

PERSISTENT OBJECT SERVICE FRAMEWORK USING
COMPONENT CONFIGURATION MODEL

BY

ARJUN CHANDRASEKAR IYER

B.Tech., Indian Institute of Technology, Kharagpur, 1996
M.S., University of Illinois at Urbana-Champaign, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

To Appa and Amma
For all their support and encouragement

Acknowledgements

I would like record my sincere thanks to my advisors, Prof. Roy H. Campbell, Prof. Richard E. DeVor and Prof. Shiv G. Kapoor for all their support and guidance throughout the course of my research. I would like to thank Fabio Kon and Manuel Roman for all their help during this work.

I would like to acknowledge the support of the National Science Foundation(NSF# 9870736) and the Defense Advanced Projects Research Agency of the Department of Defense through the Machine Tool Agile Manufacturing Research Institute.

Lastly, all my friends have been of great help and encouragement to me. My heartiest thanks to all of them.

Contents

Chapter	Page
1 Introduction	1
1.1 Persistence in Distributed Systems	2
1.2 Applicability of POS in the 2K Environment	3
1.3 Summary	4
1.4 Thesis Outline	4
2 The Persistent Object Service	5
2.1 The Common Object Request Broker Architecture	5
2.1.1 CORBA Services	5
2.2 OMG specification of the Persistent Object Service	6
2.2.1 Goals and Capabilities of the Persistent Object Service	7
2.2.2 Client view of the POS	7
2.2.3 Object Implementation	7
2.2.4 Persistent Data Service	8
2.2.5 DataStore	8
2.3 Components defined in the POS	8
2.3.1 Protocol	11
2.4 Implementation Details	12
2.4.1 Interaction between POS Components	12
2.4.2 Implementation issues in the TAO Environment	15
2.4.3 Interaction between the PO and POM	16
2.4.4 Traversal of the Data Object Graph	17
2.4.5 Interaction between the PO and the PDS	17
2.4.6 Interaction between the PDS and the DataStore	18
2.5 Issues resolved in the Implementation	20
2.6 Using the Persistent Object Service	21
3 Component Configurator Model in POS Architecture	23
3.1 Introduction to the Component Model	23
3.2 Use of ComponentConfigurator in the POS Model	26
3.3 Advantage of using the Component Configurator in the POS Model	27
3.3.1 Expressing Component Relationships	28
3.3.2 Fault-tolerance in the POS System	28
3.3.3 Automatic Reconfiguration	29

4	Application of the POS Framework in the 2k Environment	31
4.1	The Profile Repository	31
4.2	Implementation Details	33
4.2.1	Interfacing with the POS System	33
4.2.2	Flexibility of the System	33
5	Conclusion	36
5.1	POS Architecture Design	36
5.2	POS Implementation	37
5.3	Thesis Summary	37
5.4	Scope for Future Work	39
	List of References	41

List of Figures

Figure	Page
2.1 Roles in the Persistence Object Service	6
2.2 Major POS Components	9
2.3 Direct Access Protocol	11
2.4 Persistence Object Service Implementation	13
2.5 The PID Module in IDL	14
2.6 Derivation of PDS_DAI	15
2.7 IDL interfaces for PO and POM	16
2.8 IDL interface for the DAObject	17
2.9 Graph Traversal of DAObjects	18
2.10 IDL interface for the PDS	19
2.11 IDL interface for the DataStore	19
3.1 ComponentConfigurator	24
3.2 IDL interface for ComponentConfigurator	25
3.3 Component Relationships in POS Architecture	26
3.4 Psuedo-C++ code	28
3.5 Loading Dynamic Modules at Runtime	30
4.1 Profile Repository	32
4.2 Interaction between the Profile Repository and POS	34
4.3 Various Implementations of DAObjects for the Profile Repository	34
4.4 Managing different DataStores within the POS System	35

Chapter 1

Introduction

The idea of making software objects persistent has been gaining importance recently, especially with the advent of object-relational database systems. The need for saving software objects beyond the lifespan of the running application is seen in many diverse applications. Various algorithms, data structures and conversions have been used to store data, however, it is very hard to achieve portability between existing applications. Programmers spend a lot of time and effort in the design and implementation of efficient applications that implement such data storing features.

A framework for storing/restoring software objects would alleviate problems such as programming overhead and portability. Furthermore, the need for such *persistence* becomes all the more attractive in a distributed systems environment, wherein many hosts participate with different operation systems and software modules coded in different languages. The Common Object Request Broker Architecture (CORBA) [1] is an emerging open distributed object computing infrastructure, which would be an ideal architecture to base the persistence framework on, as CORBA is being standardized by the Object Management Group(OMG). CORBA specifies the Persistent Object Service(POS), whose main task is to provide uniform interfaces and mechanisms to manage persistent data. In order to organize

the objects involved in such a service, component technology can be employed through the use of *Component Configurator* objects [4].

The goal of this thesis work is the implementation of the framework for the Persistent Object Service over CORBA using *Component Configurator* objects.

1.1 Persistence in Distributed Systems

The term *Distributed Computing* simply refers to operations that run on widely separated computers that are connected via a network. Object orientation enters into the picture in the form of distributed objects and Object Request Brokers (ORBs). Distributed objects are simply objects that exist at the system level and can be accessed remotely. They can interact with other objects elsewhere in the system and elsewhere on the network. They can query other objects' capabilities or inherit them for their own use. Software components can invoke methods on remote distributed objects nearly as easily as they can on a local object. ORBs are a special kind of program-to-program middleware that is capable of supporting an object-based programming style. They are the mechanism that allows methods to be called on remote objects.

CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching. CORBA also specifies *Object Services* [2], which are domain-independent interfaces that are used by many distributed object programs. One of the services specified by OMG is the Persistent Object Service (POS), whose main task is to provide uniform interfaces and mechanisms to manage persistent data.

Persistent Objects are those objects whose life time exceeds the execution life time of programs and it is necessary to store their states in stable storage. Object states would be placed into two categories viz; one representing the persistent state which is stored in stable

storage and the other, the dynamic state which typically resides in volatile storage(memory). The dynamic states may be reconstructed from the persistent states. Of course, each object must be responsible for storing/restoring its state, but it can make use of the Persistence Object Service for the actual job which eases the programmers' burden to a great extent and makes the applications more portable. The CORBA Persistent Object Service attempts to aid the programmer in interfacing with the underlying datastores by using the service.

1.2 Applicability of POS in the 2K Environment

2K is a component-based operation system that uses reflection to manage change [3]. The 2K API is defined as a collection of IDL interfaces for different distributed services. These services are implemented by CORBA servants running on local or distributed ORBs. The reflective ORB will be customized to optimize the communication between applications and system services. The above mentioned Persistence Object Service can be used in the 2K environment to implement a Profile Repository which saves/retrieves the user profiles to/from the datastore.

The 2k operating system makes use of the *Component Configurator* [4], which is used to implement reconfigurability and fault-tolerance within the 2k system. The Component Configurator object defines a set of *hooks* to which other objects are attached, on which this object depends. Similarly it has a set of *client hooks* to which objects, which depend on this object, are attached. If any component fails, appropriate signals are forwarded to the other connected components, which may then take necessary actions to guarantee some degree of fault-tolerance.

The POS model also makes use of the Component Configurator to specify the inter-component dependence. An effort is made to make the POS system reconfigurable within the scope of the local ORB using *dynamicTAO* capabilities [5].

1.3 Summary

The outcome of this thesis is the implementation of the Persistent Object Service(POS) framework, which will greatly help programmers needing an uniform interface to achieving object persistence. The Component Configurator is used to define object dependencies within the POS system and also to aid in reconfiguration and introduce some fault-tolerance in the system.

The use of the Persistence Object Service is illustrated within the 2k environment by the proposed implementation of the Profile Repository used to store/restore user profiles.

1.4 Thesis Outline

Chapter2 of this thesis provides an overview of the Persistence Object Service(POS) as specified by the Object Management Group(OMG). Various issues in the implementation are discussed and corresponding solutions presented.

Chapter3 discusses the use of a Component Configurator in the implementation of the Persistence Object Service architecture, which would help in the specification of the framework and will facilitate automatic re-configuration and support fault-tolerance.

Chapter4 illustrates the use of the Persistent Object Service(POS) in the 2K environment by describing the implementation of a Persistence Profile Repository.

Chapter 5 presents some concluding remarks and also details the possibilities for further research.

Chapter 2

The Persistent Object Service

2.1 The Common Object Request Broker Architecture

Around 1990, the Object Management Group(OMG) introduced the Objects Management Architecture(OMA) for distributed systems [6], which defines an *abstract object model*. In 1991, OMG defined an industry standard called the Common Object Request Broker Architecture(CORBA), based upon a concrete object model derived from the OMA abstract object model. The CORBA 2.0 standard [7] specifies ways to interconnect different CORBA's either by transforming requests in a bridge/gateway or by transporting requests in the standardized Internet Inter-ORB Protocol(IIOP).

2.1.1 CORBA Services

The CORBA 1.2 standard also proposes a collection of Object Services [8] that facilitate CORBA-supported objects with additional functionality such as creating and deleting new objects (Object LifeCycle Service), looking up a server with a specific interface in another CORBA environment (Object Trading Service), or managing persistent objects (Persistence Object Service). There are two design principles that OMG follows: first, OMG intentionally

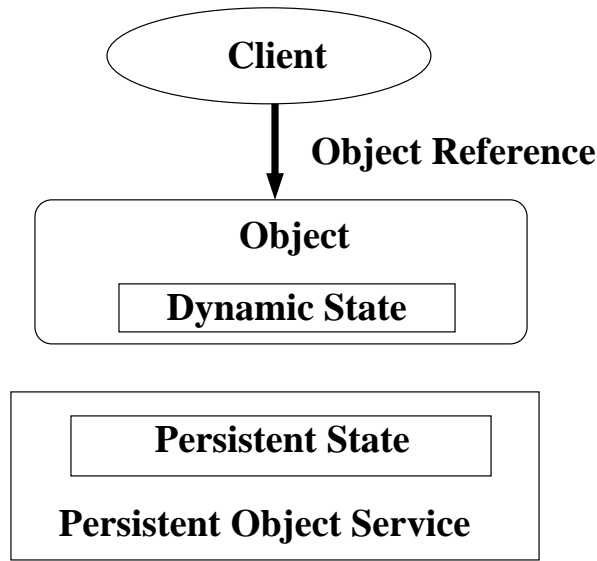


Figure 2.1: Roles in the Persistence Object Service

leaves services not fully specified; second, services may be mutually dependent and, at the same time they should be able to exist separately, thus partially covering functionality of other services. Typically an Object Service is used by inheriting a subset of IDL interfaces specifying the Object Service.

2.2 OMG specification of the Persistent Object Service

The goal of the Persistent Object Service is to provide common interfaces to the mechanisms used for retaining and managing the persistent state of objects. Figure 2.1 shows the participants in the Persistent Object Service.

As usual for Object Services, the primary task of this persistence specification is to define the interfaces that are needed to use the Persistence Object Service and the conventions for how objects can work together using it. Although the CORBA system defines object references as persistent, it defined no particular way for the object to make its state persistent. The Persistent Object Service is intended to be the most common way to implement this.

This service defines a convention to expose the persistent state from an object.

2.2.1 Goals and Capabilities of the Persistent Object Service

The Persistent Object Service plays a key role in structuring the object system. The principle requirement to be supported is the need for an object to be able to make all or part of its state be persistent. The Persistent Object Service applies object technology and principles to the storing of persistent data.

To manage object persistence, the POS defines an architecture with interfaces defined using the CORBA IDL type system. Whether detailing the particular data to be stored, describing the protocol for accessing the states or defining the convention for making state visible for client control, the same “language” is used. By accessing data through an interface, many problems of data manipulation and exchange can be avoided. For example, programs always see data in the representation that is appropriate for the machine, programming language, etc., of the application. Data can be translated as needed to facilitate use in different object types and implementations.

2.2.2 Client view of the POS

It is common for clients of objects to need to control or to assist in managing the persistence. In particular, the timing of when the persistent state is saved/restored, and the identification of which subset of the object’s state is to be stored/restored, are two aspects often of interest to clients.

2.2.3 Object Implementation

The persistent object itself has total control over its persistence. It decides which subset of its data is to be publicized as being persistent and also lays down certain rules on how

its persistence is going to be used. The object has the choice of delegating its management of the persistent data to other services or it can have fine-grained control over it. The implementation discussed in this thesis work does not delegate the management to other services.

2.2.4 Persistent Data Service

The Persistent Data Service(PDS) actually implements the mechanisms for making data persistent and manipulating it. Every PDS supports a notion of a *Protocol*, which dictates how data is to be moved from the persistent object into the persistent data service. The PDS has the responsibility of translating data from the object world above it to the storage world below it.

2.2.5 DataStore

The lowest level of interface defined is the DataStore. By having an interface hidden from all the objects and clients, the DataStore can provide service to any and all of the objects that use a DataStore interface. A DataStore can represent a filesystem, an object-oriented database or a relational database interface.

2.3 Components defined in the POS

The major components of the Persistent Object Service are shown in Figure 2.2. They are:

1. Persistent Identifier(PID): This describes the location of an objects data in some DataStore and generates a string identifier for that data.
2. Persistent Object(PO): This is an object whose persistence is controlled externally by clients. Any CORBA object is made persistent by inheriting from the PO interface.

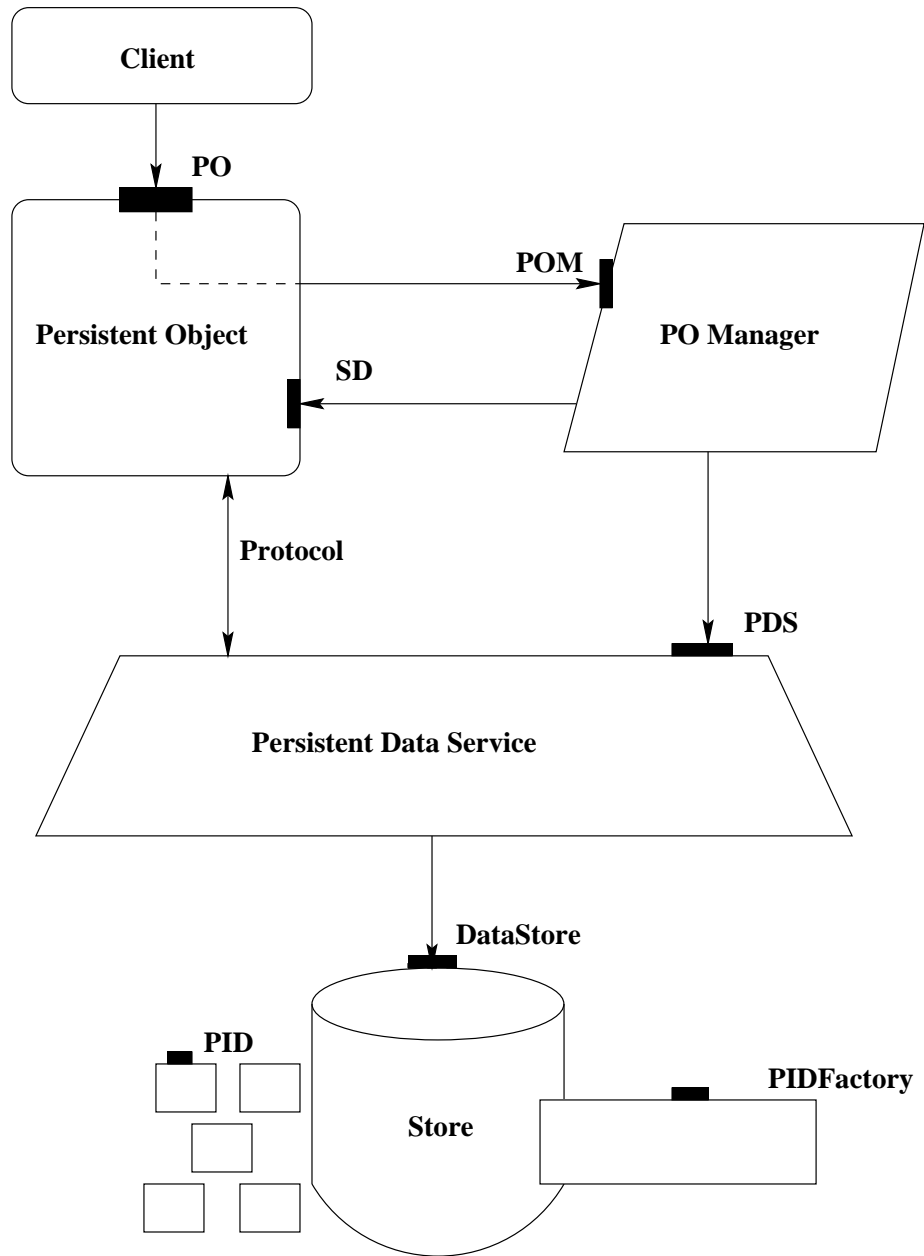


Figure 2.2: Major POS Components

3. Persistent Object Manager(POM): The Persistent Object routes all the calls through the Persistent Object Manager which provides a uniform interface. The POM then forwards the request to the correct instance of the PDS.
4. Persistent Data Service(PDS): The Persistent Data Service is responsible for acting as an interface between the persistent object and the underlying datastore. This is the heart of the Persistent Object Service.
5. Protocol: This defines a way to get data in and out of the object.
6. DataStore: This component provides one of several ways to store an objects data independently of the address space containing the object.

The Persistence Service as specified by OMG consists of three basic interfaces: *Persistent Object*(PO), *Persistent Object Manager*(POM) and *Persistent Data Service*(PDS). Fundamentally these interfaces comprise the same methods: *connect()*, *disconnect()*, *store()*, *restore()* and *delete()*.

The PDS forms a connecting bridge between the PO and the DataStore. It conforms to a particular Protocol and interfaces with a given DataStore instance. The *DataStore* actually saves and loads the data, and *Protocol* describes the way a PDS transfers data to and from the PO. Both the *DataStore* and *Protocol* are not standardized, however OMG offers three examples of *Protocol* and a specification of *DataStore_CLI*.

A POM dynamically resolves the binding between PO and its PDS, given a PID of a PO and the *Protocol* supported by the PO. Here, a PID is an identifier, uniquely denoting the object derived from PO in a database. Thus, a PID contains information about the `datastore_type`, `datastore_instance` and `data_identifier` which defines the subset of PO to be made persistent. The POM resolves the PDS in the following steps:

1. get the `datastore_type` and `datastore_instance` from the PO's PID,

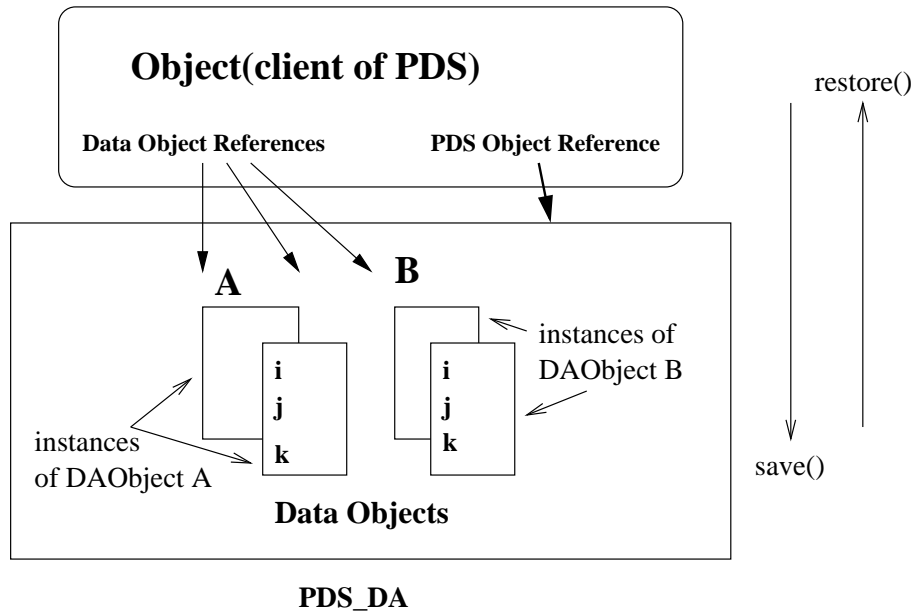


Figure 2.3: Direct Access Protocol

2. get the *Protocol* supported by the PO,
3. localize a *PDS_e* object using the pair $[DataStore, Protocol]$.

2.3.1 Protocol

OMG lays down the specification of a Direct Access Protocol(PDS_DA). The PDS_DA supports direct access to persistent data through typed attributes organized in data objects that are defined in the Data Definition Language(DDL). An object using this protocol would represent its persistent data using one or more interconnected Data Objects. The PDS_DA has two parts as shown in Figure 2.3. When connected to the PDS, the object (which is effectively the client of the PDS), has an object representing the PDS which supports the PDS_DA. The object performs operations defined in the PDS_DA interface to get references to the Data Objects in the PDS. The persistent data is manipulated by performing operations on the data references to get and set attributes on the data objects.

2.4 Implementation Details

The overall architecture of the POS system is shown in Figure 2.4. The object interactions are also depicted and, as shown, cooperation among objects is achieved through the use of the Naming Service in CORBA. In storing of the persistent state of an object, the client needs to decide which subset of the object's state is to be made persistent and also the DataStore instance which it will use to store the object's data. This is achieved by the use of the Persistent Identifier(PID), which has the corresponding attributes to reflect the client's choice. The PID IDL interface is shown in Figure 2.5.

As shown in the IDL specification, subclasses of the PID interface are created for more specific needs. A PIDFactory object is also defined which will create PID objects dynamically when the client requests it.

2.4.1 Interaction between POS Components

The POS system goes through the following sequence of operations as the client prepares to use the POS system for object storage:

1. The client of the POS system gets a PID from the PIDFactory of the specific DataStore in which it wishes to save the persistent data. It will also set the OID(object identifier) to indicate the subset of the PO's data which is to be stored.
2. Once the PID is obtained, the client will call *store()* on the Persistent Object with the PID passed in as an argument,
3. The Persistent Object(PO) will forward the call to the Persistent Object Manager(POM),
4. The POM will call *prestore()* on the PO, which will transfer the subset of PO's data to the Data Object which is used in the *DAProtocol*. The PO sets the root object of the Persistent Data Service(PDS),

Persistent Object Service Implementation Details

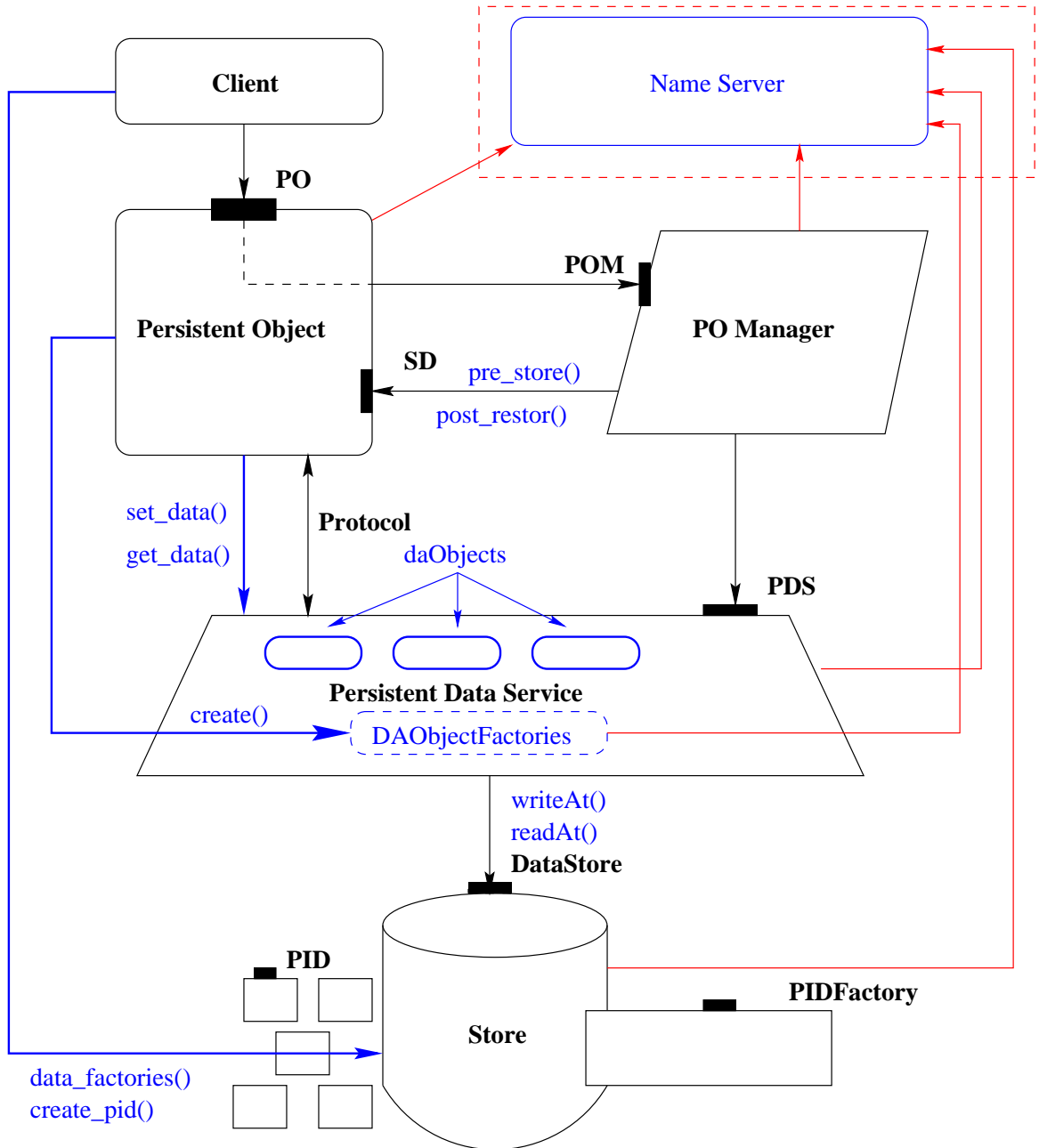


Figure 2.4: Persistence Object Service Implementation

```

interface PID
{
    attribute string DataStore_type;
    string get_PIDString();
    string getProtocol();
};

interface PID_DB : PID
{
    attribute string DataStore_name;
    attribute string key;
};

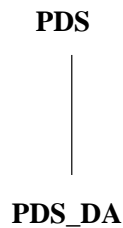
interface PID_DA : CosPersistencePID::PID_DB
{
    attribute string oid;
};

interface PIDFactory
{
    PID create_PID_from_key(in string key);
    PID create_PID_from_string(in string pid_string);
};

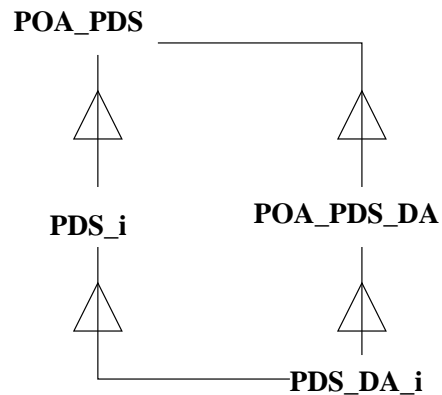
```

Figure 2.5: The PID Module in IDL

IDL Definition



TAO Implementation



PDS_DA_i has to multiply inherit from **POA_PDS_DA** and **PDS_i**.

Figure 2.6: Derivation of **PDS_DA_i**

5. The POM then forwards the *store()* call to the PDS,
6. The PDS calls the *BOSS()* function on the DAOObject to get a stream of data, which it forwards to the DataStore.
7. The DataStore instance actually stores the data into physical storage.

2.4.2 Implementation issues in the TAO Environment

Implementations for each of the IDL interfaces have to be provided in C++ within the TAO environment. For each *object* interface developed in IDL, the TAO IDL compiler produces a *POA_object* class from which the implementation (typically called *object_i*) is inherited. In case of an inheritance hierarchy in the IDL definitions, multiple inheritance is required to take care of the implementations as shown in Figure 2.6.

```

interface PO
{
    attribute CosPersistencePID::PID pid;
    PDS connect(in CosPersistencePID::PID p);
    void disconnect(in CosPersistencePID::PID p);
    void store(in CosPersistencePID::PID p);
    void restore(in CosPersistencePID::PID p);

    void pre_store();
    void post_restore();
};
interface POM
{
    CosPersistencePO::PDS connect(in CosPersistencePO::PO obj,
in CosPersistencePID::PID p);
    void disconnect(in CosPersistencePO::PO obj,
                    in CosPersistencePID::PID p);
    void store(in CosPersistencePO::PO obj, in CosPersistencePID::PID p);
    void restore(in CosPersistencePO::PO obj, in CosPersistencePID::PID p);
    void del(in CosPersistencePO::PO obj, in CosPersistencePID::PID p);
};

```

Figure 2.7: IDL interfaces for PO and POM

2.4.3 Interaction between the PO and POM

Once the client gets a PID, it can establish a connection between the PO and the underlying DataStore with the *connect()* call. Thereafter *save()/restore()* commands to the PO will store/retrieve data from and to the PO. All the calls on the PO are routed through the Persistent Object Manager(POM), which forwards the calls to the correct instance of the PDS depending upon the Protocol and DataStore involved. The IDL definitions of the PO and POM are shown in Figure 2.7.

Since the data from the Persistent Object is represented as data objects in the Direct Access Protocol, there must be a way to transfer the data from the Persistent Object to these data objects. This is achieved via the *pre_store()* call by the POM on the PO. The reverse is done through a call to *post_restore()* which will transfer the data back to the PO

```

interface DAObject
{
    void init();
    boolean dado_same(in DAObject d);
    string dado_oid();
    CosPersistencePID::PID dado_pid();
    void dado_remove();
    string BOSS();
    void invBOSS(in string str);

    void PO_to_DA();
    void DA_to_PO();
};

```

Figure 2.8: IDL interface for the DAObject

from the data objects on a restore operation. The IDL interface for the DAObject is shown in Figure 2.8.

2.4.4 Traversal of the Data Object Graph

The Persistent Object may choose to represent its data by a collection of data objects interconnected with each other. Each data object will have reference to another DAObject and so on. In this case the entire graph will have to be traversed to transfer data between the PO and the DAObject. This is accomplished by successively executing a call in each DAObject, which is depicted in Figure 2.9. In the restoration of the DAObjects, there must be a way to instantiate the appropriate DAObject instance depending upon what was stored in the DataStore. Since the C++ language does not support runtime class type information, we store a unique class identifier as a string to the data objects.

2.4.5 Interaction between the PO and the PDS

The Protocol defines the interaction between the PO and the PDS. In the Direct Access Protocol, the Persistent Object creates the DAObjects and sets the root object of the PDS

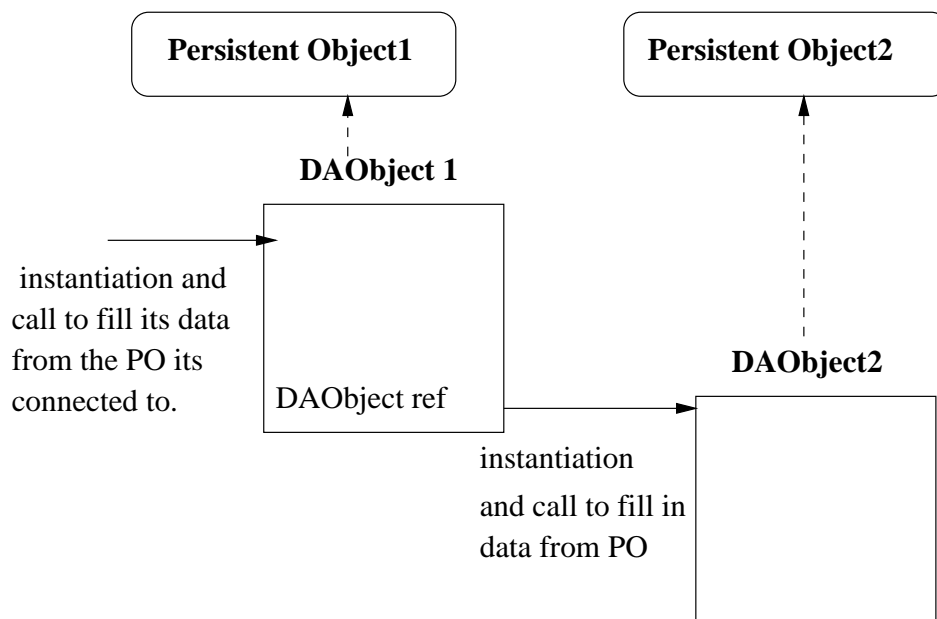


Figure 2.9: Graph Traversal of DAObjects

by a *setdata()* call on the PDS instance returned by the POM. In order to retrieve the data from the PDS, a *getdata()* is called on the PDS. Figure 2.10 depicts the IDL interface for the PDS.

2.4.6 Interaction between the PDS and the DataStore

The actual saving of the persistent data onto disk is carried out by the PDS by calling the *writeAt()* call on the DataStore interface. The DataStore may represent any storage system such a filesystem, a relational database or an object-oriented database system. The DataStore IDL is shown in Figure 2.11. The implementation should derive from the DataStore::DBInterface interface, which supports two basic functions to read/write data into the underlying physical storage.

```

interface PDS
{
    PDS connect(in CosPersistencePO::PO obj, in CosPersistencePID::PID p);
    void disconnect(in CosPersistencePO::PO obj,
        in CosPersistencePID::PID p);
    void store(in CosPersistencePO::PO obj, in CosPersistencePID::PID p);
    void restore(in CosPersistencePO::PO obj, in CosPersistencePID::PID p);
    void del(in CosPersistencePO::PO obj, in CosPersistencePID::PID p);
};

interface PDS_DA : PDS
{
    CosPersistenceDA::DAObject get_data();
    void set_data(in CosPersistenceDA::DAObject new_data);
    CosPersistenceDA::DAObject lookup(in string id);
    CosPersistencePID::PID get_pid();
    CosPersistencePID::PID get_object_pid(in CosPersistenceDA::DAObject);
    CosPersistenceDA::DAObjectFactoryFinder data_factories();
};

```

Figure 2.10: IDL interface for the PDS

```

interface DBInterface
{
    void deleteAt(in string index);
    void writeAt(in string index, in string val);
    string readAt(in string index);
    CosPersistencePID::PIDFactory get_PIDFactory();
};

```

Figure 2.11: IDL interface for the DataStore

2.5 Issues resolved in the Implementation

The OMG specification of the Persistent Object Service leaves out a lot of specifics on the implementation details. Filling in the semantic gap left in the OMG specification of the Persistent Object Service involves many decisions to be taken when designing and implementing the service. Issues had to be resolved in general areas and also in the design architecture. We describe a few of the issues and how it is resolved:

1. **Determination of Persistence Property:** The persistence of the object can be determined either *statically* or *dynamically*. In static persistence the object's persistence is determined at compile time and the object cannot cease to be a persistent object at runtime. In dynamic persistence, the object's persistence can be controlled at runtime. Since we make an object persistent by subclassing from a persistence base class, all our persistent objects are statically determined.
2. **Updating persistent object state:** The persistence of an object can be controlled either *automatically* or *explicitly*. In automatic control, update is system-controlled possibly using events. In explicit control, the update is application-controlled and the POS system described in this thesis follows this mode of update by calls to store/restore methods.
3. **The Persistent Identifier Creation:** Since, the client calls the store/restore on the Persistent Object with the PID passed in as an argument, our implementation assumes the client to have a PID instance before the call. The Persistent Object is not responsible for the creation of the PID instance.
4. **Recursive traversal of the DAObject graph:** In the implementation of the DA protocol, the OMG specifications mention the need to handle attribute values in a DAObject which are references to other DAObjects. These references may be to colo-

cated DAObjects or to remote DAObjects, which are essentially CORBA objects. The implementation proposed, provides a solution as explained in section 2.4.4.

2.6 Using the Persistent Object Service

Any CORBA object whose state is to be made persistent, can use the Persistent Object Service. The following basic steps are involved therein:

1. **Derive from PO:** The CORBA object to be made persistent has to derive from the `CosPersistence::PO` interface and therefore has to implement two methods viz; *prestore()* and *postrestore()*. If the PO conforms to the `DAProtocol`, the *prestore* and *post-restore* methods are responsible for the transfer of data from and to the PO. Thus the *prestore* creates suitable DAObjects and transfers the data from the PO to the DAObject. The *post-restore* method does the inverse, i.e. transfers data from the DAObjects back to the PO.
2. **Create DAObjects:** Depending upon what subset of the Persistent Object's state is to be made persistent, suitable subclasses of DAObject have to be defined with required attributes. The interdependence of the DAObjects also has to be defined here. Every subclass of DAObject has to define *four* methods viz; *PO_to_DA()*, *DA_to_PO()*, *BOSS()* and *invBOSS()* respectively. The first two methods determine how to ferret data to and from the PO, the next two methods transfer data out/in from/to the DAObject itself.
3. **Create DAObjectFactories:** Every subclass of DAObject created should have a corresponding Factory object which is to be registered with the Naming Service. The factory object implementation should subclass from the DAObjectFactory IDL interface and implement the *create()* method.

Thereafter, the client of the CORBA object can now store/restore persistent data from/to to the object via calls to *store()/restore()* on the PO object. Prior to this, the client must also obtain a PID from the specific DataStore object wherein it wishes to store the persistent data. The PID is obtained by calling the *create()* method of the PIDFactory interface. The reference to a PIDFactory is obtained from the *DataStore* instance wherein the persistent data is to be stored.

Chapter 3

Component Configurator Model in POS Architecture

3.1 Introduction to the Component Model

Component technology stresses the desire for independent pieces of software that can be reused and combined in different ways to implement complex software systems. Components are created by different programmers, often working in different groups with different methodologies. It is hard to create robust and efficient systems if the dynamic dependencies between the components are not well understood. Sometimes the graceful failure of one module is not properly detected by other modules leading to system failure.

In the POS architecture, each component is managed by a *Component Configurator* which is responsible for storing the dependencies between a specific component and other system components. Depending on the way it is implemented, a Component Configurator may be able manage components running on a single address space, on different address spaces and processes, or even running on different machines in a distributed system.

Figure 3.1 depicts the reified dependencies between a certain component and others.

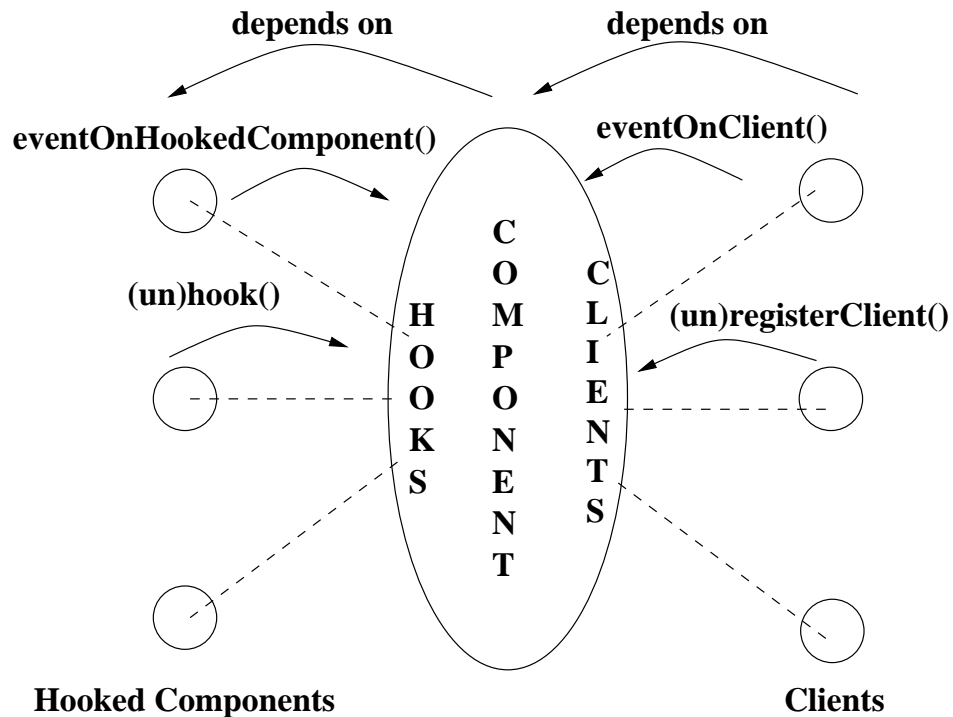


Figure 3.1: ComponentConfigurator

Each component C has a set of *hooks* to which other components can be attached. These are the components on which C depends and are called *hooked components*. There might be other components that depend on C , and these are called *clients*. In general, each time one defines that a component C_1 depends on another component C_2 , two actions are performed:

1. C_1 attaches C_2 to one of its hooks,
2. C_2 adds C_1 to its list of clients.

If any of the required components are not present at the time when the component is created, then it should create it as required. The IDL specification of the *ComponentConfigurator* class is shown in Figure 3.2.

```

interface ComponentConfigurator
{
    exception invalidArgument{};
    exception ElementExists{};
    exception NotFound{};
    exception HookBusy{};
    exception HookVacant{};

    void destroy ();
    void addHook(in string hookName) raises (ElementExists);
    void deleteHook(in string hookName) raises (NotFound);
    void hook(in string hookName, in ComponentConfigurator cc)
        raises (NotFound);
    void unhook(in string hookName) raises (HookVacant, NotFound);
    void registerClient(in ComponentConfigurator client,
                       in string hookNameInClient) raises (ElementExists);
    void unregisterClient(in ComponentConfigurator client,
                          in string hookNameInClient) raises (NotFound);

    void eventOnHookedComponent (in ComponentConfigurator hookedComponent,
                                 in Event e, in unsigned short timeToLive);

    void eventOnClient(in ComponentConfigurator client,
                       in Event e, in unsigned short timeToLive);

    string name ();
    string info ();
    DependencyList listHooks ();
    ComponentConfigurator getHookedComponent(in string hookName);
    unsigned short numberOfClients ();
    DependencyList listClients ();
    Object implementation ();
};

```

Figure 3.2: IDL interface for ComponentConfigurator

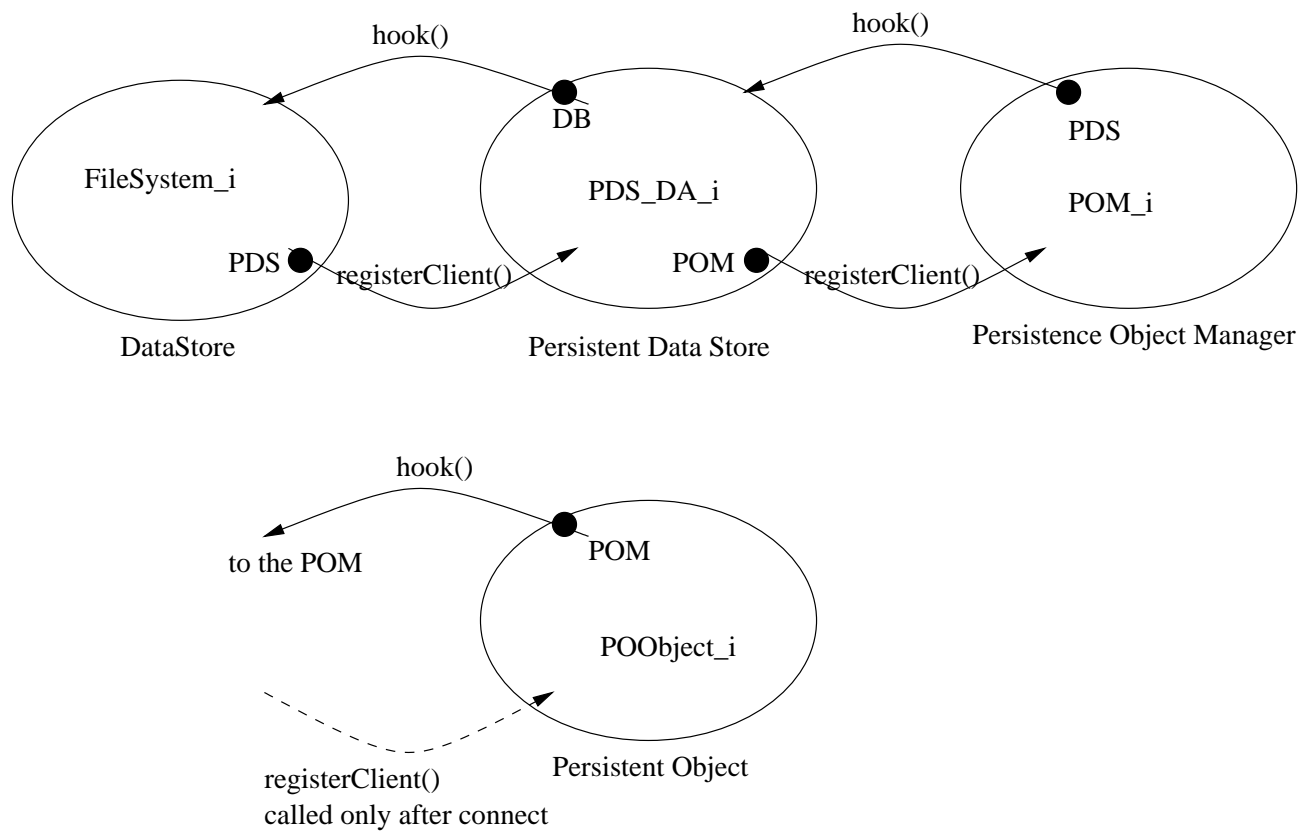


Figure 3.3: Component Relationships in POS Architecture

3.2 Use of ComponentConfigurator in the POS Model

The Figure 3.3 depicts the overall relationships amongst components in the POS model. The individual components are shown and how they connect/depend on other components in the model. The lowest component in the hierarchy is the *DataStore* interface, which does not have any of its hooks connected. Its hooks depends upon the underlying Datastore in the system, for e.g, in a system, wherein a commercial DBMS system is used to store the data, it can have a hook that represents an active DBMS running in the system.

The *Persistence Data Store(PDS)* component depends on the *DataStore* interface on which it calls the store/restore atomic operations. Thus when the PDS server is brought up, it checks for the existence of the *DataStore* interface. If any implementation is found, it

hooks on to it, otherwise it'll throw an exception upon a call to store/restore data. It would be noteworthy to add that an implementation of the DataStore can be hooked to the PDS, using the dynamic reconfigurability property of the 2K environment.

The *Persistence Object Manager(POM)* controls the overall functioning of the POS system. As shown, it depends on the PDS for routing of all the calls from the Persistent Object(PO). If the POM manages several PDS for several DataStores, each of them can be hooked on to the POM, which will then route all the calls to the appropriate PDS via the hooks registered. It registers itself as a client to the PDS.

The *Persistent Object(PO)*, upon activation will hook on the instance of the POM. However, it will register itself as a client only after a *connect()* call is made on the PO. At any time, the POM has only a single connection from a PO.

3.3 Advantage of using the Component Configurator in the POS Model

The dynamic dependence information in its turn, enables the reconfiguration of components that are already running. In addition, it provides important information for implementing fault tolerance and smooth exception handling in an environment of centralized or distributed components. The Component Configurator is used in the POS model, primarily for three reasons:

1. To express the relationship between objects in a more generic manner,
2. To introduce some level of fault-tolerance into the system and,
3. To make the system reconfigurable such that each individual component can be replaced at runtime by another specific instance either in response to component failure or for performance reasons.

```

ComponentConfigurator::~~ComponentConfigurator()
{
    for(c in hookedComponents){
c.configurator->unregisterClient(this);
    }

    for(c in clients){
        c.configurator->
            eventOnHookedComponent(this, DELETED);
    }

    //delete hooked components and clients
    //release resources
    //delete component implementation
}

```

Figure 3.4: Psuedo-C++ code

3.3.1 Expressing Component Relationships

The Component Configurator can be used to specify the relationships between the various distributed components in the POS system. More specifically, the dependancy of one component on another can very easily be modeled. This can be used to introduce fault-tolerance within the system and also establish the sequence of operations between the various components upon client interaction with the POS system.

3.3.2 Fault-tolerance in the POS System

The fault tolerance of the system, can be illustrated by means of an example. Consider the deletion of a component containing the *ComponentConfigurator* class. Different policies for dealing with component deletion can be adopted. In general, a component *C* is destroyed, an announcement must be made to components that depend on *C* and to components on which *C* depends. The following piece of pseudo-C++ code (Figure 3.4) illustrates this process with a conservative implementation of the *ComponentConfigurator* destructor. Implementations

of this destructor can be specialized to adjust its behavior to different component types and to meet special requirements. Also, different component types must implement methods such as *eventOnHookedComponent()* in proper ways to take care of the different kinds of dependencies. In the POS model described, if any component is deleted, then a invalid flag is set and any calls to the component will throw an exception. Thus, the system will not fail in case of any missing component. Components, which have specific implementations can be added at runtime to modify the behavior of the POS system. Different fault-tolerant behavior can be incorporated into the POS system by suitably modifying the behavior of the Component Configurator destructor. The present POS model adopts a more of a fault-detection feature rather than a complete fault-tolerant capability, since the components in the POS model are strongly inter-dependent on one another. Another possibility is one component firing up another instance of a component that has failed. Yet another possible implementation would have a backup instance of every component, so that the backup would be brought into action when the operating instance fails or is brought down.

3.3.3 Automatic Reconfiguration

Any of the components participating in the POS system can be reconfigured at runtime by using the functionality provided by the *dynamicTAO* ORB. The *dynamicTAO* ORB exports interfaces for loading and unloading modules into the ORB runtime and for inspecting the ORB configuration state. The architecture can also be used for dynamic reconfiguration of servants running on top of the ORB and for reconfiguring stand-alone applications.

In order to make the POS system reconfigurable, the C++ (non-CORBA) version of the *Component Configurator* is used. Every component of the POS model is registered with the local ORB domain configurator. Thereafter loadable modules have to be defined which inherit from the *Simple_Strategy_Factory* class defined in *dynamicTAO*. Once these loadable modules are defined, they can be loaded at runtime via the *dynamicTAO* reconfiguration

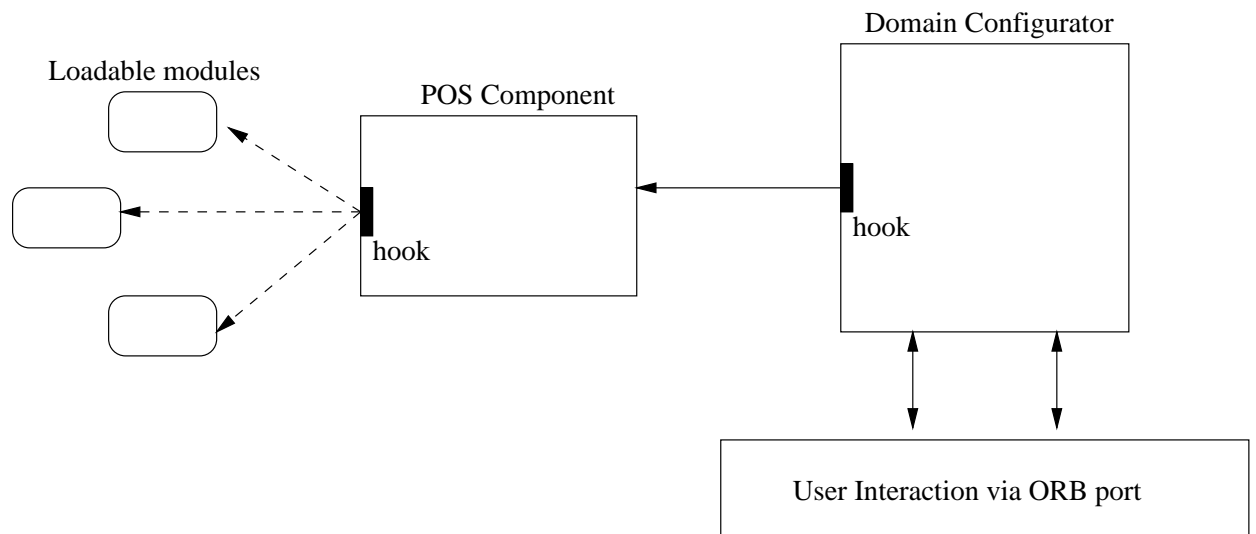


Figure 3.5: Loading Dynamic Modules at Runtime

port at the local ORB. Figure 3.5 illustrates the scenario. Several different loadable modules can be specified to customize the behaviour of the component under various operating parameters.

Chapter 4

Application of the POS Framework in the 2k Environment

The Persistence Object Service(POS) can be used in the 2k distributed environment to implement a *Profile Repository*, wherein the user profiles can be saved/restored from the underlying DataStore. This work presents one possible model, for the repository. In general systems that use the POS architecture should be customized to suit one's requirements. The profile repository can form part of an *Environment Service*, which is responsible for managing groups of components as a collective entity and for helping the components interface with the 2K operating system.

4.1 The Profile Repository

The Profile Repository represents a storage of all possible user profile information in a distributed system. The IDL specification of the Profile Repository is shown in Figure 4.1.

```

module twokProfileService{

    struct Desc {
        CosNaming::Name comp_name;
        string comp_reference;
    };

    typedef sequence<Desc> ListOfDescs;

    struct Profile {
        ListOfDescs default_components_specs;
        string username;
    };

    exception UnknownObject {};

    typedef sequence<Profile> ListOfProfiles;

    interface Profile_Repository {
        Profile get_profile(in string username)
        raises(UnknownObject);
        void set_profile(in Profile theProfile)
        raises(UnknownObject);
        void list_profiles();
        void list_a_profile(in Profile profile);
        void initialize_repository();
        long init_naming_service();
    };
};

```

Figure 4.1: Profile Repository

4.2 Implementation Details

A profile is a system entity that keeps information about distinctive features of a user. The Profile Repository is in ways similar to the Implementation Repository and the Interface Repository. The Profile Repository exports methods such as *get(user_name, passwd)* and *set(user_name, passwd)*.

There is an interceptor at the profile repository that does the login checks (username, password). The interceptor is initialized when the orb is initialized and stays there until the orb dies. The Profile Repository CORBA object implementation will be the Persistent Object(PO) in the POS model. It will inherit from the *CosPersistencePO::PO* interface and thereby inherit the methods that any PO object needs to inherit.

4.2.1 Interfacing with the POS System

Figure 4.2 depicts the interaction between the clients of the Profile Repository, the Profile Repository and the POS system. Once the Profile Repository interface inherits from the *CosPersistencePO::PO* interface, any client of the PO can call the *store()* function to make the state of the Profile Repository persistent. In order to store the data of the Profile Repository in the *DataStore*, the *DAObject* implemented should provide suitable methods which will stream out the data in a consistent way, so that the object can be reconstructed any time later, from the char bytes, when a restoration operation has to be carried out. Figure 4.3 shows the implementation details of streaming the Profile Repository data using a *DAObject*-protocol described previously.

4.2.2 Flexibility of the System

The flexibility of the POS model makes it easier to add another *DataStore* implementation if required. If one needs the option of either saving the persistent data in the filesystem or in

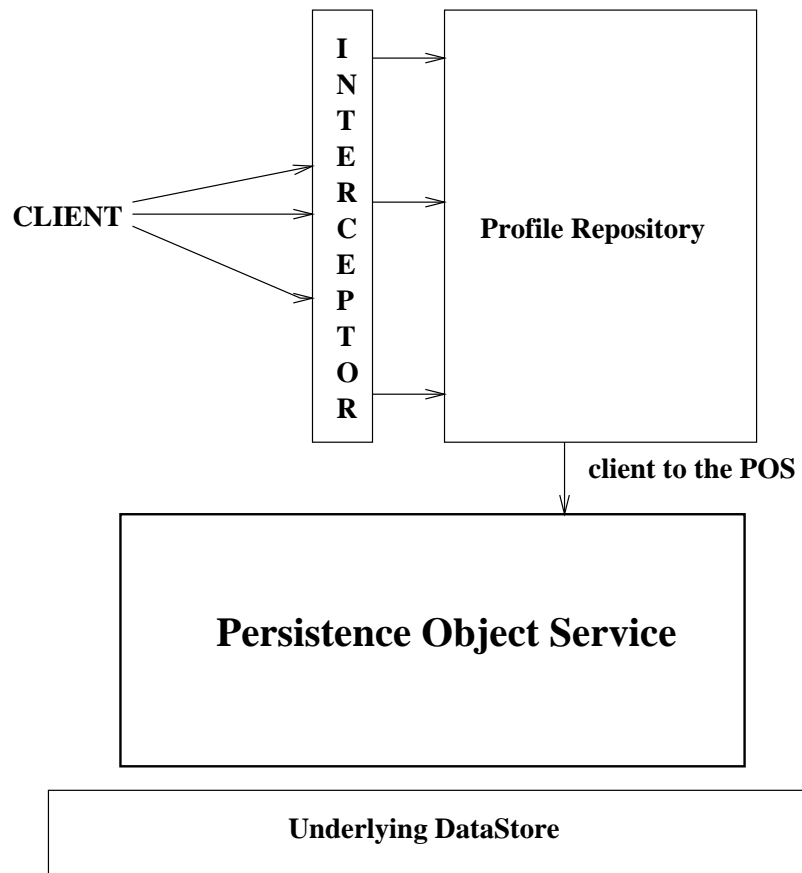


Figure 4.2: Interaction between the Profile Repository and POS

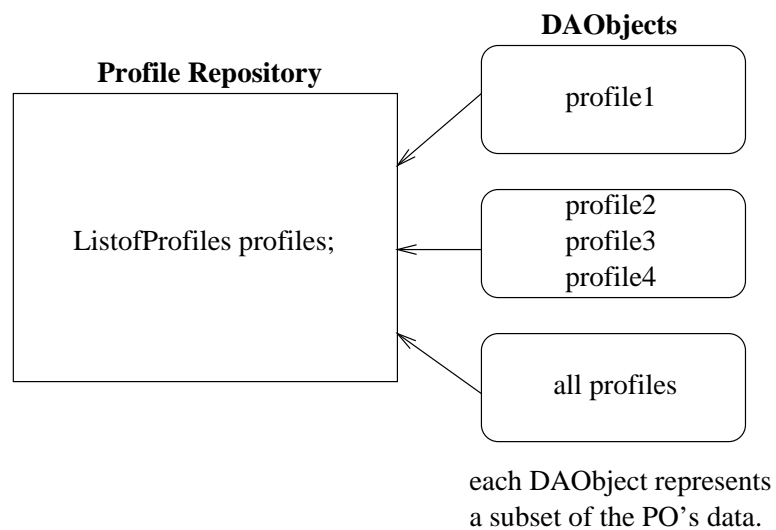


Figure 4.3: Various Implementations of DAOObjects for the Profile Repository

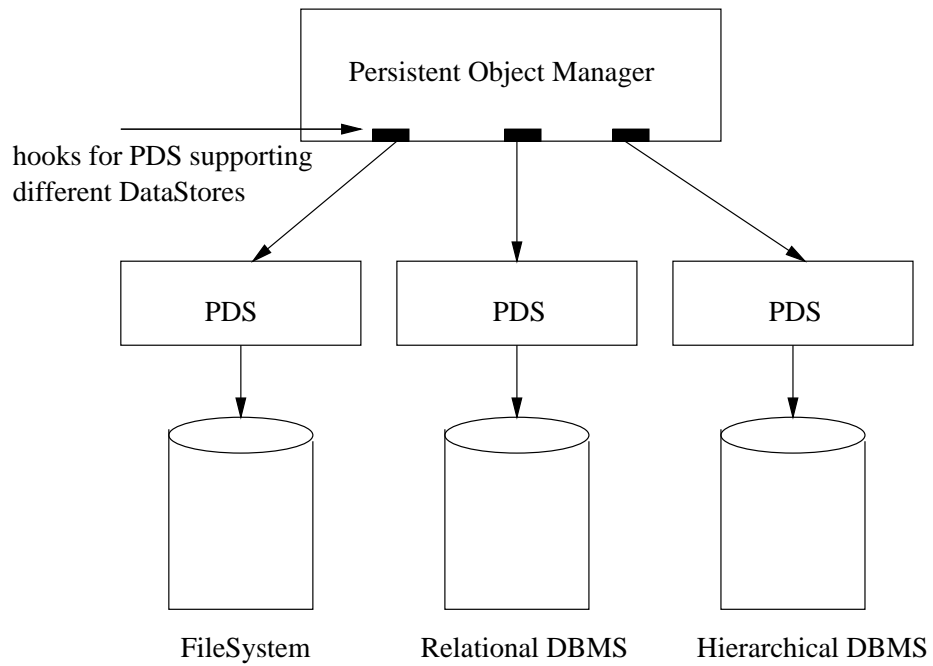


Figure 4.4: Managing different DataStores within the POS System

a commercial DBMS, the functionality can be added by implementing another *DBInterface* instance and linking it to the underlying DBMS(Figure 4.4). The client of the POS will specify, via the *CosPersistence::PO::PID* interface, which DataStore is to be used for storage. The POM will accordingly forward the call to the appropriate PDS, which will connect to the correct *DBInterface* instance.

Chapter 5

Conclusion

The Persistence Object Service(POS) provides a uniform interface to achieve persistence in a distributed environment. It can be very effectively used as tool in a distributed systems programming environment, to interface and manage a set of datastore objects offering diverse storage capabilities.

The OMG specification of the Persistence Object Service lays down the interfaces for four main components viz; *Persistent Object*, *Persistence Object Manager*, *Persistence Data Service* and the *DataStore* respectively. In addition to the above interfaces, OMG also describes a possible *Protocol*, which dictates the interaction between the Persistent Object and the Persistence Data Service. The OMG specification describes the *Data Access Protocol(DA)*, which involves the specification of the persistent datasets of a persistent object via the use of DAOjects.

5.1 POS Architecture Design

The initial design stage documented in Chapter2 requires an understanding of the purpose and relationship of the POS components. The interaction between the components is described in Chapter2 and require additional methods to that specified in the OMG specifi-

cations. The design specifies the exact sequence of operations on any initiation action from the client. The Direct Access(DA) Protocol is elaborated to define the interaction between the DAObject, Persistent Object(PO) and the Persistent Data Service(PDS). The design includes an efficient way of recursively traversing the DAObject graph.

The use of the Component Configurator in the POS model was immediately appreciated and its use in representing the relationship between the various objects is beneficial in the design stage and simplifies coding. The Component Configurator also helps in making the system more fault-tolerant and reconfigurable.

5.2 POS Implementation

The various IDL interface implementations are coded in C++ within the TAO environment. The CORBA Naming Service is used to store any global references of the POS objects. Section 2.4 explains the implementation details.

The implementation of the CORBA Component Configurator is used to describe the relationship between the objects. The *hooks* of the Component Configurator are used to specify the dependency of one object on another. The Component Configurator is subclassed to implement different strategies on component destruction.

The implementation of the POS model can be utilized in any distributed system to store the data of any distributed object. In the present work, we describe its use in storing the user profile data of a Profile Repository, in the 2K distributed operating system.

5.3 Thesis Summary

The result of the present thesis work is twofold viz; (1) implementation of a framework for the Persistent Object Service in CORBA and (2) the illustration of a practical use of the

Component Configurator in the Persistent Object Service. The following section presents a summary of the various tasks implemented and thereafter we present some scope for future work in this area.

1. IDL interfaces for the OMG Persistence Object Service specification

The OMG specifications are studied in detail and the points which needed clarification are noted, to be resolved later. Once the overall picture of the POS architecture is understood, the interfaces are defined in IDL(section 2.3). Certain methods are added to some of the interfaces as required. Interfaces, which were purposely left unspecified by OMG, are defined and their interaction with the other interfaces in the POS system understood. Some of the interfaces are subclassed to either add to the basic functionality or to implement some properties of a *Protocol*.

2. Interaction between the POS components

Thereafter, the interaction between the various POS components are studied and discrepancies resolved. A notion of a client establishing a connection, then storing/restoring data, and thereafter terminating the connection, is established. More specifically the interaction between the PO, POM and PDS is carefully studied and resolved(section 2.4.1).

3. Implementing the DA Protocol

The Direct Access protocol as specified by OMG is implemented as the standard protocol in the POS system(section 2.3.1). The DAObject interface specification is outlined, from which other specific DAObjects inherit. The interaction between the PO, DAObject and PDS is defined. Methods are added to the DAObject interface, which allowed for the data to be passed from the PO to the DAObject and thereafter, from the DAObject to the PDS, which finally forwards it to the DataStore. Another issue here, is to

design an implementation, which will allow one to manage a network of DObjects, one referencing the other(section 2.4.4). This is also defined as explained in one of the preceding chapters.

4. Fault Tolerance and Reconfigurability of the POS system

The interdependence of the components in the POS system is studied further, to ascertain the vulnerable points in the system. This is analyzed and the problem solved by the use of *Component Configurator* objects, which make it easier to incorporate fault-tolerance in a graph of interconnected objects. The use of these Component Configurator also provided a very easy way to make the system reconfigurable at runtime(section 3.3).

5. Implementing persistence in the Profile Repository

The Profile Repository in the 2K system is used to maintain the user profiles of users registered in the 2K system. The POS framework developed could be employed immediately to store/restore the user profiles in an underlying DataStore. The flexibility of the POS framework, proved to be of immense use, as the data of the Profile Repository could now be saved in the filesystem or in a commercial DBMS as required. In the implementation designed(section 4.2), the filesystem is used to store the user profiles which could be restored anytime later upon restart of the Profile Repository server.

5.4 Scope for Future Work

In the present work, the specifications of OMG is followed in implementing the Direct Access(DA) protocol, which dictates the interaction between the Persistent Object(PO) and the Persistent Data Service(PDS). Several other protocols can be designed, to improve flexibility. Other CORBA services such as the *Relationship Service* and the *Externalization Service*

could be used within the POS framework. The Relationship Service could be used to describe the relationship between the various DAObjects and the Externalization Service could be used to stream data to and from the DAObjects.

List of References

- [1] “The common object request broker: Architecture and specification,” July 1995.
- [2] “Common object service specification,” March 1995.
- [3] F. Kon, A. Singhai, R. H. Campbell, D. Carvalho, R. Moore, and F. J. Ballesteros, “2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments,” in *ECOOP’98 Workshop on Reflective Object-Oriented Programming and Systems*, (Brussels, Belgium), July 1998.
- [4] F. Kon and R. H. Campbell, “On the Role of Inter-Component Dependence in Supporting Automatic Reconfiguration,” Tech. Rep. UIUCDCS-R-98-2080, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1998.
- [5] M. Román, F. Kon, and R. H. Campbell, “Supporting Dynamic Reconfiguration in the *dynamicTAO* Reflective ORB,” Tech. Rep. UIUCDCS-R-98-2085, Department of Computer Science, University of Illinois at Urbana-Champaign, February 1999.
- [6] R. M. Soley, ed., *Object Management Architecture Guide*. John Wiley & Sons, 3rd ed., 1990.
- [7] “Common object services volume 1, omg 94-1-1,” 1994.
- [8] “Object service architecture, omg 92-8-4,” 1992.
- [9] H. Robert, “Simple object persistence with store table,” *Journal of Object-Oriented Programming*, vol. 10, pp. 49–50, Jun 1997.
- [10] V. Srinivasan and D. T. Chang, “Object persistence in object-oriented applications,” *IBM Systems Journal*, vol. 36, no. 1, pp. 66–87, 1997.
- [11] C. K. Hess and R. H. Campbell, “Media Streaming Protocol: An Adaptive Protocol for the Delivery of Audio and Video Over the Internet,” in *ICMCS’99*, (Florence, Italy), IEEE, June 1999.
- [12] F. Kon, D. Carvalho, and R. Campbell, “Automatic Configuration in the 2K Operating System,” in *Proceedings of the ECOOP’99 Workshop on Object Orientation and Operating Systems*, (Lisbon), June 1999.

- [13] F. J. Ballesteros, C. Hess, F. Kon, S. Arévalo, and R. H. Campbell, "Object Orientation in Off++ - A Distributed Adaptable μ Kernel," in *ECOOP'99 Workshop on Object Orientation and Operating Systems*, (Lisbon), June 1999.
- [14] F. Kon and R. Campbell, "A Framework for Dynamically Configurable Multimedia Distribution," in *Proceedings of the ECOOP'99 Workshop for PhD Students in Object Oriented Systems*, (Lisbon), June 1999.
- [15] M. Román, F. Kon, and R. H. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: the *dynamicTAO* Case," in *Proc. ICDCS'99 Workshop on Middleware*, (Austin, TX), June 1999.
- [16] F. Kon and R. H. Campbell, "Supporting Automatic Configuration of Component-Based Distributed Systems," in *Proc. 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, (San Diego, CA), May 1999.
- [17] F. J. Ballesteros, S. Arevalo, F. Kon, and R. H. Campbell, "Towards a grand unified framework for mobile objects," in *III ECOOP Workshop on Mobility and Replication*, (Brussels, Belgium), July 1998.
- [18] F. Kon, R. H. Campbell, B. Srinivasan, R. Chandra, and A. Viswanathan, "Dynamic Reconfiguration of Scalable Internet Systems with Mobile Agents," Tech. Rep. UIUCDCS-R-99-2105, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1999.
- [19] F. J. Ballesteros, C. Hess, F. Kon, and R. H. Campbell, "The Design and Implementation of the Off++ and vOff++ μ kernels," Tech. Rep. UIUCDCS-R-98-2086, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1999.
- [20] F. J. Ballesteros, F. Kon, and R. H. Campbell, "A Detailed Description of Off++, a Distributed Adaptable Microkernel," Tech. Rep. UIUCDCS-R-97-2035, University of Illinois at Urbana-Champaign, August 1997. Also available at <http://choices.cs.uiuc.edu/2k/off++>.
- [21] R. B. Moore, "An extensible architecture for distributed object system interoperability," Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1998. Available at <http://choices.cs.uiuc.edu/2k>.
- [22] R. H. Campbell, K. Nahrstedt, and M. D. Mickunas, "2K: A Component-Based Network-Centric Operating System." Project home page: <http://choices.cs.uiuc.edu/2K>, 1998.
- [23] F. Kon, "dynamicTAO: Adding Dynamic Flexibility to ACE/TAO." Project home page: <http://choices.cs.uiuc.edu/2K/dynamicTAO>, 1998.
- [24] F. Kon, "ComponentConfigurator Source Code and Documentation." Project home page: <http://choices.cs.uiuc.edu/2K/ComponentConfigurator>, 1998.

- [25] F. Kon, “Distributed Configuration Protocol.” Project home page: <http://choices.cs.uiuc.edu/2k/dynamicTA0/DCP.txt>, June 1998.
- [26] R. Campbell, M. D. Mickunas, and K. Nahrstedt, “Dynamic Resource Management for a Network-Centric Operating System.” NSF/CISE-funded project, no. 98-70736, 1998.
- [27] A. Singhai, A. Sane, and R. Campbell, “Quarterware for Middleware,” in *Proc. 18th International Conference on Distributed Computing Systems (ICDCS)*, pp. 192–201, IEEE, May 1998.