SUPPORTING INTEROPERABILITY IN CORBA VIA OBJECT SERVICES

Jaromír Adamec, Michael Gróf, Jan Kleindienst,
František Plášil, Petr Tůma

Tech.Report No 114 October 1995

Charles University Prague
Department of Software Engineering
Malostranské nám. 25
118 00 Prague 1
Czech Republic

ver 1.11.

Supporting Interoperability in CORBA via Object Services[3]

Jaromír Adamec[1], Michael Gróf[1], Jan Kleindienst[2], František Plášil[1,2], Petr Tůma[1]

[1] *Charles University,*
Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranské nám. 25, 118 00 Prague1,
Czech Republic
e-mail:{adamec, grof, plasil, tuma}
@kki.ms.mff.cuni.cz

[2] *Institute of Computer Science*
Czech Academy of Science
Pod vodárenskou věží
18000 Prague 8
Czech Republic
e-mail: {kleindie,plasil}@uivt.cas.cz

Abstract. This report is a study of three of the CORBA object services. Based mainly upon [OMG 92, OMG94a, OMG94b], the study aims to identify the main issues of the design and implementation architecture of the persistent, events, and replication object services. The reseach was done with an emphasis upon employing the services in a halfbridge and gateway CORBA implementation which are the main goals of the TOCOOS project.

1 Introduction

1.1 The role of interoperability and services in CORBA

The ability of two or more objects, protocols, applications, or environments to exchange information and mutually use resources despite differences in their design is called *interoperability*. CORBA 1.2 ([OMGa] ,[OMGb]) specifies the semantic of object services and outlines the mechanisms of request invocation in order to make client calls to remote servers transparent via Object Request Broker (ORB) [OMG93]. The document [OMG92] suggests about 20 different object services (naming, event, replication, transaction, persistence, etc.), that are intended to provide a standardized layer exporting the CORBA advanced functionality, independed of the vendor and the underlying environment. The interfaces of the object services are specified in Interface Description Language (IDL) that allows binding to

[3] This work was done as a part of TOCOOS, a project funded by the COPERNICUS Program, project CP 940247

various underlying programming languages such as C++, Pascal, or Smalltalk. So far, only a fraction of the object services has been fully specified: event services, lifecycle services [OMG94a], events, and persistent services [OMG94b] to name the most of them.

The CORBA vendors come up with different communication and dispatching protocols what makes interoperability among two or more CORBAs impossible without specialized transformations. Looking for solutions to the CORBA interoperability, OMG Task Force submitted the CORBA 2.0 [OMG94d] request for proposal, intending to add more functionality to CORBA 1.2 that would help integrate multiple ORBs into larger hierarchies and allow transparent access to CORBA object servers in cooperating systems. The new CORBA architecture that comprises interoperability features is denoted *CORBA 2.0* and objects supported by CORBA 2.0 are called Universal Networked Objects (UNO).

To overcome the "standardization gap" of the CORBA networking layer, CORBA 2.0 specifies the Internet Inter-ORB Protocol (IIOP). Even though the IIOP specification defines a canonical request format and standardizes request dispatching, it still lacks sophisticated security and runtime support [OMG94c].

1.2 Supporting interoperability by services

The interoperability in CORBA 2.0 is defined by several approaches. One of them is utilizing the common IIOP protocol, which is considered as a low-level solution. Another, high-level solution according to CORBA 2.0 [OMG94d], would be using request-level gateways, that are constructed at the CORBA application level (above the layer of object services) to transform client request from one CORBA format to another. Such gateways should be designed in a such a way that allows for taking the advantage of the underlying object services. That would reduce the amount of code needed for a gateway construction and contribute to an improvement of the bridge code portability.

Out of the twenty above mentioned object services, there are four of them that are vital for a gateway construction:

Lifecycle service support creating, deleting, copying and moving of the gateway. If the gateway is overloaded, it could be copied and the copy would take over a portion of the incoming requests.

Persistent object service ensure that the gateway will be triggered off from its previous state by automatic storing of the relevant persistent objects at a stable (persistent) store. The state of the gateway will be either checkpointed at well defined time events or automatically. That would improve the gateway fault-tolerant capabilities.

Replication object service improve the throughput of an outgoing request by creating gateway replicas. The number of replicas will vary in time as a function of request load. Thus, there should be a possibility to span new replicas and merge the obsolete ones.

Event services allow for multicasting communication of separated gateways inside the same CORBA. Different gateways can thus set up a shared communication channel through which they may exchange relevant information such as location of remote servers.

1.3 The structure of the report

In Chapter 1, the relation of object services to inter-ORB gateway is showed. The reasons why object services are important to the gateway construction are discussed. Chapter 2, Metaservices, explains our view onto metaservices, comparing it to the Simmel's concept of Kala metaservice [Sim92]. Replication and the associated inherent problems, such as maintaining replicas, locking, and keeping consistency, are outlined in Chapter 3. Push model, pull model, and the concept of typed and untyped event channels is discussed in Chapter 4. Persistence, as a key service for supporting the gateway construction, is discussed in Chapter 5.

2 Metaservices

2.1 Kala approach

The concept of *metaservice* was introduced by Simmel in [Sim92]. The basic idea of metaservices, as stated in [Sim92], is "instead of looking for the one-size-fits-all service, we should look for one common way to express any variation of this service. This leads to the idea of metaservices functionality that can be used to implement specific services."

In Simmel's approach, metaservices are defined by the Kala abstractions: *basket*, *monad*, *kin*, *handle*. Kala creates those abstractions upon hardware persistent storage devices such as disks and, basically, implements manipulation with untyped data on the bases of bits and pointers.. The operations upon those abstraction are called metaservice primitives. In [OMG92a] Simmel claims that Kala provides functionality enough to implement CORBA services. The Kala functionality is in [OMG92a] for that purpose divided into two categories: *access services* and *visibility services*. The first category, access services, is claimed to support object persistence, object relationship, object events, and object recovery. The other category, visibility services, claims to support object transactions, object sharing (concurrency control), object versioning and configuration management, object access control and security, and object licensing.

All in all, [OMG92a] proposes Kala as the base to standardize primitives to express and implement the semantics of object services, thus, at same time, providing a base for standardizing on low-level interoperability.

There are several weak points of the Kala's metaservices approach: the list of supported object services does not include replication, and, moreover it does not provide means for low-level implementation architecture change: for example, the there is a variety of techniques know for implementing persistency; they include some low-level concepts like lazy or early pointer swizzling. Another key issue is that in the future some other object services can be defined and there is no guarantee the functionality of the Kala abstractions will be sufficient to implement those new object services. Put simply, Kala seems to us too "persistence-oriented".

2.2 Generalized metaservices

On the other hand, we feel like it is meaningful to identify a common basis for specific kinds of object services; this base should not be a subject of modification both when porting object services implementation to a different hardware architecture, and when adjusting a particular object service's semantics. Such an adjusting or detailing is necessary as the object services' semantics is in [OMG 94a,OMG94b] not fully defined, or, better put, it skips some very important details (e.g. Persistent store manager, persistent store, fault tolerance in events, etc.). Therefore, to reflect such a stable object services' base, we provide a new concept of generalized metaservices:

When a problem P is to be solved by running a particular software, we say that the corresponding software architecture represents a model of the problem. Naturally, there exists a number of models of P, usually having some common parts; similarly, there exists a number of implementations of a particular model. Given a problem P, we understand *generalized metaservices* (*metaservices* for short) as the software building blocks to be most likely present in all potential models of P.

Supposing a model is a composition of modules, a module can correspond to a metaservice, can be model specific or just implementation dependent. An implementation dependent module can be called from a metaservice, from a model specific module, or from another implementation dependent module. But a model specific module cannot be called from a metaservice. A metaservice can be called from a model specific module or from a metaservice. For simplicity, we assume metaservices cannot be nested.

By convention we expect a metaservice is to be specified as a module in IDL with one interface. The methods of the interface are called *metaservice primitives*.

3 Replication

There can be different motivations for object replication. One can be performance enhancement: Data that are shared between a large client community should not be held at a single server, since this computer will act as a bottleneck that slows down responses. Another motivation is improvement of fault tolerance. When the computer with one replica crashes, system can proceed computation with another replica. Still another motivation is using replicas to access remote object. When a remote object is to be accessed, a local replica reflecting remote object's state is created and used instead of the remote object.

3.1 Replication model

There are two distinct models of replication. Which model is adopted depends on what motivation of replication is focused and other system issues. The first model can be denoted as revealed or explicit replication. With this model, applications are free in creating replicas and connecting them into an ensemble of replicas. The application is free in decision whether access whole ensemble as single object or individual replicas. The ensemble has no ability of creation or deletion of objects. The type of the whole ensemble is not based on types of replicas but rather on the protocol for distribution of updates among replicas. The type of individual replicas can vary.

The second model can be named as hidden replication. Objects created according this model have ability of creation and deletion of replicas. Application cannot access individual replica; whole replicated object is to be used instead. The fact of replication is as hidden from the application as possible. The hidden replication model results in a more compact replicated object with full control over creation and deletion of replicas. With such an organization, a policy can be devised under which replicas are automatically created and deleted to meet some general criteria e.g. to minimize system respond time.

Replicas of an ensemble implemented according the explicit model are more loosely coupled; the ensemble has no control over replicas creation, deletion, placement, or their internal implementation. Therefore, an application programmer is allowed to create replicas meeting more closely his requirements.

3.2 Setting replication attribute

Any distributed system supporting replicated objects must implement a mechanism which makes created object to be replicated. There are several ways to do it.

3.2.1 Static setting

The simplest case is to set the replication attribute statically during compilation time. This requires to associate replication the attribute with classes rather than individual instance. The most typical way to establish such an association is the usage of inheritance. Each instance of a class implemented as replicable becomes replicable automatically during its initialization.

3.2.2 Semidynamic setting

A variation on the approach described in 3.2.1 is to set statically only an ability of replication. An object to be replicated must be an instance of a replication-aware class created e.g. by employing inheritance, but the object becomes replicated after explicit activation of the replication mechanism.

3.2.3 Dynamic setting

Dynamic setting of the replication attribute is the most general approach. With this approach, no a priory knowledge about what object will be replicated is required at the compile-time; any object can be converted into replicated implementation during run-time when desired. Obviously, the run-time representation of the object must contain information comprehensive enough to achieve such a level of generality. Specially, detailed information about types of object’s data members and types and other characteristic (reader/writer property, data members affected by given writer method ...) of object’s methods.

3.2.4 Our approach

In our implementation, we decided to implement the semidynamic approach (3.2.2). The static approach (3.2.1) we have found too limiting, and, at the same time, the fully dynamical approach (3.2.3) requires too detailed run-time information not available under most C++ compilers and CORBA implementations.

3.3 Creating and deleting replicas

During the life time of a replicated object, creating and deleting replicas are essential operations to reflect external requirements. The way to create a new replica mainly depends on what replication model has been adopted. With the explicit model, a replica is created as a regular object and then connected to the replica ensemble of a replicated object. The connection can be accomplished by assigning the ensemble name while the replication of the object is set on or by passing the new replica reference as a parameter of a replicated object’s method ’connect_new_replica’. With hidden model is a method ’create_new_replica’ provided as a part of the standard replication control interface. This replicated object’s method will create a new replica in a specified place (e.g. via a parameter of the method). With either model, there are several issues discussed in next sections.

3.3.1 Naming

In both models of replication, the replicated object has two levels of abstraction: internally, every replica is considered to be an object and therefore it should be named. On the other hand, from the outside, the whole replicated object is understood as a single object having single name.

When the replication support is an integral part of the system design, dealing with this bi-level naming creates no significant problem. But when implementing replication on the top of an existing system which typically has only one level of object abstraction and naming, the problem of assignment of separate internal and external names must be solved.

The following schemes can be proposed: When at most one replica is allowed per naming domain (name scope), all the replicas can share the same name which serves at the same time as the global name of the whole replicated object. Individual replicas are distinguished by involving naming domains identifiers in this scheme. This solution is suitable for the replication models where all replicas are considered to be equal; no replica is selected as the master or coordinator.

When one replica is designed to be the master, a special dummy object bearing the global replicated object name can be created. This object is used only for locating the master. After the master is contacted, the dummy object is no longer needed. The dummy object separated from the master is needed because different replicas can become to be the master during the life time of a replicated object and the underlaying system does not necessarily have to allow changing the name of a replica and also e.g. broadcasting the awareness of the change to other replicas may be a problem.

3.3.2 Initialization

When a new replica is connected to a replicated object, the replica's state must be initialized so that it is consistent with the state of the remaining replicas. When the consistency among replicas is maintained with the state change propagation mechanism (see 3.5.2), the same mechanism can be employed in initialization of new replicas.

An alternative approach has to be used for operation propagation (3.5.1). Typically, a form of linearization of the object state is used. This mechanism can be shared with persistency and transaction control implementation.

3.3.3 Maintaining list of replicas

For updating replicas, keeping their consistency and other activities, it is necessary to maintain a list of the replicas which is the ensemble currently consisting of. In models having one master replica, the replica list is typically maintained at the master replica. In case of failure, there should be available another copy of this list as the only way to recover it is to use a message broadcast which can be unacceptable in wide area networks.

Another approach is to distribute the replica list among replicas. In such an implementation, each replica contains references to a fixed number of its neighbors. Again, the recovery of the list can be complicated when corrupted by a failure.

To make replicated object fault tolerant, the replica list should be held on every replica. Thus, if the master is lost any replica can easily replace it. To make the failure recovery more effective, the replica list can be cached on proxies created when binding a replicated object. With this approach, the problem of updating replica lists held on proxies must be solved.

3.4 Scope of replicas

Another important system decision is where can be replicas distributed. This section analyses the problem.

3.4.1 One address space

All replicas are forced to share the same address space. This can be meaningful only when different replica implementations within one ensemble are allowed. With such an organization, an operation can be performed on the replica where the operation is performed most effectively. When all replicas are implemented in the same way or no information about properties of implementation alternatives is available, it makes no sense to maintain more than one replica within one address space.

3.4.2 Single server implementation

This allows replicas to be distributed over many instances of the same server. The advantage of this variant is that all the information about possible replica behavior is known to the programmer at the application design time.

3.4.3 Multiple server implementations

No restriction is imposed on where replicas can be placed as long as they obey protocols for replica updating and keeping consistency.

3.5 Updating (state change propagation)

When implementing replicated object the basic task is to propagate updates made on one replica to the remaining ones. This can be accomplished in two different ways.
This section deals with the problem of performing one operation on whole ensemble of replicas. The problem of consistent invocation of a operation sequence on the ensemble is covered in Sect. 3.6.

3.5.1 Operation propagation

With this approach, operation calls performed on one replica are propagated to the remaining ensemble members. In other words, every operation is performed on all the replicas. More formally, consider the following invocation of a method op on the replicated object obj:

obj.op(x1, x2, . . . , xn)

To propagate the invocation, the (n+1)-tuple <op, x1, x2, . . . , xn> must be distributed among replicas. This is trivial if the parameters x1, x2, . . . , xn are data (integers, arrays, structures . . .), but some difficulties arise if the parameters are objects (object references). First, it must be ensured that the object references are valid in all the address spaces where the replicas are located. How difficult it is to overcome this problem depends mainly on system's strategy of remote object usage. When, for instance, remote object can be freely bound using proxy mechanism this problem is not hard one.

Another problem with object reference parameter (xobj) is more fundamental one: In the distributed environment, it cannot be guaranteed synchronous invocation of the operation on all replicas. Therefore the state of the object xobj can be changed during the invocation distribution. If such a change is not prevented it can possibly lead to inconsistencies among replicas. This effect can be prevented by locking the object xobj or by making snapshot of its state and using this snapshot in the operation op instead of the object xobj itself. Both these solutions suffer from the same problem.

In a general case the lock/snapshot operation should be recurrently applied on the whole transitive closure of the xobj's dependent objects. On the other hand, in a special case, there is a certain generally indeterminable level of objects in the transitive closure the state of which the operation op depends on. This level can be relatively shallow, possibly, the operation op does not depend on the object xobj's state at all. Therefore the general lock/snapshot operation performed on the whole transitive closure can be unacceptably costly in comparison with what depth is really needed.

3.5.2 State change propagation

The other known approach is to propagate information about the state change caused by an operation call rather than the operation itself. In a naive straightforward implementation, the whole state is propagated after every update. This is unacceptable for large object. A more sophisticated implementation distributes only the difference from the previous state.

3.5.3 Our approach

The both approaches (3.5.1, 3.5.2) have their own advantages and disadvantages. The main advantage of operation propagation is that a distributed system usually support remote operation invocation as its elementary service. Therefore an implementation of operation distribution is quite straightforward. Disadvantages of operation propagation was discussed in Sect. 3.5.1.

The problems inherent to operation propagation do not arise in state change approach. On the other hand, an implementation is more complicated because to make it effective, a special state change propagation protocol must be designed for each distributed class.
Former discussion implies that employing just one of the approaches is not practically sufficient. Therefore, both of them should be implemented and let the application programmer to choose appropriate one. Such a choice can be made either on per class or per method base. We have adopted the per method choice, as it allows for greater amount of flexibility without any significant system design complication.

3.6 Keeping consistency

The previous section (3.5) deals with the problem of propagating the change made on a replica to the rest of the ensemble. Because invocations on replicas are performed basically synchronously, the problem of consistency [Mul94, CDK94] must be solved. In general, two conditions have to be met in order to keep replicas' states consistent. Let *s* be the set of requests processed by a replicated object:

- The set of requests *s* must be totally ordered. In other words, updates must be performed on all the replicas ensemble in the same order. Thus, the set of requests is considered to be a sequence.

- The sequence of requests *s* must be causally ordered. When the two requests comes from the same processor, their causal order is expected to be equal to their instruction flow order. Two requests originating from different processors are causally orderer iff one is sent as result of the fact that the other has been sent. Note, that the total ordering does not imply the causal ordering.

These requirements can be relaxed in some circumstances. The most important relaxation is allowing read requests to be processed in any order. Some requests have the special property that result of their sequential invocation does not depend on any particular order. Such requests are called commutable.

3.6.1 Centralized locking

The simplest method to achieve consistency for replicated object with master replica is centralized locking [IBM93a, IBM93b]. Nothing special is required for reading methods with this approach. Such an invocation is simply passed to a replica. On the other hand, write methods must use locking. On the beginning of a writing method the lock operation must be performed. The lock call contacts the master replica to get permission. When the master has already locked the replicated object, the next lock operation fails. Symmetrically, at the end of a write method lock should be released by the unlock call. At this point, the operation is propagated accordingly to the section 3.5. The unlock operation must block until all replicas are updated otherwise proper ordering could be violated.

3.6.2 Logical clock

The previous section (3.6.1) described an approach assuming one centralized authority is used to ensure proper invocation ordering. There are alternatives that do not require such a centralized authority. In these purely distributed schemes, proper ordering can be ensured by explicit assigning unique identifier to requests. Replicas then process requests accordingly to the order defined by the identifiers.

With such a scheme, an essential concept is stability. A request is said to be *stable* at the given replica when no request with lower identifier can be subsequently delivered to the replica. The consistency requirement can be formulated more specifically by using the concept of stability: A replica next processes the stable request with smallest identifier.

Two separate problems have to be solved in order to implement this scheme: First, system-wide unique identifiers conformed to causal ordering must be assigned in a distributed way. This can be accomplished using logical clock. The logical clock [Mul94] is a counter maintained on each computer which is incremented with every outgoing request. The value of the counter is added to every request as a timestamp. When a request is received at a processor the value of its logical clock is adjusted so that it is at least greater by one then the request's timestamp. This adjustment ensures timestamps to be causally ordered. In order to make the timestamps unique, a processor identifier is appended as the lowest order bits to every timestamp.

The other problem is to implement stability test on the basis of defined timestamps. The main difficulty is that requests can be delayed for an arbitrary length of time without being considered faulty. To overcome this difficulty additional assumptions have to be adopted:

- FIFO Channels. Messages between a pair of processors are delivered in the order sent. This can be easily ensured by attaching sequence numbers to the messages between every pair of processors.

- Failure Detection Assumption. A processor p detects that a processor q failed only after p has received q's last message sent to p.

With these assumptions, the stability test is performed as follows: Every client periodically makes some - possibly null - requests to the replicated object. A request is stable at replica r if a request with larger timestamp has been received by r from every client running on a non-faulty processor.

3.7 Exceptional state handling

Real-world distributed systems consist of components which are inevitably faulty. One of the purposes to use replicated object is fault tolerance improvement. This requires the implementation of replicated objects to be able to recover after component fault as transparently as possible. There are several critical issues of achieving fault tolerance:

- partitioning
- reelecting master after crash
- distinguishing master crash from partitioning
- terminating transaction after client's crash
- continuing transaction after master's crash
- rebinding after replica's crash

3.8 Replication vs. persistency

When implemented replication and persistency at the same time, these two mechanisms can interfere with each other. With a straightforward mechanism, the persistent attribute is assigned to every replica independently and therefore all replicas are stored to the persistent store. In other words, the state of a replicated object is stored in persistent store many times although once would be sufficient.

The cooperation of replication and persistency must be handled as a separated issue. When a persistent object is implemented with one master replica, internally, only the master can be assigned with the persistent attribute in order to make the whole replicated object persistent. When all replicas are considered to be equal, a protocol must be implemented to make distributed agreement among replicas about what replica should be stored to the persistent store.

3.8.1 Using replication to implement persistency

There is a way to employ replication to implement persistency. Such an implementation of persistency can be easy and previously (3.8) described interference problem of replication and persistency does not arise. The idea is simple: build a long-living server as an abstraction of a persistent memory presumably disk space. If an object is to be persistent simply one replica is created in such a server.

3.8.2 Using common features

Both replication and persistency have common features which can be employed even if replication and persistency are implemented separately. Both replication and persistency requires an object's ability to linearize its state and store it as a sequence of bytes and restore another object's state later according to the previously stored information.

Another common set of services is implementation of transactions. In the case of replication it is required for keeping consistency among replicase. In the case of replication to keep the permanent storage image of an object consistent.

3.9 Proposed implementation

This section describes which of the approaches described so far in the Chapter 3, we have decided to use in our implementation. We have chosen the explicit model of replication (3.1) as this decision matches with the general philosophy of CORBA. Replication attribute will be set on the semidynamic basis as described in 3.2.4. Both the operation propagation and the state change propagation approaches will be implemented (3.5.3). Keeping consistency will be based on the centralized locking approach (3.6.1).

The first proposal of the replicated object interface is as follows:

```
interface Replicated_object
{
   void connect_new_replica(in Replicated_object ensemble);
   void disconnect();

   void get_state(out sequece<octet> buffer, out long size);
   void update_state(in short upd_id, in sequence<octet> buffer, in long size);
   const short UPDATE_ALL_STATE=0;

   void lock();
   void unlock_abort();
   void unlock_propagete_val(in short upd_id, in sequence<octet> buffer, in long size);
   void unlock_propagate_op();

   void notify(short nf_id, any data);
}
```

Every replica must inherit the Replicated_object interface. The replica becomes a part of an ensemble by calling connect_new_replica on the ensemble (on any replica which belongs the the ansable). This call also sets the replication mechanism on. The replication mechanism is set off and the replica is disconnected from the ensemble by calling disconnect.

Every descendant class must define the methods get_state and update_state. The method get_state should encode all the object's state into a sequence of bytes. The method update_state updates the object state according to the contents of the buffer. The way to do it is determined by the parameter upd_id. Only one value for this parameter is defined by the constant UPDATE_ALL_STATE. This constant is used when the contents of the buffer have been generated by the get_state method. Alternatives to the approach, e.g. defining a particular constant associated with each of the methods, can be defined in descendant classes.

Every "writer".method must start with the lock call and must be terminated with one of the unlock_... methods. The unlock_abort method should be used when no updates are required; the programmer is responsible for restoring the previous state. The unlock_propagate_val method can be used when the state change propagation (3.5.2) is to be performed. This call leads to invocation of the update_state method on all the replicas in the same ensemble. The unlock_propagate_op() is an implementation of operation propagation (3.5.1). This call leads to invocation of the proper method on all the replicas in the same ensemble.

The method notify is used to inform replicas about exceptional states, e.g. an replica with lost connection. The notification codes will be specified in the future. This interface proposal can be subject to a change.

3.10 Metaservices designed to support implementation

In the implementation, the following metaservices are expected to be used: The replica identification metaservice. This metaservice is intended to overcome the naming problem as described in Sect. 3.3.1 in a way independent of the underlaying system. In addition, the replica finding metaservice will be used to identify the most suitable replica the client is to be bound to. Finally, the network failure detection metaservice will be used to detect network failures and to classify them in a system independent way.

4 Events

4.1 Basic principles & concepts

Event service is one of the object services built upon CORBA distributed objects environment. Event service does not require any CORBA extensions. Basically, event service is designed for two purposes. The first purpose is to send information about an event without interest in receiver’s identity. The second is reverse situation: receiving information without an interest in the sender’s identity. For example, if a document has been changed, it is useful to inform the make system about this change. The make system is not interested who has changed the document, but all dependent documents must be recompiled.

The event service defines two kinds of objects: the supplier and the consumer. Suppliers produce event data and consumers process event data. Event data are transferred from suppliers to consumers using standard CORBA requests.

There are two approaches to initiate event communication (*communication* for short). The "push" model allows a supplier to send event data to a consumer. The "pull" model allows a consumer to request event data from a supplier. The pull model can use both the blocking and non-blocking communication methods.

The communication can be generic or typed. In the generic communication, only single untyped parameter can be transferred. Suppliers and consumers must "marshal" and "unmarshal" event data before send and after receive. In the typed communication, a new IDL interface is defined and event data are passed as arguments of functions.

Event channel is an object allowing multiple suppliers to communicate with multiple consumers. The event channel acts as the supplier and consumer simultaneously. An event channel can support generic or typed communication. A supplier or a consumer communicates with the channel using push or pop strategy. Event channels are standard CORBA objects and all communication is done by using standard requests.

4.1.1 Push model

In the push model, suppliers push event data to consumers. This is done by invoking *push* operation on the *PushConsumer* interface. To establish push style communication, the consumer must send the *PushConsumer* interface reference to a supplier. This interface defines operation *push*, which can be used to transfer untyped event data, and the operation *disconnect*, which allows to break the communication. The supplier can send the *PushSupplier* interface reference to the consumer. This interface defines only the *disconnect* operation. If a supplier does not send the *PushSupplier* interface reference to a consumer, the consumer cannot break the communication.

4.1.2 Pull model

In the pull model, consumers pull event data from suppliers. This is done by invoking *pull* or *try_pull* operations on the *PullSupplier* interface. To establish pull style communication, the supplier must send the *PullSupplier* interface reference to a consumer. This interface defines operations *pull* and *try_pull*, which can be used to obtain untyped event data, and operation *disconnect*, which allows for breaking the communication. The operation *pull* blocks the consumer if no event data are ready. The operation *try_pull* returns boolean flag *has_events*, and eventually the data. The consumer can send the *PullConsumer* interface reference to the supplier. This interface defines only the *disconnect* operation. If a consumer does not send the *PullConsumer* interface reference to a supplier, the supplier cannot break the communication.

4.1.3 Event Channel

Event channel is a service that intermediate event data transfer among suppliers and consumers. The event channel communicates with suppliers as it was a "proxy_consumer" and with consumers as it was a "proxy_supplier". In every relation, push style or pop style communication can be used. The event channel provides asynchronous communication; it does not need to supply event data to its consumers at the same time when it consumes data from its suppliers.

Event channels are created with no connection to any suppliers or consumers. They define interfaces to create proxy suppliers and proxy consumers. There are two steps to establish communication through an event channel: The first step is to obtain a proxy consumer or proxy supplier. The second step is to connect with the corresponding proxy. Two step connection process is useful if an application needs to connect two or more event channels together. The application can obtain proxy consumer interface from one event channel, proxy supplier interface from second event channel, and connect these proxy interfaces.

The basic concept of the event channel is schematically illustrated on Fig.4.1.

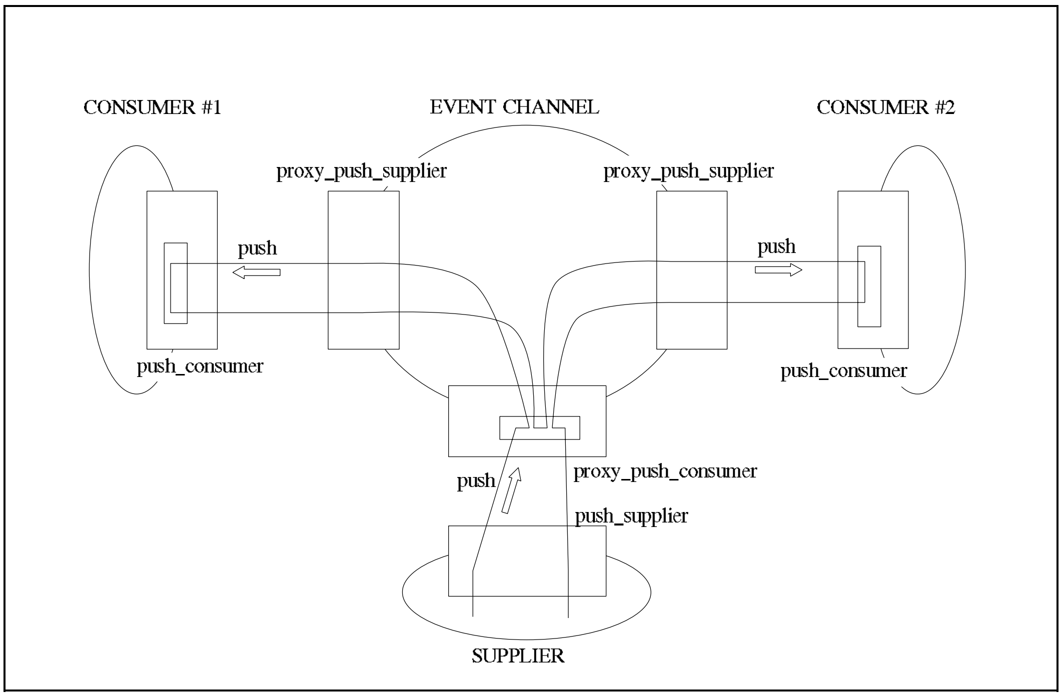

Figure 4.1 - The event channel concept

The figure illustrates an event channel, which is connected with one supplier, and two consumers. In all three connections, the push communication model is employed. If the supplier wants to send an event (event data flow is represented by arrows), it invokes the *push* method on the *proxy_push_consumer* interface. This interface is a part of the event channel object. The event channel now invokes the *push* method on *push_consumer* interfaces of all connected consumers. Note that *push_supplier* and *proxy_push_supplier* interfaces defines only the *disconnect* method, and thus they play no role in event delivery process.

4.2 Potential enhancements to event channel concept

OMG specification defines only the basic set of interfaces; the semantics of methods on those interfaces is specified only partially. To satisfy specific application requirements, the basic channel concept must be enhanced in many ways. Next paragraphs show some potential enhancements to the event channel specification.

4.2.1 Setting the semantics of delivery

Some event channel implementations can support only one specific event delivery semantics. For example, the channel can deliver event data *at least once* to every connected consumer, with no event history (this means that event data are delivered only to consumers, which are connected at the time of the event delivery, and they are not delivered to consumers connected later).

If the event channel supports a more complex delivery semantics (for example event history), there can be a need of runtime setting of some parameters, e.g. lifetime of events. If the channel supports event history, suppliers can be interested in event lifetime setting. Thus, event channel must provide a method to control delivery semantics. One possible solution is to define additional methods on appropriate proxy interfaces. Another approach can allows for controlling the channel by optional *control* fields in event data structure. In principle, the specific form of the event channel control is left on the application programmer.

4.2.2 Persistent event channel

For some applications, one can find it useful to make the event channel persistent. In this case, event data will survive system crashes, and the event channel will exist even if none of its clients is running. An event channel is a standard CORBA object, and thus it can be made persistent using CORBA object persistency service.

4.3 Detailed description of services

4.3.1 push_supplier

```
interface PushSupplier
  {
   void disconnect_push_supplier();
  };
```

The *PushSupplier* interface is inherited by the supplier, which uses the push model of communication. The supplier can send the *PushSupplier* interface reference to the consumer, allowing it to break the communication by using the *disconnect_push_supplier* method.

4.3.2 push_consumer

```
interface PushConsumer
  {
   void push(in any data) raises(Disconnected);
   void disconnect_push_consumer();
  };
```

The *PushConsumer* interface is inherited by the consumer, which uses the push model of communication. The supplier invokes *push* method on the *PushConsumer* interface, and generic event data are passed as argument to this invocation. In addition, the *PushConsumer* interface defines method *disconnect_push_supplier*, which allows the supplier to break the communication.

4.3.3 pull_supplier

```
interface PullSupplier
  {
   any pull() raises(Disconnected);
   any try_pull (out boolean has_event) raises(Disconnected);
   void disconnect_pull_supplier();
  };
```

The *PullSupplier* interface is inherited by the supplier, which uses the pull model of communication. The consumer invokes *pull* or *try_pull* method on the *PullSupplier* interface, and generic event data are passed as return value of this invocation. In addition, the *PullSupplier* interface defines method *disconnect_pull_supplier*, which allows the consumer to break the communication. If event data are not available, the *pull* method will block the consumer. The *try_pull* method allows the consumer to detect presence of event data by using boolean return value *has_event*.

4.3.4 pull_consumer

```
interface PullConsumer
  {
   void disconnect_pull_consumer;
  };
```

The *PullConsumer* interface is inherited by the consumer, which uses the pull model of communication. The consumer can send the *PushConsumer* interface reference to the supplier, allowing it to break the communication.

4.3.5 event_channel

```
 interface EventChannel
  {
   ConsumerAdmin for_consumers();
   SupplierAdmin for_suppliers();
   void destroy();
  }
```

The *EventChannel* interface is the basic interface of the newly created event channel. It provides methods for adding consumers and suppliers, and finally for destroying the channel. By invoking *for_consumers* method, the client obtains reference to the *ConsumerAdmin* interface. This reference can be passed to application, which wants to connect a number of consumers; however it is not permitted to connect any supplier. Similarly, by invoking *for_suppliers* method, the client obtains reference to the *SupplierAdmin* interface. This reference can be passed to the application, which wants to connect a number of suppliers, however it is not permitted to connect any consumer.

4.3.6 supplier_admin

```
interface SupplierAdmin
 {
  ProxyPushConsumer obtain_push_consumer();
  ProxyPullConsumer obtain_pull_consumer();
 }
```

The application, which possess the reference to the *SupplierAdmin* interface can obtain push or pull proxy consumers and thus connect new suppliers to the event channel.

4.3.7 consumer_admin

```
interface ConsumerAdmin
 {
  ProxyPushSupplier obtain_push_supplier();
  ProxyPullSupplier obtain_pull_supplier();
 }
```

An application, which passes the reference to the *ConsumerAdmin* interface can obtain push or pull proxy suppliers and thus connect new consumers to the event channel.

4.3.8 proxy_push_supplier

```
interface ProxyPushSupplier: EventComm:: PushSupplier
 {
  void connect_push_consumer(in EventComm::PushConsumer push_consumer)
            raises(AlreadyConnected);
 }
```

The interface *ProxyPushSupplier* inherits from the interface *PushSupplier*. The additional method *connect_push_consumer* provides the associated event channel with the reference to the *PushConsumer* interface, allowing the channel to push event data to the consumer.

4.3.9 proxy_push_consumer

```
interface ProxyPushConsumer: EventComm:: PushConsumer
  {
   void connect_push_supplier(in EventComm::PushSupplier push_supplier)
              raises(AlreadyConnected);
  }
```

The interface *ProxyPushConsumer* inherits from the interface *PushConsumer*. The additional method *connect_push_supplier* provides the associated event channel with the reference to the *PushSupplier* interface, allowing the channel to break the communication.

4.3.10 proxy_pull_supplier

```
interface ProxyPullSupplier: EventComm:: PullSupplier
  {
   void connect_pull_consumer(in EventComm::PullConsumer pull_consumer)
              raises(AlreadyConnected);
  }
```

The interface *ProxyPullSupplier* inherits from the interface *PullSupplier*. The additional method *connect_pull_consumer* provides the associated event channel with the reference to the *PullConsumer* interface, allowing the channel to break the communication.

4.3.11 proxy_pull_consumer

```
interface ProxyPullConsumer: EventComm:: PullConsumer

  {
   void connect_pull_supplier(in EventComm::PullSupplier pull_supplier)
              raises(AlreadyConnected);
  }
```

The interface *ProxyPullConsumer* inherits from the interface *PullConsumer*. The additional method *connect_pull_supplier* provides the associated event channel with the reference to the *PullSupplier* interface, allowing the channel to pull event data from the supplier.

4.4 An example of event service[4]

--- file supplier.idl ---

```
#include "events.idl"
```

[4] This example was created by Jan Alexander

```
interface Supplier: CosEventComm::PushSupplier {
  readonly attribute short value;

  void change_data( in short new_value );
};
```

--- file supp_i.h ---

```
#include "supplier.hh"

class PushSupplier_i : public SupplierBOAImpl {
  EventChannelRef event_channel;
  SupplierAdminRef supplier_admin;
  ProxyPushConsumerRef proxy_push_consumer;
  short value;
public:
  PushSupplier_i( char* hostname );
  ~PushSupplier_i( );

  virtual short value( CORBA::Environment &env );

  virtual void disconnect_push_supplier( CORBA::Environment &env );
  virtual void change( short new_value, CORBA::Environment &env );
};
```

--- file supp_i.cc ---

```
// For readability, exception handling is omitted from this code

#include "supp_i.h"

PushSupplier_i::PushSupplier_i( char* hostname )
{
  event_channel = EventChannel::_bind( "", hostname, IT_X );
  supplier_admin = event_channel->for_suppliers( IT_X );
  proxy_push_consumer = supplier_admin->obtain_push_consumer( IT_X );
  proxy_push_consumer->connect_push_supplier( this );
  value = 0;
}

PushSupplier_i::~PushSupplier_i( )
{
  // Here we will release all objects (proxy_push_consumer ...)
}
```

```
short PushSupplier_i::value( CORBA::Environment &env )
{
  return value;
}

void PushSupplier_i::disconnect_push_supplier( CORBA::Environment &env )
{
  // Here we will do something when we are disconnected from event channel
}

void PushSupplier_i::change( short new_value, CORBA::Environment &env )
{
  // Here we will send notification event to all consumers
  value = new_value;
  // notification_data is struct any filled with message informing about
  // change on value
  proxy_push_consumer->push( notification_data );
}
```

--- file consumer.idl ---

```
#include "events.idl"

interface Consumer: CosEventComm::PushConsumer {
};
```

--- file cons_i.h ---

```
#include "consumer.hh"

class PushConsumer_i : public ConsumerBOAImpl {
  EventChannelRef event_channel;
  ConsumerAdminRef consumer_admin;
  ProxyPushSupplierRef proxy_push_supplier;
public:
  PushConsumer_i( char* hostname );
  ~PushConsumer_i( );

  virtual void push( CORBA::any &data, CORBA::Environment &env );
  virtual void disconnect_push_consumer( CORBA::Environment &env );
};
```

--- file cons_i.cc ---

```
// For readability, exception handling is omitted from this code

#include "cons_i.h"
```

```
PushConsumer_i::PushConsumer_i( char* hostname )
{
  event_channel = EventChannel::_bind( "", hostname, IT_X );
  consumer_admin = event_channel->for_consumers( IT_X );
  proxy_push_supplier = consumer_admin->obtain_push_supplier( IT_X );
  proxy_push_supplier->connect_push_consumer( this );
}

PushConsumer_i::~PushSupplier_i( )
{
  // Here we will release all objects (proxy_push_supplier ...)
}

void PushConsumer_i::disconnect_push_consumer( CORBA::Environment &env )
{
  // Here we will do something when we are disconnected from event channel
}

void PushConsumer_i::push( CORBA::any &data, CORBA::Environment &env )
{
  if ('Value has changed') {
    //Do  something  with  changed  value  (for  example  get  new  value  from
supplier)
    // ....
  }
}
```

4.5 Typed Event Communication - basic ideas and concepts

Typed event communication is an enhancement to the previously described generic (untyped) event communication. It allows to define IDL interface for event types, and, consequently, to react on an event by calling an operation on the associated interface.

4.5.1 Typed push model

In the push style typed event communication, both supplier and consumer must support a mutually agreed interface <I>. The interface <I> may contain any operation with the following restriction:

- all parameters must be of the mode "in"
- no return values are permitted (return value must be of void type)

To establish push style typed communication, each consumer must send the TypedPushConsumer interface reference to its supplier. This interface defines the operation *get_typed_consumer*, which can be used to obtain the reference to the interface <I>, and the operation *disconnect*, which allows for breaking the communication. Once the supplier has the reference to the interface <I>, it can invoke operations on the interface <I>. The supplier can send the PushSupplier interface reference to the consumer. This interface only defines the *disconnect* operation. If the supplier does not send PushSupplier interface to a consumer, the consumer cannot break the communication.

4.5.2 Typed pull model

In the typed pull model, the interface <I> can be converted to interface Pull<I> by using the following scheme:

For every operation in the <I> interface, the interface Pull<I> contains two operations:

- *pull_o*, with all "in" parameters converted to "out" parameters. When the operation is called, it returns event data in its "out" parameters. If the data are not available, consumer’s thread blocks.

- *try_pull_o*, with all "in" parameters changed to "out" parameters.

To establish pull style typed communication, the supplier must send the *TypedPullSupplier* interface reference to a consumer. This interface defines the operation *get_typed_supplier*, which can be used to obtain the reference to the interface Pull<I>, and operation *disconnect*, which allows to break the communication. The operations *pull...* block the consumer if event data are not ready. The operations *try_pull...* return boolean flag *has_events*, and eventually the data. The consumer can send the PullConsumer interface reference to the supplier. This interface only defines the *disconnect* operation. If a consumer does not send the *PullConsumer* interface to a supplier, the supplier cannot break the communication.

4.5.3 Typed event channel

In principle, typed event channel provides the same services as the untyped channel. It allows multiple suppliers to communicate with multiple consumers.

4.6 Comparing untyped and typed models

Typed event communication can be interpreted as a small extension to the untyped one. In the untyped model, the application programmer is forced to define his own semantics of the event data structure. He can marshal information into the event structure while the supplier is preparing data to be sent, and unmarshal event structures immediately after receiving them. In the typed model, the programmer can define an IDL interface, and the CORBA system will marshal and unmarshal data automatically. There are no other differences between the typed and the untyped model of the event communication.

4.7 Implementing untyped model

The most important part of event service implementation is the event channel implementation. OMG documents do not suggest any specific form of the event channel implementation. Basically, an event channel must distribute event data between suppliers and consumers, but semantics of delivery and quality of service is left up to the individual implementation of the event channel. For example, an event channel can distribute events only between suppliers and consumers, which are simultaneously connected to the event channel. Another event channel implementation can accumulate all event data (store the event history) and deliver all the accumulated data to a newly connected consumer.

In our approach, the event channel has *at least one* delivery semantics and no event data accumulation. If an application requests another event channel semantics, the proposed implementation can be extended.

The implementation constructs an event channel from the set of objects (Fig. 4.2).

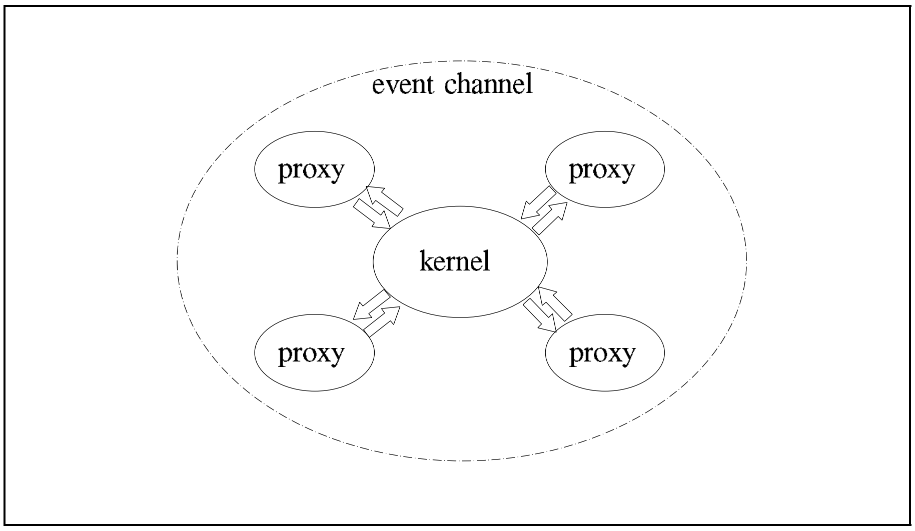

Figure 4.2 - The event channel implementation

The *kernel* object implements the central administration unit of the channel and it is connected with a number of *proxy* objects. The kernel object contains the lists of connected suppliers and consumers and it communicates with all proxy objects. A proxy object is connected with an individual supplier or consumer. There are four proxy object types, one for each of the four types of communication (push/pull, consumer/supplier).

Due to the event interface being rather low-level and very generic, all the functionality will be implemented in a form of a single module. The interface of this module, compliant with [OMG94a], represents the event metaservice.

5 Persistence

5.1 The key issues of the persistent service

We will start this section by defining several important concepts used here:

persistent object - an object, the lifetime of which can exceed the lifetime of the application it is used in.

persistent state (of a persistent object) - the tuple of values corresponding to the object's attributes. Certain object attributes, usually with a very limited lifetime, can be considered auxiliary; their values may not be embodied in the object's persistent state.

dependencies (of a persistent object PO) - the set of all the objects targeted by an reference attribute of PO.

transitive closure of dependencies (of a persistent object PO) - the set of all objects reachable from PO via reference attributes transitively over all nested dependencies;; this is an analogy of the "deep copy" concept. Sometimes, we find it useful to limit the depth of nesting to a maximum n; we call a subset of the transitive closure defined by this limit *n-th level dependencies PO*.

5.1.2 Determining object persistence

5.1.2.1 Static

At the compilation time, some of the application objects are statically denoted as being persistent, typically by inheriting from a persistent base class. Such objects will hold their persistent property forever within the time scope the application and there is no way they can cease being persistent at runtime.

This approach is relatively easy to implement. However, the necessity to set the object persistent property statically and impossibility to give it up later is not very convenient for a user. Thus, this black-or-white approach is not desirable.

5.1.2.2 Semidynamic

This approach is a modification of the static approach with a runtime enhancement: statically denoted objects can choose to *dynamically* turn the persistent property on and off. This provides the user with the runtime ability to decide about the object persistence. Yet this control is limited to those classes of objects which have been *statically preselected* by inheriting from the persistent object base class; again, this forces the user to take early decisions.

5.1.2.3 Dynamic

This highly desirable approach, which is suggested by the CORBA specification of persistent service [OMG94b] and implemented for example in Eiffel, allows the user to decide *dynamically* on the persistent feature of *all* objects (denoted as orthogonal persistence). Objects can turn their persistent feature on and off an arbitrary number of times within the lifetime of the application.

5.1.3 Creating a persistent object

In this paragraph, we analyze the process of creating a persistent object, the persistent property of which is turned on at the time of its creation.

5.1.3.1 Static object

A static persistent object is always created via a static constructor call. The constructor contains a piece of code that performs the act of persistent object creation. The drawback of the static creation is that the control of the destructor call is very limited, if impossible.

Another problem arises with static persistent objects global in scope. The constructor of such an object is called before the main () function of the program starts; in this case the persistent service module might not necessarily be properly initialized

5.1.3.2 Dynamic object

In the case of dynamic object creation, there seem to be two philosophically different approaches: either all intelligence of the persistent object creation is hidden inside the constructor body, resp. inside *all* the constructors of the corresponding class(es). Alternatively, all the work of object creation is performed by an overloaded `new` operator.

5.1.3.2.1 "Smart constructor" approach

The syntax of creating a new persistent object is the same as it would be for creating a classical volatile memory object:

```
Persistent Object *po = new Persistent Object ();
```

The code in the smart constructor body must ensure the following actions:

1) Assign a PID to the current *Persistent Object* instance,
2) add the current instance to the persistence store under the obtained PID , (call `this->save_state()`)
3) update internal tables.

5.1.3.2.2 "Smart new" approach

The `new` operator can be overloaded either on "per class" or "per program" basis.

a) Per class overloading

If there is a `new` operator overloaded *per class*, then it inherently knows the type of the class that should be created. In this case, the syntax look the same as for the smart-constructor approach:

```
PersistentObject *po = new PersistentObject ();
```

Thus, the user must define a **new** operator for every class that can have persistent instances.

b) Per program overloading

Considering the case of having only one `new` operator overloaded per program (i.e.. the one that hides the standard library `new` operator), we come to conclusion, that there is a need of explicitly supplying the information about the class type of the object being created as the second argument of the `new` operator. The C++ language offers no way of retrieving and processing the type information at runtime. Thus, the example from a) would look as follows:

```
PersistentObject *po = new ("PersistentObject") PersistentObject ();
```

Looking at the code fragment, there is a threat of creating inconsistency in the `new` operator arguments by mistake. This could be overcome using a macro call.

5.1.4 Saving and loading object state

For saving and loading object persistent state, there are two principal approaches: First, by providing specialized object methods for every persistent object which serve to save and restore the object's persistent state. For simplicity, we assume this to be done by two "comprehensive" methods *save_state* and *load_state* (5.4.1) designed to save/restore all the persistent object state at once. Basically, the methods can be made public, and thus a part of the object IDL interface, or they can be made accessible to the underlaying persistency control only - e.g. by using the friend construct in C++. While making *save_state* and *load_state* public works nicely also for distributed calls, the later alternative is limited for the server side only.

Second, saving and loading of an persistent object's state can be done by placing the object into a persistent address space. A number of approaches towards the persistent address space architecture exist; to those known at most belong:

- a fragmented address space (just the designated sections of an application address space are persistent); in this case such a section often stores an object cluster, e.g. [AJJ+92,AJL92];

- continuous address space; the entire address space or its single continuous part is subject to persistency.

5.1.5 When to update the persistent state

The updating of persistent state can be application controlled or system controlled. In the former case, updating is done by explicit calls of *save_state* and *load_state* (5.4.1) from the application.

When updating is system controlled, it is alternatively done in one of the following three methods:

- Updating is done at system well-defined moments, e.g. when deleting an object, an error/exception is raised, the end of the application is reached, etc. Basically, this approach is used in [ORBIXa,ORBIXb];

- Updating is transaction based; it means that the updating is done at the end of the out-most level transaction. Naturally, this strategy requires the persistency implementation to cooperate with transaction support. In Orbix, this can be achieved by employing the filter concept [ORBIXa,ORBIXb];

- True persistent virtual memory is employed; the state of an object stored in such memory is implicitly persistent and is up-to-date at all times.

5.1.6 What is to be saved

In addition to the issue of when to update the persistent state (5.1.5), the question arises of *what exactly is to be saved.* The following alternatives can be identified:

- Just the persistent state of an object is saved (the object's dependencies are not saved);

- The persistent state of an object is saved together with the persistent state of the object's dependencies. The dependencies can be stored up to n-th level; in general, the transitive closure can be stored.

To implement storing of dependencies the following techniques are used:

- The current status of relations among objects is not evaluated; dependencies are stored implicitly by defining groups of object; such a group must be closed with respect to inter-object relations. As examples see clusters in COOL [HMA90,AJJ+92,ALJ92] or groups in SOM [IBM93a,IBM93b].

- Full persistent state is stored with explicit evaluation of dependencies. This can be either application controlled (explicit) or system controlled (based on hardware-supported reference identification). In the later case, the techniques of early or late pointer swizzling are typically used [SKW92,VD92].

5.2 Implementation architecture

The proposed architecture can be divided into three parts hierarchically related with respect to their functionality. The Typed Data Part encapsulates the metaservice responsible for accessing datastores; its only purpose is to provide a common interface for accessing various datastore classes. The Persistent Object Part uses the underlying Typed Data Part to store the contents of individual persistent objects. The Persistent Service Part coordinates the work of the previous two parts to provide the client with a simple-to-use persistent service interface.

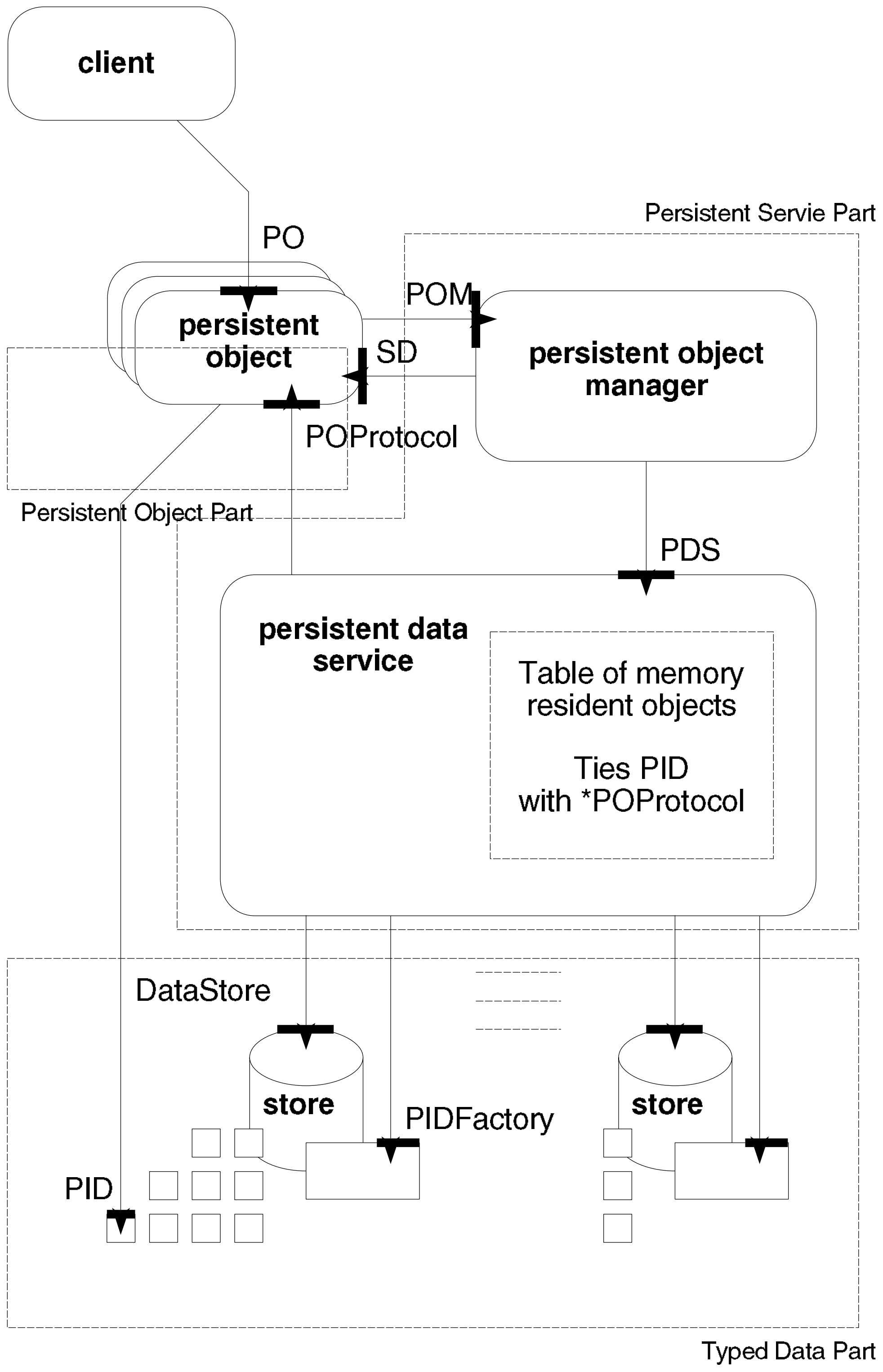

Figure 5.1 Persistent service architecture

5.3 Typed Data Part

5.3.1 Store Access Module

The Store Access Module represents the metaservice used to access data on an external storage media. Apart from the storage access primitives, the module is also capable of providing single level transactions necessary for the crash recovery support in the hierarchically higher parts.

The module consists of four interfaces. The first two interfaces, called *PID* and *PIDFactory*, introduce the notion of a unique persistent identifier, as defined in [OMG94b]. The other two, called *DataStore* and *DataStoreFactory*, provide means of accessing the external store.

```
module StoreAccess {

    typedef unsigned long       ulong;
    typedef sequence <octet>    tBuffer;

    exception Internal { };
    exception InvalidMediaID { };
    exception StoreIsFull { };
    exception NoTransaction { };
    exception NestedTransaction { };

    interface PID : CosPersistencePID::PID {
        attribute string media_ID;
        attribute ulong location_ID;
        void rewrite (void)
            raises (Internal, StoreIsFull);
        void rewind (void)
            raises (Internal);
        void save (in tBuffer data)
            raises (Internal, StoreIsFull);
        tBuffer load (in ulong length)
            raises (Internal);
        void remove ( );
    };

    interface PIDFactory {
        PID get_root_PID ( )
            raises (Internal);
        PID create_unique_PID ( )
            raises (Internal, StoreIsFull);
        };

    interface DataStore {
        void trans_begin ( )
            raises (Internal, StoreIsFull, NestedTransaction);
        void trans_commit ( )
            raises (Internal, StoreIsFull, NoTransaction);
        void trans_abort ( )
```

```
            raises (Internal, NoTransaction);
        PIDFactory get_PIDFactory ( );
        void destroy_on_remove ( );
    };

    interface DataStoreFactory {
        DataStore get_DataStore (in string media_ID)
            raises (Internal, InvalidMediaID);
    };
};
```

5.3.1.1 The *PID* interface

The *PID* interface masks the differences among various storage media classes, thus creating the unified abstraction of a datastore containing records addressable by PIDs. The primitives are not aware of the structure of the data being stored - at this level, each record is simply a stream of bytes of an arbitrary length.

Each *PID* instance denotes a single location in a store. Apart from the *location_ID* itself, the *PID* contains two strings identifying the store class and instance. Thus, the persistent identifier is global in scope.

```
interface CosPersistencePID::PID {
    attribute string datastore_type;
    string get_PIDString ( );
};

interface PID : CosPersistencePID::PID {
    attribute string media_ID;
    attribute ulong location_ID;
    void rewrite (void)
        raises (Internal, StoreIsFull);
    void rewind (void)
        raises (Internal);
    void save (in tBuffer data)
        raises (Internal, StoreIsFull);
    tBuffer load (in ulong length)
        raises (Internal);
    void remove (void);
};
```

The *rewrite* and *save* methods are used to put the data into a store location denoted by the *location_ID* attribute. Rewriting a location erases any previously stored data and prepares it to be filled with subsequent *save* calls. The *rewind* and *load* methods are used to retrieve the data in a similar manner.

5.3.1.2 The *PIDFactory* class

As described in [OMG94b], *PID* instances are created by an appropriate factory class. Once created, the *PID* remains valid until a *PID::remove* call is issued, as described in [OMG94a].

```
interface PIDFactory {
    PID get_root_PID ( )
        raises (Internal);
    PID create_unique_PID ( )
        raises (Internal, StoreIsFull);
};
```

As the persistent IDs are being devised by the store itself, an extra care needs to be taken to provide the client with means of obtaining the store contents. By a convention, the *get_root_PID* method returns a single *PID* denoting a special record to be used for the purposes of maintaining a directory of stored data. It is up to the client to specify a strategy for the root record usage.

5.3.1.3 The *DataStore* class

Each instance of the *DataStore* class represents a single datastore capable of storing and retrieving the persistent data.

```
interface DataStore {
    void trans_begin ( )
        raises (Internal, StoreIsFull, NestedTransaction);
    void trans_commit ( )
        raises (Internal, StoreIsFull, NoTransaction);
    void trans_abort ( )
        raises (Internal, NoTransaction);
    PIDFactory get_PIDFactory ( );
    void destroy_on_remove ( );
};
```

As the persistent service is expected to be secure with respect to system failures, a certain level of fault tolerance is required from the underlying *DataStore* as well. Thus, a simple one level transaction mechanism is introduced - any changes made to the store contents after the *trans_begin* call are take effect after the *trans_commit* call successfully returns.

A particular *DataStore* implementation is always associated with a corresponding implementation of the *PIDFactory* and *PID* classes. The store provides its client with means to obtain an appropriate *PIDFactory* by calling the *PIDFactory::get_PIDFactory* method.

5.3.1.4 The *DataStoreFactory class*

The only purpose of the *DataStoreFactory* is to manufacture *DataStore* instances to be used by the persistent service.

```
interface DataStoreFactory {
    DataStore open_DataStore (in string media_ID)
        raises (Internal, InvalidMediaID);
};
```

Once created, a *DataStore* object remains valid until a *DataStore::remove* call is issued. Removing the store, however, does not destroy its data, unless the *DataStore::destroy_on_remove* method has been called prior to *DataStore::remove*.

5.3.1.5 Comments

5.3.1.5.1 Exploiting smart stores

The *DataStore* interface provides the datastore itself with no information about the data to be stored. This makes the exploitation of possibly existing smart features of the underlying store (e.g. in case of using a database as a datastore) very difficult, if not impossible. It might also hinder any attempts to optimize the process of storing the data.

A solution to this problem could be based on user enhancing the *PID* and possibly other related interfaces by inheritance.

5.3.1.5.2 Using several stores at once

Complex client applications are likely to use several *DataStore* instances simultaneously. Since both the client and the persistent service implementation itself need to be able to locate the appropriate *DataStoreFactories*, a need for a common interface arises.

As [OMG94a] indicates, the Trading Object Service is meant to provide the necessary interface. This service, however, has not been standardized at the time of our persistent service design. Currently, we are evaluating the option of using the trading service specification as standardized in [OMG95a].

Another interface standardized by OMG is the generic factory. After being expanded with a support for registering specific object factories, the generic factory interface might be used for the purpose of obtaining specific *DataStore* instances as necessary.

5.3.2 Data Storage Module

Usually, the *DataStore* interface will be used to store single pieces of typed data, i.e. chars, integers, strings etc. The purpose of the Data Storage Module is to provide an interface to store these types easily and effectively.

In the first version of the implementation, a simple set of macros can provide the necessary functionality, as is indicated bellow:

```
#define SaveVar(PID,Var,Bfr,IT_env) { Bfr._buffer = (unsigned char *) &Var;
  Bfr._length = sizeof(Var); Bfr._maximum = sizeof(Var);
  PID->save (Bfr,IT_env) }
#define LoadVar(PID,Var,Bfr,IT_env) { Bfr = PID->load (sizeof(Var),IT_env);
  memcpy ((char *) &Var, Bfr._buffer, Bfr._length); }
```

5.3.2.1 Comments

Perhaps the only drawback of this implementation is an absence of a canonical form of the stored data. Until now, we have not encountered a serious need for such a form.

5.4 Persistent Object Part

5.4.1 Object Storage Module

This module is responsible for saving and restoring the persistent attributes of an individual object, together with exporting the type and object reference information.

As the only entity authorized to access the persistent object's internal state is the object itself, the services of this module are exported in a form of methods provided by each persistent object. Using the terms introduced in [OMG94b], this is a part of the proprietary protocol.

```
module ObjectStorage {

    typedef CosNaming::Name TID;

    interface POListItem {
        attribute PO object;
        attribute POListItem next;
    };

    interface POList {
        attribute POListItem first;
        attribute POListItem last;
    };


    interface POProtocol {
        TID get_TID ( );
        void save_state (StoreAccess::PID pid);
        void load_state (StoreAccess::PID pid);
        void get_references (out POList references);
        void set_references (inout POList references);
    };
};
```

5.4.1.1 The *POProtocol* class

This is the interface each persistent object needs to be equipped with in order to be able to cooperate with the persistent service.

```
interface POProtocol {
    TID get_TID ( );
    void save_state (StoreAccess::PID pid);
    void load_state (StoreAccess::PID pid);
    get_references (out POList references);
    void set_references (inout POList references);
};
```

The implementation of the persistent data service needs to be able to identify object classes at runtime. As the C++ language does not provide the runtime type information to the user, the *POProtocol* class exports the *get_TID* method to do this - the method returns a user-supplied identifier of the object's type.

The *save_state* and *load_state* methods access the object's persistent state minus the references to other persistent objects. The *get_references*, resp. *set_references*, methods store, resp. restore, the references to other persistent objects.

The *set_references* method is expected to remove the processed references from the reference list passed as the inout parameter. In class hierarchy implied by inheritance, this action is necessary in order to be able to reuse the predecessor's *get_references* and *set_references* methods.

5.4.1.2 The *POList* and *POListItem* classes

These two classes are used in exporting the list of references to other persistent objects. We prefer the dynamically linked list of items over the IDL sequence, as the operation of concatenating and separating several linked lists, which is likely to be used in the *get_references* and *set_references* methods, is less time- and memory-consuming compared to the same operation performed on sequences.

```
interface POListItem {
    attribute POProtocol object;
    attribute POListItem next;
};

interface POList {
    attribute POListItem first;
    attribute POListItem last;
};
```

5.4.2 Object Factory Module

When restoring a transitive closure of a persistent object, the persistent data service implementation needs a way to create (rebuild) object instances to be filled with data from the datastore. Thus, a generic object factory is required capable of creating an object instance of the type corresponding to the given type ID.

In practice, the generic factory interface is more suitable for the purpose of obtaining specific object instances after being extended with a support for registering specific object factories.

5.5 Persistent Service Part

This part incorporates both the Persistent Data Service and the Persistent Object Manager; both specified in [OMG94b].

According to the OMG specification, the main role of the Persistent Object Manager is to dispatch a function call to the appropriate Persistent Data Service. Since we have only one Persistent Data Service instance, the interface of the Persistent Object Manager simply passes all requests through to the instance.

5.5.1 Reference Management Module

5.5.1.1 Unique object identifiers

As the implementation of the persistent service is expected to work with the C++ objects as well as with CORBA objects, we can not use the CORBA object IDs to identify the object instances. We need another kind of an identifier associated with every persistent object to encode inter-object relations in persistent store.

Fortunately, there is no need to devise another mechanism of object identification. Each persistent object is already associated with its PID, all we have to do is enhance the PID semantics to act as an object identifier in addition to store location identifier.

5.5.1.2 List of memory resident objects

In an application, we need to prevent the persistent service from creating several object instances using the same persistent object image in the datastore. As PID is used as a persistent object identifier, a list of <PID, object reference> pairs maintained by the reference management module can be used to check whether an object has been already loaded.

5.5.2 Persistency Management Module

This module is the heart of the persistent service functionality. It exports the following specialized version of the Persistent Data Service *CosPersistencePDS::PDS* interface [OMG94b]:

```
module PersistencyManagement {

    interface PDS {
        PDS connect (in POProtocol object, in PID pid);
        void disconnect (in POProtocol object, in PID pid);
        void store (in POProtocol object, in PID pid);
        void restore (in POProtocol object, in PID pid);
        void delete (in POProtocol object, in PID pid);
    };
};
```

5.5.2.1 Connecting and disconnecting the object

As defined in [OMG94b], the *connect*, resp. *disconnect*, method turns on, resp. off, the automatic updating of the object's persistent state image in the datastore. In our implementation environment, the persistent data service has no way of detecting the moments when the object is modified. Thus we can only update the object image at system and transaction well-defined moments (5.1.5).

5.5.2.2 Storing the object

The persistent data service architecture dictates a necessity of storing and restoring object's transitive closure at once. Thus, the *store* request is implemented in the following steps:

1. The list of memory resident objects (5.5.1.2) is searched to find out if the object has a PID associated with it. A new PID is assigned if necessary.

2. The store location associated with the PID is emptied using the *DataStore::rewrite* call.

3. The *POProtocol::get_TID* method is called to get the object's type ID. The type ID is put into the store location associated with the PID.

4. The object is asked to save its state into the store location associated with the PID by calling its *POProtocol::save_state* method.

5. A list of references to object's dependencies is retrieved by calling *its POProtocol::get_references* method.

6. By issuing recursive calls to *PDS::store*, the transitive closure of the object's dependencies is saved.

7. The list of references obtained in step 5 is converted into a list of PIDs using the list of memory resident objects. The list of PIDs is written into the store location asociated with the object's PID.

5.5.2.3 Restoring the object

The *restore* request is implemented in the following steps:

1. The store location associated with the PID is prepared for reading by calling the *DataStore::rewind* method.

2. The object type ID is read from the store location associated with the object's PID.

3. The list of memory resident objects is searched to check if the object already is in memory. The generic object factory is called to create a new object if necessary.

4. The object is asked to load its state from the store location associated with the PID by calling its *POProtocol::load_state* method.

5. A list of PIDs of object's dependencies is retrieved from the store location associated with the object's PID.

6. By issuing recursive calls to *PDS::restore*, the transitive closure of the object's dependencies is loaded.

7. The list of PIDs obtained in step 5 is converted into a list of references using the list of memory resident objects. The object's attributes containing references are set by calling its *POProtocol::set_references* method.

5.5.2.4 Deleting the object

When issuing the *delete* call, the object is removed from the list of memory resident objects and its PID is destroyed using the *PID::remove* call. This results in the object being disassociated with its persistent state, the transient state of the object is not affected by this call.

6 Conclusion

This report is a study of three of the CORBA object services. Based mainly upon [OMG 92, OMG94a, OMG94b], the study aims to identify the main issues of the design and implementation architecture of the persistent, events, and replication object services. The research was done with an emphasis upon employing the services in a halfbridge and gateway CORBA implementation which are the main goals of the TOCOOS project.

After decoding all of the subtleties in the event service specification [OMG94a], the event service seems to be a relatively close and compact problem. As to replication, the service lacks OMG specification; its design thus needs an additional effort to overcome the specification gap. Finally, we have found the persistent service design to be a challenge with respect to so many low-level decisions that need to be identified and taken. All in all, a lot of work remains to be done in the persistency and, particularly, replication field.

ACKNOWLEDGEMENT

The authors would like to thank to Jan Alexander, Antonin Brcak, Michal Fadljevic, and Nguyen Duy Hoa for their overall intellectual contribution to many discusions. Moreover, Jan Alexander deserves a special credit for providing the demonstration program in Sect.4.4, Antonin Brcak for his contribution to Chapter 3 architecture, Nguyen Duy Hoa for the enourmous effort devoted to the implementation of the Data Store, and Michal Fadljevic for preparing the references.

References:

[AJJ+ 92] Paulo Amaral, Christian Jacquemot, Peter Jensen, Rodger Lea, and Adam Mirowski. Transparent object migration in COOL-2. Technical report, Chorus Systemes and Universite Pierre et Marie Curie, 1992.

[AJL92] Paulo Amaral, Christian Jacquemot, and Roger Lea. A model for persistent shared memory addresing in distributed systems. Technical report, Chorus systemes, 1992.

[AK85] D. A. Abramson and J. K. Keedy. Implementing a large virtual memory in a distributed computing system. In Proceedings of the 18th Hawaii Int. Conference on System Sciences, pages 515-522, 1985.

[ALJ92] Paulo Amaral, Rodger Lea, and Christian Jacquemot. Implementing a modular object oriented operating system on top of CHORUS. Technical report, Chorus Systemes, 1992.

[ANS94] American National Standards Institute ANSI. Basic Reference Model of Open Distributed Processing-Part 3: A Perspective Model, 1994.

[BM92] A. L. Brown and R. Morrison. A Generic Persistent Object Store. Software Engineering Journal, 7(2):161-168, 1992.

[Bou94] Francois Bourdon. The Automatic Positioning of Objects in COOL V2. Technical report, Service d‘Etudes communes des Postes et Telecommunications, 1994.

[Bul94] Groupe Bull. Compound LifeCycle Addendum OMG 94-5-6, 1994.

[CBHS93] Vinny Cahill, Sean Baker, Chris Horn, and Gradimir Starovic. The Amadeus GRT - Generic Runtime Support for Distributed Persistent Programming. Proceedings of OOPSLA-93, pages 144-161, 1993.

[CDK94] G. Coulouris, J. Dollimore, and T. Kindberg. Distributed Systems Concepts and Design. Addison-Wesley, second edition, 1994.

[Cor95] Oracle Corporation. Object Query Service OMG 95-1-3. 1995.

[CP89] D. E. Comer and L. L. Peterson. Understanding Naming in Distributed Systems. Distributed Computing, 3(2), 1989.

[dA93] Paulo Fernando V.C. Cardoso do Amaral. PAS: A Framework for studying the implementation of multiple address spaces. Technical report, Universite Pierre et Marie Curie - Paris, 1993.

[DdBF+92] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally Persistent Operating System. Computer Systems, 7(3):289-312, 1992.

[DRH+ 92] A. Dearle, J. Rosenbergr, F. Henkens, F. Vaughan, and K. Maciunas. An Examination of Operating System Support for Persistent Object System. In 25th Hawaii International Conference on System Services, volume 1, pages 779-789. IEEE Computer Society Press, 1992.

[Ede92] Daniel R. Edelson. Smart Pointers: They're Smart but They're Not Pointers. UCSC- CRL-92-97, Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz, 1992.

[Fra95] Framingham Corporate Center. Object Services RFP 5. OMG 95-3-25, 1995.

[FS94a] Paulo Fereira and Marc Shapiro. Garbage Collection and DSM Consistency. Technical report, INRIA, 1994.

[FS94b] Paulo Ferreira and Marc Shapiro. Garbage Collection of Persistent Objects in Distributed Shared Memory. Technical report, INRIA, 1994.

[HCF+ 95] D. Haginont, P. Y. Chevalier, A. Freyssinet, S. Krakowiak, S. Lacourte, J. Mossiere, and X.Rousset de Pina. Persistent Shared Object Support in the Guide System - Evaluation and Related Work. Technical report, Bull-IMAG/Systems, 1995.

[HK93] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In Proceedings of the 1993 Summer Usenix Conference, June 1993.

[HKPCS95] J. Hans, A. Knaff, E. Perez-Cotes, and F. Saunier. Arias: Generic Support for Persistent Runtimes. Technical report, Bull-IMAG/Systems, 1995.

[HMA90] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov. COOL: A Kernel Support for Object-Oriented Environments. Technical report, INRIA and Chorus Systemes, 1990.

[HRH95] Michael Hollins, John Rosenberg, and Michael Hitchens. Breaking the Information Hiding Barrier. Technical report, University of Sydney and University of Western Sydney and Australia, 1995.

[IBM93a] IBM Corp. SOMobjects Developer Toolkit - Collection Classes Reference Manual, 1993.

[IBM93b] IBM Corp. SOMobjects Developer Toolkit - Programmers Reference Manual - Version 2.0, 1993.

[KB92] J. K. Keedy and P. Broessler. Implementing Databases in the MONADS Virtual Memory. In Proceedings of the Fifth International Workshop on Persistent Object Systems, Springer Workshops in Computing series, San Miniato (Pisa) Italy, 1992.

[KN93a] Y. A. Khalidi and M. N. Nelson. A Flexible External Paging Interface. In Proceedings of the Usenix conference on microkernels and other architectures, September 1993.

[KN93b] Y. A. Khalidi and M. N. Nelson. The Spring Virtual Memory System. Technical Report SMLI-93-9, Sun Microsystems, 1993.

[KR89] J. L. Keedy and J. Rosenberger. Support for objects in the MONADS architecture. In Proceedings of the Int. Workshop on Persistent Systems, pages 202-213, 1989.

[LXC93] Swee Boon Lim, Lun Xao, and Roy Campbell. Distributed Access to Persistent Objects. Technical report, University of Illinois at Urbana Champain, Department of Computer Science, 1993.

[MCCK94] R. Morrison, R. C. H. Connor, Q. J Cutts, and G. N. C. Kirby. Persistent Possibilities for Software Environments. IEEE Computer Society Press, pages 78-87, 1994.

[MGH+ 94] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. In Proceedings of Compcon Spring 1994. IEEE, February 1994.

[MGNS91] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented Objects for Distributed Abstraction. Technical report, INRIA, 1991.

[Mul94] S. Mullender, editor. Distributed Systems. Addison-Wesley, second edition, 1994.

[OMGa] Common Object Model Specification, Draft version 0.2, OMG 94-10-9.

[OMGb] Common Object Model Specification, Draft version, OMG 94-9-14.

[OMG92] Object Service Architecture, OMG 92-8-4, August 1992.

[OMG92a] Kala-Standardizing on Object Meta Services, Brief Response to the OMG services, Request for Information, OMG 92-4-5, 1992

[OMG93] Common Object Request Broker: An Architecture and Specification, OMG 93-12-29, 1993.

[OMG94a] Common Object Services Volume I, OMG 94-1-1, 1994.

[OMG94b] Persistent Object Service Specification, OMG 94-10-7, 1994.

[OMG94c] Comparsion of the OMG and ISO-FCCITT Object Models, OMG 94-12-30, 1994.

[OMG94d] Universal Networked Objects, ORB 2.0 RFP Submission, OMG 94-9-32, 1994.

[OMG95a] ODP Trading Function Final Draft, ISO/IEC DIS 13235, OMG 95-7-6, 1995

[OMK93] D. B. Orr, R. W. Mecklenburg, and R. Kuramkote. Strange Bedfellows: Issues in Object Naming Under Unix. In Proceedings IEEE93, pages 141-145, Department of Computer Science, University of Utah, Salt Lake City, UT 84112, USA, 1993.

[ORBIXa] Orbix, Programmer's Guide. IONA Technologies Ltd. Dublin, 1994

[ORBIXb] Orbix, Advanced Programmer's Guide. IONA Technologies Ltd. Dublin, 1994

[PSWL94] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna. Technical report, Computing Laboratory, University of Newcastle upon Tyne, 1994.

[RHB+ 90] J. Rossenberg, F. Henskens, A. L. Brown, R. Morrison, and D.Munro. Stability in a Persistent Store Based on a Large Virtual Memory. In Security and Persistence, Workshops in Computing, pages 229-245. Springer-Verlag, 1990.

[SDP92] Marc Shapiro, Peter Dockman, and David Plainfosse. SSP Chains: A Robust Distributed References Supporting Acyclic Garbage Collection. Technical report, INRIA, 1992.

[SDP93] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of the Arjuna Distributed Programming System. Technical report, Computing Laboratory, University of Newcastle upon Tyne, 1993.

[SF94] Marc Shapiro and Paulo Ferreira. Larchant-RDOSS: A distributed shared persistent memory and its garbage collector. Technical report, INRIA, 1994.

[SG91a] S. S. Simmel and I. Godard. The Kala Basket - A Semantic Primitive Unifying Object Transactions, Access Control, Versions and Configurations. In OOPSLA, 1991.

[SG91b] S. S. Simmel and I. Godard. The Kala Basket: A Semantic Primitive Unifying Object Transactions, Access Control, Versions, and Configurations. In OOPLSA'91, pages 230- 246, 1991.

[SGH+ 91] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An Object-Oriented Operating System - Assessment and Perspectives. Technical report, INRIA, 1991.

[SGM89] Marc Shapiro, Philipe Gautron, and Laurence Mosseri. Persistence and Migration for C++ Objects. In ECOOP'89, Nottingham (GB), July 1989.

[Sha94a] Marc Shapiro. A Binding Protocol for Distributed Shared Objects. Technical report, INRIA, 1994.

[Sha94b] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. Technical report, INRIA, 1994.

[Sim92] S. S. Simmel. Providing commonality while supporting diversity. Hotline on Object-Oriented Technology, 3(10), August 1992.

[SKW92] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient, Portable Persistent Store. In A. Albano and R. Morrison, editors, Persistent Object Systems, Workshop In Computing, pages 11-33. Springer-Verlag, 1992.

[SPFA94] Marc Shapiro, David Plainfosse, Paulo Ferreira, and Laurent Amsaleg. Some Key Issues in the Design of Distributed Garbage Collection and References. Technical report, INRIA, 1994.

[Tan95] A. S. Tanenbaum. Distributed Operating Systems. Prentice Hall, 1995.

[VD92] F. Vaugham and A. Dearle. Supporting Large Persistent Stores using Conventional Hardware. In A. Albano and R. Morrison, editors, Persistent Object Systems, Persistent Object Systems, Workshop In Computing, pages 35-53. Springer-Verlag, 1992.