Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

AEG CORBA-sation

From ANSAware to CORBA

Nicola Howarth

Abstract

This document describes a design for the “CORBA-isation” of ANSAware. This includes the addition of a front-end for CORBA IDL, and the replacement of PREPC by enabling stubs to be invoked directly. All of the above is implemented over ANSAware 4.1.1.

Further work will include support for IIOP within ANSAware/RT.

APM.1550.00.01 **Draft** 7th November 1995

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

Distribution:
Supersedes:
Superseded by:

From ANSAware to CORBA

From ANSAware to CORBA

Nicola Howarth

APM.1550.00.01

7th November 1995

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	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

This document is aimed at system programmers with a familiarity of both ANSAware [ANSA 93] and CORBA [OMG 93].

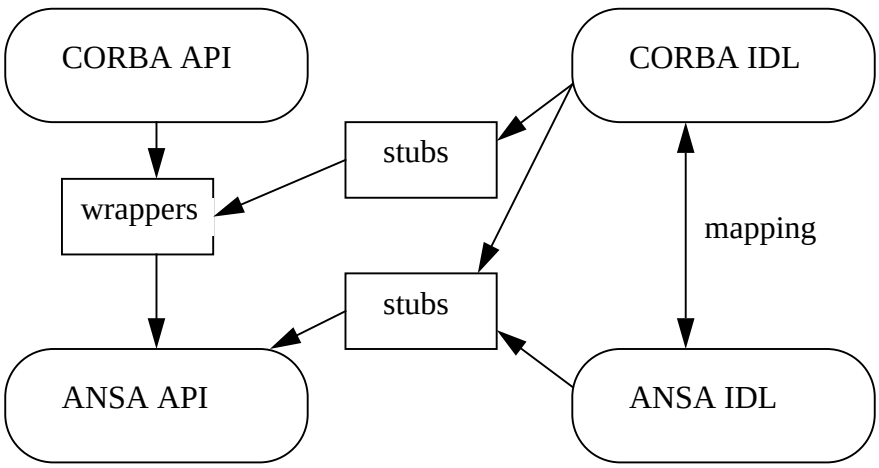
The work fulfills the requirements laid out by AEG.

The primary goal is to define ANSAware services using CORBA IDL. Further work provides some alignment of ANSAware with CORBA API.

The work is based on the following:

- ANSAware 4.1.1 - a flexible ORB
- ANSA IDL - this is semantically similar to CORBA but with different base types and constructors, and different syntax.
- ANSA PREPC - this is used to add objects to C, to add invocation and exception handling, and to provide consistency with the IDL.

The major tasks are to support CORBA IDL and API over ANSAware 4.1. This is done in two stages: firstly CORBA IDL with ANSA API, and secondly with CORBA API. The latter is implemented largely by providing additional stubs and wrappers to interwork with ANSAware. The third task is to implement an IIOP over ANSAware R/T.



2 Object Model

2.1 CORBA IDL

Support for CORBA IDL with language bindings for the C programming language.

The type ANY will not be supported. There is no equivalent within ANSAware, and since trading will be carried out using the ANSAware trader with ANSAware interface references, there is no requirement for the ANY type.

2.2 CORBA API/C

Remove DPL syntax and PREPC tool.

Support CORBA C language binding invocation (calling the stub directly).

The following CORBA functions for the C programming language should be supported:

- BOA_create
- BOA_dispose
- BOA_get_id
- BOA_set_exception
- ORB_object_to_string
- ORB_string_to_object
- Object_is_null
- Object_is_equal
- Object_duplicate
- Object_release
- exception_value
- exception_free

These should be provided by wrapping up existing ANSAware nucleus functions.

The following functions will NOT be supported (either because their function is unclear, they do nothing, they relate to handling the ANY type, or they map to non-existent repository functions):

- BOA_impl_is_ready (status with respect to withdraw)
- BOA_deactivate_impl (map to local withdraw)
- TypeCode_equal (this and the following relate to type ANY)
- TypeCode_kind
- TypeCode_param_count

- TypeCode_parameter
- ORB_get_default_context (this could be implemented as a null operation for future enhancements)
- Object_get_implementation
- Object_get_interface

Support will also NOT be provided for the dynamic invocation interface, including:

- create_request
- add_arg
- send
- send_multiple_requests
- get_response
- get_next_response

Functions which do nothing may be implemented as stubs.

2.3 IIOP

Further investigation is required to determine whether this can be supported within ANSAware 4.1. ANSAware 4.1 was designed to support multiple protocols, but this feature has not been used. An alternative is to implement IIOP within ANSAware/RT, which does provide support for protocol layers. In the latter case the CORBA IDL and API code will be ported to ANSAware/RT.

Sun's IIOP which is public domain software is not appropriate since it can only be used in applications, and not within an environment such as ANSAware.

Performance figures would be subject to the performance of IIOP. This can only be tested on SUN's IIOP.

2.4 ANSAware Enhancements

2.4.1 Elimination of GEX-related code

This will apply only to ANSAware/RT.

2.4.2 Inclusion of Timed REX Protocol

This only applies to ANSAware/RT. There are no CORBA compliant functions here, only ANSA nucleus functions.

Parallel protocol operation (GIOP/TREX/REX) is possible in ANSAware/RT.

2.4.3 Trader log file clean-up

This will be provided.

2.4.4 Security

Provision should be made for the addition of encryption functions prior to transmitting data across the network.

3 Schedule

This section is very much a draft and will be added to in due course.

Table 3.1: Work Schedule

month	time allowed	hours	total	spent	total spent	comments
06/95	0.50	75	75	21.5	21.5	IDL FE 0.5/2 pm
07/95	0.5	75	150	68	89.5	IDL FE 1.0/2 pm
08/95	1.0	150	300	136.5	226	IDL FE 2/2 pm
09/95	0.75	113	413	75.5	301.5	ANSA API BE 0.75/1.75 pm
10/95	1.0	150	563	195	496.5	ANSA API BE 1.75/1.75 pm
Deliver CORBA IDL FE and C ANSA API BE						
11/95	1.0	150	713			CORBA API BE 1/1.75 pm
12/95	0.75	113	826			CORBA API BE 1.75/1.75 pm
Deliver CORBA API						
01/96	1.0	150	976			IIOP for AW/RT 1/1.75 pm
02/96	0.75	113	1089			IIOP for AW/RT 1.75/1.75 pm
Deliver AW/RT with IIOP						
03/96	1.0	150	1239			enhancements/tidy up 1 pm
04/96	1.0	150	1389			documentation 1 pm
05/96	0.75	112	1501			support 0.75 pm
Total	10.00	1501				

Each item under “comments” in Table 3.1 gives the task to be done in the current month, and indicates the proportion of the task completed by the end of the month, in terms of person-months, e.g. 0.5/2 pm indicates that 0.5 of a month of a 2-month task should be completed.

3.1 IDL Front End and ANSA API

The ANSA “CORBA API” should not require any changes to the existing ANSAware API. For the CORBA compiler, we are using SUN’s IDL compiler front end, which comes with a dummy back end. This carries out parsing and scope checking, and generates an abstract syntax tree (AST) from which further work can be completed.

The tasks that will need to be completed are:

- Modify SUN back end to produce ANSA stubs.

- Convert the ANSAware examples' IDLs to CORBA.
- Test with ANSAware examples.
- Generate additional examples to cover CORBA features not supported by ANSAware. Each element of the CORBA IDL syntax should be covered.
- Modify the ANSAware build tree system to take both CORBA and ANSA IDLs
- Modify the ANSAware build tree system to build the Sun front end (in C++).

3.2 CORBA API

This involves the removal of any requirement for the DPL syntax PREPC tool, and the addition of CORBA functions.

- Changes to object invocations to call a C function directly. This will also involve:
  - — modifications to the handling of exceptions - CORBA and ANSA exception handling should be combined;
  - — implementation of the CORBA context clause, if not already complete.
- Extensions to the object invocation to include invocations on the trader and other tools.
- Generation of examples to test the new invocations.
- Implementation of CORBA functions which equate to ANSAware functions.
- Modification of examples to replace remaining DPL statements with CORBA equivalents.
- Implementation of CORBA functions not previously covered by ANSAware.
- Generation of further examples to test both new CORBA functions and ANSAware replacements not covered previously.

3.3 IIOP

Further investigation is required to determine whether this can be supported within ANSAware 4.1. ANSAware 4.1 was designed to support multiple protocols, but this feature has not been used. An alternative is to implement IIOP within ANSAware/RT, which does provide support for protocol layers. In the latter case the CORBA IDL and API code will be ported to ANSAware/RT.

Sun's IIOP which is public domain software is not appropriate since it can only be used in applications, and not within an environment such as ANSAware.

Performance figures would be subject to the performance of IIOP. This can only be tested on SUN's IIOP.

This section outlines the tasks required to implement the IIOP on ANSAware/RT.

- Port the CORBA IDL and CORBA API to ANSAware/RT, if AW/RT is used.

- Incorporate IIOP (using the DIMMA work as a template and wrapping C++ code with C)
- Change stubs to marshall into IIOP on-the-wire format

4 CORBA IDL

4.1 Requirement

The requirement is to implement a CORBA IDL stub generator which will generate C stubs to inter-work with ANSAware 4.1.1. The full CORBA IDL should be supported, with the exception of the type ANY.

4.2 Design Choices

Four possible routes were considered:

- the generic stub compiler
- modifications to stubc
- modifications to existing work based on the ANSA Phase 3 AST work
- making use of the Sun front end CORBA IDL compiler and generating a new back end

The generic stub compiler is not appropriate here. The term “generic” applies to the stub language, and not to the IDL. It could be used to generate CORBA IDL from ANSAware IDL, but not to generate the stubs themselves, as it only accepts ANSAware IDL as input.

If stubc were to be modified, this would involve changes to the parser and all the internal storage functions. The additional functionality needed to support the larger CORBA IDL would require further code to generate the required output. In other words, every part of the existing stubc would need modification, and the parser, in particular, would need a complete re-write.

The original ANSA Phase 3 AST work took the DPL syntax and parsed it into an abstract syntax tree (AST). Further work has been carried out to generate a reduced tree based on the DIMMA IDL only, and to generate a back-end which produces stubs, written in C++. If this route were to be used, then a new parser would be required, along with changes to the AST to handle the greater scope of CORBA IDL. The back-end would need to be re-written to produce C output code.

Sun Microsystems Inc have a CORBA IDL compiler front end which generates an AST. This is much more complex and at the time of writing not directly compatible with the AST discussed above (this situation is likely to change after future work). Sun also provide a template back end. The front end has already been used by APM to generate stubs for the WWW from CORBA IDL. This route removes the requirement for writing a parser, which would be a significant amount of work. In addition it carries out syntax and scope checking. We already have some expertise in using the front end. In fact the WWW work produced stubs partly in C, and while not compatible with ANSAware, comparisons can be drawn. This will make modifications easier.

The existence of a CORBA IDL parser and a template back end, and some experience in using the combination, make the last route the most attractive. It should produce the most comprehensive solution, with the minimum of effort. In addition SUN are maintaining the front end as CORBA evolves, and providing bug fixes and so on.

4.3 The Scale of the Problem

Given that we now have a CORBA IDL parser and scope checker, this section outlines the tasks that need to be done in order to generate ANSAware stubs from CORBA IDL.

CORBA IDL is much more comprehensive than ANSAware IDL, although most of CORBA can be mapped to an ANSAware equivalent. Many of the “new” features identified in this section are not new at all, but are covered by ANSAware in a different way. However since they are different, they are effectively new for the purpose of this exercise. There are also some features of ANSAware IDL which do not easily map to CORBA.

The first stage then is for an evaluation of the relationship between CORBA and ANSAware IDLs, and for the undertaking of some sort of mapping between the two. Since invocations are to be written in C, and these must in turn map on to the stubs, the API must also be considered in part at this point. The requirement is therefore to consider how operations declared in a CORBA IDL interface may be invoked, and how data may be passed to them, and returned from them. This includes the use of context and exception statements. This will be carried out in two parts. Initially the mapping will be to DPL-type statements, so that testing of the IDL can be carried out with minimum effort. Subsequently the operation invocations will be written in C. A comparison also needs to be drawn between the basic data types of the two IDLs.

Once such a comparison has been completed, and the format of the stubs and marshalling functions defined, then the back end of the CORBA IDL compiler can be implemented. This must then be tested, which is likely to involve changes to the original ANSAware examples.

This then identifies three main stages of effort:

1. A comparison between CORBA and ANSAware IDLs, and the generation of a mapping between the two. This includes:
   (i) comparison of basic types
   (ii) mapping of CORBA features not directly supported by ANSAware
   (iii) consideration of the C API
2. Implementation of back end to generate C stubs from the Sun front end.
3. Modification of the ANSAware examples to CORBA IDL to test the implementation.

4.4 Comparison of CORBA and ANSAware IDLs

4.4.1 Basic Types

The conversion from CORBA basic types to those of ANSAware is as given below. Where there is no direct mapping, the nearest is given in parentheses.

Table 4.1: CORBA and ANSAware Basic Types

CORBA	ANSAware
boolean (TRUE or FALSE)	Boolean (ansa_TRUE or ansa_FALSE)
unsigned short	Short Cardinal
unsigned long	Cardinal
short	Short Integer
long	Integer
float	Real
double	Long Real
octet	Octet
char	Char
string (section 4.4.1.1)	String
enum (section 4.4.1.2)	Enumeration
array (section 4.4.1.3)	Array
sequence (section 4.4.1.4)	Sequence
struct (section 4.4.1.5)	Record
discriminated union (section 4.4.1.6)	Choice
any (section 4.4.1.7)	no equivalent in ANSAware

4.4.1.1 string

Examples of strings in CORBA are:

```
string MyStrA;
```

The ANSAware equivalent is:

```
MyStrA : TYPE = STRING
```

The C equivalent for the ANSAware version is:

```
typedef ansa_String MyStrA;
```

CORBA also supports a bounded string where the maximum length of the string is defined:

```
string <10> MystrB;
```

In the CORBA C mapping this is treated in the same way as an unbounded string.

4.4.1.2 enum

A CORBA example:

```
enum Colour { Red, Green, Blue };
```

maps easily to the ANSAware equivalent:

```
Colour : TYPE = { Red, Green, Blue };
```

The CORBA C mapping #defines each enumerator with an appropriate unsigned integer value, but does not specify the use of the enum statement. ANSAware uses the enum statement, and it is proposed that this should be used also for the CORBA back end.

4.4.1.3 *array*

A CORBA example:

```
octet Address[16];
```

maps to ANSAware:

```
Address : TYPE = ARRAY 16 of OCTET;
```

These can map directly to C arrays.

4.4.1.4 *sequence*

CORBA supports both bounded and unbounded sequences. For a bounded sequence, the maximum size is specified. If no maximum size is specified, the size of the sequence is unspecified (unbounded). Prior to passing such a sequence as a function argument (or as a field in a structure or union), the length of the sequence, the maximum size of the sequence, and the address of the buffer to hold the sequence must be set. CORBA sequences are as follows:

```
typedef sequence <float, 10> Vector ;
typedef sequence <float> Vector;
```

An ANSAware sequence has a length and a buffer:

```
Vector : TYPE = SEQUENCE OF REAL;
```

and has a C equivalent:

```
typedef struct {
     ansa_Cardinal length;
     ansa_Real *data;
} Vector;
```

As can be seen, this does not include the maximum size specified by CORBA. To provide full CORBA functionality here, the structure must be modified to include the maximum size.

There are several options here. One is to ignore the maximum size altogether, even when used in CORBA IDL. This is not satisfactory if CORBA compliance is required. The alternative is to modify the structures generated from the IDL to include the maximum size. This will require some modification to existing ANSAware examples to ensure that the maximum size is set correctly.

The resulting C equivalent for bounded and unbounded sequences should be:

```
typedef struct {
     ansa_Cardinal max_length;
     ansa_Cardinal length;
     ansa_Real * data;
} Vector;
```

This does not make any use of the bounded sequence, but complies with the CORBA C language stub mapping. It is up to the user to set each of the elements in the structure. An instance of the type might be declared as:

```
Vector x = { 10L, 0L, (ansa_Real *)NULL);
```

Before passing `&x` as an in parameter, the programmer must set the `data` member to point to a `ansa_Real` array of 10 elements, and must set the `length` member to the actual number of elements to transmit.

There appears to be an inconsistency in CORBA here since it is possible to specify the maximum length in the IDL, but it does not appear in the C language stub mapping, and there is consequently no way of checking that the maximum size is not exceeded. Similarly, since the buffer is a pointer, rather than a fixed size array, it does not matter if the maximum size is exceeded, provided that the data pointed to is of sufficient size.

In order to reduce the potential quantity of code for sequences, they will be implemented in two stages. A sequence of a particular type will generate a generic `typedef` for a sequence of that type. Particular instances can then be "typedef'ed" to that generic type. For example:

```
IDL  typedef sequence<long,10> vec10;

C    typedef struct {
          CORBA_unsigned_long _maximum;
          CORBA_unsigned_long _length;
          CORBA_long *buffer;
     } CORBA_sequence_long;
     typedef CORBA_sequence_long vec10;
```

Care must be taken not to generate the typedef for `CORBA_sequence_long` twice, given an IDL of the form:

```
typedef sequence <long, 10> vec10;
typedef sequence <long, 20> vec20;
```

4.4.1.5 struct

A CORBA structure is of the form:

```
struct CreateRecord {
     AccountNumber acct;
     PersonalIdentificationNumber pin;
};
```

which has an ANSAware equivalent as a RECORD:

```
CreateRecord : TYPE = RECORD [
     acct : AccountNumber,
     pin : PersonalIdentificationNumber
];
```

Although different in format, these two have basically the same content.

Each of these maps directly to a C struct statement.

4.4.1.6 discriminated union

A CORBA discriminated union maps to an ANSAware CHOICE. A CORBA example is:

```
union Foo switch ( long ) {
     case ‘a’: long x;
     case ‘b’: Foo y;
};
```

which maps to C as follows:

```
typedef struct {
     CORBA_long _d;
     union {
          CORBA_long x;
          Foo y;
     { _u;
} Foo;
```

An example of the nearest equivalent in ANSAware is:

```
ListResult : TYPE = CHOICE OpStatus OF {
     OpSuccess => AccountRecord,
     OpFailure => OpReason,
};
```

which translates to C as:

```
typedef struct ListResult {
     OpStatus designator;
     union {
          AccountRecord u_OpSuccess;
          OpReason u_OpFailure;
     } u;
} ListResult
```

It can be seen that the C code is similar in the two cases.

4.4.1.7 *ANY*

The type *any* will not be supported in this implementation.

4.4.2 CORBA features handled differently by ANSAware

4.4.2.1 *Inheritance*

In CORBA, an interface can inherit from another interface, and if the *idl* for that interface is defined in another file, then that file can be included, e.g.:

```
file simple.idl
     interface simple { .... }
     interface simple2 : simple { ... }

file simple2.idl
     #include “simple.idl”
     interface simple3 : simple2 { ... }
```

In `simple.idl`, interface `simple2` inherits from interface `simple`. In file `simple2.idl`, interface `simple3` inherits from `simple2`, and the *idl* which holds `simple` and `simple2` has been `#include`’d.

The ANSAware `IS COMPATIBLE` with is the nearest equivalent of CORBA inheritance, with the `FROM` clause being replaced by the `#include` when the

interface to be inherited from is in another `idl` file. The `#include` feature also replaces the ANSAware `NEEDS` statement.

4.4.2.2 *naming*

The names of all operations and definitions within a module are prefixed by the name of the module.

The names of all operations defined within an interface are prefixed with the name of the interface.

For example, the following section of code:

```
const short l = 4;
enum N { zero, un , deux, trois };
module M {
     const short l = 4;
     enum N { zero, un , deux, trois };
     interface A {
          const short l = 4;
          enum N { zero, un , deux, trois };
          N foo(in N x, out N y, inout N z);
     };
};
```

gives rise to the following names:

```
#define l 4
typedef CORBA_enum N;
#define M_l 4
typedef CORBA_enum M_N;
typedef CORBA_Object M_A;
#define M_A_l 4
typedef CORBA_enum M_A_N;
extern M_A_N M_A_foo( M_A_N x, M_A_N *y ....)
```

In ANSAware, the names of operations within an interface are preceded by the name of the interface. CORBA takes this a stage further with modules, which do not exist in ANSAware, and also by prefixing the names of arguments to operations.

4.4.2.3 *const*

CORBA supports constant definitions of the form:

```
const short x = 4;
```

There is no equivalent in ANSAware. In C, the constant is translated into a #define:

```
#define x 4
```

4.4.2.4 *attribute*

In CORBA an interface can have attributes as well as operations. Such attributes are defined as part of an interface. An attribute definition is logically equivalent to declaring a pair of accessor operations; one to retrieve the value of the attribute, and one to set the value. A `readonly` attribute has only a `retrieve` value operation. For example:

```
interface foo {
     enum material_t { rubber, glass };
     struct positon_t {
          float x, y;
     };
     attribute float radius;
     attribute material_t material;
     readonly attribute position_t position;
     ...
};
```

The attribute declarations are equivalent to:

```
...
float          _get_radius();
void           _set_radius(in float r);
material_t     _get material();
void           _set_material(in material_t m);
position_t     _get_position();
...
```

There is no equivalent in ANSAware. However the back end of the IDL compiler should be able to provide this feature with little difficulty.

4.4.2.5 *oneway*

When a client invokes an operation with the CORBA *oneway* attribute, the invocation semantics are best-effort, which does not guarantee delivery of the call; best-effort implies that the operation will be invoked at most once. An operation with the *oneway* attribute must not contain any output parameters and must specify a *void* return type. An operation defined with the *oneway* attribute may not include a raises expression; invocation of such an operation, however, may raise a standard exception.

The CORBA *oneway* attribute is the equivalent of an ANSAware *announcement*, as opposed to an *interrogation*.

4.4.2.6 *void*

In CORBA the type of an operation which has no return value is *void*. In ANSAware IDL this is expressed as:

```
RETURNS [];
```

4.4.2.7 *operations - basic invocation and parameters*

A CORBA interface with one simple operation, for example:

```
interface Echo {
     string Echo ( in string Src );
};
```

is mapped to C as follows:

```
CORBA_string Echo_Echo(CORBA_Environment *ev,
          CORBA_string Src);/* for the server */
CORBA_string Echo_Echo(Echo ref, CORBA_Environment *ev,
          CORBA_string Src);/* for the client */
```

A similar interface in ANSAware is mapped as follows:

```
void I_Echo_Echo AW_P((                /* client stub */
     ansa_InterfaceFrame *_frame,
     ansa_Voucher *_v,
     ansa_InvocationType _it,
     ansa_String Src /* (IMMUTABLE) */
));

ansa_Status C_Echo_Echo AW_P((         /* client stub */
     ansa_Voucher *_v,
     ansa_String *_R1
));

int Echo_Echo AW_P((                   /* server stub */
     ansa_InterfaceAttr *_attr,
     ansa_string Src /* (IMMUTABLE) */,
     ansa_String* _R1
));
```

As can clearly be seen, there is a significant difference between the two. The most obvious difference is that CORBA maps to two functions, one for client and one for server, while ANSAware maps to three functions. The CORBA functions both returns a type reflecting that suggested in the IDL, while the ANSAware functions are each typed differently.

There is no problem in producing a back end to the compiler that will produce ANSA-like C code. However care must be taken with the form of the invocation, in particular with the return value, and with the local parameters, i.e. the parameters other than those specified by the application programmer.

The points to consider are as follows:

- the “type” of the function
- the CORBA in, out and inout parameter types
- the CORBA environment parameter
- the ANSA local parameters

As explained above, in CORBA the operation type, described in the IDL, is passed through as the return type of the C function. In ANSAware the functions’ return types are used for exception handling, so another route must be found for the return value. ANSAware operations separate out the input and output parameters as:

```
Echo : OPERATION [ Src : STRING ]
          RETURNS [ STRING ];
```

In other words, the input parameters are passed following the word `OPERATION`, and the return values are passed following the word `RETURNS`. There is no limit to the number of items which may be returned by an ANSAware operation. In order to map CORBA to ANSAware, it is proposed that the function type in CORBA is equated to a `RETURNS` type in ANSAware, where this item is the first in the `RETURNS` list (in fact the IIOP protocol mandates this). If the type is void, then it will not appear in the list.

As already explained, ANSAware separates the input and output parameters, and these equate to the CORBA “in” and “out” parameters. In both CORBA and ANSAware, input parameters are passed by value, and output parameters by reference. In addition CORBA supports the “inout” parameter, which can be passed in both directions. The CORBA C mapping states that an “inout”

parameter must also be passed by reference. In ANSAware the output parameters are specified by type only, and cannot be used as input parameters. The “inout” parameters must therefore be mapped to two parameters, one each for input and output. The input part will follow normal input parameters, and the output part will follow normal output parameters. In the final CORBA API this mapping could take place in the stub along, and a single parameter be used in the API.

The CORBA invocations take one or two additional parameters. For both client and server, the parameters specified in the IDL are preceded by a CORBA environment variable. For the client, this in turn is preceded by a reference, which is similar to the interface reference in ANSAware. The environment variable is used to transmit exception information. This is covered in more detail in section 4.4.2.8 below.

The ANSAware function for the server, here `Echo_Echo`, takes an additional argument `ansa_InterfaceAttr *_attr`. However this is used only internally, so provided it is included in the argument list, no further action need be taken with it.

The operations invoked by the client are `C_Echo_Echo` and `I_Echo_Echo`. Both of these take an `ansa_Voucher *_v`, and the latter also takes `ansa_InterfaceFrame *_frame` and `ansa_InvocationType _it`. The `_v` argument is used to communicate between the two invocations, and has no direct relevance to the application program. However it must be included in the invocation. The `_it` argument is the invocation type and must be generated by the back end.

To summarise: the additional parameters required by ANSAware must be included. In addition, the function type should be added as the first `RETURN` parameter. The CORBA environment parameter is discussed in the following section.

If the implementation of Echo_Echo in the server has the same name as the stub in the client, then these must have the same parameters, so that it is possible to directly invoke operations within the same executable.

4.4.2.8 operations - exception and raises

In CORBA IDL, exceptions are described and raised as follows:

```
exception NoSuchAccount { long reason; };
exception InvalidPin { long reason; string msg; };
interface Account {
     void Credit ( in AccountNumber acct,
                   in PersonalIdentificationNumber pin,
                   in long Amount )
                   raises ( NoSuchAccount, InvalidPin );
```

Exceptions are declared with a name and arguments. An operation signature identifies which exceptions may be raised by that operation. In the C mapping, each exception state yields a `#define` and a `typedef struct`:

```
#define ex_NoSuchAccount “ex_NOSuchAccount”
typedef struct NoSuchAccount {
     CORBA_long reason;
} NoSuchAccount;

#define ex_InvalidPin “ex_InvalidPin”
typedef struct InvalidPin{
     CORBA_long reason;
     CORBA_string msg;
} InvaildPin;
```

The raises clause has no effect on the C operation signature, i.e. it is ignored. When a user-defined exception occurs the server sets information on the exception into the environment variable, which can later be checked by the client.

ANSAware exception handling is intended to cover system errors. The CORBA and ANSAware exceptions are distinct, and can remain so. It remains only to determine the position of the CORBA environment parameter in the invocation. This should be as in the CORBA C mapping, i.e. the first parameter for the server, preceding the input parameters, and the second argument for the client, preceded only by the interface reference.

4.4.2.9 operations - context

An operation which takes a context clause has the following form:

```
void op1 (unsigned long s) raises (excl) context (“accuracy”);
```

The runtime system guarantees to make the value (if any) associated with each string in the context available to the object implementation when the request is delivered.

In a mapping to C, the context can be represented as an additional input parameter. On invocation, the parameter should consist of a pointer to a structure of the following type:

```
struct context {
     int length;
     char **data;
};
```

where the struct has a length which is the number of items, and an array of character strings. If the operation has a context, and that context is not specified in the invocation (or rather, the pointer has a null value), or if any of the string values are null, then the null will be replaced by the value of the environment variable(s) specified by the name in the IDL context.

The context parameter will appear as the final parameter to the operation.

4.5 ANSA API

Changes to the API will be done in two stages. In the first stage, the changes will be the minimum required to the current ANSAware API to support the CORBA IDL. This has been discussed in the previous sections, and will be summarised later in this section. The second stage will remove the need for ANSAware's DPL syntax and associated tool PREPC by making invocations

directly in C. Additional CORBA functions for local object management will be provided. This second stage is discussed in chapter 5.

The major change to the ANSA API is due to enhancements in the invocation. The basic ANSAware invocation is:

```
! {results} <- ifref$op ( arguments ) [exceptions]
```

where:

- `{results}` is a list of results (out arguments)
- `ifref` is the interface reference
- `op` is the operation name
- `arguments` is a list of arguments
- `exceptions` has the following form:
  `Continue statuslist1 Abort statuslist2 Signal statuslist3`

To this we need to add the following facilities, described in section 4.4.2 above:

- the function type - the first item in the list of results
- `inout` parameters (`in` and `out` parameters are covered by arguments and results) - treated in the same way as an `out` parameter
- the environment parameter - the ANSA API this should precede all other results (including the function type)
- the context parameter - the final input parameter

Take for example a simple ANSAware invocation:

```
! { obuf } <- ifref$Echo (ibuf)
```

Here we have a single result `obuf`, an interface `ifref`, an operation `Echo` of no type, and a single input parameter `ibuf`. For a CORBA invocation, with the minimum additions, this will become:

```
! { ev, obuf } <- ifref$Echo ( ibuf )
```

A more complicated example would be set up as follows:

```
! { ev, function_type, out_args, inout_args} ->
          ifref$Op ( in_args, context_arg )
```

where `ev` has type `CORBA_Environment`, and the context has type `CORBA_context`.

The basic types will be aligned from CORBA to ANSA and the application program may use either. Similarly types ANSA_Environment and ANSA_context will equate to the CORBA versions.

5 CORBA API

The work required for the CORBA API falls into two areas:

- the removal of the need for PREPC for CORBA applications
- support for CORBA functions

This does not imply complete removal of PREPC. ANSAware itself will still need to be built with PREPC. Non-CORBA applications will still require PREPC. And since CORBA does not provide all the functionally which PREPC provides, there may still be some requirement for PREPC in certain applications.

Take a typical example, the Echo example. Its use of PREPC is primarily as follows:

1. to create an instance of an Echo server (with given concurrency), and to export the interface reference corresponding to this instance to the trader
2. to withdraw an interface reference from the trader, and to destroy the interface instance
3. to import an interface reference from the trader
4. to invoke an operation on an interface reference

These four operations form the basic use of PREPC, and it is these that must be implemented in some way without the need for PREPC. (Note that invocations on the trader actually fall under the fourth point, that of operation invocation on an interface reference). The most straightforward way of implementing each of these operations is to invoke a function which carries out the instructions normally inserted by PREPC. This function will have to be created for each interface by the IDL compiler, along with the stub and marshalling functions.

A point considered was whether the *create* function should be combined with the *trader export* function, since when a interface is to be exported, this normally occurs immediately after creation. However not all interfaces are exported to the trader (for example individual accounts in the Simple Bank example), and so a separate *create* function must exist.

The second major topic, that of support for certain CORBA functions, will be implemented either by an implementation of the CORBA function directly, or where this function is interface dependent, it may again be created by the IDL compiler.

This work is covered in the following sections. Section 5.1 covers the implementation and creation of C functions to replace the basic PREPC facilities, in order to support CORBA-style invocations. Section 5.2 covers the implementation of specific CORBA functions.

5.1 Replacement of basic PREPC facilities

5.1.1 The PREPC Create and Export statements

The Create statement permits the creation of interface instances in a program. The Echo example is used here:

```
! USE TypeName
! DECLARE { ir } : TypeName SERVER
ansa_InterfaceRef ir;
! { ir } :: TypeName$Create(concurrency [, arguments])
```

An interface of type `TypeName` is instantiated, with the resulting interface reference returned in `ir`. `Create` is the only way of creating new interface references. The `concurrency` argument indicates how many simultaneous invocations for operations in this interface will be queued for this instance. If the optional `arguments` are provided, then the user must supply routines for the creation and destruction of interface instance state.

An example of ANSA IDL using Create is found in the Echo server:

```
! USE Echo
! DECLARE { ir } : Echo SERVER
ansa_InterfaceRef ir;
! { ir } :: Echo$Create(16)
```

This creates an instance of the Echo interface, with concurrency of 16.

The C code generated by the `Create` statement is (for the example above):

```
ansa_InterfaceRef ir;

{
ansa_Status _stat;
ansa_StatePtr _state = (ansa_StatePtr)0;
ansa_StateDestructor _destproc;
_destproc = (ansa_StateDestructor)0;
_stat = binder_createService(concurrency, nullNonce,
          Echo_Dispatcher, _state, _destproc, (ansa_ChannelId*)0,
           &(ir) );
thread_setExceptionCode(0);
thread_set_ExceptionState((ansa_StatePtr)0);
switch(_stat) {
     case ok:
          break;
     default:
          instruct_Abort("Prepc.createService", _stat );
          break;
     }
}
```

The nearest CORBA equivalent is the BOA_create function, which is used to create an arbitrary object reference (as opposed to an interface instance). The BOA_create function takes the following form:

```
CORBA_Object CORBA_BOA_create (
     CORBA_Object            boa,
     CORBA_Environment       *ev,
     CORBA_ReferenceData     *id,
     CORBA_InterfaceDef      intf,
     CORBA_ImplementationDefimpl
);
```

Here the function returns a CORBA_Object. The arguments are:

- `CORBA_Object boa` refers to the Basic Object Adapter Interface (BOA) on which the operation is invoked.
- `CORBA_Environment *ev` contains details of any exceptions.
- `CORBA_ReferenceData *id` is “immutable identification information, chosen by the implementation at object creation time” [OMG 93].
- `CORBA_InterfaceDef intf` refers to the Interface Repository.
- `CORBA_ImplementationDef impl` refers to the Implementation Repository.

It should be noted that this area of the CORBA specification is likely to be extensively modified, and so it is not considered necessary to conform to it in detail. Indeed other existing CORBA implementations do not conform.

In a mapping from the BOA_create function to ANSA, the type of the function logically maps to an `ansa_InterfaceRef`. The arguments are less straightforward:

- boa - no equivalent for ANSAware - a possibility here is to provide some #defined constant such as `ANSA_BOA` and pass that, but ignore it
- *ev - this will be used to handle exceptions - see section 5.1.3 on invocations.
- *id - no equivalent, but may be of use to specific applications (eg. to distinguish between bank accounts in the simple bank example)
- intf - no equivalent
- impl - no equivalent

None of these cover the concurrency used by PREPC. Now it would be possible to assume a concurrency of one, but this is building in an unnecessary restriction. Instead one of the arguments with no ANSA equivalent ccould be used for concurrency. An additional problem is that no interface has been specified. This is easily resolved by invoking *create* as an operation on the interface in question, here Echo. A simple invocation of the *create* function becomes:

```
ansa_InterfaceRef ir;
ir = CORBA_Echo_create( ev, 16 );
```

If the interface name should at some future time be required (for example if a repository were to be added), then the create function itself could know it’s IDL type name (e.g. by a #define within the function when it is generated by the IDL compiler).

The function can be implemented by the IDL compiler at the same time as the marshalling and stub functions.

This provides a satisfactory invocation for the *create* function when used alone, i.e. when the interface reference is not exported to the trader. When an

export is required, it is valid to combine the two functions, providing additional information to the trader by extra arguments. The *Export* function is of the following form:

```
! {} <- traderRef$Export( “Echo”, “/ansa/testservices”, propbuf,
                ir )
```

Of the arguments, the first, specifying the type name, and the last, giving the interface reference, are already included in the *create* invocation. This leaves the specification of the trader reference itself, and the context and property list. An alternative, and preferred, method is to provide a function library with such functions as Export (see section 5.1.4).

The ANSAware trader is not the same as the CORBA implementation repository, but has similarities, in that it provides a mapping from a server's name to the executable code which implements that server. The trader reference therefore maps logically onto the *impl* argument of *BOA_create*. The context and property list form additional arguments. They do not map to CORBA because the ANSA trader does not map to CORBA, and this is an implementation of CORBA basics over ANSAware, rather than a complete conversion of ANSAware to CORBA.

The combined *Create* and *Export* statements become:

```
ansa_InterfaceRef ir;
char propbuf[1024];
ir = CORBA_Echo_create( ev, 16, &traderRef, “/ansa/testservices,
                            propbuf );
```

One problem remains: how to handle arguments given to the Create statement. PREPC generates an invocation to a user-supplied routine for the creation of interface state. This function returns an ansa_StatePtr, and this is in turn passed to the binder_createService function. Since the user must supply this routine, then the user can also invoke it, and pass the ansa_StatePtr to the Create function, rather than the arguments themselves. When no arguments are used, an ansa_StatePtr(0) should be passed. So the final version of the create function is of the form:

```
ansa_InterfaceRef CORBA_Echo_create (
     CORBA_Environment       *ev,
     ansa_Cardinal           concurrency,
     ansa_StatePtr           state,
     ansa_InterfaceRef       traderRef,
     ansa_String             context,
     ansa_String             proplist
);
```

In the Echo example with no arguments, but including an export to the trader, this becomes:

```
ansa_InterfaceRef ir;
ir = CORBA_Echo_create( ev, 16, ansa_StatePtr(0), traderRef,
          “/ansa/testservices”, propbuf );
```

In an example where no export is required, but where there are arguments, this becomes:

```
ansa_InterfaceRef ir;
ansa_Integer value;
ansa_StatePtr _state = TypeName__Create( value );
ir = CORBA_TypeName_create( ev, 1, _state, ansa_InterfaceRef(0),
          ansa_String(0), ansa_String(0) );
```

5.1.2 The PREPC Withdraw and Destroy statements

The Withdraw statement permits a service provider to discontinue service on a particular service offer.

```
ansa_InterfaceRef ir;
! traderRef$Withdraw(ir)
```

An example of ANSA IDL using Withdraw and Destroy is again found in the Echo server:

```
ObjState *p;
     p = (ObjState *)state;
     if( p->export == ansa_TRUE )
!         traderRef$Withdraw( p->ref )
!    {} :: Echo$Destroy( p->ref )
     free ((char *)p);
```

This checks that the interface was exported to the trader, and if so, withdraws the offer. It then destroys the interface. The C code generated is:

```
ObjState *p;
p = (ObjState *)state;
if (p->export == ansa_TRUE)
{    ansa_Status _stat; int _sg5;
tlab5:
     _stat = binder_withdrawService(&(traderRef), &(p->ref));
     thread_setExceptionCode(0);
     thread_setExceptionState((ansa_StatePtr)0);
     switch(_stat) {
          case ok:
               break;
          default:
               instruct_Abort("Prepc.withdrawService", _stat);
               break;
          case transmitTimeout:
          case invalidNonce:
          case illegalOperation:
          case illegalInterface:
          case abnormalReturn:
          case gexEmptyGroup:
          case gexObsoleteReference:
          { _sg5 = Signal_binder_relocate(_stat, &(traderRef));}
          if (_sg5 == ExceptionAbort)
               instruct_Abort("Prepc.Relocate", _stat);
          else if (_sg5 == ExceptionRetry)
               goto tlab5;
          break;
     }
}
{
ansa_Status _stat;
_stat = binder_destroyService( &(p->ref) );
```

```
thread_setExceptionCode(0);
thread_setExceptionState((ansa_StatePtr)0);
switch(_stat) {
     case ok:
          break;
     default:
          instruct_Abort(“Prepc.destroyService”, _stat);
          break;
     }
}
free ((char *)p);
```

As can be seen, most of this code is taken up with error handling.

The nearest CORBA equivalent, and the partner of BOA_create, as this is the partner of PREPC Create, is BOA_dispose. This has the form:

```
void CORBA_BOA_dispose {
     CORBA_Object       obj
};
```

which translates to ANSA as:

```
void CORBA_Echo_dispose( ansa_InterfaceRef ir );
```

However these functions are normally only used for managed services, i.e. when used with the factory.

5.1.3 PREPC Invocations

This facility is used to cause the execution of an operation; the thread making the invocation is suspended pending the arrival of a reply (or until an error occurs).

```
ansa_InterfaceRef ir;
! { r1, ..., rn } < ir$Op ( a1, ..., am )[ ExceptionList ]{ retry }
```

where

- `{ r1, ..., rn }` are the results returned from the operation
- `ir` is the interface reference
- `Op` is the operation name
- `(a1, ..., am)` are the arguments passed to the operation
- `[ExceptionList]` describes how to handle exceptions
- `{retry}` is used to specify the number of times the underlying execution protocol should attempt to communicate with a remote service without receiving a reply.

An example of an invocation found in the Echo client is:

```
! {obuf} <- intRef$Echo(ibuf) Continue ok Abort illegalOperation,
                illegalInterface Signal abnormalReturn {5}
```

which generates the following C code:

```
{    ansa_Status _stat = ok; int _sg2;
     ansa_InterfaceFrame _frame;
     ansa_Voucher _v;
tlab2:
     _frame.fr_retries = 5;
     _frame.fr_ifref - &(intRef);
     _frame.plug = 0;
     _frame.ForceSingleton = 0;
     I_Echo_Echo(&(_frame), &_v, CALLtype, buf );
     _stat = C_Echo_Echo(&_v, &(obuf));
     thread_setExceptionCode(0);
     thread_setExceptionStte((ansa_StatePtr)0);
     switch(_stat) {
          case ok:
               break;
          case abnormalReturn:
               {
               _sg2 = Signal_Echo_Echo(_stat, &(intRef),
                              (buf), &(obuf));
               }
               if (_sg2 == ExceptionAbort)
                    instruct_Abort("Echo.Echo", _stat);
               else if(_sg2 == ExceptionRetry)
                    goto tlab2;
               break;
          case illegalOperation:
          case illegalInterface:
          default:
               instruct_Abort("Echo.Echo", _stat);
               break;
     }
}
```

It can be seen here that the operation arguments (inputs) are passed by value to the function `I_Echo_Echo`, while the results (outputs) are passed (by reference) from function `C_Echo_Echo`.

Details of CORBA operations are given in section 4.4.2.

CORBA invocations have the following components:

- a function type, which may be void
- an operation name
- in, out and inout parameters
- an environment parameter, for exception handling
- a context, related in this case to the Unix environment (not the same as the ANSA context)

These must be supported, and in addition ANSAware requires:

- an interface reference
- a retry count - this is optional in ANSAware, but could still be supported for CORBA applications - in fact it could be embedded into the CORBA context

A C representation of an invocation of a CORBA function over ANSAware should therefore be similar to the following:

```
res1 = opname( ifref, ev, inarg1, ..., inargx, &outarg1,
     &...,&outargy, &inoutarg1, &..., &inoutargz, context)
```

The Echo example given above would then become:

```
obuf = EchoEcho( intRef, ev, ibuf, (CORBA_context)retry );
```

The implementation of the C invocations described above will be carried out by a function generated by the IDL compiler, which will consist of code similar to that originally generated by PREPC. The handling of exceptions differs considerably between ANSAware and CORBA, and is discussed in section 5.1.5.

5.1.4 The PREPC Import statement

This statement is used to set up a binding to a specified interface type within a given trading context, subject to any user-supplied constraints. It takes the form of a standard PREPC invocation:

```
! { ir } <- traderRef$Import("typeName", ansa_String Context,
               ansa_String Constraints )
```

This would convert to a C invocation in the CORBA style as:

```
ir = TraderImport( traderRef, ev, "typeName",
               ansa_String Context, ansa_String Constraints,
               (CORBA_context)0, ansa_Cardinal(1);
```

Note that the ANSA and CORBA contexts are not the same thing. The section on invocations above indicates that such a function would be generated by the IDL compiler. In the case of the trader, however, which will still be built using PREPC, a function library will be generated to provide such functions as Import.

5.1.5 Exceptions

Exceptions are handled in a completely different manner in ANSA and CORBA. ANSA provides the applications programmer with three options for handling errors: to continue, abort, or signal the error to a user-provided handler. This provides good flexibility with minimum of programming effort. CORBA simply returns the error in a structure, and allows the programmer to handle that error within the application code. This does not preclude the same actions as with ANSAware, but the handler appears embedded in the application program, rather than being kept separate.

The implementation of CORBA exception handling is a simplification of the existing code, since the error handling code will be removed. The current content of such code needs to be clearly documented however, so that the application programmer can make use of the various functions provided within ANSAware for handling exceptions.

Exceptions are passed to invoking code through the environment argument, ev, which is of type CORBA_Environment (using the Echo example again):

```
typedef struct CORBA_Environment {
     CORBA_exception_type _major;
     CORBA_exception_type _minor;
     CORBA_exception_value _value
     CORBA_string _reason;
} CORBA_Environment;

CORBA_Environment *ev;

obuf = EchoEcho( intRef, ev, ibuf, (CORBA_context)0, 5 );
switch ( ev._major ) {
     case CORBA_NO_EXCEPTION:/* successful outcome */
          break;
     case CORBA_USER_EXCEPTION:/* a user-defined exception */
          .....              /* take whatever action required */
          break;
     default:               /* standard exception */
          break;
}
/* free any storage associated with exception */
CORBA_Exception_free( &ev );
```

The CORBA specification indicates that the CORBA_Environment structure should include at least the exception type. Other elements are application-dependent. On return from an invocation, the _major field indicates whether the invocation terminated successfully; _major can have one of the values CORBA_NO_EXCEPTION, CORBA_USER_EXCEPTION, or CORBA_SYSTEM_EXCEPTION; if the value is one of the latter two, then any exception parameters signalled by the object can be accessed.

Three functions are defined on a CORBA_Environment structure for accessing exception information; their signatures are:

```
extern CORBA_char *CORBA_exception_id(CORBA_Environment *ev);
extern void *CORBA_exception_value(CORBA_Environment *ev);
extern void CORBA_exception_freef(CORBA_Environment *ev);
```

CORBA_exception_id() returns a pointer to the character string identifying the exception. If invoked on a CORBA_Environment which identifies a non-exception, (_major==CORBA_NO_EXCEPTION) a NULL is returned.

CORBA_exception_value() returns a pointer to the structure corresponding to this exception. If invoked on a CORBA_Environment which identifies a non-exception or an exception for which there is no associated information, a NULL is returned.

CORBA_exception_free() returns any storage which was allocated in the construction of the CORBA_Environment. It is permissible to invoke CORBA_exception_free() regardless of the value of the _major field.

User defined exceptions are identified in the raises clause in the IDL (see section 4.4.2.8. The IDL compiler generates a structure for each exception:

```
#define ex_NoSuchAccount “ex_NOSuchAccount”
typedef struct NoSuchAccount {
     CORBA_long reason;
} NoSuchAccount;
```

The _reason field in the CORBA_Environment structure should be set to the string describing the exception, here “ex_NoSuchAccount”. This is returned by the CORBA_exception_id() function. The _value field should be set to point to an instance of the structure generated by the IDL compiler, and it is this pointer that is returned by the CORBA_exception_value() function.

System errors may have a similar structure defined, or the _value field may be null. In the latter case, the _minor field should indicate the number of the system error involved. In the case of ANSAware errors, the fields should be set up as follows (for an example of an illegal operation):

```
ev->_major = CORBA_SYSTEM_EXCEPTION;
ev->_minor = illegalOperation;
ev->_reason = ex_illegalOperation;
```

where the latter is defined as:

```
#define ex_illegalOperation “ex_illegalOperation”
```

This will require some additional definitions to ANSAware’s existing errors, and the initialising of the CORBA_Environment structure when an error occurs.

An additional function is provided to simplify the setting up of a CORBA_Exception when an exception occurs. This is:

```
extern void CORBA_BOA_set_exception( CORBA_exception_type major,
     CORBA_BOA_exception_type minor,
     CORBA_exception_value value,
     CORBA_string reason );
```

The illegalOperation example above then becomes:

```
CORBA_BOA_set_exception( CORBA_SYSTEM_EXCEPTION, 0, 0,
                              ex_illegalOperation );
```

5.2 Implementation of CORBA functions

Most of the following functions refer to CORBA object references. For the purpose of this work, the object reference is approximately the equivalent of the ANSA interface reference.

5.2.1 Basic Object Adapter

The Basic Object Adapter Interface (BOA) provides various functions:

- generation and interpretation of object references
- authentication of the principal making the call
- activation and deactivation of the implementation
- activation and deactivation of individual objects, and
- method invocation through skeletons.

It has no equivalent in ANSAware, although it is possible such an interface might be created in order to implement the following functions.

5.2.1.1 BOA_create, BOA_dispose

These functions are discussed in sections 5.1.1 and 5.1.2.

5.2.1.2 *BOA_get_id*

There is no equivalent in ANSAware. If this function is invoked it will return the internet address plus the local interface address.

5.2.2 Object functions

A group of functions to carry out operations on CORBA objects will equate to functions operating on ANSAware interface references. In particular:

```
string ORB_object_to_string( ansa_InterfaceRef *ifref );
```

returns a string representation of an interface reference.

```
ansa_InterfaceRef *string_to_object( string str );
```

returns an interface reference which equates to the given string.

```
boolean is_nil( ansa_InterfaceRef *ifref )
```

returns ansa_FALSE since ANSA has no concept of a null interface reference.

```
boolean object_is_equal( ansa_InterfaceRef ifref1,
                         ansa_InterfaceRef ifref2 )
```

will return ansa_TRUE if, and only if, the two interface references specified can be used to direct invocations to the same interface instance (equivalent to ANSAware's `ifref_cmpIdentify()`).

Note: Is this too strong?

```
ansa_Status object_duplicate( ansa_InterfaceRef *to,
                              ansa_InterfaceRef *from )
```

copies the "from" interface reference into the "to" interface reference; storage will be allocated as required (as ANSAware's `ifref_copyRef()`).

```
void release( ansa_InterfaceRef ifref );
```

will free a previously copied or parsed interface reference (as ANSAware's `ifref_freeRef()`).

5.2.3 Exception functions

The following functions are described in section 5.1.5:

```
extern CORBA_char *CORBA_exception_id(CORBA_Environment *ev);
extern void *CORBA_exception_value(CORBA_Environment *ev);
extern void CORBA_exception_free(CORBA_Environment *ev);
extern void set_exception( CORBA_exception_type major,
     CORBA_BOA_exception_type minor,
     CORBA_exception_value value,
     CORBA_string reason );
```

6 GIOP Support

CORBA2 defines the Universal Networked Objects (UNO) for interoperability. Its main interoperation protocol is called GIOP, and its mapping and implementation over TCP is called IIOP.

UNO defines a standard format for interface reference representation, called IOR, which is a type string plus a sequence of tagged protocol profiles. Each profile itself is an opaque octet sequence, and can be used to encapsulate any address information.

IIOP has a special mapping for its profile, called IIOP IOR profile, which has the following fields:

- *version*, 2 octets
- *host*, a character string
- *port*, 2 octets
- *object_key*, an opaque octet sequence

GIOP defines the following components:

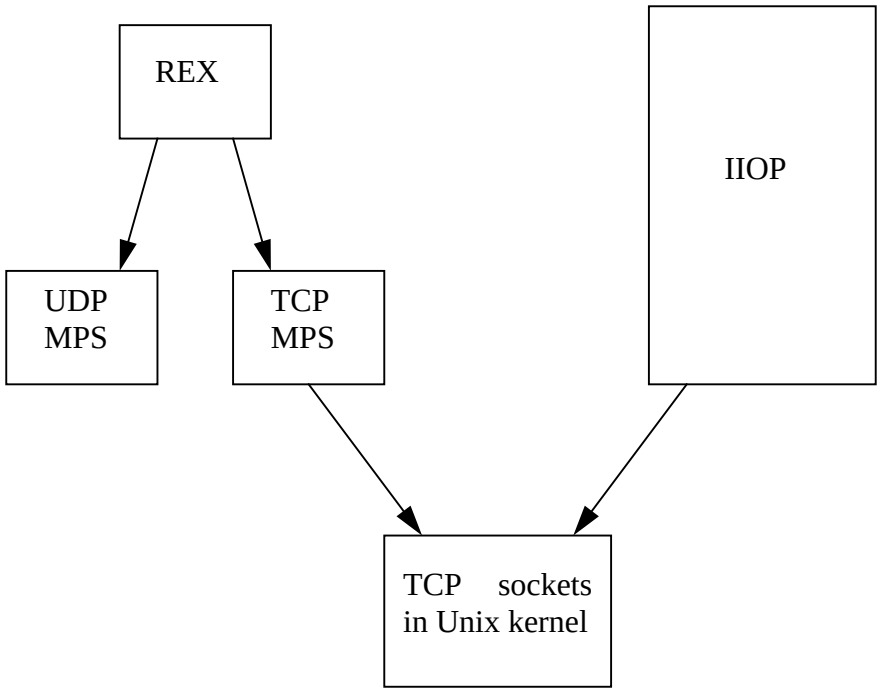
- Common Data Representation (CDR), which defines the wire format (as an octet stream) for all CORBA IDL types. Data in the octet stream are kept in alignment and in local host byte order.
- RPC Messages, defines seven types of message for RPC processing. Each message has a common header plus a message type specific header.
- Connection management, GIOP is designed to execute with a connection oriented protocol. Conventions are defined to open, close and multiplex network connections.

IIOP further details the GIOP connection management in the case of TCP.

6.1 Building IIOP into ANSAware

In ANSAware, the protocol REX provides a service for process-to-process interactions across a network. It is a remote procedure call protocol with extensions to support some of the generalisations of “procedure calls” permitted by the ANSA computational model. Below REX are the various Message Passing Services (MPS), UDP and TCP. An implementation of the IIOP would cover the tasks done both by REX and the MPS, as shown in figure 6.1.

Figure 6.1: Rex, MPS and IIOP



7 ANSAware Enhancements

The following enhancements are included:

- elimination of GEX code (AW/RT only)
- inclusion of timed REX protocol (AW/RT only)
- trader logfile clean-up
- security ??????

7.1 Elimination of GEX-related code

7.2 Timed REX Protocol

7.3 Trader Logfile clean-up

7.4 Security ???????

References

[OMG 93]

The Common Object Request Broker: Architecture and Specification; **OMG Document Number 93.12.43,** Revision 1.2, Draft 29 December 1993.

[ANSA 93]

An Overview of ANSAware 4.1, **RM.099.02**, APM Ltd., Cambridge U.K., February 1993