInterest(in string UserId, in long category);

// Seller:
    string NewItem(in string title, in string descr,
                  in string details, in long category, in string seller,
                  in short days, in string location);
    item_seq GetItemsForSale(in string UserId);

// Bidder:
    boolean PlaceABid(in string Bidder, in string ItemNo,
                     in double amount);
    item_seq GetInterests(in string UserId);

    // ...
};

interface DataRetrieval
{
    item GetItem(in string ItemNo);
    itemDetails GetItemDetails(in string ItemNo);
    item_seq SearchItem(in string text);
    bid firstBid(in string ItemNo);
    bid currentBid(in string ItemNo);
    bid_seq BidHistory(in string ItemNo);
    cat GetCategory(in long catNo);
    user GetUser(in string UserId);
    // ...
}

```