Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: Cambridge (0223) 323010
INTERNATIONAL: +44 223 323010
FAX: +44 223 359779
E-MAIL: apm@ansa.co.uk

ANSA Phase III

A Real Time Programming Model for CORBA

Nicola Howarth, Guangxing Li

Abstract

This document aims to identify the requirements and limitations of the work required to add performance features to Orbix, with particular reference to the RIDE real time extensions to ANSA.

APM.1059.00.04	**Draft**	27 January 1994

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

A Real Time Programming Model for CORBA

Request for Comments (confidential to ANSA consortium for 2 years)

A Real Time Programming Model for CORBA

Nicola J. Howarth
Guangxing Li

APM.1059.00.04

27 January 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Copyright © 1993 Architecture Projects Management Limited

Architecture Projects Management Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

Page	Section	Title
1	**1**	**Introduction**
3	**2**	**RIDE**
3	2.1	Introduction
3	2.2	The Problem Space
3	2.2.1	The ANSA Object Execution Model
4	2.2.2	ANSAware Object Execution Model Deficiencies for Realtime Applications
4	2.3	The RIDE Engineering Model
4	2.3.1	RIDE Objects
5	2.3.2	Object Invocation
5	2.3.3	Priority and Deadline Scheduling Policies
6	2.3.4	Programming Extensions for the RIDE Engineering Model
9	2.4	The RIDE Communication System
10	2.5	Temporal Synchronisation
10	2.5.1	Timed Automata Synchronization Mechanism
12	2.5.2	Virtual Time
13	2.6	Summary of RIDE extensions
13	2.6.1	The Invocation
13	2.6.2	Entry Creation
13	2.6.3	Interface Creation and Entry Bindings
13	2.6.4	Application Task Rendezvous
13	2.6.5	Resource Reservation
14	2.6.6	Communications
14	2.6.7	Temporal Synchronization Mechanisms
14	2.6.8	Future Extensions
17	**3**	**Orbix**
17	3.1	Goals
17	3.2	CORBA and Orbix
17	3.3	CORBA components
18	3.4	Orbix, ANSA and C++
18	3.5	Orbix Concepts
18	3.5.1	Invocations
18	3.5.2	Interfaces
19	3.5.3	Binding and Exceptions
19	3.5.4	Orbix Daemon and the Implementation Repository
20	3.5.5	Interface Repository
20	3.6	Programming in Orbix
20	3.6.1	IDL
21	3.6.2	Binding and invocation
22	3.6.3	Implementation by a server
23	3.6.4	Registration, binding and server activation

24 3.6.5 Exception Handling
25 3.6.6 Inheritance
25 3.6.7 Filtering
26 3.6.8 Smart Proxies

27 4 An Orbix Example
27 4.1 The SimpleBank Example
27 4.2 Interfaces
29 4.3 Orbix server
29 4.4 Clients
29 4.4.1 The Bank Manager client
30 4.4.2 The ATM client
31 4.5 Implementation
32 4.6 Summary of source files
32 4.7 Execution trace
35 4.8 Stub files
35 4.8.1 Typedef and Exception statements
35 4.8.2 Interface statements

41 5 Comparison of Orbix and ANSAware
41 5.1 Language Compatibility
41 5.2 Tool Compatibility
42 5.2.1 STUBC and PREPC
42 5.2.2 The Trader
44 5.2.3 The Factory
44 5.2.4 The Nodemgr
45 5.3 Communicating between Orbix and ANSAware
45 5.3.1 Underlying communications
45 5.3.2 Support for Groups
46 5.3.3 Multiple protocols
46 5.4 Support for Storage
47 5.5 Orbix threads with ANSAware real time scheduling

49 6 Implementing RIDE on Orbix
49 6.1 The Object-Oriented Approach
49 6.2 Invocations
50 6.3 The Entry Interface
51 6.3.1 The Interface Definition
52 6.3.2 Usage from a Client
53 6.3.3 The Implementation by a Server
53 6.4 Resource Allocation
56 6.5 rpc Attributes
57 6.6 Parallel Protocol Stack
57 6.7 Synchronization
57 6.7.1 The Interface Definition
58 6.7.2 Usage from a Client
59 6.8 Virtual Time
59 6.8.1 The Interface Definition
59 6.8.2 Usage from a Client

1 Introduction

The Performance Group is currently working on both a programming model and an engineering model for real time performance in managed open systems.

The objectives of the work are set out in the document "*A Performance Framework*" [APM 93a]. This technical report introduces the ANSA work programme activities on real time and performance management aspects of open, distributed systems. It forms a statement of baseline and scope for work on real time and performance management. The emphasis of the document is on parameters, scope, and interaction and interference between timeliness, performance and other open, distributed system control functions.

The high level engineering design is set out in "*An Architecture for Real-Time: Engineering Aspects*" [APM 93b]. It describes the architecture of a distributed environment for real-time applications, covering the entire system environment.

The high level design requires a programmer's interface to the engineering components. This high level design has previous been implemented using ANSAware with a RIDE prototype. The purpose of this document is to explore how to use CORBA as the programmer's interface, using Orbix, which is fully CORBA compliant.

The aim of this document is therefore to identify the requirements and limitations of the work required to add performance features to Orbix, with particular reference to the RIDE real time extensions to ANSA. This is exploratory work, intended to identify the problems involved both with developing a real-time programming language, and with the use of existing tools and their suitability (or lack of it).

In the process of carrying out this work, an additional benefit will be the identification of deficiencies in CORBA in relation to real-time programming, and an understanding of the problems involved in integrating different tools, here Orbix and ANSAware, the former generating C++, the latter written in C.

Section 2 introduces the RIDE programming model, and identifies the required extensions to the existing ANSAware Testbench. This provides the problem space.

Section 3 gives an overview of Orbix, identifying its principle features and providing an introduction to writing programs in Orbix.

Section 4 describes how to program the ANSAware SimpleBank example using Orbix. It also outlines the code generated by Orbix, and traces through the execution path.

Section 5 compares Orbix with ANSAware, since the use of Orbix will replace many of the ANSAware features (in particular the Trader), without replacing their full functionality. Identification will also be made of the facilities offered

by Orbix which are not supported by ANSAware, in particular the Interface Repository.

Section 6 outlines a possible approach to implementing the computational features of RIDE using Orbix, and identifies some of the benefits and drawbacks of such an approach.

There are two principle benefits from taking the approach outlined above. Firstly the work will identify real time extensions required for CORBA, and secondly it will provide a basis for prototyping a declarative real time framework. It forms part of the overall real time prototype. Since this will have been developed on a CORBA-compliant system, it should be easily portable between such systems.

It is assumed that readers of this document will already be familiar with the ANSA architecture and with ANSAware.

2 RIDE

2.1 Introduction

RIDE (Real tIme Distributed system Environment) is a distributed system environment developed for the task of programming and executing large realtime application [Li93]. The RIDE approach is to provide functional extensions and elaborations to a non-realtime system environment, namely ANSA. RIDE is an ANSAware-based prototype of the ANSA High Level Performance Engineering model.

This section does not attempt to explain in detail the full methodology and reasoning behind RIDE, but to provide a brief summary of those components necessary to implement the computational model of RIDE using Orbix. It is therefore not complete in itself. The RIDE engineering model is described primarily in terms of its interfaces, in order to identify the interaction required with e.g. PREPC, or Orbix. The details of RIDE implementation are not discussed.

2.2 The Problem Space

RIDE makes use of an object-based programming model from the ANSA architecture. The RIDE Realtime Programming Model describes the structure of RIDE objects, along with object invocation mechanisms, the handling of priorities and deadlines, resource allocation, scheduling mechanisms and policies, and the application's control over scheduling.

For realtime applications, the execution aspect of objects has fundamental effects on the *predictability* of computational activities. Realtime object execution models must address not only how computational activities are carried out, but also how shared resources are used. Distributed realtime systems must provide support for the specialised requirements of realtime communication, tasking, scheduling, and control, and these requirements must be explicitly addressed in an object execution model.

2.2.1 The ANSA Object Execution Model

The ANSA Object Execution Model can be summarised as follows:

- objects export services through interfaces
- threads are created either explicitly for concurrent computational activities, or implicitly by the invocations between objects.
- the capsule is in charge of the management of resources (tasks, buffers etc.) in the system, and of their allocation to different threads.

The system behaviour is therefore dependent on the system's resource management policy, and the infrastructure offers no mechanism by which to interface with this management. The resulting behaviour is non-

deterministic, and depends entirely on system workload. No guarantees can be made.

2.2.2 ANSAware Object Execution Model Deficiencies for Realtime Applications

The time-sharing characteristics of tasking and scheduling can be summarised as follows:

- multiplexing of one thread queue. The queue is used for all interfaces within a capsule; all system tasks are homogeneous and are allocated for serving any thread.
- thread enqueue policy (and thus request service scheduling policy) is First In First Served (FIFS).

The ANSAware tasking system is efficient at the task/thread resource sharing based on this single capsule-wide thread queue with a pool of tasks, but imposes severe constraints on flexible and realtime scheduling. It precludes the possibility of preallocating tasks for realtime services. If all system tasks have been assigned to some time-consuming non-realtime threads, newly arrived realtime requests must wait for the completion of non-realtime requests. This design also precludes the possibility that an application may perform its own resource management, synchronisation and scheduling on the basis of services and tasks.

The simple FIFS thread enqueue policy therefore precludes realtime performance, when the object is executed in an open environment where both time-constrained and non-constrained operations may be requested dynamically.

2.3 The RIDE Engineering Model

The RIDE Computational Model is based on the ANSA Computational Model, but with several extensions to support the requirements of realtime.

2.3.1 RIDE Objects

The ANSA object is consists of data, one or more tasks of execution, and a set of exported interfaces. RIDE adds a further abstraction, the *entry*, as the basic mechanism for realtime scheduling. An entry is an activity queue with a record of control data. It may be created dynamically, and interfaces of an object may be bound to it. When an interface is bound to an entry, each operation request on the interface is transferred to an activity enqueued on the entry. Any thread representing a computational activity is also spawned with an entry identifier. The entry is an engineering concept which is confined within a capsule. It is discussed here briefly since the computational model is required to provide a mechanism by which entries can be created.

The entry is the mechanism on which flexible tasking is based. System tasks may be allocated for each individual entry. The tasks allocated are dedicated to execute the threads on the entry. When executing a thread, a task is also allowed to rendezvous with other entries dynamically. A *rendezvous* of a task with an entry means that the task waits to accept and execute one thread on the entry. Different control parameters may be selected for each entry to choose a thread enqueue policy, a task/entry rendezvous policy, and to enforce concurrency controls.

In RIDE, with data, interface, entry and tasks encapsulated within a capsule, there is a choice of how many entries are allocated, which interface is attached to which entry, how many tasks are allocated to an entry, whether a task rendezvous with a specific entry, and what kinds of resource scheduling policies are used.

The decision to allocate a new entry for a set of interfaces reflects the need to separate these interfaces from others for the purpose of resource management.

The number of tasks allocated to an entry not only enforces the real concurrency allowed for the execution of threads on the entry, but also affects the realtime scheduling properties, such as preemptivity.

The flexibility for allowing a task to rendezvous with an entry enables an application to have complete control over its virtual processor(s) based on its knowledge of the system state.

The user control over system tasking behaviour is further enhanced by the scheduling policy/mechanism separation in the RIDE architecture.

2.3.2 Object Invocation

To support the realtime requirements of a system, there must be some means of enabling the urgency of a computational activity to be spread among all the nodes it needs to access. This urgency information should be used by used by the system resource scheduler to resolve resource contention so that important or more urgent computational activities have better access to system resources. In RIDE, this is done by allowing the association of an optional priority and/or deadline with each invocation. As the invocation crosses the physical boundary and becomes a thread in the called object, this priority and/ or deadline is also passed and becomes a property of the thread, which may then be used as scheduling parameters at the server site.

Allowing explicit invocation priority has several benefits:

- extra flexibility in conjunction with the server scheduler in determining how the invocation is to be processed
- a low-priority invocation may be sent from a high-priority task without modifying the server task's priority
- a low-priority thread may send a high-priority invocation to a server indicating an urgent situation

The effect on the system resource management is also determined by the scheduling policy and the resources allocated for the service.

2.3.3 Priority and Deadline Scheduling Policies

The following terms need to be defined:

- *static priority* - that which is declared at the creation of a task.
- *dynamic priority* - the static value potentially enhanced by a rendezvous or an explicit change of priority.

There are three types of priority protocols defined in RIDE. With *priority inheritance* a serving task may take into account the priority of an invocation, and use this priority as its dynamic priority. RIDE defines two levels of priority inheritance schemes: *(basic) priority inheritance,* and *transitive priority inheritance*.

With *(basic) priority inheritance*, a server task with low priority raises its priority to the higher priority of an invocation request prior to starting the service, and returns to its original value on completion.

Transitive priority inheritance is an extension of the basic scheme, covering the situation where there are no waiting server tasks, and a high priority request arrives. Here the invocation priority is compared with those of the running tasks. If all of the running servers are running at a lower priority than the new invocation, one of these tasks is chosen to inherit the invocation priority. If any of the running servers are running at a higher priority than the new invocation, then the invocation is queued in the entry.

The *Priority Ceiling* protocol associates each entry with a fixed priority ceiling value, which specifies an upper-bound priority that applies to all invocations on the interfaces bound to an entry. While a task is executing a thread on the entry, its priority is raised to the ceiling priority. If an invocation has a higher priority than the ceiling priority, it is rejected.

The deadline associated with an invocation specifies a bound on the completion time of the requested operation. An earlier deadline has a higher priority than a late one, and is scheduled on this basis. Pre-emption is possible if the task scheduler provides an earliest deadline first pre-emptive scheduling service, and serving tasks are allowed to inherit thread deadlines. This *deadline inheritance* can be handled in the same way as the priorities defined above.

Priority and deadline scheduling can be combined to provide alternative scheduling models. One possibility is *priority first, then deadline*, in which deadlines are only used when two threads have the same priority. Another combination is *deadline first, then priority*, where priorities are only used in the case of an unsatisfiable deadline.

2.3.4 Programming Extensions for the RIDE Engineering Model

2.3.4.1 The Invocation and Entry Interface

The ANSA Testbench currently uses an Interface Definition Language (IDL) to define ANSA interfaces, and a preprocessor (PREPC) which scans C programs for embedded statements (DPL) which augment the original program to bind interfaces and invoke remote operations. IDL has an associated compiler, STUBC, which generates stub code from IDL definitions.

An ANSA invocation is as follows:

```
{result} <- IfRef$Operation(arguments)
```

The RIDE extensions of DPL allow realtime attributes to be attached to invocations. A realtime invocation is as follows:

```
{results} <- IfRef$Operation(arguments) rtAttrs rt_attrs
```

The parameter `rt_attrs` may optionally have values for priority, deadline, deadline type, timeout, RPC protocol id, etc. The `rtAttrs` item itself is optional, allowing non-realtime invocations to retain their original form.

An entry may be created by:

```
entry = Entry(enqueue_policy, rendezvous_policy, control);
```

The enqueue_policy argument selects which thread enqueue policy is used. The rendezvous_policy argument selects which priority inheritance protocol is used for task/thread rendezvous. The control argument defines the enqueue_policy related arguments, for example the allowed priority range of all invocations on the entry, the priority ceiling values and so on.

An interface may be created and bound to an entry:

```
{IfRef} :: IfName$Create(arguments)

EntryBind(entry, IfRef)
```

System tasks may be allocated to an entry:

```
TaskSpawn(entry, tasks, arguments);
```

Six thread enqueue policies are defined:

- FIFS - First In, First Served
- PB - static Priority Based
- DB - earliest Deadline first
- PDB - static Priority first, then earliest Deadline Based
- DPB - earlier Deadline first, then static Priority Based
- USER - application supplied policy

The behaviour of the user policy is defined by the control argument in the entry create operation. This argument provides an upcall function and a thread enqueue policy identifier selecting one of the other five policies listed above. The upcall function is called whenever an invocation arrives on the entry. It returns a flag indicating if the invocation is either:

- schedulable - to be queued and served
- ignored - a bogus client is detected
- unschedulable - not enough resource to serve the invocation.

In the case of schedulable, the function also returns the value of the invocation deadline and/or priority to be used by its selected thread enqueue policy. The RIDE system scheduling does not provide *guaranteed scheduling*, but rather allows an application to provide its own schedulable test algorithm, with its own standard of guarantee and resource management policy, and integrate it with the RIDE infrastructure.

Six task/thread rendezvous protocols (policies) are defined:

- rendezvous_N - the null protocol (the priority and deadline of a thread have no effect on its serving task)
- rendezvous_PI - the priority inheritance protocol
- rendezvous_TPI - the transitive priority inheritance protocol
- rendezvous_C - the priority ceiling protocol
- rendezvous_DI - the deadline inheritance protocol
- rendezvous_PDI - the priority and deadline inheritance protocol

Initially a RIDE capsule has a default system entry, and all ANSA system services (e.g. Capsule, Object, and Notification interfaces) are bound to this entry. The default system entry has a FIFS enqueue_policy, and a

rendezvous_N rendezvous policy. Any newly created interface reference is by default bound to the system entry. The `EntryBind` function is used to force the interface to be bound to a specific entry. An interface may be reset, and bound back to the system entry, by the operation `UnBind(IfRef)`. An entry may be closed by `EntryClose(entry)`.

2.3.4.2 *Application Task Rendezvous*

In addition to allocating system task(s) on an entry for service requests, RIDE also allows tasks to rendezvous with entries at run-time. The interface is as follows:

```
Accept(entry_set, timeout)
```

An `entry_set` may be a single entry, or the union of several entries. The task waits for at most `<timeout>` to serve one request on any entry of the entry_set.

Figure 2.1 shows an application task rendezvous to implement a bounded buffer. The application task rendezvous has the following characteristics:

- The rendezvous is transparent to the client.
- The `Accept` statement ensures that only one request is executed in the server. Other requests are queued, until the server task executes a subsequent `Accept` statement.
- The server performs its own synchronisation. It accepts requests only when it is able to complete immediately. In contract, if system tasks are used, the synchronisation much be checked inside each operation. Synchronisation is performed by suspending the task at a condition variable or semaphore. This method is expensive on system resources, and cannot be solved by simply limiting the number of concurrent requests allowed. Application task rendezvous may circumvent the problem by synchronising before a request starts execution, and not after.
- The server may initiate object invocations like other client tasks.
- The server may perform its resource management when not responding to external requests. It is therefore possible to have interface-specific tasks with pre-allocated resources and optimized synchronisation management.

2.3.4.3 *End-to-End Scheduling*

The client-server model generally supports *many-to-one* interactions. This is natural for server design, enabling the sharing of server resources, but needs revising for realtime use since realtime applications often require client-based resource preservation and guarantee. This is often known as an *end-to-end* guarantee, and requires end-to-end scheduling.

The RIDE architecture does not provide end-to-end scheduling directly. However applications requiring this can be designed on top of the RIDE mechanisms. ANSA allows the dynamic creation and passing of interface references, so it is possible to have a resource management interface which provides an operation of the following signature:

```
getIfRef :OPERATION [ resource-requirements ]
         RETURNS [ service-interface ]
```

Before a client uses the server, it uses getIfRef to prepare the server as to its resource requirements. The management interface can then create a new interface instance of the server, allocate required resources (tasks, entry and

Figure 2.1: A Bounded Buffer

```
- - IDL Interface
BufferIn : INTERFACE
BEGIN
    In : OPERATION [in : INTEGER] RETURNS [];
END.
BufferOut : INTERFACE
BEGIN
    Out : OPERATION [] RETURNS [INTEGER];
END.
- - server.dpl - server program
#define MaxBuf 10
static int count = 0;
DECLARE ir_in : BufferIn SERVER
DECLARE ir_out : BufferOut SERVER
static RIDE_Entry in_entry, out_entry;
int BufferIn_In(ansa_InterfaceAttr *_attr, int in)
{
    count++;
}
int BufferOut_Out(ansa_InterfaceAttr *_attr, int *out)
{
    count--;
}
static int svr_task()
{
    for(;;) {
        if (count == 0)
            Accept(in_entry,0);
        else if (count == MaxBuf)
            Accept(out_entry,0);
        else Accept(in_entry | out_entry, 0);
    }
}
Body()                    - - server program starts here
{
    create ir_in and ir_out interface reference;
    create in_entry and out_entry;
    bind ir_in to in_entry, ir_out to out_entry;
    spawn the server task svr_task;
}}
```

application level resources) to the interface instance, and pass that reference back to the client. The client is then able to use the server with guaranteed resources.

2.4 The RIDE Communication System

The ANSA Testbench communication system implements three protocol layers:

- Message Passing Services (MPS): these provide an interface to the transport protocols provided by the underlying operating system.
- Execution Protocols (REX/GEX): these form part of the mechanism which facilitates the invocation of ANSA operations.

- Sessions: these are used to store the end-to-end state required for a remote invocation and to synchronise the execution of the tasking and the communication systems.

The ANSA communication system design makes efficient use of resources by heavy use of multiplexing. This raises problems for realtime applications:

- there is no association between the (interface level) channels and MPS channels, and the two level modules have no interactions when channels are created and destroyed. This means that although it is possible to distinguish interfaces providing realtime services from those providing non-realtime services at a high level, communication to/from these interfaces may share the same MPS communication channel.
- IDL and PREPC provide no way of selecting between multiple execution protocols.

The RIDE communication system attempts to overcome these two problems as follows:

- MPS interface is redesigned as connection-based, maintaining simple states of its channels. The Execution Protocol is extended to use this connection-based interface. The result is a parallel protocol stack.
- IDL stubc and PREPC compilers are extended to allow an application to select a dedicated execution protocol on a per-call basis.

To support this, DPL is extended to understand the connection-based nature of communications. A client may set up a private MPS communication channel to a server interface as follows:

```
IfRef$Connect(arguments)
```

The arguments are expected to include communication QoS in the future, when the underlying operating system can provide the required service. Such private channels can be released similarly:

```
IfRef$Disconnect()
```

It is not necessary for every client to call the IfRef$Connect operation to set up its own private MPS channel; a default channel may be used.

2.5 Temporal Synchronisation

RIDE uses a *timed automata* model for providing temporal synchronization. The timed automata synchronization facility is designed to fit the ANSA Computational Model. A timed automata is intended to provide a temporal synchronization service, and to determine whether an activity requesting an invocation may proceed immediately, or some other action must be taken. The main function of a timed automata is to delay a synchronization operation until a synchronization condition is satisfied.

2.5.1 Timed Automata Synchronization Mechanism

There are three major functional requirements for the timed automata synchronization mechanism:

- to provide a facility for defining and signalling events
- to define the allowed transitions

- to provide a facility for defining and waiting on synchronization conditions

ANSA announcements and interrogations can both be used for the first of these. The second is defined by a timed automata specification language. ANSA interrogations are able to provide the third.

The synchronization services are provided by the *timed automata interface*. Timed automata interfaces are specified by the timed automata specification language. A timed automata is an instance of a timed automata specification which encapsulates its own states and timers. A timed automata specification consists of a set of states, a set of timers, a set of transitions on these states, and a set of timing guards. In addition, named subsets of the set of states may also be defined. Figure 2.2 shows the syntax of the timed automata specification language. The BNF notation is extended by the use of "..." to denote a list of zero or more of the surrounding items.

Figure 2.2: Timed Automata Specification Language Syntax

```
TA_Interface ::= InterfaceName : TA_INTERFACE = Spec
Spec ::= BEGIN Body END.
Body ::= TimerSpec StateSpec SetSpec GuardSpec TransitionSpec Initialization
TimerSpec ::= | TIMER : TimerList;
StateSpec ::= STATE : StateList;
SetSpec ::= SetDecl ... SetDecl
SetDecl ::= SET SetName : StateList;
GuardSpec ::= GuardDecl ... GuardDecl
GuardDecl ::= GUARD StateName TimerExpr;
TransitionSpec ::= TransitionDecl | TransitionDecl ... TransitionDecl
TransitionDecl ::= TRANSITION TranName TranList;
TranList ::= TranItem | TranItem ... TranItem
TranItem ::= (StateName, TimerExpr, Action, StateName)
Actoin ::= | < TimerList >
TimerExpr ::= | TimerOp | (TimerExpr) | TimerExpr & TimerExpr | TimerExpr ‘|‘ TimerExpr
TimerOp ::= TimerItem TimerOperator TimerItem
TimerItem ::= TimerName | Number
TimerOperator ::= >= | <=
Initialization ::= STARTAT StateName Action;
TimerList ::= NameList
StateList ;;+ NameList
NameList ::= Identifier | Identifier, ... Identifier
SetName ::= Identifier
StateName ::= Identifier
TranName ::= | Identifier
```

A client may issue the following operations to a timed automata service:

Signal an Event. State transition operations on a timed automata are given by the TRANSITION attributes defined on the interface.

```
{} <- IfRef$Signal(Transition)
```

For example, invoking the transition operation with the tuple (*s1, t, act, s2*) will change the state of the transition to *s2*, provided the current state is *s1* and the timing constraint *t* is satisfied. In addition, the timers in the *act* are reset to zero.

Wait on Synchronization Conditions. There are two kinds of synchronization conditions that can be applied to a timed automata. One of these corresponds to each TRANSITION attribute, and the other corresponds

to each SET attribute. There are also two combined synchronization operations that can be applied. The operations are:

```
Wait on transition:
     {state} <- IfRef$Await(TRANSITION, Transition, timeout)
Wait on set:
     {state} <- IfRef$Await(SET, Set, timeout)
Combined operations:
     {state} <- IfRef$SigAwait(Transition,Relation,Name,timeout)
     {state} <- IfRef$AwaitSig(Transition,Relation,Name,timeout)
```

The `Await` operation can choose either TRANSITION or SET as the required synchronization condition. If TRANSITION is chosen, the operation will wait until a transition of the name `Transition` occurs in the timed automata. The `timeout` argument takes its intuitive meaning - a positive value will force the operation to return when the timeout expires; a zero value enables the operation to wait until the required synchronization condition is satisfied. The result parameter `state` contains the state of the timed automata when the operation returns. If the SET is used with the `Await` operation, it waits until the state of the timed automata is a member of the Set. A convention is adopted to allow a single state to denote the state set of itself. The `SigAwait` and `AwaitSig` operations combine the effect of a `Signal` and an `Await` operation. The `SigAwait` first signals a transition `Transition`, and then awaits the required synchronization condition. The `Relation` is either TRANSITION or SET, and the `Name` is a `Transition` or `Set` correspondingly. The `AwaitSig` first awaits a synchronization condition and then signals a transition.

All operations on a timed automata are atomic - there is no concurrent execution of operations.

Inquiry. The following operation can be used to get the current state of a timed automata.

```
{state} <- IfRef$State
```

2.5.2 Virtual Time

Associated with each timed automata are some time management operations to enable an application to define and use its own virtual time. A virtual time y is a linear function of the real time x, i.e. $y = ax+k$, in which $a=m/n$. By default, the virtual time of a timed automata is directly mapped to the real time, i.e. $y=x$. The following operations can be applied to control the clock tick of the virtual time of a timed automata:

```
Set the virtual time function:
     {} <- IfRef$Time_Virtual(m,n,k)
Freeze the virtual time by delay:
     {} <- IfRef$Time_Freeze(delay)
Resume the virtual time:
     {} <- IfRef$Time_Resume()
Catch up the delayed virtual time:
     {} <- IfRef$Time_Catchup(delay)
```

The virtual time function can also be set at creation time:

```
{IfRef} :: IfName$Create(arguments)
IfRef$Time_Virtual(m,n,k)
```

2.6 Summary of RIDE extensions

2.6.1 The Invocation

An ANSA invocation is as follows:

```
{result} <- IfRef$Operation(arguments)
```

The RIDE extensions of DPL allow realtime attributes to be attached to invocations. A realtime invocation is as follows:

```
{results} <- IfRef$Operation(arguments) rtAttrs rt_attrs
```

The parameter `rt_attrs` may optionally have values for priority, deadline, deadline type, timeout, RPC protocol id, etc. The `rtAttrs` item itself is optional, allowing non-realtime invocations to retain their original form.

2.6.2 Entry Creation

An entry may be created by:

```
entry = Entry(enqueue_policy, rendezvous_policy, control);
```

The enqueue_policy argument selects which thread enqueue policy is used. The rendezvous_policy argument selects which priority inheritance protocol is used for task/thread rendezvous. The control argument defines the enqueue_policy related arguments, for example the allowed priority range of all invocations on the entry, the priority ceiling values and so on.

2.6.3 Interface Creation and Entry Bindings

An interface may be created and bound to an entry:

```
{IfRef} :: IfName$Create(arguments)
EntryBind(entry, IfRef)
```

System tasks may be allocated to an entry:

```
TaskSpawn(entry, tasks, arguments);
```

2.6.4 Application Task Rendezvous

In addition to allocating system task(s) on an entry for service requests, RIDE also allows tasks to rendezvous with entries at run-time. The interface is as follows:

```
Accept(entry_set, timeout)
```

An `entry_set` may be a single entry, or the union of several entries. The task waits for at most `<timeout>` to serve one request on any entry of the entry_set.

2.6.5 Resource Reservation

The ability to create an interface instance of a server with the required resources allocated to it is as follows:

```
getIfRef :OPERATION [ resource-requirements ]
         RETURNS [ service-interface ]
```

2.6.6 Communications

A client may set up a private MPS communication channel to a server interface as follows:

```
IfRef$Connect(arguments)
```

Such private channels can be released similarly:

```
IfRef$Disconnect()
```

2.6.7 Temporal Synchronization Mechanisms

2.6.7.1 Wait on Synchronization Conditions:

```
Wait on transition:
     {state} <- IfRef$Await(TRANSITION, Transition, timeout)
Wait on set:
     {state} <- IfRef$Await(SET, Set, timeout)
Combined operations:
     {state} <- IfRef$SigAwait(Transition,Relation,Name,timeout)
     {state} <- IfRef$AwaitSig(Transition,Relation,Name,timeout)
```

2.6.7.2 Inquiry

```
{state} <- IfRef$State
```

2.6.7.3 Virtual Time

```
Set the virtual time function:
     {} <- IfRef$Time_Virtual(m,n,k)
Freeze the virtual time by delay:
     {} <- IfRef$Time_Freeze(delay)
Resume the virtual time:
     {} <- IfRef$Time_Resume()
Catch up the delayed virtual time:
     {} <- IfRef$Time_Catchup(delay)
Set the virtual time on creation:
     {IfRef} :: IfName$Create(arguments)
     IfRef$Time_Virtual(m,n,k)
```

2.6.8 Future Extensions

This summary is based on the original RIDE specification. Future work is likely to make use of additions outlined below. These should be considered when porting the engineering model of RIDE onto Orbix.

2.6.8.1 Entry

An `entry` object is specified by a variable `entry_attr` of entry attributes type `entry_attr_t`. Operations on an `entry` include:

- create a default entry attributes object:
  `result_code = entry_attr_create(entry_attr)`
- set or obtain the default attributes of an entry attributes object
  `result_code = entry_attr_set_anAttr(entry_attr, an_attr)`
  `an_attr = entry_attr_get_anAttr(entry_attr)`
- create a new entry object:
  `result_code = entry_create(entry_attr, entry)`

- close an entry:
  `result_code = entry_close(entry)`
- bind an interface to an entry:
  `result_code = entry_bind(interface_ref, entry)`
- unbind an interface to an entry:
  `result_code = entry_unbind(interface_ref, entry)`
- spawn tasks on an entry:
  `result_code = task_spawn(entry, tasks, task_attr)`
- create a default entry set attributes object:
  `result_code = entry_set_attr_create(entry_set_attr)`
- set or obtain the default attributes of an entry set attributes object:
  `result_code=entry_set_attr_set_anAttr(entry_set_attr, an_attr)`
  `an_attr = entry_set_attr_get_anAttr(entry_attr)`
- create an entry set object:
  `result_code = entry_set_create(entry_set_attr, entry_set)`
- add an entry to an entry set:
  `result_code = entry_set_add(entry, entry_set)`
- remove an entry from an entry set:
  `result_code = entry_set_remove(entry, entry_set)`
- close an entry set:
  `result_code = entry_set_close(entry_set)`
- application controlled rendezvous:
  `result_code = accept(entry_set, timeout)`

Possible entry attributes are thread queueing policy, task/thread rendezvous policy, ceiling value, priority low bound, priority upper bound, user scheduling, and so on.

Possible entry set attributes are entry join policy and so on.

2.6.8.2 *Call Attributes Object*

RPC calls are allowed to be associated with call attributes objects:

- create a default call attributes object:
  `result_code = rpc_attr_create(rpc_attr)`
- set or obtain the default attributes of a call attributes object:
  `result_code = rpc_attr_set_anAttr(rpc_attr, an_attr)`
  `an_attr = rpc_attr_get_anAttr(entry_attr)`
- associate a call attributes object with a RPC:
  `! xxx = yyy(zzz) rpc_Attr rpc_attr`

Possible call attributes are priority, deadline, deadline type, timeout, rpc protocol id and so on.

2.6.8.3 *Parallel Protocol Stack*

A channel is defined by a channel attributes object. It also implied an interface attributes object.

- create a default channel attributes object:
  `result_code = channel_attr_create(channel_attr)`

- **set or obtain the default attributes of a channel attributes object:**
```
result_code = channel_attr_set_anAttr(channel_attr, an_attr)
an_attr = channel_attr_get_anAttr(channel_attr)
```
- **create a default interface attributes object:**
```
result_code = interface_attr_create(interface_attr)
```
- **set or obtain the default attributes of an interface attributes object:**
```
result_code=interface_attr_set_anAttr(interface_attr, an_attr)
an_attr = interface_attr_get_anAttr(interface_attr)
```
- **interface create:**
```
! interface = yyy(interface_attr)
```
- **channel create:**
```
! interface$channel_create(channel_attr)
```
- **channel close:**
```
! interface$channel_close()
```

3 Orbix

3.1 Goals

This chapter investigates Orbix with a view to implementing the engineering model of RIDE in Orbix.

3.2 CORBA and Orbix

In December 1991 the USA based Object Management Group (OMG) released a document entitled “The Common Object Request Broker: Architecture and Specification”. This describes and specifies an Object Request Broker (ORB), which is a layer of software responsible for locating and sending requests to software entities in a network.

In CORBA the only requirement for an object to be remotely accessible is that it have an interface which is described by the CORBA Interface Definition Language (IDL). The only functions that can be invoked on a remote object are those described by the IDL interface to that object.

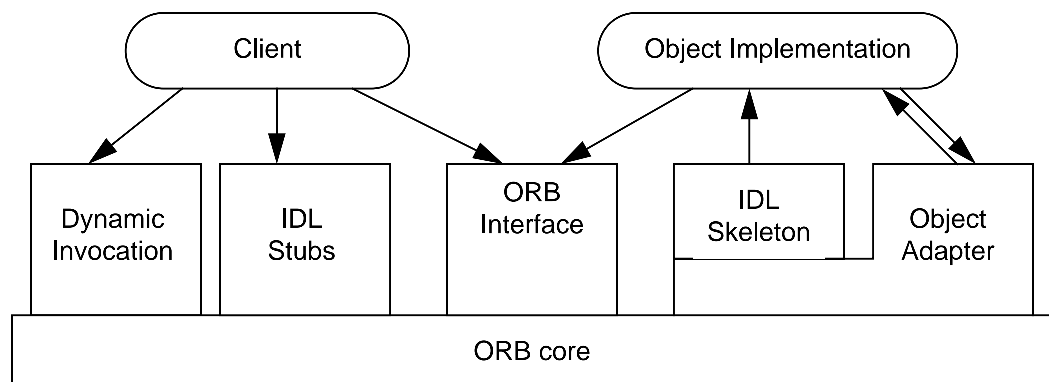
When a client issues a request to a server, CORBA is responsible for locating the object and delivering the request to it. All details of the server are transparent to the client, other than the interface that the server offers.

Orbix has been designed as a fully CORBA-compliant system, and implements most of the features of CORBA, including the passing of interface references, and exception handling. It includes support for the C++ programming language.

3.3 CORBA components

The are five components of the CORBA interface [IONA 93] shown in Figure 3.1 and listed below:

Figure 3.1: CORBA Components



- When a client issues a request to a server, code to marshall the request parameters into a single line buffer is needed before the request can be transmitted across the network. This is provided by the "*IDL stubs*" component of CORBA. (An IDL stub is needed for each type of server on the network.) These stubs are generated at compile time and statically linked with the client.
- CORBA defines a *Dynamic Invocation Interface* (DII) which allows clients to build and send requests at runtime. This enables a client to interact with servers of which it had no knowledge at compile time.
- The *IDL Skeleton* is responsible for decoding the request on arrival at the server. It unmarshalls the parameters and calls the appropriate server operation.
- The *Object Adapter* is the primary method by which the server interacts with the ORB, providing the required interfaces for services such as security, activation and deactivation.
- The *ORB Interface* provides access to key ORB services for both clients and servers. It provides support for string represenntation management of object references and the initialisation of requests to be sent to servers.

3.4 Orbix, ANSA and C++

Orbix is a C++ implementation of CORBA, and relates most closely to C++ terminology. A full discussion of the relationship between C++ and ANSA is given in *Mapping ANSA Concepts to C++* [MAC93]. The current section identifies the differences in terminology between Orbix and C++.

Components of an Orbix program are called objects, and interfaces to remote objects are specified using IDL. An IDL interface consists of operation and attribute (property) specifications. These interfaces are translated by the Orbix IDL compiler to produce a C++ class corresponding to each IDL interface. In Orbix terminology, such C++ classes are called IDL C++ classes. The IDL C++ class lists the functions that clients of the interface can use, and these functions must be defined in C++ by the implementor of the interface.

3.5 Orbix Concepts

3.5.1 Invocations

Objects in Orbix may be any size. The only requirement for an object to be remotely accessible is that it has an interface described by the CORBA IDL. The only functions that can be invoked on a remote object are those described by the IDL interface to that object. Orbix also provides complete location transparency of objects, and support for both single and multiple inheritance of IDL interfaces.

3.5.2 Interfaces

When a client makes an invocation on an IDL interface, the invocation is sent to a local representative, or proxy, for that interface. This proxy is responsible for contacting the actual object. The proxy is an instance of a C++ class and presents the client with an identical interface to the IDL interface. The code for a proxy is generated automatically.

An interface is a description of a service, and says nothing about the implementation of that service. An interface can have more than one implementation, and a single implementation can have multiple interfaces. An interface is related to a specific implementation using a *TIE*. A TIE, which, like the proxy, is generated automatically, is responsible for decoding the incoming request and sending it to the correct implementation.

3.5.3 Binding and Exceptions

To use a remote object, the client “binds” to the server. This binding process is not described in CORBA, but is an essential part of remote access, and a flexible binding interface is provided by Orbix. The following example shows how a client binds to a server:

```
Bank *bundesBank = Bank::_bind(hostname, IT_X);
```

_bind is a static member function defined for each IDL interface. It takes as an optional parameter the name of the host where the server can be found.

The IT_X parameter is the CORBA exception object which is used to indicate if an exception occurred in carrying out the access (which may be remote). The following example demonstrates a full interaction:

```
char *hostNameList = IT_lookup(“Bank”);
Bank *bundesBank = Bank::_bind(hostname, IT_X);
TRY {
     Account *ireland;
     ireland = bundesBank->newAccount(“Punt”, IT_X);
     ireland -> makeLodgement(100000.00, IT_X);
} CATCHALL {
     cerr << “couldn’t open account or make lodge.” << endl;
     exit(1);
} ENDTRY
```

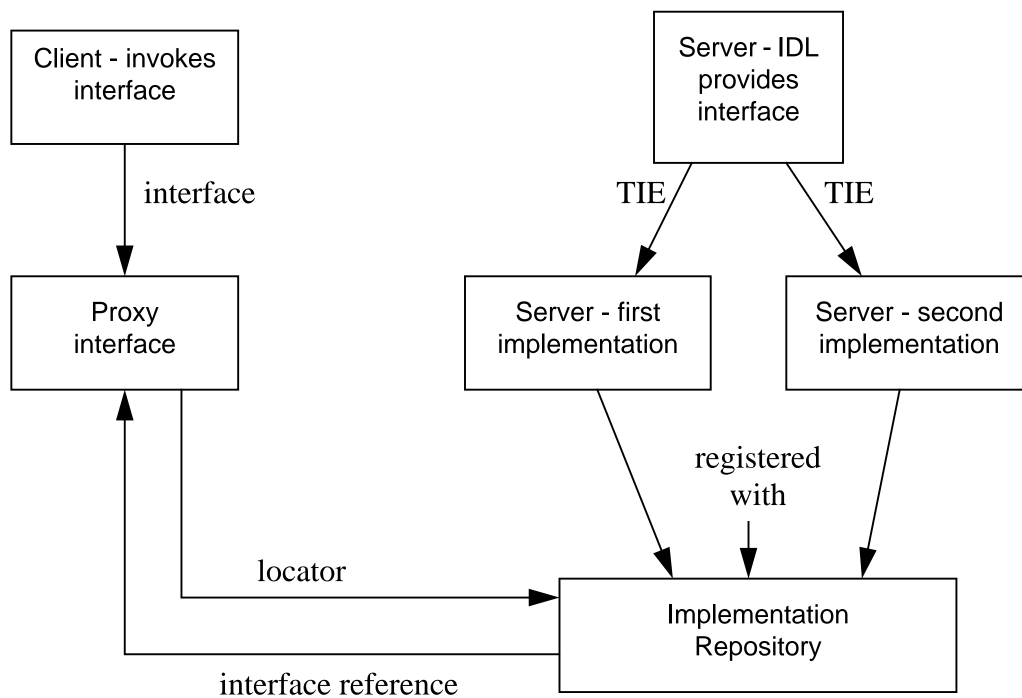
The full CORBA exception model is supported, and a `CATCHALL` macro is provided for catching and decoding user defined exceptions.

3.5.4 Orbix Daemon and the Implementation Repository

When a request arrives at a node the server handling that request must be activated. This is carried out by the Orbix Daemon (orbixd), which is a “super-server” and listens for incoming requests. When a request arrives, orbixd examines it and determines for which server it is bound. Orbixd then interrogates the Implementation Repository. In certain modes of operation (shared activation mode, see section 3.6.4), if the required server is already launched and running when a function invocation arrives for one of its objects, then Orbix will route the invocation to the running server.

The Implementation Repository is a simple database which associates interface names (servers) with their executable images. When a request arrives the orbixd checks if the Implementation Repository has an entry for the interface, and if so, activates the executable image and forwards the request to that image.

Figure 3.2: Binding in Orbix



3.5.5 Interface Repository

The Interface Repository is the component of Orbix which provides persistent storage of module and interface definitions. Given a pointer to an object, the object's type and all information about that type can be determined at runtime by calling functions defined by the Interface Repository. The Interface Repository in Orbix is not yet implemented.

3.6 Programming in Orbix

3.6.1 IDL

A basic Orbix IDL file representing a simple banking application might be as follows:

```
// IDL
// in e.g. "bankService.idl"

interface account { // bank accounts.
     readonly attribute float balance;
     void Credit(in float Amount);
     void Debit(in float Amount);
}

interface bank { // a factory for bank accounts
     exception reject { string reason; };
     // make a new account to be associated with the person or
     // company whose name is given. Reasons for rejecting the
     // request are indicated by the exception
     account newAccount (in string name) raises (reject);

     // delete an account
     void deleteAccount (in account a);
};
```

The `deleteAccount` operation has no explicit, programmer defined, exception associated with it. An operation request can still fail, however, by raising one of the standard system exceptions, one of which indicates that there is no such object. An attempt to request `deleteAccount` for a non-existent account would cause this standard exception to be raised.

When this IDL source file is passed through the IDL compiler, three files are generated:

- bankService.idl.h - a common header file used by both clients and servers for the banking service
- bankService.client.cc - an implementation file containing the proxy support for the banking service
- bankService.server.cc - an implementation file for servers of the banking service

The contents of these files are not of direct relevance to the application programmer. They effectively form the "stubs" by which client and server communicate.

3.6.2 Binding and invocation

The following example demonstrates possible use of the banking service by a client wishing to open an account and then perform operations on that account:

```
// C++
#include "bankService.idl.h"
#include <stream.h>

main() {
     // Bind to *any* bank service
     bank *b = bank::_bind();
     // Obtain a new bank account
     account *a = b->newAccount("John");
     a->Credit (56.90);
     count << "Current balance is " << a->balance();
     // etc . . . .
```

The static member function `bank::_bind()` is used to ask Orbix to try to find any server currently offering the `bank` interface. Orbix will search for a `bank` object, activating one if necessary, and will raise an exception if none can be found. `_bind()` is not always required before communicating with a particular object; if a call to another object returns a reference to it then this will result in the creation of a proxy for it and no call to `_bind()` is required.

3.6.3 Implementation by a server

There are two methods for relating implementation classes to their IDL C++ classes: the BOAImpl approach and the TIE approach.

3.6.3.1 The BOAImpl approach

Orbix generates a C++ class for each IDL interface using the name of the interface appended with the letters BOAImpl (e.g. the class `accountBOAImpl` is generated from the IDL interface `account`). The implementor's class should then inherit from the BOAImpl-class. A new header file, e.g. `bankServer.h`, creates classes `banker` and `acc`, which redefine each of the functions inherited from their respective BOAImpl classes. In addition they may add constructors, destructors, functions and member variables. It is this file which is included in an implementation of the server.

3.6.3.2 The TIE approach

The `DEF_TIE` macro indicates that a particular class implements a specific IDL interface. The macro takes two parameters: the name of the IDL C++ class, and the name of the C++ class which implements this interface:

```
//C++
// Indicate that acc implements account
class acc;
DEF_TIE(account,acc)
// we now have a class TIE(account,acc)

// Indicate that banker implements bank
class banker;
DEF_TIE(bank, banker)
// we now have a class TIE(bank, banker)
```

This method provides total separation of the class hierarchies for the IDL C++ classes and for the C++ classes used to implement the IDL interfaces. The `DEF_TIE` macro builds the associations between the two classes.

Consider an IDL operation which returns a reference to an `account` object. In the IDL C++ class, this is translated into a function returning an `account*` pointer. Using the `TIE` approach, the actual object to which a pointer is to be returned would be of type acc - which is not a derived class of account. To overcome this problem the server should create an object of type `TIE(Account,acc)`. This `TIE` object references the `acc` object, and a pointer to the `TIE` object should be returned by the function.

The easiest way to code the server is to have it create the `TIE` object when the `acc` object is created, and to have it always use the pointer to the `TIE` object. All invocations on the `TIE(account,acc)` object are automatically passed by it to the `acc` object. The alternative is for the server to directly access the `acc` object, and for it to create a `TIE` object only when it needs to return a pointer compatible with account*.

3.6.3.3 *Server implementation code*

The header file (e.g. bankServer.h) therefore provides a link between implementation classes and their IDL C++ classes, using either the BOAImpl or TIE approach. In either case it defines the implementation classes. A source file (e.g. bankServer.cc) includes the header file and provides the actual implementation code for the implementation classes, providing code for each of the operations specified in the header file.

The third file is the mainline for the server application, and constructs, in this case, a new `bank` object:

```
// C++
#include "bankService.idl.h"
#include <stream.h>

main() {
     bank *b;                          // IDL C++ class

     // create a new banker object
     b = new banker;
     // although not strinctly necessary in this case, we will
     // name the object:
     b->_marker("College Green");

     // OR using the TIE approach: create a bank TIE and
     // explicitly give it a name:
     b = new TIE(bank, banker)
          (new banker(), "College Green");

     // wait for incoming requests
     CORBA::Orbix.impl_is_ready();

     // resume here when Orbix shuts us down
     cout << "bank shutting down . . ."

}
```

The function call CORBA::Orbix.impl_is_ready() indicates that the server has completed its initialisation and is ready to receive operation requests on its objects.

3.6.4 Registration, binding and server activation

Registration of a server is achieved by the `putit` command:

```
% putit bank /usr/users/fred/banker
```

The file `/usr/users/fred/banker` is then registered as the implementation code of the server called bank at the current host. The `putit` command does not execute the file: this can be done directly from the shell, or the server will be automatically launched by Orbix in response to an incoming operation invocation.

Any command line parameters should be appended after the file specification. These are then given to the server *every* time it is run by Orbix.

In addition `putit has` several optional switches:

- selection of host

- selection of communications protocols
- application to be launched in own xterm
- specify a marker to use unshared activation mode
- use per-method call activation mode

Normal use is *shared activation mode*, where all of the objects managed by the same server on a given machine are contained in the same process. With *unshared activation mode*, individual objects of a server are registered with the Implementation Repository, and a separate process will be created per active registered object. With *per-method call activation mode*, individual operation names are registered. A process is created to handle each individual operation call, and the process is destroyed once the operation has completed.

In a similar manner a client, when binding to a server, can specify the name given with the marker for unshared activation mode, and also the internet host name of the node on which to find an object:

```
_bind(char * name="my_object", char* host="myhost",
          CORBA::Environment *env =
               CORBA::default_environment)
```

Orbix also provides a locator facility where the locations of servers can be recorded and then looked up by clients. If the host name parameter is not given to `_bind()` then Orbix itself will use the locator. The *lookup()* operation on the locator checks to see if the server has been registered on the local host before checking other locations.

3.6.5 Exception Handling

The following is an example of a client with exception handling:

```
// C++
#include "bankService.idl.h"
#include <stream.h>

main()
     bank *p;
     account *a;
     float f;

     TRY {
          // Bind to *any* bank service
          p = bank::_bind();

          // Obtain a new account
          a = p->newAccount ("Joe", IT_X);
     }
     NONE {
          TRY {
               a->Credit( 56.90 IT_X);
               f = a->balance (IT_X);// get balance
          }
          CATCHANY {
               count << "Error on crediting account() "
                    << "or balance()\n"
                    << IT_X << endl;
```

```
                exit(1);
            }
            ENDTRY
            count << "Current balance is " << f << endl;
        }
        CATCHANY {
            count << "Error on newAccount() (or bind)"
                << IT_X << endl;
            exit(1);
        }
        ENDTRY
    }
```

A number of requests can use the same `IT_X` variable. Each attempt at a request immediately aborts if the value of the exception pointer is not nil, so any failure will cause subsequent requests not to be attempted, until the exception pointer is reset.

3.6.6 Inheritance

The Orbix IDL allows a new interface to be refined by extending the functionality provided by an earlier one. This is similar to the way in which a C++ class can be derived from its base class. Both IDL and C++ support multiple inheritance, allowing an interface to have several immediate base interfaces.

3.6.7 Filtering

Orbix allows a programmer to specify that additional code should be executed before or after the normal code of an operation [IONA 93a]. There are two forms of such filtering: per-process, and per-object. Per-process filters apply to all of the objects in an address space; per-object filters apply to individual objects.

In addition to executing its own code, a pre-filter is able to determine whether an invocation should continue.

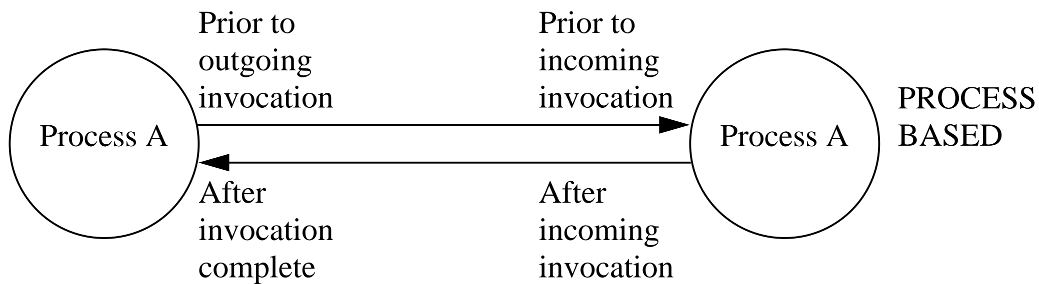
3.6.7.1 Per-process filters

Per-process filters monitor all incoming and out-going operation requests to and from an address space. Each process can have a chain of such filters, with each element of the chain performing its own actions. Each filter of the chain can monitor four points:

- pre incoming: before any incoming operation on any object in the address space
- post incoming: after any incoming operation on any object in the address space
- pre out-going: before any operation from this address space to any object in another address space
- post out-going: after any operation on any object in another address space

A pre-in filter can create a thread to handle an incoming request. This is a documented part of Orbix but at the current time is not yet correctly implemented.

Figure 3.3: Per-process filters



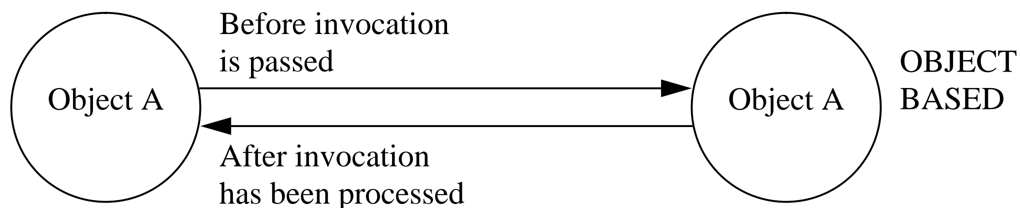
3.6.7.2 *Per-object filters*

Per-object filtering allows a filter to be associated with a particular object. The following filtering points are supported:

- per-object pre: a filter applied to operation invocations on a particular object - before they are passed to the target object
- per-object post: a filter applied to operation invocations on a particular object - after they are processed by the target object

Per-object filtering can only be used if the TIE approach has been used to associate the target object's class with its IDL C++ class.

Figure 3.4: Per-object filters



3.6.8 Smart Proxies

Proxy classes for IDL interfaces are automatically generated by the IDL compiler, and are used to support invocations on remote interfaces. They are therefore normally hidden from the programmer.

It can sometimes be of benefit to implement proxy classes manually. A typical example is where it is desirable to “cache” some information from a remote object locally at a client site. In the simple bank example, the balance of an account could be cached at a client so that requests to obtain the balance could be immediately satisfied, provided that changes to the balance cause the cached value to be refreshed.

4 An Orbix Example

4.1 The SimpleBank Example

This section describes how to program the ANSAware SimpleBank example using Orbix. It also outlines the code generated by Orbix, and traces through the execution path.

The SimpleBank example is a simple banking service to provide a (centralised) Automatic Teller Machine (ATM) management system for a bank. In towns and cities all around the country are a network of ATMs connected to a central node. There are also bank managers and their assistants who administer bank accounts, and are authorised to examine their balances and so on.

For each account the management system maintains the following information:

- account number
- personal identification number (four digits)
- current balance
- time of last change to balance (by account creation or credit/debit)

The following sections of this chapter outline the way this example can be implemented using Orbix and the TIE approach. A full description of the ANSAware method is given in [APA 93].

4.2 Interfaces

This section defines the interfaces required by the system.

An account interface is presented to ATMs to permit a user to credit money to an account, debit money from an account, and list the current details of that account.

In order to enable a user to access his account, a further interface is required, whereby the user provides a personal identification number (PIN) in addition to the account number. This is the SBank interface.

Finally an SBankMgmt interface is provided to managers of the bank. This interface contains operations to create a new account (with opening balance), destroy an account, interrogate a single account, list all accounts with their details, and shut down the service.

The ANSAware example of the management service maintains a master copy of the set of accounts in a disc file and logs all changes as they occur. The Orbix version has been simplified and this feature has been omitted.

The idl file for this example, SimpleBank.idl, is therefore as follows:

```
typedef unsigned long AccountNumber;
typedef unsigned long PersonalIdentificationNumber;

exception NoSuchAccount();
exception InvalidPin();
exception InsufficientFunds();
exception ResourcesExhausted();
exception StaleAccountReference();

interface Account
{
     struct AccountRecord {
          string owner;
          float balance;
          string lastaccess;
     };
     void Credit( in float Amount )
                              raises( StaleAccountReference );
     void Debit( in float Amount ) raises( InsufficientFunds,
                              StaleAccountReference );
     void List( out AccountRecord List_R1 ) raises(
                              StaleAccountReference );
};

interface SBank
{
     Account Access ( in AccountNumber acct,
                        in PersonalIdentificationNumber pin )
                        raises (NoSuchAccount, InvalidPin,
                                 ResourcesExhausted );
};


interface SBankMgmt
{
     struct CreateRecord {
          AccountNumber acct;
          PersonalIdentifcationNumber pin;
     };
     struct FullRecord {
          AccountNumber acct;
          PersonalIdentificationNumber pin;
          string owner;
          float balance;
          string lastaccess;
     };
     typedef sequence<FullRecord> ListAllResult;
     void Create( in string owner, in float balance,
                  out CreateRecord Create_R1)
                  raises ( ResourcesExhausted );
     void Destroy ( in AccountNumber acct )
                  raises ( NoSuchAccount );
     void ListOne ( in AccountNumber acct,
                  out FullRecord ListOne_R1 )
                  raises ( NoSuchAccount );
     void ListAll ( out ListAllResult ListAll_R1 );
};
```

4.3 Orbix server

The orbix server, in file SBankServer.cc, creates three objects.

Firstly, it creates an `AccountList` object, to be shared by the SBankMgmt and SBank objects, which will hold the list of accounts:

```
AccountList* myList = new AccountList;
```

The SBankMgmt and SBank objects are then created:

```
SBankMgmt* mySBankMgmt = new TIE(SBankMgmt, SBankMgmt_i)
                            (new SBankMgmt_i(myList));
SBank* myBank = new TIE(SBank, SBank_i) (new SBank_i(myList));
```

The `SBankMgmt* mySBankMgmt` is initialised to point to a TIE object which points in turn to the new `SBankMgmt_i` object. The `TIE(SBankMgmt, SBankMgmt_i)` construct is a CPP macro call that expands to the name of a C++ class which represents the relationship between the `SBankMgmt` and `SBankMgmt_i` classes. This class is defined by the `DEF_TIE(SbankMgmt, SBankMgmt_i)` macro call, and it has a constructor which takes a pointer to a `SBankMgmt_i` object as a parameter. Here, this pointer is returned by `new`. The `SBank* myBank` is similarly initialised.

The server then indicates to Orbix that the server's initialisation is complete:

```
Orbix.impl_is_ready("SBank", IT_X);
```

The executable file generated from this code must be registered with the Implementation Repository using the `putit` command:

```
putit SBank /path/SBankServer
```

This specifies the server's name as SBank, but does not specify any host name on which to execute the `putit` command. By default, the command is executed on the local machine.

Since neither `-marker` nor `-method` were specified, shared activation mode will be used, with all of the objects managed by the server contained in the same process.

Similarly, the communications protocol has not been specified. The default is `XDR/TCP`, i.e. XDR encoding above TCP.

4.4 Clients

The two client programs, one for the bank manager and the other for the ATM, are similar to the ANSAware client programs `manager.dpl` and `teller.dpl`, in so far as the general operation and error checking is concerned. The differences are in the binding and the operation invocations.

4.4.1 The Bank Manager client

The bank manager client attempts to bind to the SBankMgmt object in the SBank server:

```
SBankMgmt* BankMgmt;
TRY {
     BankMgmt = SBankMgmt::_bind(":SBank", nil, IT_X);
} CATCHANY {
     cerr << "Bind to object failed" << endl;
     cerr << "Unexpected exception " << IT_X << endl;
     exit(1);
} ENDTRY
```

Here the `SBankMgmt* BankMgmt` returned from the `_bind` call will be a pointer to the `SBankMgmt` interface, specified as `SBankMgmt::_bind()`. This on its own would bind to any `SBankMgmt` service, but in this case the named server "`SBank`" has been specified. The host parameter, which specifies the internet host name of a node on which to find the object, has been left null, so Orbix will attempt to find the object's server.

The individual commands which may be given by the bank manager are implemented as invocations of operations provided by the `SBankMgmt` interface. For example:

```
TRY {
     BankMgmt->Destroy(accno, IT_X)
} CATCHANY {
     cerr <<"Destroy("<<accno<<") failed, reason: " <<IT_X<<endl;
     exit(1);
} ENDTRY
```

Here the Destroy operation on the `BankMgmt` interface is invoked, giving the account number as an argument, and checking for errors.

4.4.2 The ATM client

The teller client attempts to bind to the SBank object in the SBank server:

```
SBank* Bank;
TRY {
     Bank = SBank::_bind(":SBank", nil, IT_X);
} CATCHANY {
     cerr << "Bind to object failed" << endl;
     cerr << "Unexpected exception " << IT_X << endl;
     exit(1);
} ENDTRY
```

Here the same server is specified as in the previous example, but the `SBank` interface is chosen rather than the `SBankMgmt` interface.

The teller client next attempts to access a particular account, using the `Access` operation on the `SBank` interface:

```
Account* Acc;
TRY {
     Acc = Bank->Access(accno, pin, IT_X);
} CATCHANY {
     cerr << "Access(" << argv[1] << ", " << argv[2] <<
                  ") failed, reason: " << IT_X << endl;
     exit(1);
} ENDTRY
```

Having successfully obtained a pointer to the required account interface, the teller client can carry out operations on that interface. For example:

```
(void) sscanf(argv[4], "%f", &value);
TRY {
     Acc->Debit(value, IT_X);
} CATCHANY {
     cerr << "Debit(" << form("%.2f", value) <<
               ") failed, reason: " << IT_X << endl;
} ENDTRY
```

4.5 Implementation

An implementation C++ and header file exist for each of the four classes: SBankMgmt, SBank, Account and AccountList. To identify the first three as implementation sources, the file names have "_i" appended to them, e.g. SBank_i.cc. As the files for each of the classes are similar in nature, only that of SBank will be handled here.

The header file is as follows:

```
#include "SimbleBank.idl.h"
#include "AccountList.h"

class SBank_i {
private:
     AccountNumber highest_account_number;
     AccountList* my_list;
public:
     SBank_i(AccountList* new_list);
     virtual ~SBank_i();
     virtual Account* Access(AccountNumber acct,
          PersonalIdentificationNumber pin, Environment& env);
};
DEF_TIE(Sbank, SBank_i)
```

The first include statement includes the definition of the IDL C++ class generated from the `SBank` interface by the IDL compiler.

The three public functions consist of the creator and destructor for the class, and the `Access` operation.

Finally the link between the IDL and its implementation is made with the `DEF_TIE` statement.

The C++ file includes code for the creator and destructor in addition to that required to implement the `Access` function:

```
// ctor
SBank_i::SBank_i(AccountList* new_list) {
     my_list = new_list;
}
// dtor
SBank_i::~SBank_i() {}
```

4.6 Summary of source files

This section summarises the source files required to generate the Simple Bank example, and identifies the purpose of each of them.

SimpleBank.idl	IDL for all of the interfaces
SBankServer.cc	server for SBank and SBankMgmt
SBankTeller.cc	ATM client
SBankManager.cc	bank manager client
AccountList.cc	implementation of the operations on the “AccountList” list of accounts, e.g. add, find, remove
AccountList.h	header file defining the AccountList class
Account_i.cc	implementation of the account interface with operations Credit, Debit and List
Account_i.h	header file defining the Account_i class
SBankMgmt_i.cc	implementation of the SBankMgmt interface with operations Create, Destroy, ListOne and ListAll
SBankMgmt_i.h	header file defining the SBankMgmt_i class
SBank_i.cc	implementation of the SBank interface with the Access operation
SBank_i.h	header file defining the SBank_i class

4.7 Execution trace

First the bank server is registered using the `putit` command:

```
putit SBank /path/SBankServer
```

When the server is executed as a result of an incoming invocation from a client, it creates a new TIE (of class `TIE(SBank, SBank_i)`) to an object of class `SBank_i`, and a new TIE (of class `TIE(SBankMgmt, SBankMgmt_i)`) to an object of class `SBankMgmt_i`, and waits on `CORBA::Orbix.impl_is_ready()`. For simplicity we shall deal only with the SBank object here, and exception handling has been omitted:

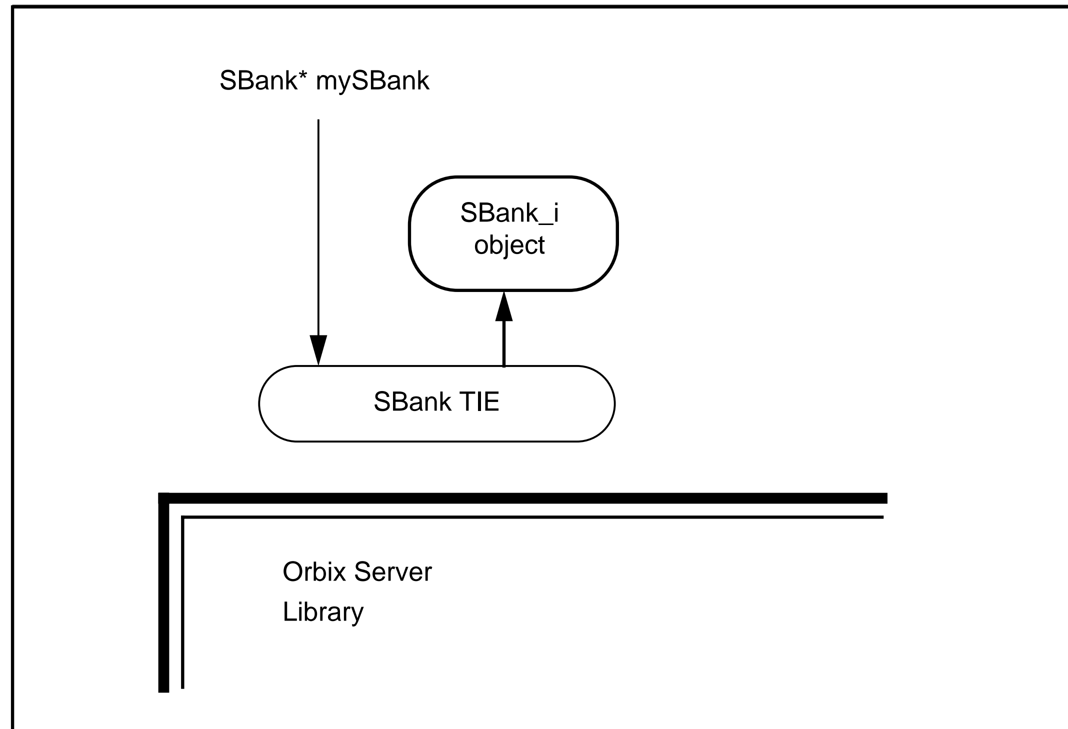
```
// C++
main() {
      AccountList* myList = new AccountList;
      SBank* mySBank = new TIE(Sbank, SBank_i)
                               (new SBank_i(myList));
      CORBA::Orbix.impl_is_ready();
```

The state of a server is shown in figure 4.1.

The server is now quiescent, waiting for incoming requests. If `impl_is_ready()` times out, the server will terminate.

Consider the ATM client: it first binds to the bank service which has been registered as SBank:

Figure 4.1: SBank server




```
// C++
main() {
      SBank* Bank;
      Bank = SBank::_bind(":SBank");
```

Since the server name ":SBank" has been given, Orbix will choose the server with this name on any machine. No object name has been given, so Orbix will choose any `SBank` object within the chosen server. In this case there is only one.

We assume that the `SBank::_bind(":SBank")` call will bind to our newly created `TIE(SBank, SBank_i)` object. That is, the result of the binding is an automatically generated proxy object, which acts as a "stand-in" for the remote `SBank_i` object in the server. The pointer `Bank` within the client is now a remote pointer, as shown in figure 4.2.

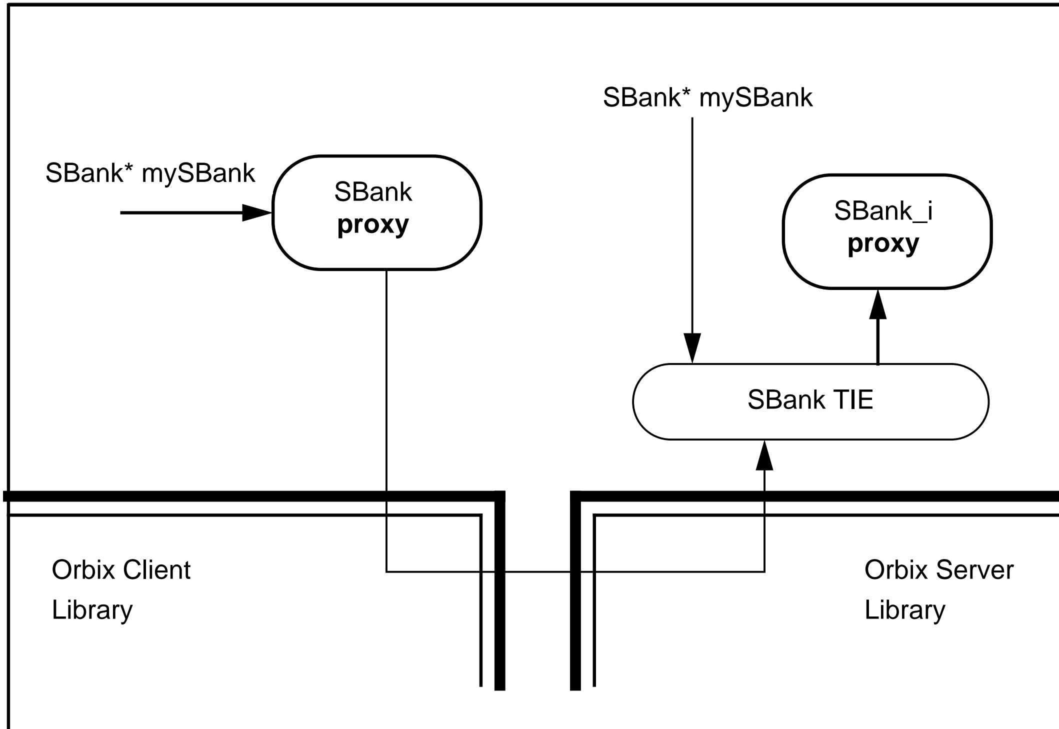
The programmer is not normally aware of the proxy object: it is managed automatically by Orbix. Also, although the client programmer need not be aware of the `TIE` object, all remote operation invocations, (inter-machine, or inter-context within a machine) on the `SBank_i` object will go via it.

The ATM client now proceeds by asking the bank to access an account, and to credit a sum to that account:

```
// C++
Account* Acc;
Acc = Bank->Access(accno, pin);
Acc->Credit(value);
```

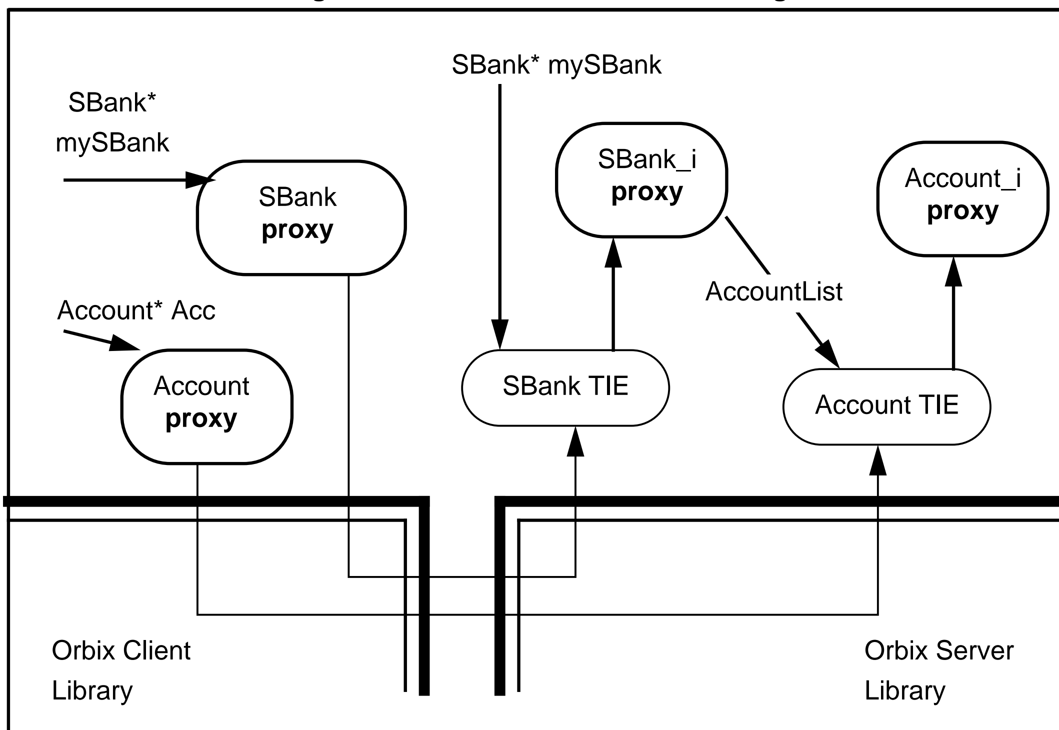
When the `Bank->Access` call is made, Orbix will launch the bank server (if it is not already running) and wait until the server calls `impl_is_ready()`, as explained earlier. Subsequently, within the server, `SBank_i::Access` is called (via the TIE), which looks up the `Account` object and associated `TIE` object in

Figure 4.2: SBank Binding



the `AccountList`. `Access` returns the `Account` reference (in C++, a pointer to the TIE) back to the client. At the client side, a new proxy for the object referenced by the pointer `Acc` is automatically created. `Acc` becomes a remote pointer.

Figure 4.3: SBank and Account Binding



4.8 Stub files

When the IDL source file SimpleBank.idl is passed through the IDL compiler, three C++ source files are generated:

- SimpleBank.idl.h — a common header file used by both clients and servers for the banking service
- SimpleBank.client.cc — an implementation file containing the proxy support for the banking service
- SimpleBank.server.cc — an implementation file for servers of the banking service

The SimpleBank.idl file, which is used to generate the files above, consists of `typedef` statements, `exception` statements and `interface` statements. The other two files consist of C++ classes generated from the interface statements.

The following sections deal primarily with the `SBank` class, since this relatively straightforward. The other classes generate more complex code, but that for `SBank` covers most of the main points.

Much use is made of the `Request` object. This is built by specifying its target object's reference, and is used to define the target object, the operation name, and the parameters of the call. Operations which can be carried out on this `Request` object include: r.setOperation ("opname"), r.getOperation(), r.invoke, r.assert, r.convertToReply etc. These are described below as they are used. The parameters are read from and written into the Reqest object as if it was an i/o stream, using the << and >> operators.

4.8.1 Typedef and Exception statements

The `typdef` statements are transferred directly from SimpleBank.idl to SimpleBank.idl.h.

Each `exception` statement is expanded to a C++ class with operations. For example the statement

```
exception NoSuchAccount
```

becomes

```
#ifndef NoSuchAccount_defined
#define NoSuchAccount_defined

static const char* ex_NoSuchAccount = "NoSuchAccount";
class NoSuchAccount : public Exception {
public:

     void encodeOp (Request &IT_r, Environment &IT_env) const;
     void decodeOp (Request &IT_r, Environment &IT_env);
     void decodeInOutOp (Request &IT_r, Environment &IT_env);
     NoSuchAccount () : Exception (ex_NoSuchAccount) {}
};
```

4.8.2 Interface statements

Four C++ classes are defined for each interface. For example the interface SBank

```
interface SBank
{
    Account Access( in AccountNumber acct,
                        in PersonalIdentificationNumber pin)
        raises (NoSuchAccount, InvalidPin, ResourcesExhausted);
};
```

generates classes `SBank`, `SBankProxyFactoryClass`, `SBank_dispatch`, and class `SBankBOAImpl`. The `SBank` class is described in the following sections.

4.8.2.1 *SimpleBank.idl.h*

In the `.IDL.h` file, the SBank classes are defined as shown below. The class `SBank` inherits from the class `Object`, and as such has the following operations in addition to those defined in the IDL:

_duplicate - when a server returns a pointer to a newly created object which it wishes to retain, it must call _duplicate() on the object to prevent the object being destroyed when Orbix calls _release() on the returned pointer. The _duplicate operation in the SBank class invokes the Object::_duplicate operation.

_bind - used to ask Orbix to find any server currently offering the specified interface. This operation invokes the New operation on the Factory class, and returns either nil, or a pointer to the new object (having used _duplicate on the object).

_isRemote - identifies whether an object is remote or local to the client. It invokes Object::_isRemote.

_deref - returns a pointer to the object if local, otherwise to a proxy object. It invokes Object::_deref.

_release - calling this operator on a pointer to an object reduces the object's reference count by one, and deletes the object if the reference count is zero. This should be used instead of delete. The implementation is not defined.

_castdown. This invokes the _narrow operation on the Factory class, giving a pointer to the current object.

_narrow - used to enable casting of references. Its implementation is not defined.

The class SBankProxyFactoryClass inherits from the class ObjectFactoryClass. It has the following operations:

New - returns a pointer to a new SBank class.

IT_castUp.

The class SBank_dispatch inherits from the PPTR class, and has the following operations.

SBank_dispatch. Three definitions are given for different numbers of arguments. The implementation is not defined.

dispatch.

IT_deref.

Finally the class SBankBOAImpl inherits from the SBank class. The SBankBOAImpl class is not used here since we are using the TIE approach.

4.8.2.2 *SimpleBank.client.cc*

In the SBank class, additional operations `SB_release`, `SBank_duplicate` and `SBank_getBase` are defined. The inherited operations `SBank::_release` and `SBank::_narrow` are defined. In the SBank_dispatch class, the `SBank_dispatch::dispatch` operation is defined.

The majority of the code relates to the Access operation on the `SBank` class. The details with respect to SBank are given below; a summary of the code is given here:

1. Check for previous error or null proxy (i.e. failed to bind). If either occurs, then return according to the type of the operation.
2. Create a Request object, and pass the arguments to it.
3. Invoke the operation.
4. If no errors, decode the results and return.
5. If user defined errors, decode them and return.
6. If other errors, return according to the type of the operation.

In the case of `SBank::Access`, the operation takes three arguments: the account number, the pin, and an optional environment variable. An initial check is made on the environment argument `IT_env`, and on the proxy address of the object (`SBank`). If the bind to `SBank` failed, then `IT_env` will hold an error, and the proxy reference will be null. In either of these cases a new (empty) Account is returned (since operation Access returns an Account*)..

```
Account* SBank:: Access(AccountNumber acct,
     PersonalIdentificationNumber pin, Environment &IT_env) {
     if (IT_env || m_isNull) return new Account;
```

If the bind had completed successfully, and no subsequent errors were recorded in the environment variable `IT_env`, then a Request object `IT_R` is created, and the account number and pin sent to it.

```
     Request IT_r (this, "Access",false,false);
     if (!IT_r.isException ()) {
          IT_r << acct;
          IT_r << pin;
     }
```

Once the parameters are inserted, the request can be sent, using `IT_r.invoke`. If no exception has been raised, then the pointer to the account object returned is unmarshalled (using `Factory.unMarshal`), duplicated and returned.

```
        IT_r.invoke (Flags(0),IT_env);
        if (!IT_env) {

             Account* IT_result;
             IT_r.assert ("");
             char * IT_l0;
             IT_r.decodeStringOp (IT_l0);
             IT_result = (Account*) Factory.unMarshal
                            (IT_l0,Account_IMPL, IT_env);
             delete[] IT_l0;
             if (IT_result)
             IT_result->_duplicate ();
             return IT_result;
        }
```

If a user exception was given (e.g. InvalidPin), then this is decoded and returned in the environment variable

```
        else if (IT_env._major == StExcep::USER_EXCEPTION) {
             char* IT_str = IT_env.exception_id ();
             if (!strcmp (IT_str, ex_NoSuchAccount)) {
                  Exception *IT_ex = new NoSuchAccount;
                  Environment IT_pe2;
                  ((NoSuchAccount*)IT_ex) ->
                            decodeOp (IT_r, IT_pe2);
                  IT_env = IT_ex;
             }

             else if (!strcmp (IT_str, ex_InvalidPin)) {
                  Exception *IT_ex = new InvalidPin;
                  Environment IT_pe2;
                  ((InvalidPin*)IT_ex) -> decodeOp (IT_r, IT_pe2);
                  IT_env = IT_ex;
             }

             else if (!strcmp (IT_str, ex_ResourcesExhausted)) {
                  Exception *IT_ex = new ResourcesExhausted;
                  Environment IT_pe2;
                  ((ResourcesExhausted*)IT_ex) ->
                                      decodeOp (IT_r, IT_pe2);
                  IT_env = IT_ex;
             }
        )
   ;
        return new Account;
   }
```

Finally if an error occurred other than a user exception, the default new Account is returned.

There are two final entries for SBank in Simplebank.client.cc.

An instance of the SBankProxyFactoryClass, `SBankProxyFactory`, is created.

A client dispatch function SBank_dispatch::dispatch is given which causes a run time exception and returns false.

4.8.2.3 *SimpleBank.Server.cc*

This file holds the `SBank_dispatch_impl` operation. A summary of the code is as follows. Details for `SBank` are provided below.

1. Obtain the name of the operation.
2. Check the name against each valid operation name until a match is found, and if no match is found, return a run time exception.
3. Set up local variables for the arguments and read the given arguments into them from the Request object.
4. Invoke the implementation of the operation.
5. If no errors have occurred, decode the result and write it into the Request object.
6. If errors have occurred, decode them into the request object, or generate a system exception for unknown errors.

The first task is to obtain the name of the invoked operation, using `IT_r.getOperation`:

```
#define SBank_dispatch_impl

boolean SBank_dispatch::dispatch (Request &IT_r,
 boolean isTarget, void *pp) {
    if (!pp)
         pp = m_obj;
    char *s = IT_r.getOperation ();
```

The operation name is compared in turn against each of the available operations (in the case of `SBank` there is only one: `Access`). If no match is found, then a run time exception is instigated. If a match is found, then the `IT_R.assert` operation is invoked to set up local arguments, and then the arguments are read from the Request object using the `>>` operator.

```
    if (!strcmp(s,"Access")) {
         Account* IT_result;
         Environment IT_env;
         if (!IT_r.assert ("Ro~Access~+acct{ul},+pin{ul},
                                   >{O~Account},N{}"))
              return true;
         unsigned long acct;
         IT_r >> acct;

         unsigned long pin;
         IT_r >> pin;
```

Assuming no errors have occurred so far, then the actual implementation of the required operation is invoked, using the local copies of the arguments retrieved from the Request object.

```
         if (!IT_env)
              IT_result = ((SBank*)(void* /*gnu!!*/)pp)->
                                   Access ( acct, pin, IT_env);
```

If no errors have been registered, then the response is converted to a suitable form with `IT_r.convertToReply`, and written out to the Request object, at which point the local result buffer is released.

```
            if (!IT_env) {
                IT_r.convertToReply ("O~Account");

                IT_r << (Object*)IT_result;
                if (IT_result) IT_result->_release ();
                }
```

If user defined exceptions have been raised, e.g. “No such account”, then these are handled. The exception is converted to an appropriate form using the `IT_r.convertToException` operation, and compared in turn with each expected exception. If a match is found, then this is encoded into the Request object. If no match is found, a system exception is generated.

```
            else {
                IT_r.convertToException (IT_env);
                if (!strcmp (IT_env.exception_id(),
                                    ex_NoSuchAccount)) {
                    Environment IT_pe2;
                    ((NoSuchAccount*)IT_env.exception_value()) ->
                                        encodeOp (IT_r, IT_pe2);
                }
        else
            if (!strcmp (IT_env.exception_id(), ex_InvalidPin)) {
                Environment IT_pe2;
                ((InvalidPin*)IT_env.exception_value()) ->
                                        encodeOp (IT_r, IT_pe2);
            }
        else
            if (!strcmp (IT_env.exception_id(),
                                        ex_ResourcesExhausted)) {
                Environment IT_pe2;
                ((ResourcesExhausted*)IT_env.exception_value())->
                                        encodeOp (IT_r, IT_pe2);
            }

        else IT_r.makeSystemException (IT_env);
        }
        return true;
    }
```

Here a run time exception is returned if the operation name was not recognised.

```
    else if (isTarget)
        IT_r.makeRuntimeException2 ();
    return false;
    }
```

5 Comparison of Orbix and ANSAware

5.1 Language Compatibility

Orbix is a C++ implementation of CORBA, while the existing ANSAware code is written in C. It is possible, however, for ANSAware to be used with C++. A C++ compiler will compile C code, although some changes may need to be made to ANSAware (e.g. a frequently used variable name *new* is a reserved word in C++).

In some cases it may be possible to build ANSAware using a C compiler, and just compile the main() module by a C++ compiler. The relies on C and C++ compilers producing compatible code.

Any C++ support modules can be called from *main*(), and replacements for any ANSAware modules can be put into a library which is linked in prior to libansa.a. This effectively means that a C++ program is able to make use of ANSAware libraries where the code is written in C. Orbix objects should therefore be able to invoke functions from the ANSAware libraries, hence Orbix objects can run over ANSAware.

There are three areas where special care must be taken with the implementation of this combination of C and C++:

- Choice of compiler, to ensure compatibility between C and C++ compilers, if this route is taken. The gnu gcc and g++ compilers are most likely to be compatible. (Or re-compile ANSAware using a C++ compiler.)
- Data interaction between C and C++, in particular where C++ classes are passed into or through C functions. Since C++ classes are not recognised by C, any data within these classes must be referenced by function call, and not directly. Where C++ classes are passed to C++ functions via C functions, these C functions must not dereference the pointers.
- Function calls between C and C++ functions must be carried out in a manner recognizable to both languages, and any arguments must be passed in a compatible manner.

5.2 Tool Compatibility

ANSAware provides an engineering nucleus for distributed programming, with various tools and support features [OVW 93]. These include:

- **STUBC** - the utility which compiles an interface specification written in (ANSAware) IDL into stub routines and header files in C for inclusion in programs which will provide and use that interface.
- **PREPC** - the preprocessor which extracts control commands from C programs and translates them into calls to the stub routines prepared by STUBC. Control commands exist for declaring interfaces, performing trading functions and calling operations in local or remote interfaces.

- **Trading** - traders give access to information about available services. Trading matches offers and requests for particular services, using service names, interface types and service properties in combination as selection criteria. In this way, the separate parts of a distributed application can find each other on demand.
- Objects are created by **Factories** in two stages. First a capsule is created, containing object templates. A capsule can then be instructed to make objects from the templates. These objects can then offer their services to the trader or to other objects.
- **Node Managers** provide a method for controlling the services available on a network. They use factories and traders to bootstrap and control both static and dynamic services.

5.2.1 STUBC and PREPC

Orbix has its own IDL which effectively replaces both STUBC and PREPC. The Orbix IDL generates code to marshall the request parameters into a single line buffer prior to transmitting the request across the network. These stubs are generated at compile time and statically linked with the client application. The stubs make use of the proxy and TIE approach. Because all invocations make use of a proxy, no preprocessing is required. The Orbix IDL thus replaces both STUBC and PREPC.

5.2.2 The Trader

The Orbix Implementation Repository is able to locate available services (objects) both by object name and by use of a marker, a name given to a specific instance of an object, However it provides no further constraints on the choice of specific object. The Trader facilities are therefore only partially achieved. This lack of ability to match service properties offered by the trader is a particular loss. This ability would greatly facilitate the matching of real-time requirements to services offering particular QoS guarantees.

5.2.2.1 Trader Tools comparison

A list of the ANSAware Trader facilities and the extent to which they are available in Orbix is given below:

- The *Register* operation causes a new service offer to be placed in the trader’s database. This is invoked from a server when the server is activated (from command line, factory etc.). In Orbix registration is carried out by means of the *putit* command from the command line.
- The *Lookup* operation causes the trader to search its database for a matching offer according to a specified policy. Among other arguments, a set of matching constraints may be specified for the offer’s property values. Orbix provides a *_bind* function whereby the client specifes the generic name (type) of the object, and may optionally specify a name (marker) denoting a specific instance of the server. Thus some specialisation is provided but without the matching of constraints provided by ANSAware.
- The *Delete* operation causes the trader to delete all offers in its database having the specified interface reference and matching constraints. Orbix provides a *rmit* command entered from the command line to remove registration of a server.

- *TrCtxt*, *Ctxtcl* - an interface to manipulate the context name space of the trader, and a client to exercise these operations. The nearest equivalent in Orbix is the hierarchical structuring of server names, in a similar manner to Unix directories. The command *mkdirit* creates a new registration directory, and then *putit*, *rmit* etc. are used in the normal way but specifying the full "path" of the server. Similarly *rmdirit* is used to remove a registration directory. The directory must first be empty.
- *TrFed, TrType*, *typecl*- these concepts are not supported in Orbix.
- *trclient* - provides user level access to, and manipulation of, the Trader's database. In addition to the *putit* and *rmit* commands, Orbix provides *lsit* and *catit* commands. *lsit* lists either a specific, or all, registrations. *catit* provides full information about a given registration. Each of these is executed from the command line.

5.2.2.2 *Constraint matching facilities*

The primary deficiency of Orbix in comparison with the ANSAware Trader is therefore the lack of constraint matching facilities. It would of course be possible to create a "Trader database" within Orbix, whereby servers offering specific "features" could be identified. On creation of a server it could identify itself to the "Trader" along with any guaranteed QoS or other information, and its Orbix marker. A client would request from the "Trader" the marker of a server offering specific QoS etc. The client could then bind to a server giving the marker returned.

5.2.2.3 *Registration*

The other major difference between the ANSAware Trader and the facilities provided by Orbix relates to identifying what is running at any particular time. The Implementation Repository contains the necessary information to relate server (and marker) names to the shell commands, but it does not contain the information on what is currently running. Only the Orbix Daemon knows what is running, and this information is not accessible to the application programmer.

There is also a fundamental difference between the *register* operation in ANSAware and the *register* operation in Orbix. In ANSAware, unless you are using proxy offers, *register* means "here is a running server that supports a given interface", while in Orbix it means "here is a binary file which when executed will support a given interface" (or "will have a given server name"). This affects both the Register and Delete operations.

This difference in registration between the ANSAware Trader and the Orbix Implemenation Repository is both an advantage and a disadvantage. On the one hand the Implementation Repository is aware of all registered services, not simply those currently running. On the other hand it is not possible to identify which services are currently running.

5.2.2.4 *The Interface Repository*

A further difference is the inclusion in Orbix of CORBA's Interface Repository, which, when implemented, will provide persistent storage of module and interface definitions. Given a pointer to an object, the object's type and all information about that type can be determined at runtime by calling functions defined by the Interface Repository. Although Orbix itself only provides minimal type checking, the future availability of the Interface Repository will provide the information for type checking code.

5.2.3 The Factory

In ANSAware, dynamic instantiation of a service is accomplished by instantiating a capsule which has an object within it that supports the desired interface type. The process is thought of from the point-of-view of creating an *object* using a particular template.

Orbix supports two levels of “factory”:

1. The first relates to the three different mechanisms for launching servers, giving the programmer control over how servers are implemented as processes by the underlying operating system (see section 3.6.4). Depending on the mechanism used, new processes may be created for individual objects or operations.
2. The second makes use of the C++ *new* operator. In C the *malloc* function knows nothing about the type of the variable being allocated, but simply takes *size* as a parameter. In contrast, the *new* operator knows the class of the object to be allocated, and automatically calls the class’s constructor to initialise the memory it allocates. The simple bank example in the Orbix Programmer’s Guide [IONA 93] uses the *new* operator to generate new bank accounts. The Interface Repository holds interface types; when a new account is created in this manner it is simply a new instance of a known type, and is therefore not registered with the Interface Repository.

These two levels roughly correspond in ANSAware to the creation of a new capsule, and the creation of a new object within a capsule.

The implementation of factory facilities is therefore rather different in Orbix to that of ANSAware, but is probably sufficient for most uses.

5.2.4 The Nodemgr

The ANSAware Node Manager provides an interface for the creation, simple monitoring and termination of services. It uses the factory for the actual mechanics of creation and destruction; the service it provides beyond the simple functionality of the factory is a database for describing and managing services. Each service description is identified by an alias: each alias can be given various attributes that specify how it is to be managed. An alias may be defined to the Node Manager as permanent, which indicates that the Node Manager will automatically restart the appropriate service if it terminates. In addition, dynamic service creation is provided by using the proxy export facility given by the trader’s federation facilities.

Orbix does not have a Node Manager. Some of the features are supported by the three mechanisms for launching servers, as in the case of the factory (see section 5.2.3).

The only other feature supported by Orbix that relates to the use of different nodes (hosts), is the locator facility, whereby the locations of servers can be recorded and then looked up by clients. The locator can be used explicitly by application programs, and is used by Orbix when the host name parameter is not specified to *_bind()*. This does not provide any of the facilities of the ANSAware Node Manager, but is mentioned here since it is a host-specific facility provided within Orbix.

5.3 Communicating between Orbix and ANSAware

Orbix object references are pointers to Orbix IDL C++ classes. As such, they cannot be dereferenced and used by C functions. However as pointers, they can be passed from a C++ function to a C function, then back to another C++ function for dereferencing and use. If the constraint is made that all clients and servers will be written in Orbix, and hence C++, then for the purposes of passing a reference as an address of an object, there is no problem. Problems may arise, however, in so far as the ANSAware nucleus uses information stored in the interface reference. Obviously Orbix object references will not hold the same information (although there is some similarity).

A solution to this problem would be to create an Orbix object to hold any state required by the ANSAware nucleus which is normally part of the interface reference, but which is not part of the Orbix object, and all functions needed to access this state. This class can then be inherited by all Orbix objects, such that all Orbix objects thus hold the necessary information and access functions.

The mandatory component of an ANSAware interface reference comprises the *communication* state maintained by the underlying run-time system [IREF 91]. Since the communications is now to be entirely carried out by Orbix, this part of the interface reference is no longer required. This precludes many of the features of ANSAware, for example, migration, and support for groups.

The optional aspects of the interface reference are:

- *interface* state, stored and retrieved by the nucleus on behalf of the interface, but created, modified and read by the interface's code component
- *global* state, managed entirely by the interface's code component

Both of these should be handled directly by Orbix.

It would therefore appear that there is no need for an "interface reference" in any form, and that the Orbix object reference is sufficient on its own. Should it become apparent during implementation that this is not entirely the case, then a solution of the type discussed could be implemented.

Should the Orbix work be taken further, it would be necessary to identify what modifications to the ANSAware nucleus would be necessary in order to accommodate the use of C++. In addition many areas of the ANSAware nucleus may become redundant.

5.3.1 Underlying communications

The underlying communications is handled entirely by Orbix, and is not accessable by the application program. This provides difficulties should the application programmer which to place any constraints on the network. This problem has not yet been resolved.

5.3.2 Support for Groups

It may be possible to provide support for groups making use of multi-cast requests. Multi-cast requests are supported, using the `send_multiple_requests`() static function defined in class `CORBA`:

```
// C++ - in class CORBA
static ORBStatus send_multiple_requests (
     Request        *reqs[], // array of ptrs to requests
     long           count,   // number of requests
     Flags          invoke_flags = 0,
     Environment    &env = default_environment
);
```

Each of the requests in the `reqs` array will be made in parallel. It behaves like `send()` in that it does not wait for the requests to complete before returning to the caller. The caller can use `get_response()` or `get_next_response()` to determine the outcome. `get_next_response()` is defined as a stataic function in class `CORBA`:

```
// C++ - in class CORBA
static ORBStatus get_next_response (
     Request *&req,
     const Flags &response_flags = 0,
     Environment &env = default_environment
);
```

`get_next_response()` can be called many times to determine the outcomes of the individual requests specified in a `send_multiple_requests()` call: `req` is an inout parameter which is changed to point to the `Request` whose completion is being reported. Alternatively, `get_response()` can be called on the individual `Request` objects in the array.

5.3.3 Multiple protocols

Orbix does support multiple protocols, in the sense that at on registration (`putit`), the -c switch can be used to specify the communications protocol used by the server. The default is XDR/TCP, that is, XDR encoding above TCP, and this is currently the only protocol supported by Orbix. Other protocols should become available at a later date. In addition it will be possible for users to add their own protocols, since Iona will publish the interfaces to allow customers to add the necessary modules.

5.4 Support for Storage

The main aspects of storage are: activation, passivation, and migration. Activation has already been discussed (see section 3.6.4). Passivation will depend on the way the server has been written. The `impl_is_ready()` call returns only when a timeout occurs or an exception occurs while waiting for or processing an event. The timeout parameter indicates the number of milliseconds to wait between events: a timeout will occur if Orbix has to wait longer than the specified timeout for the next event. A timeout of zero indicates that `impl_is_ready()` should timeout and return immediately whenever there is no pending event. Since activation can also specify the host on which the server is to run, this effectively provides a mechanism by which migration can be achieved.

5.5 Orbix threads with ANSAware real time scheduling

Little information is currently available on the handling of Orbix threads, and the current implementation of Orbix does not correctly support threads. It is therefore difficult to pursue this aspect of Orbix until further information is available.

6 Implementing RIDE on Orbix

6.1 The Object-Oriented Approach

There are two possible approaches to the problem. Firstly there is the existing RIDE implementation, where the extensions are built into the nucleus. This is currently implemented on ANSAware 3.0, so would need re-working for ANSAware 4.1, in addition to changing the user interface for Orbix. The second approach is to implement RIDE on top of the ANSAware nucleus, but using an object oriented approach.

An object-oriented approach has many benefits. It will provide a basis for an object-oriented interface to the nucleus, which will subsequently become more modular. Individual modules may then be replaced with ease. The current ANSAware implementation of the nucleus comprises a large, heavily integrated system, which is difficult to compartmentalise, and it is consequently difficult to extract sections of it. A modular approach would greatly ease future development, providing a much higher level of flexibility. Any additions should, therefore, aim to provide this flexibility at an early stage. An object-oriented approach will help to achieve this.

There are two ways of implementing such an object-oriented solution:

- use Orbix IDL to generate Orbix C++ classes
- implement C++ classes directly

If the Orbix IDL approach is used, this can facilitate development, since Orbix carries out additional checking, and is able to produce template files for the server. There is no overhead, since Orbix supports local object invocations. When trying to bind to an object, Orbix first looks within the caller's address space, then for a server on the same host, then finally for servers registered from other hosts (assuming no specific host has been specified). If the object is found in the local address space, then a direct call is made, and no use is made of proxies, network protocols or sockets. The Orbix IDL approach is therefore the most appropriate, since it will deal with both local and remote objects in the optimum manner.

The following sections examine the RIDE interface described in chapter 2, and look at ways of implementing this using Orbix.

6.2 Invocations

The suggested invocation to support real-time parameters is one which has an additional optional argument, of type `rtAttrs`:

```
{results} <- IfRef$Operation(arguments) rtAttrs rt_attrs
```

These real-time attributes are specified at invocation, rather than on binding.

There are several options for implementing this using Orbix:

- pass an additional argument as part of the operation invocation
- use a “trader” prior to binding
- use a pre-filter

The first option, where the real-time attributes simply form an additional argument to the operation, implies that any computation must be done at run-time by the server, thus reducing its performance. This is not an appropriate solution for a real-time system.

Both second and third options specify the attributes at binding, rather than invocation. This implies that it may be necessary to renegotiate the binding on a subsequent invocation with different real time attributes.

The second option makes use of the “marker” name given to a particular instance of a server. Given a “trader” which identifies markers with real-time attributes, a client can look up the appropriate server instance and bind directly to that. This places an additional requirement on the client, but the server is then known to be capable of providing the necessary QoS. If the server’s marker is known in advance then no call to the trader is required (but the servers in the system must be “known”). The QoS in this case depends on the object, rather than the operation. An intelligent trader could identify a server instance appropriate to a particular operation with QoS requirements, and if per-method call activation mode is used (see section 3.6.4), a process will be created for each invocation, thus preventing invocations to other operations within the server (see also section 6.4).

The pre-filter (section 3.6.7) is executed after an operation invocation but prior to the operation itself. The pre-filter can check with the trader for the marker name of an appropriate server, and create a thread to handle the request. This can be done either by directing the operation to the object to which the client is currently bound, or by binding to the appropriate object and re-issuing the invocation. The former provides some optimisation. Care needs to be taken here to ensure that a loop does not occur, particularly when several objects may meet the requirements. This approach makes the checking transparent to the client.

This third option would appear to be preferred, since the process is transparent to the client, but unfortunately Orbix threads do not yet work correctly. The second option is logically similar, and could be used as a temporary solution for the purposes of demonstrating RIDE using Orbix.

A potential problem with the use of different instances of a server for different invocations occurs where servers need to share state. If this is the case, then a separate object would be needed to hold the state, such that the servers may each access the same state. Some care with multiple use and locking will need to be exercised here.

6.3 The Entry Interface

The initial implementation of RIDE defined operations on entries to:

- create/close an entry
- bind/unbind an interface to an entry
- allocate system tasks to an entry
- enable tasks to rendezvous with entries at run-time.

The first three of these bullets are represented by five functions in the proposed modified version of RIDE, and can be implemented directly as operations on the Orbix object *entry*. There are two possible ways of doing this:

- Generate an *entry* object which supports all five of these operations. This would have to allocate space within itself for each new entry created, and keep track of entries.
- Generate an entry object to carry out the operations of the second two bullets, and a factory object to create and delete entries.

The second approach makes it much easier to create and close entries, and simplifies the entry object itself. The factory could keep track of entries in a list if this were required.

6.3.1 The Interface Definition

In Orbix this would have the following IDL:

```
// IDL
// in e.g. "entry.idl"
interface entry { // an individual entry
     void bind ( in interface_ref iref );
     void unbind ( in interface_ref iref );
     void task_spawn ( in tasks t, in task_attr ta );
};

interface make_entry { // a factory for making entries
     entry create ( in entry_attr ea );
     void close ( in entry e );
};
```

The entry is created with a set of attributes of type *entry_attr*.

The binding to an ANSAware interface reference is replaced to a binding to an Orbix interface reference, which is represented as a C++ pointer to the corresponding IDL C++ class (*bank *b* in section 3.6.2). The implication here is that since we are passing a pointer to a C++ class, the pointer can only be used in a C++ function. But since in this Orbix implementation, both clients and servers will be written in C++, it is not anticipated that this will be a problem. The pointer may possibly be passed through the ANSAware nucleus C functions, but it will then simply be a pointer, and so long as these functions to not attempt to dereference it, its integrity should not be invalidated.

The *task_spawn* operation allocates system tasks to an entry.

An entry attributes object can be created, and its default attributes set and read:

```
// IDL
// in e.g. "entryattr.idl"
interface entryattr { // an individual entry attribute object
     void set ( in entry_attr ea );
     void get ( out entry_attr ea );
};

interface make_entryattr { // factory for making entry attributes
     entryattr create ();
     void close ( in entryattr ea );
};
```

An entry set is used to allow tasks to rendezvous with entries at run-time. The entry set is a union of one or more entries. The entry set may be created, and have entries added to it and removed from it:

```
// IDL
// in e.g. "entryset.idl"
interface entryset { // an individual entry set
     void add ( in entry e );
     void remove ( in entry e );
};

interface make_entryset { // factory for making entry sets
     entryset create( in entry_set_attr esa );
     void close ( in entryset es );
};
```

The *entry set* is created with attributes in a similar way to the *entry*:

```
// IDL
// in e.g. "entrysetattr.idl"
interface entrysetattr { // an individual entry attribute object
     void set ( in entry_set_attr esa );
     void get ( out entry_set_attr esa );
};

interface make_entrysetattr { // factory for making entry
                            // set attributes
     entrysetattr create ();
     void close ( in entrysetattr esa );
};
```

6.3.2 Usage from a Client

The entry and entry attributes factories could be used as follows:

```
// C++
#include "entry.idl.h"
#include "entryattr.idl.h"

main() {

// Bind to any make_entryattr service
make_entryattr *mea = make_entryattr::_bind();
// Create an entry attribute and set its default
entryattr *ea = mea->create_entryattr();
ea->set( <default attributes> );

// Bind to any make_entry service
make_entry *me = make_entry::_bind();
// Create a new entry
entry* e = me->create_entry( ea );

// etc. . . . .
```

Entry sets would be handled in a similar fashion.

6.3.3 The Implementation by a Server

The implementation of the server for entry and make_entry is covered here. Those for the other objects would be along similar lines.

The server implementation falls into three areas/files:

- the service header file, which defines the classes with their constructors and destructors, and defines the TIE is this approach is used
- the service implementation file, with the code which implements the service
- the mainline code, which creates the TIE, and informs Orbix when ready for incoming requests

6.4 Resource Allocation

RIDE does not provide directly for resource management, but applications that require the function can be designed on top of RIDE, making use of the fact that ANSAware supports the dynamic creation and passing of interface references. Orbix too supports the dynamic creation of a server, in that, depending on the activation mode, a new process is created from the executable file on binding to a server.

An example of a resource management requirement might be: a client requires a database operation to be performed rapidly, thus requiring total use of a disc until the operation is complete.

The existing RIDE implementation suggests a resource management interface which provides an operation of the form:

```
getIfRef :     OPERATION [ resource-requirements ]
               RETURNS   [ service-interface]
```

Prior to using the server, the client tells the server about its resource requirements using the getIfRef operation. The management interface then creates a new interface instance of the service, allocates the required resources, and returns the interface reference to the client. The client may then use the server with guaranteed resources.

In Orbix we would need a management object to bind to a server, and to invoke a management operation on that server which would attempt to allocate the required resources (tasks, entry and application level resources). If successful at obtaining the resources, the management operation would register the new instance of the server with the “trader”, with its marker, and return the Orbix interface reference to the client. If unsuccessful, the management operation would provoke an exception.

This are can be implemented this in Orbix as follows:

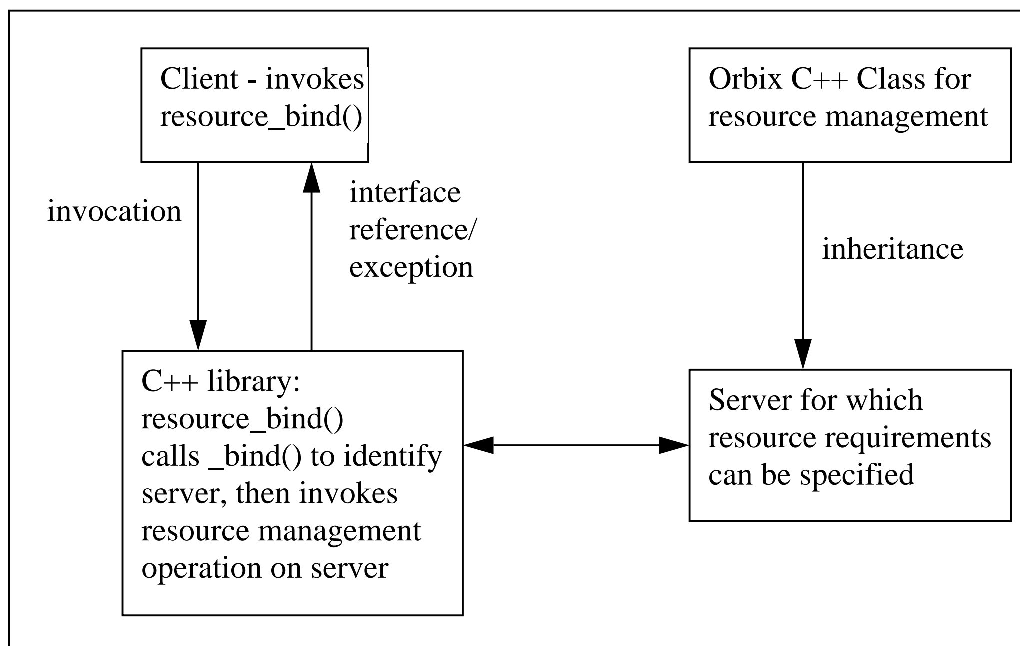
Each server for which resource requirements can be specified should inherit from an Orbix C++ class which provides a resource management interface. On binding to the server, a new instance of the server will be created (assuming unshared activate mode). Immediately following the _bind(), the resource allocation operation should be invoked to allocate the necessary resources, returning as described above. Because inheritance is used, servers can either utilise the basic resource allocation operation unchanged, or they can modify it to provide additional server-specific facilities.

An variation on this approach is to provide a resource management object, with an operation which takes as input an object type and resource requirements, and returns an interface reference. This operation would carry out the _bind() and the resource allocation operation invocation as an atomic action, and would replace the usual call to _bind(). If this were an independent Orbix object, the client would still need to bind to it, and there would be no advantage. A more suitable solution would be for such an operation to be implemented as a C++ function, incorporated into a general-purpose library and linked with the client at build time.

The components of this approach are summarised below:

1. An Orbix C++ Class contains a resource management operation.
2. Each server for which resource requirements can be specified inherits from the Orbix C++ Class generated in (1).
3. The C++ function resource_bind() invokes _bind() to identify a suitable object, and then invokes the resource management operation on that object.
4. The client invokes a resource_bind() operation, rather than the usual _bind().

Figure 6.1: Resource Allocation



An outline of the code is provided (this code is intended as a guide only):

```
// IDL
// in e.g. "rsrc_mgmt.idl"
// standard IDL file for the rsrc_mgmt interface

interface rsrc_mgmt {
    int get_resources ( in Resources res_list );
};
```

```
// C++: the file resource_mgmt.h
// ties the rsrc_mgmt class to the class that implements it
#include "rsrc_mgmt.idl.h"
// Indicate that getres implements rsrc_mgmt
class getres;
DEF_TIE (rsrc_mgmt, getres)
// We now have a class TIE(rsrc_mgmt, getres)
class getres {
public:
     virtual void get_resources (Resources res_list,
          CORBA::Environment &env = CORBA::default_environment );
};
```

```
// C++: the file resource_mgmt.C
// the getres server
#include "rsrc_mgmt.h"
// implementation of class getres;
// default constructor and destructor
getres::getres() {}
getres::~getres() {}
// implementation of get_resources
void getres::get_resources (Resources res_list,
          CORBA::Environment &env = CORBA::default_environment )
{
// do the resource allocation here
}
```

```
// IDL
// in e.g. "aServer.idl"
// a new IDL to define the derived Orbix class
// include the resource management class
#include rsrc_mgmt.idl
// derive a new class from the resource management class
interface new_rsrc_mgmt : rsrc_mgmt {
     // no new operations needed
};
```

```
// C++: the file myServer.h
// ties the derived class to the class that implements it
#include "aServer.idl.h"
#include resource_mgmt.h"
// Indicate that new_getres implements new_rsrc_mgmt
class new_getres;
DEF_TIE (new_rsrc_mgmt, new_getres)
// We now have a class TIE(new_rsrc_mgmt, new_getres)
class new_getres {
public:
     virtual void get_resources (Resources res_list,
          CORBA::Environment &env = CORBA::default_environment );
};
```

```
// C++: the file myServer.C
// the server for the derived class
#include "myServer.h"
// implementation of class new_getres;
// default constructor and destructor
new_getres::new_getres() {}
new_getres::~new_getres() {}
// implementation of get_resources
void new_getres::get_resources (Resources res_list,
         CORBA::Environment &env = CORBA::default_environment )
{
     // either do the non-standard resource allocation here, or
     // invoke the standard function:
     getres *gr = getres::_bind();
     gr->get_resources( res_list );
}
```

```
// C++: the file res_bind.h
// the C++ header file defining resource_bind()
class Get_Res
{
public:
     Get_Res ();
     ~Get_Res();
     int resource_bind( char *obj, char *marker,
                             Resources res_list );
};
```

```
// C++: the file res_bind.C
// the C++ function resource_bind()
#include "res_bind.h"
int resource_bind( char *obj, char *marker, Resources res_list )
{
     new_getres *ngr = new_getres::_bind( marker );
     ngr->get_resources( res_list );
}
```

```
// C++: an extract from the client program
{
     result = resource_bind( aServer, "this_one", res_list );
}
```

6.5 rpc Attributes

RPC calls are allowed to be associated with call attributes objects. These objects are similar to the entry attribute objects, and may be created, set or read:

```
// IDL
interface rpc_attr
{
     int rpc_attr_create {
          out rpc_attr rpca
     };
     int rpc_attr_set_anAttr {
          out rpc_attr rpca,
          in an_attr aa
     };
     an_attr rpc_attr_get_anAttr {
          in rpc_attr rpca
     };
};
```

The original RIDE method of associating a call attributes object with an RPC was to specify the attributes object as an argument to the invocation, in the same way as the real-time attributes were specified in section 6.2. A similar solution should be applied.

6.6 Parallel Protocol Stack

Channel and interface attributes can be created, set and read in a similar manner to entry and RPC attributes. It is however unclear as to what form these may take in an Orbix environment. Private channels similarly have no role, since Orbix handles all communications.

6.7 Synchronization

6.7.1 The Interface Definition

The synchronization services are provided by the timed automata interface. An instance of this interface needs to define timers, states, transitions on the states, and a set of timing guards. Named subsets of the set of states may also be defined. The interface needs to support operations for state transition, wait on transition or set, combinations of these operations, and enquiry. In addition some sort of initialisation will be needed to set up the state table.

We therefore need an interface for the timed automata interface, and a factory to generate such interfaces. This approach enables the client to access more than one timed automata specification, and is therefore more flexible then simply generating a single timed automata interface. The factory solution also enables initialisation to be carried out on creation, so no additional operation is required.

Possible IDL specifications are:

```
// IDL
// in e.g. TimedAutomata.idl

interface timed_automata { // instances of timed automata
     readonly attribute int current_state;

     void Signal ( in Transition t );
     void Await_Transition ( in Transition t, in Timeout to );
     void Await_Set ( in Set s, in Timeout to );
     void SigAwait_Transition ( in Transition t1,
                         in Transition t2, in Timeout to );
     void SigAwait_Set ( in Transition t1, in Set s,
                         in Timeout to );
     void AwaitSig_Transition ( in Transition t1,
                         in Transition t2, in Timeout to );
     void AwaitSig_Set ( in Transition t1, in Set s,
                         in Timeout to );
     };

interface ta_factory { // a factory for timed automata

     timed_automata new_timed_automata ( in string name );
     void delete_timed_automata ( in timed_automata ta );
};
```

A further Orbix class will need to be established for *timers*, and the underlying code for these will probably change depending on the host machine. The Orbix interface references for these will be passed in a list to the Initialise operation.

The list of *states* is simply a list of state names. It is unclear whether this is actually required, except as a consistency check for the other data.

Each *set* consists of a set name and a list of states. A list of sets is passed to the Initialise operation.

Each *guard* consists of a guard name and a timer expression, where a timer expression relates a state to a timeout. A list of guards is passed to the Initialise operation.

A *transition* consists of a transition name, an initial state, a timer expression, an action, and a new state. The *action* is a list of timers. Invoking a transition will cause a state to change to the new state, provided it was formally in the initial state, and the timing constraint specified by the timer expression was satisfied. In addition timers in the action list are reset to zero.

The start state is the initial state for this instance of the timed automata.

6.7.2 Usage from a Client

The following code generates a new timed automata service and makes use of it:

```
// C++
#include "TimedAutomata.idl.h"
#include <stream.h>

main() {
     // Bind to any ta_factory service
     ta_factory *fac_ta = ta_factory::_bind();

     // Obtain a new instance of a timed automata
     timed_automata *ta = fac_ta->new_timed_automata ("my_ta");

     ta->Initialise ( all the initialisation data );
     count << "Current state is " << ta->get_State();

     // etc . . . . .
```

6.8 Virtual Time

6.8.1 The Interface Definition

The virtual time functions again require a virtual time object, and a factory to create such objects.

```
// IDL
// in e.g. "vtime.idl"
interface vtime { // an individual virtual clock
     void set ( in int m, in int n, in int k );
     void freeze ( in int delay );
     void resume ();
     void catchup ( in int delay );
};

interface make_vtime { // a factory for making virtual clocks
     vtime create ( in int m, in int n, in int k );
     void close ( in vtime vt );
};
```

6.8.2 Usage from a Client

The following code demonstrates the creation of a clock and its use:

```
// C++
#include "vtime.idl.h"

main() {
     // Bind to any make_vtime service
     make_vtime *mvt = make_vtime::_bind();

     // Obtain a new instance of a virtual clock
     vtime *vt = mvt->create ( 1, 1, 0 );

     vt->freeze ( 10 );
     vt->resume ();
     vt->catchup ( 5 );

     // etc . . . . .
```

References

[APA 93]

ANSAware 4.1 Application Programming in ANSAware, **RM.102.02**. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1993.

[APM 93a]

A Performance Framework, **APM.1051.06**, APM, Francis Wai et. al., APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England. 1993

[APM 93b]

An Open Architecture for Real-Time: Engineering Aspects, **APM.1072.02**, Guangxing Li, APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England. 1993

[IONA 93]

Orbix: Programmer's Guide, Iona Technologies Ltd, Dublin, Republic of Ireland, 1993.

[IONA 93a]

Orbix Advanced Programmer's Guide, Iona Technologies Ltd, Dublin, Republic of Ireland, 1993.

[IREF 91]

ANSAware 4.0 Interface References, **RC.268.01**, Cosmos Nicolaou, APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England. 1991

[Li 93]

Guangxing Li. *Supporting Distributed Realtime Computing*. PhD thesis, Computer Laboratory, University of Cambridge, New Museum Site, Pembroke Street, Cambridge, CB2 3QG, England, July 1993.

[MAC 93]

Mapping ANSA Concepts to C++, **TR.036.00**. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1993.

[OVW 93]

An Overview of ANSAware 4.1, **RM.099.02**. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England. 1993