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**

ANSA Phase III

A Stub Compiler for CGI and HTTP: The Programmer's Guide

Nigel Edwards

Abstract

Deploying new and customised applications and services in the Internet is difficult. Yet many organisations require more than the basic functionality provided by standard applications such as HTTP servers (WWW servers).

The tool kit described in this document demonstrates how it can be made relatively easy to program applications in WWW which use libwww and, in particular, CGI and HTTP. By using the concept of client and server stubs we have provided a layer of abstraction which protects the programmer from the underlying details of the protocols. This means that programmers have less work to do to write their applications and because they have less code to write, they are less likely to make mistakes.

APM.1465.01 **Approved** 21st September 1995
Technical Report

Distribution:
Supersedes:
Superseded by:

A Stub Compiler for CGI and HTTP: The Programmer's Guide

A Stub Compiler for CGI and HTTP: The Programmer's Guide

Nigel Edwards

APM.1465.01

21st September 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

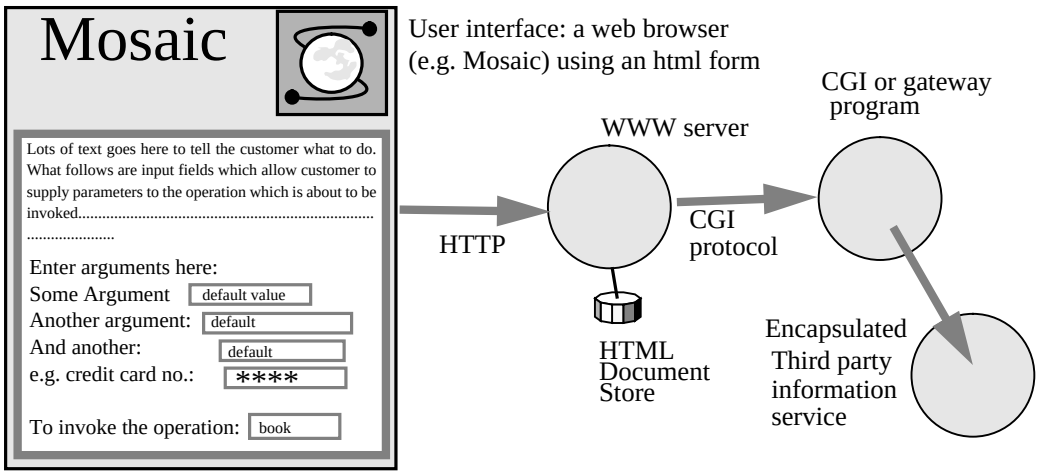
TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

The tool described in this paper is intended to assist those writing Common Gateway Interface (CGI) programs for the World-Wide Web (WWW). CGI is a standard for external gateway programs to interface with HTTP servers. As shown in figure 1.1, this allows WWW to encapsulate third party information services such as databases. CGI defines the format of the data stream between the WWW server and the gateway or CGI program. It also defines the environment variables available to the gateway program. The CGI and HTTP (HyperText Transfer Protocol) protocols are described in [McCOOL] and [BERNERS-LEE 95] respectively.

Figure 1.1: CGI in WWW[a]



a. Mosaic is a trademark of the University of Illinois

Clients for encapsulated information services are implemented by HTML (Hypertext Markup Language) “Forms” [CONNOLLY]. An HTTP server returns an HTML form to a WWW browser which asks the user for the parameters needed by the CGI program. The user supplies the parameters by filling out input fields in the displayed from. The form then sends these back to the HTTP server which forks the CGI program and passes it the parameters. The HTTP server passes any results from the CGI program to the client in the form of another document (generally an HTML document).

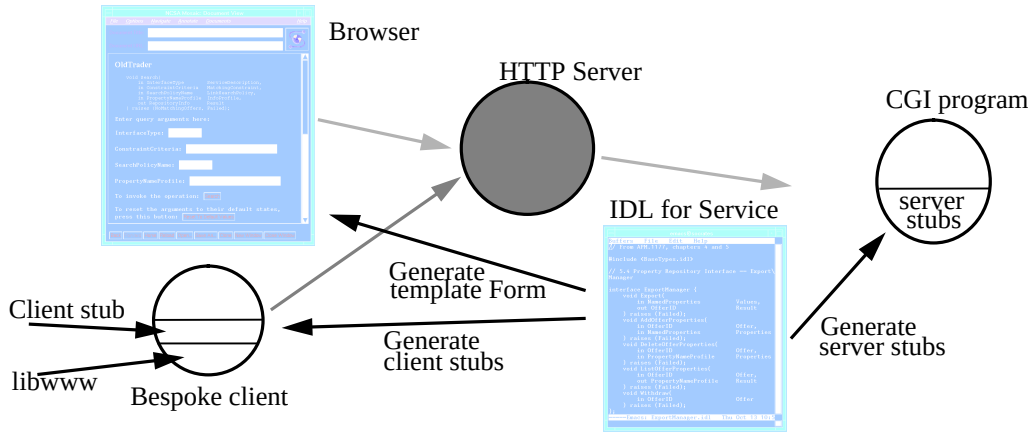
CGI programs are not only used as gateways. They are also used without invoking external services, to extend the functionality of a server and to implement some HTTP methods (such as DELETE).

There are few tools to assist CGI programmers; they must explicitly parse the incoming data stream to extract parameters. This is analogous to CORBA programmers having to write their own unmarshalling routines for servers.

We have developed a stub compiler for CGI. From a CORBA IDL description of the CGI program it generates a server stub, a client stub and a template

HTML form (see figure 1.2). Currently all CORBA IDL primitives and data types are supported, except the “ANY” data type.

Figure 1.2: CGI CORBA IDL Compiler



Server stubs make the job of the CGI programmer easier: they unmarshal the incoming data stream.

Client stubs encapsulate libwww (the underlying library used by many WWW applications) in much the same way as IDL stubs and skeletons encapsulate the programmer from the API of the underlying ORB in CORBA. Consequently, it becomes very easy to write bespoke client applications for CGI services: for example, the following program is a CGI clients for the Echo program running on <URL:http://www.ansa.co.uk:8080/cgi-bin/Echo>[1].

Figure 1.3: A simple CGI client program

```
#include <CEcho.h>
PUBLIC char * HTAppName = "Echo client"; /* Application name */
PUBLIC char * HTAppVersion = "0";  /* Application version */
static CORBA_Environment ev = {CORBA_NO_EXCEPTION, 0, 0};
int main(int argc, char* argv[]){
  Echo ref = Stub_bind(“http://www.ansa.co.uk:8080/cgi-bin/Echo”,
  printf(Echo_Echo(ref, &ev, "hello world\n"));
  HTAnchor_delete(ref);
}
```

Template HTML forms are supported for the simplest CORBA data types (e.g. strings and longs). Template forms generated by the stub compiler are guaranteed to produce a parameter stream consistent with the CGI server stub generated from the same IDL definition. This helps to ease the problem of keeping an HTML form consistent with its corresponding CGI program. Template forms need to be annotated with text to explain the input fields to humans.

The programming model provided by the stub compiler is very close to the CORBA programming model. The main difference is that CGI programs are stateless: they are forked for each invocation by the HTTP server. This means

1. At the time of writing there is no HTTP server at www.ansa.co.uk:8080, our home page is: <URL:http://www.ansa.co.uk/>

that any state change made by the client has to be stored externally (e.g. on the local file system).

Please remember that this is a prototype, not a product, so there is much scope for tidying up the code and improving on-the-wire formats. If you have any comments we would appreciate them, we will also do our best to answer any questions. The stub compiler is based on the public domain front end CORBA IDL compiler implemented by SunSoft. The only language supported at present is C.

The document continues as follows:

- §2 describes how to run the supplied test applications;
- §3 describes how to build an application and how the client and server stubs work;
- §4 gives an overview of the stub compiler, so that you can customise the back end;
- §5 describes the routines provided by the stub library;
- §6 states our conclusions and discusses possible future directions.

2 Running the test applications

./README_FIRST explains the platform requirements and what you need to do to build the software.

The software comes with a number of test applications which you should be able to run if you have built it successfully. The most useful of the test applications is the Echo service which can be used to test server performance. The SimpleBank application is the canonical ANSA example which we use to evaluate platforms such as commercial Object Request Brokers. It demonstrates how the toolkit might be used to build and deploy a real application. The other applications have been used to test the software, they do not do anything more useful.

Most of the test clients are called "**client**" and take no parameters. They can be run by changing to the appropriate directory under **./ansa_test** and typing "**client**". Many of the directories contain a file called **results.log** containing sample output from the client program.

The exceptions to this are the client programs in **./ansa_test/Echo** and **./ansa_test/simplest**.**./ansa_test/SBank** does not have any client programs: access to this application is only through using HTML forms.

2.1 Echo

The Echo software comes from ANSAware [APM 93] and is useful for measuring system performance. There are five client programs: **client**, **techo**, **tsink**, **tsource** and **trivial**. The programs **techo**, **tsink** and **tsource** time their execution and print performance statistics. These take the following command line flags:

- **-n<number>**: specifies the number of invocations
- **-b<buffer size>**: specifies the amount of data to be sent or source (range 1 to 24000.

Hence `techo -b10000 -n5` will cause techo to invoke Echo five times with a buffer of 10K.

2.2 simplest

The program `client` expects two integer arguments, of which the second must non-zero.

Hence `client 45 22` produces the following output:

```
First incremented is 46
Second incremented is 23
First divided by second is 2
Remainder is 1
```

3 Building a CGI application

This chapter explains how to build a CGI application using the stub compiler. You should also read the "readme" files in the distribution:

- **./README_FIRST** explains the platform requirements and what you need to do to build the software;
- **./README** is top level readme file for the stub compiler front end from Sunsoft, it gives pointers to the rest of SunSoft's documentation;
- **./docs/CLI** describes the command line arguments for the stub compiler.

This chapter assumes you understand the CORBA Interface Definition Language (IDL) ([OMG 93] is available via anonymous ftp if you don't).

The software distribution comes with a number of test applications which will be built for you when you build the software. The remainder of this chapter uses one of these test applications, Echo, to show what you need to do if you want to build you own applications. SimpleBank is used to illustrate how to build a stateful application.

The Echo sources can be found in directory **./ansa_test/Echo**. The source files are:

- **Makefile** — the rules used by make;
- **Echo.idl** — the interface definition of the Echo service;
- **server.c** — the CGI server program for the Echo service
- **client.c**, **techo.c**, **tsink.c**, **tsource.c**, **trivial.c** —client programs using the Echo service.

The easiest way to build a new application is to create a new directory at the same level in the build tree as Echo, for example: **./ansa_test/NewApp**. Into this directory copy the makefile from the directory **./ansa_test/Echo**. Assuming that you are going to write a single client application, called "client.c", edit the new file "**./ansa_test/NewApp/Makefile**":

- Change all occurrences of the string "Echo" to "NewApp";
- Delete the make rules for **trivial**, **techo**, **tsink** and **tsource**;
- Remove **trivial**, **techo**, **tsink** and **tsource** from the definition of "TARGETS" and "CLIENT_APPS".

You now have to create the following files:

- **NewApp.idl** — the interface definition for the new service
- **server.c** — the CGI server program for the new service
- **client.c** — a CGI client program

Once you have created these files, the command "make" in the directory will build your application. The command "make install" will install it (see §3.4 and §3.5) The remainder of this chapter gives more details on what needs to be

done to create the three files listed above and explains how the stubs drive the underlying software and protocols.

3.1 The ".idl" file

This file conforms to the standard rules for CORBA interface definitions (see [OMG 93]). This file NewApp.idl will define the interface offered by you new CGI program. The Echo.idl file is shown below:

```
interface Echo{
 string Echo(in string Src);
 void Sink(in string Src);
 string Source(in long Length);
 string Reverse(in string Src);
};
```

This says the Echo service has four operations. The Echo operation takes a string and returns a string; the Sink operation takes a string and returns nothing; the Source operation takes an integer (the length of the sourced string) and returns a string; the Reverse operation takes a string and returns a (reversed) string. By timing the execution of these operations we can measure the performance of a system (see **`./ansa_test/Echo/techo.c`**).

User and system exceptions are supported and are discussed in §3.8.

3.2 Stub compiler command line arguments

The stub compiler takes all command line arguments defined in file **`./docs/CLI`**. In addition the following arguments are supported by the backend:

- **`-Wb,h`** stops html file generation
- **`-Wb,o`** stops generation of everything except the html form which is written to standard output (this enables generation of forms on the fly)
- **`-Wb,p<n>`** sets the server port to n
- **`-Wb,n<name>`** sets the server machine name
- **`-Wb,c`** stops generation of client stubs (e.g. CEcho.c and CEcho.h).
- **`-Wb,s`** stops generation of server stubs (e.g. SEcho.c and SEcho.h)

Suppose there is an HTTP server running on port 8000 on a machine called www.ansa.co.uk. The following command creates the files CEcho.c, CEcho.h, SEcho.c, SEcho.h, Echo.html (the client stub, server stub[1] and HTML form):

```
idl -Wb,p8000 -Wb,nwww.ansa.co.uk Echo.idl
```

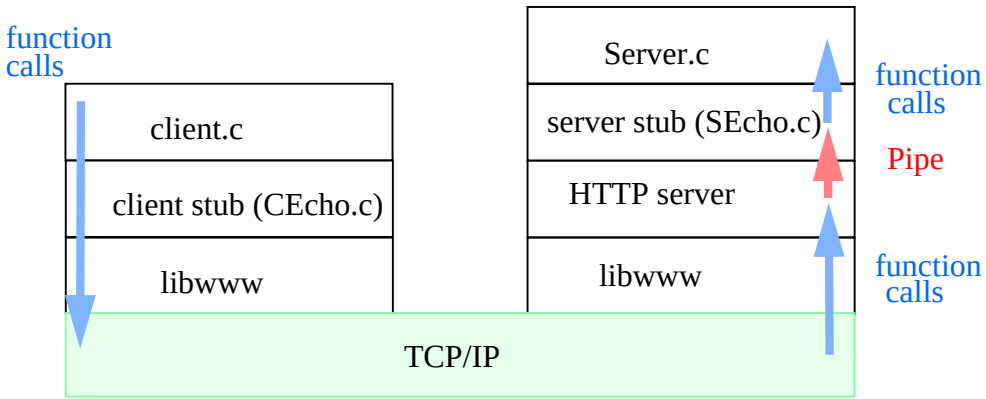
The `-Wb,p` and `-Wb,n` arguments are required unless generation of the html form is suppressed.

1. CORBA [OMG 93] refers to the client stub as the "IDL stub" and the server stub as the "IDL skeleton".

3.3 The client (client.c)

Figure 3.1 shows the arrangements of the various protocol stacks involved when a client program invokes a CGI program.

Figure 3.1: Protocol stacks for CGI invocation



The client stub consists of a stub library (see §5) and the client code generated by the stub compiler from the IDL file (e.g. CEcho.c). The library `libwww` is the standard WWW library used by many WWW applications (at the time of writing, the client uses version 2.17). This library is bundled with the distribution and will be linked with your client code by `make`.

One of the simplest clients we have written is the program client.c (in **`./ansa_test/Echo`**). Most of the code for this program is shown in figure 3.2 (the program invokes the other Echo operations, but these invocations are not shown). The program fragment shown receives a string from the user typing at the keyboard and invokes the Echo operation with the string as an argument, printing the result. The input loop is terminated when the user types control-D.The remainder of this section discusses the statements in figure 3.2 which directly affect the invocation. These statements are printed in bold.

The file **`CEcho.h`** contains the declarations of the operations in the Echo interface and also the types and operations needed by the stub library and libwww. This file is generated by the stub compiler from Echo.idl.

The variables **`HTAppName`** and **`HTAppVersion`** are global variables representing the client application name and version. These are passed to a remote HTTP server as part of the HTTP request.

The **`CORBA_Environment`** variable, **`ev`**, is used to handle exceptions (we will explain how in §3.8).

Note: The present implementation requires the initialisation which is rather ugly — there is no reason why this requirement could not be removed (other than lack of time).

The variable **`ref`** is an (unbound) reference to the remote resource. The type declaration of **`Echo`** is meant to indicate that the reference should only be bound to a resource of type Echo. The current implementation does not enforce this. The initialisation of **`ref`** to NULL is recommended, but not required.

The statement "**`ref = Stub_bind(SERVER, &ev);`**" binds ref to the resource instance by invoking various routines within libwww to convert the textual representation of the URL into something which can be used to invoke the

Figure 3.2: Source code fragment for client program

```
#include <CEcho.h>
PUBLIC char * HTAppName = "Echo client"; /* Application name */
PUBLIC char * HTAppVersion = "0";  /* Application version */

CORBA_string obuf;
char ibuf[1024];

int main(int argc, char* argv[]){
 CORBA_Environment ev = {CORBA_NO_EXCEPTION, 0, 0};
 Echo ref = NULL;
 int n;

 ref = Stub_bind(SERVER, &ev);
 handle(&ev);
 printf("Ctrl-D to end the input loop\n");
 printf("> "); fflush(stdout);
 while(n = read(0, (void *) ibuf, sizeof(ibuf))) {
      ibuf[n] = '\0';
      obuf = Echo_Echo(ref, &ev, ibuf);
      handle(&ev);
      printf("%s", obuf);
      Stub_free(obuf);
      printf("> ");
      fflush(stdout);
 }
/* Stuff here deleted*/
 HTAnchor_delete(ref);
}
```

resource. In the code shown, SERVER is defined at compile time to be the URL of the resource, for example:
<URL:http://www.ansa.co.uk:8000/cgi-bin/Echo>
Any valid URL would be perfectly legal in this statement, for example the following is legal:

```
ref = Stub_bind("http://outlaw.ansa.co.uk:8000/cgi-bin/Echo",&ev);
```

Note: It would be perfectly possible to build an implementation which did not require this statement, embedding the call to **`Stub_bind()`** within the call of **`Echo_Echo()`**. The only tricky part would be avoiding the overhead of calling Stub_bind each time the reference was reused. Lack of time has prevented us doing this or investigating how much overhead the operation imposes.

The call of **`handle()`** (described in §5.2.10) is the simplest way of handling an exception. If no exception has occurred the statement has no effect. If an exception has occurred **`handle()`** prints the exception and forces the program to **`exit()`**. In §3.8 we explore more sophisticated ways of handling exceptions.

The statement "**`obuf = Echo_Echo(ref, &ev, ibuf);`**" invokes the Echo operation in the remote resource. **`Echo_Echo()`** is an operation provided by the client stub, CEcho.c. It creates a buffer which will be used as the body of the HTTP request. Into this buffer, it marshals the CORBA_Environment variable **`ev`**, the operation name **`Echo_Echo`** and the parameter **`ibuf`**. It then uses various operations in libwww to pass the request to the server (establishing a TCP connection). Once the HTTP request has completed, the client stub

unmarshals the results from the result buffer. It first unmarshals the environment variable **ev**, and checks to see if an exception has occurred. If no exception has occurred (or if a system exception has occurred returning **COMPLETED_YES** or **COMPLETED_MAYBE** — see §3.8), the result is unmarshalled. In this case there are no **out** or **inout** parameters, so only the value to be returned and assigned to **obuf** needs to be unmarshalled. Finally the client stub garbage collects any buffers and **Echo_Echo()** returns.

Note: The HTTP method used for all requests is **POST**, which is overloading the **POST** method. Since HTTP supports the concept of extension methods a better solution would be to modify libwww to support user defined methods (such as **Echo_Echo**). We believe the modifications to libwww and CERN's HTTP server to support this would be trivial. The tricky part is getting a good mapping of HTTP's standard methods (e.g. GET, PUT, DELETE) into IDL [EDWARDS 95].

The last statement in figure 3.2, is a call to **HTAnchor_delete()**. This is a routine in libwww which garbage collects the **ref** data structure. It should only be called when **ref** will be no longer used.

3.4 The server (server.c)

Figure 3.1 shows the arrangements of the protocol stacks for a server receiving an invocation. The HTTP server (probably using libwww) receives the invocation and does some initial parsing of it. When it has determined that the invocation is intended for a CGI program, it forks the appropriate server program and writes the request buffer down a pipe to the server program. It reads whatever the server program writes to the output pipe and returns it to the client as the HTTP reply using the TCP connection.

The server program consists of the code written by the application programmer (e.g. **./ansa_test/Echo/server.c**) and the server stub. The server stub itself is made up of the stub library and the server code generated by the stub compiler (e.g. SEcho.c).

Figure 3.3 shows most of the source code for the Echo service which needs to be written by the application programmer (for brevity **Echo_Source()** and **Echo_Sink()** are omitted). The statements which are specifically concerned with driving the underlying protocol stacks are in bold font.

SEcho.h includes the definitions required by the underlying Stub library.

HTAppName and **HTAppVersion** are discussed in §3.3

The declarations of **Echo_Echo()** and **Echo_Reverse()** are very similar to the corresponding declaration in the IDL file. The main differences are:

- Inclusion of the environment variable **ev** in the parameter list;
- Prefixing of identifiers and type names by the scoping string Echo_ and CORBA_.

The only use for **ev** in the current implementation is throwing exceptions (see §3.8). The scoping string is applied to comply with the scoping rules defined in [OMG 93] for the IDL to C mapping[2].

Note: The above declarations are generated automatically by the stub compiler for the file **SEcho.h**; the stub compiler could be adapted easily to generate a template s**erver.c** file containing the appropriate operation declarations, time did not permit this.

2. As noted in chapter 5 of [OMG 93], this convention means that it is advisable to avoid the indiscriminate use of underscores for identifiers in IDL specifications.

Figure 3.3: Source code for Echo Server

```
#include <SEcho.h>
PUBLIC char * HTAppName = "Echo server"; /* Application name */
PUBLIC char * HTAppVersion = "0";  /* Application version */

static char buffer[32000];

CORBA_string Echo_Echo(CORBA_Environment *ev, CORBA_string Src){
 if(stub_browser) printf("Echo::\n%s", Src);
 return Src;
}

/*Echo_Source and Echo_Sink deleted*/

CORBA_string Echo_Reverse(CORBA_Environment *ev, CORBA_string Src)
 register char *sp;
 int n;

 for (sp = buffer, n = strlen(Src) - 1; n >= 0; n--)
   *sp++ = Src[n];
 *sp = '\0';
 if(stub_browser) printf("Result of Echo_Reverse\n%s", buffer);
 return buffer;
}
```

The **stub_browser** variable indicates whether the incoming invocation has been made from an ordinary browser using an HTML form or from a programmatic client such as that shown in figure 3.2. The two cases need to be treated differently: a browser expects HTML (or plain text etc.), a client like that shown in figure 3.2 expects to use its stubs to unmarshal the results. This approach has a number of imitations which are discussed in §3.11.

If **stub_browser** is true, the server stub does not marshal any results. Instead it expects the application to write HTML [CONNOLLY] to the standard output. This is the function of the **printf()** statements following the test of the **stub_browser** variable. The server stubs provide HTML wrappers so that the HTML source to the browser will look something like that shown in figure 3.4. Only the text between the <PRE> and </PRE> needs to be generated by the application.

Figure 3.4: HTML returned in reply to invocation of Echo_Echo

```
<HTML>
<HEAD>
<TITLE>Result for Echo</TITLE>
</HEAD>
<BODY>
<PRE>
Echo::
hello world</PRE>
</BODY>
</HTML>
```

Note: At the time of writing we have not determined whether having the server stub generate these HTML wrappers is useful. Future implementations may not generate any wrappers, or may generate different wrappers. §3.10 shows how to override the <PRE>..</PRE> wrappers.

If **stub_browser** is false, the request has come from a client program (like that shown in figure 3.2) rather than a browser. In this case the server does not print anything, rather the server stubs marshal any results into a buffer and write them into the standard output. The latter is redirected to a pipe connected to the HTTP server. Whatever it receives on its pipe, the HTTP server passes to the client program over their TCP connection. Anything that is written to standard output (e.g. using **printf()**) will be returned to the client which will try to unmarshal it as a result. This means that programmers should not write to the standard output if the **stub_browser** variable is false, otherwise the client marshalling routines may get confused[3].

All the CGI environment variables as specified in [McCOOL] are available to server programmers. These can be accessed within a C program in the normal way (i.e. using the **getenv()** routine).

You should always keep in mind that your programs are stateless: a new process is forked to service each invocation. Any state which needs to be stored must be stored externally (either in the third party application to which your CGI application is acting as a gateway, or in the local file system). Thus although CORBA attributes are supported, it does not make sense to try to set an attribute, unless you implement a means of storing and retrieving state between invocations. The SimpleBank example shows how to do this (§3.10).

When the CGI program is forked (e.g. the Echo server) it examines the **Accept** header fields of the HTTP request.These are contained in the **HTTP_ACCEPT** environment variable. The client stubs will include the media type **application/x-corba** in this header, ordinary browsers will not. Thus if the server stub finds the media type **application/x-corba** is acceptable, it sets **stub_browser** to false and writes the string "**Content-type: application/x-corba**" into the output pipe (machine parseable results will follow). Otherwise **stub_browser** is set to false and the stub writes into the output pipe the string "**Content_type: text/html**" followed by the HTML preamble shown in figure 3.4 up to and including the element **<PRE>**.

Next the server stub reads the input data from the input pipe and places it in a buffer. Unmarshalling can now begin. First it unmarshals the environment variable, **ev**, which is used to throw an exception if an error occurs. This is the CORBA_Environment variable as specified in [OMG 93] which is used in exception handling. It should not be confused with UNIX environment variables.

Next the operation name is unmarshalled and compared with a table of operation names supported by the interface. If a match is found (e.g. **Echo_Echo**) the remainder of the buffer is passed to the routine responsible for dispatching that operation e.g. **S_Echo_Echo**. This routine unmarshals the parameters from the buffer and invokes the application level operation **Echo_Echo**.

When **Echo_Echo** returns the action taken by the **S_Echo_Echo** is determined by the **stub_browser** variable. If it is true it does nothing (expecting the program to have written to the standard output using printf()), if it is false,

3. Currently the client side marshalling routines use colons (":") to delimit fields, so provided no colon characters are written to the standard output, the client side marshalling routines will ignore whatever is in the **printf()** statements. This is sometimes useful in debugging: the **Stub_dump_buffer()** routine can be called from within the client stub to dump the entire contents of a buffer including whatever was in the **printf()** statements. This is discussed in §5.6.6.

the environment variable and results are marshalled into a buffer and then the entire contents of the buffer written into the output pipe.

S_Echo_Echo then returns. If **stub_browser** is true the server stub writes the HTML post-amble into the output stream starting from the element **</PRE>**. Finally the server CGI program **exits()**.

CGI programs are installed by **make** in the directory indicated by the **make** variable **SCRIPT_DIR** in the file **./ansa_make_vars.**

3.5 The ".html" file — and HTML forms client

The HTML form created by the stub compiler from the Echo IDL definition is shown in figure 3.5. This form can be used to drive the corresponding CGI server program by using an ordinary browser such as Mosaic[4].

Figure 3.5: HTML form fragment generated from Echo.idl

```
<head>
<TITLE>Input for Echo</TITLE>
</head>
<BODY><H1>Input for Echo</H1>
<HR>


<H2> Operation Echo</H2>
<FORM METHOD="POST" ACTION="http://www.ansa.co.uk:8000/cgi-bin/Echo">
<P>Enter arguments here:<P>
<INPUT NAME="Operation" TYPE=hidden VALUE="Echo_Echo">
CORBA_string Src: <INPUT SIZE=10 NAME="Src"> <P>
To invoke Echo_Echo: <INPUT TYPE="submit" VALUE="Echo_Echo"><P>
</FORM>
<HR>

-- Sink and Source operations deleted --

<H2> Operation Reverse</H2>
<FORM METHOD="POST" ACTION="http://www.ansa.co.uk:8000/cgi-bin/Echo">
<P>Enter arguments here:<P>
<INPUT NAME="Operation" TYPE=hidden VALUE="Echo_Reverse">
CORBA_string Src: <INPUT SIZE=10 NAME="Src"> <P>
To invoke Echo_Reverse: <INPUT TYPE="submit" VALUE="Echo_Reverse"><P>
</FORM>
</BODY>
```

The URLs embedded in the form are set by the **-Wb,p** and **-Wb,n** flags given to the stub compiler as command line arguments. The HTML forms will be installed by **make** in the directory indicated by the **make** variable **FORMS_DIR** in the file **./ansa_make_vars**.

Any form generated by the stub compiler should be regarded as a template only. That is you should consider editing it, adding text to help the user of the form to decide what inputs to provide. You may also need to adjust the size of the input windows. **./ansa_test/SBank** contains some examples of edited template forms.

Finally note that the stub compiler will only generate forms for operations with simple data types as parameters "in" and "inout" parameters. This is because humans are unlikely to type complicated data types correctly. Currently the data types supported are: long, short, unsigned long, unsigned short, float, double, char, boolean, octet, string and enum. In addition any

4. Mosaic is a trademark of the University of Illinois.

typedef resolving to one of the above types is allowed and operations returning the value void are allowed. Use of any other data types will result in suppression of form generation for that operation. This does not apply to “out” parameters — arbitrary data types are allowed.

Note: The list of acceptable data types is probably too generous, in particular humans are not very good at typing floating point numbers into an HTML form! The structure and layout of the generated form may change as we get more experience with using the stub compiler.

3.6 Inheritance

Inheritance as defined in [OMG 93] is supported. An example of its use is the interface simplest which is defined in **./ansa_test/simplest/simplest.idl**. The definition is shown in figure 3.6.

Figure 3.6: The use of inheritance in an interface definition

```
#ifndef SIMPLEST_IDL
#define SIMPLEST_IDL
#include “struct.idl”
interface simplest: foo{
  Long inc(in long a);
  long div(in long a, inout long b);
};
#endif //SIMPLEST_IDL
```

The interface `simplest` inherits the interface foo. The latter is defined in the file struct.idl, hence the need for the #include statement. The interface `simplest` is itself inherited by a number of other interfaces, so the #ifndef and associated statements are needed to avoid problems arising from multiple inclusions.

To tell the stub compiler where to find this include file, the following declaration is added to **./ansa_test/simplest/Makefile:**

```
IDL_INCLUDES= -I../Struct
```

The CGI server program for `simplest` needs to implement all the operations declared in the interface foo.

Note: We depart from the CORBA specification slightly by prefixing inherited operations with the name “**foo**_”; the CORBA specification says it should be prefixed with “**simplest**_”. For example the file **./ansa_test/simplest/server.c** contains an operation inherited from foo called **foo_op3**; [OMG 93] says it should be called **simplest_op3**. Time has prevented us from fixing this.

3.7 Passing URLs as arguments

URLs can be passed as arguments; this allows a CGI program to invoke another CGI program potentially run by a different HTTP server. An example of this is contained in the directory **./ansa_test/TypeTest/**.

The main difficulty is that the CGI program needs to be linked with both the client and the server stubs. At present this can require manual editing of the client stubs to avoid name clashes over local and remote operation names. This is only needed if the object is also a client of objects of the same type.

For example, the CGI program **TypeTest** invokes an operation called **TypeTest_string_op** on another (remote) CGI **Typetest** program. To do this it calls a local operation in its client stub called **RTypeTest_string_op()** instead of **TypeTest_string_op()**. If it tried to invoke an operation called **TypeTest_string_op()** the compiler would interpret this a local invocation of its own operation of the same name. This would generate a compilation error (the remote invocation takes an extra argument — the reference to the resource).

3.8 Handling Exceptions

The standard CORBA IDL to C mapping for user and system exceptions is supported. A number of routines are provided by the stub library to make handling exceptions easier; these are described in §5. Support for exceptions is provided by using setjmp(), allowing programmers to halt execution of an operation immediately. The directory **./ansa_test/exceptions** contains some examples of handling both user and system exceptions. In this directory the file **simple.idl** contains the following fragment:

```
exception simple_error {
     long reason;
};
interface simple {
     long op(in long a, out simple b, inout long c)
          raises (simple_error);
};
```

The above defines a user exception “simple_error” and a single operation “op” which can return the exception. It is usual to return an exception only if an error has occurred, normally the operation returns a long. Exceptions are mapped to C structures, so the above exception declaration generates the following declaration in the header files:

```
#define ex_simple_error “ex_simple_error”
typedef struct simple_error {
 CORBA_long reason;
} simple_error;
```

The string “ex_simple_error” can be thought of as the exception’s type code, it is used to determine the type of the returned exception.

Variables of type **CORBA_Environment** are used to carry exceptions information:

```
typedef struct CORBA_Environment{
 CORBA_unsigned_long _major;
 CORBA_void *_exception;
 char *_id;
 jmp_buf env;
}CORBA_Environment;
```

The **_major** field can take one of three values: **CORBA_NO_EXCEPTION**, **CORBA_SYSTEM_EXCEPTION** and **CORBA_USER_EXCEPTION**. If the **_major** field indicates an exception has occurred, the **_id** field will contain the “type code” for the exception and **_exception** points to an area of memory containing the value of the exception.

The implementor of **op()** could raise the exception **simple_error** by writing the following code:

```
static simple_error my_excp2= {55};
.
.
CORBA_long simple_op(CORBA_Environment *ev, CORBA_long a,
                     simple *b, CORBA_long *c){
.
.
 if (error_condition)
     CORBA_throw_immediate(ev, &my_excp2, ex_simple_error);
 /*never get here if the exception is raised*/
 *c = 100;
 *b = Stub_bind(SERVER, ev);
 return a;
}
```

The function **CORBA_throw_immediate()** is described in §5.2.6, it takes the exception and its "type code", attaches them to the **CORBA_environment** variable **ev** and uses **setjmp()** to halt execution and raise the exception. The client programmer can handle this exception with the following code (the functions **CORBA_exception_id()** and **CORBA_exception_value()** are respectively the data accessor functions for the **_id** and **_value** fields of **CORBA_Environment** — see §5.2.2 and §5.2.3):

```
void handle_exception(CORBA_Environment *ev){
 switch(ev->_major){
 case CORBA_NO_EXCEPTION:
     break;
 case CORBA_SYSTEM_EXCEPTION:
     .
     .
     break;
 case CORBA_USER_EXCEPTION:
     if (strcmp(ex_simple_error, CORBA_exception_id(ev)) == 0){
     simple_error *q = (simple_error *)CORBA_exception_value(ev);
     fprintf(stderr, "operation failed reason: %i\n", q->reason);
     exit(-1);
     }
     .
     .
     break;
 default:
      fprintf(stderr, "Unknow value of _major %i\n", ev->_major);
      exit(-1);
      break;
 }
 CORBA_exception_free(ev);
}

int main(int argc, char* argv[]){
 .
 .
 result = simple_op(ref, &ev, a, &b, &c);
 handle_exception(&ev);
```

3.9 Use of system exceptions in stubs and the library

All CORBA system or standard exceptions defined in [OMG 93] correspond to the following structure:

```
typedef struct CORBA_std_ex{
     CORBA_unsigned_long _done;
     unsigned long _minor;
}CORBA_std_ex;
/*Permissible values of _done*/
#define CORBA_COMPLETED_YES 1
#define CORBA_COMPLETED_NO 2
#define CORBA_COMPLETED_MAYBE 3
```

The **_done** field indicates whether or not a request has been completed successfully. For example, if the exception occurs in the client stub before the request is sent to the server it should be set to **CORBA_COMPLETED_NO**.

Each module in the stub library defines an exception with a **_minor** field which is unique within the stub library.

For example the file **./ansa_stubs/src/Cmarshall.c** contains the following declaration:

```
/*Standard exception return for this module*/
static CORBA_std_ex excp = {CORBA_COMPLETED_NO, 3};
```

The corresponding declaration in the file **./ansa_stubs/src/util.c** is:

```
/*Standard exception return for this module*/
static CORBA_std_ex excp = {CORBA_COMPLETED_NO, 5};
```

Within the above two modules the following two statements are used to raise the standard exceptions **NO_MEMORY** and **MARSHALL**:

```
CORBA_throw_immediate(ev, &excp, NO_MEMORY);
CORBA_throw_immediate(ev, &excp, MARSHAL);
```

By reading the value of the **_minor** field you can tell in which module the exception has been raised. The stub library **handle**() function can be used to do this (see §5.2.10). If a system exception occurs it prints the name of the exception and the value of the **_minor** field before executing **exit()**.

The stub compiler tries to generate a different **_minor** field for each client and server stub by keying off a process identifier. For example the file CEcho.c will contain four declarations like the following:

```
static CORBA_std_ex excp = {CORBA_COMPLETED_NO, 29898};
static CORBA_std_ex excp1 = {CORBA_COMPLETED_NO, 29899};
static CORBA_std_ex excp2 = {CORBA_COMPLETED_MAYBE, 29900};
static CORBA_std_ex excp3 = {CORBA_COMPLETED_MAYBE, 29901};
static CORBA_std_ex excp4 = {CORBA_COMPLETED_MAYBE, 29902};
```

Different parts of the CEchco.c module will raise a different one of the above exceptions, so you can use the **handle()** function to help identify which line of code generated the exception.

3.10 Stateful interaction — the SimpleBank Application

The SimpleBank is the canonical ANSA example which we have used to evaluate a number of CORBA offerings (commercial and public domain). The code for SimpleBank is contained in **`./ansa_test/SBank`**.

The standard distributed object technique is to implement an SBank object which has a single operation “Access”. The customer supplies a PIN and Account Number, if these are valid the SBank object returns an object reference to the customer’s account. The customer can then invoke `debit()`, `credit()` and `list()` operations on it.

Implementing this in WWW is hard since it does not really support distributed objects. If we choose to regard an account as state plus functions to access that state, creating an account would involve cloning a CGI “account” program at a particular URL e.g. <URL:http://www.ansa.co.uk/accounts/nje> and creating some state for that program (the initial balance) somewhere on the file system. This would be extremely inefficient. (Since all CGI programs are stateless they have to store their state externally — in the local file system.)

A better method is to implement the bank's state as a file system (for large amounts of state a database would offer better performance), and implement a CGI program which operates on this state.

The management operations present no conceptual problems: operations are implemented to `create`, `delete` and `display` the state of the bank.

The operations available to customers are more interesting. In our Orbix™ implementation the Access operation returns an object reference to an account object (provided the account number and PIN are valid)[5]. The WWW version returns an HTML form which has the PIN and Account number embedded in them as hidden parameters. This means that customers do not have to provide their PIN and account number for each operation, but the CGI program has them available as hidden parameters which it uses to check that the access is authorised.

For comparison the Account interface for the Orbix implementation was defined thus:

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

For the WWW implementation it was changed as shown below:

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

Thus the WWW implementation will return nothing to the bespoke client teller application, which should remember the PIN and account number for

5. Orbix is a Registered Trademark of Iona Technologies Ltd.

the customer, so that they do not have to type it in for each operation. In contrast, the HTML form returned by this operation is generated on the fly and has the account number and PIN embedded in it as hidden parameters, so it can be used only to access a particular account. The form embedded in the code (**server.c**) was generated by editing the template form, **SimpleBank.html**, generated by the stub compiler. Note that the generation of the form begins by generating </PRE> and ends with <PRE> to override the default formatting generated by the stub compiler.

The directory **./ansa_test/SBank** contains three HTML files: **Bank.html Manager.html Teller.html**. The files **Manager.html** and **Teller.html** were generated by editing the template HTML file, **SimpleBank.html**, generated by the stub compiler. The file **Bank.html** is the entry point for the application. It contains links to the other HTML files which drive these applications. **You will need to edit these files to change the URLs to point at your local web server (they are not created at installation time).** In addition the bank creates persistent state and uses lock files in directory **/tmp**. If you want to change this you will need to edit the `#define state` and `#define lock` statements in **server.c**. Build the application once you have done this (e.g. `make; make install`). Now accessing **Bank.html** should allow you to drive the demonstration.

This application was developed in a very short time (a few hours). The hardest part was dealing with reading and writing the state of the bank onto the local file system and creating lock files to prevent corruption in the event of concurrent access. We did not implement any bespoke client applications for this demonstration.

3.11 Limitations and known bugs

This section details some of the limitations and know bugs of the current implementation, and describes points where we have deviated from [OMG 93]. Some of these limitations are intrinsic to the differences between the CORBA computational model and the stateless execution associated with CGI. Other limitations could be removed given more time.

The type `ANY` is not supported.

The implementation does not support cannot exceptions with no members: e.g. `exception NoSuchAccount {};` will not work

CORBA context objects are not supported, they could be used to present the CGI environment variables currently accessible using **getenv()**.

Support for arrays caused some difficulty since C cannot return an array from a function. If a single dimensional array is defined to be the result of the operation, then a pointer to the first element of the array is returned, rather than the array itself. Returning multi-dimensional arrays is not supported.

There is no dynamic allocation of memory for arrays passed as parameters since IDL requires the dimensions of all arrays to be fixed. Memory for arrays returned as operation result values is dynamically allocated and must be freed by the programmer when no longer required.

As discussed in §3.6 the scoping of names of inherited operations is not compatible with that described in [OMG 93].

All strings and sequences are treated as unbounded. When returning sequences or strings as **inout** or **out**, a new storage area is allocated which is at least as big as the size of the sequence or string. The programmer is responsible for freeing the old buffer area (used before the invocation).

There is no concept of an ORB pseudo object providing **object_to_string()** and **string_to_object()** operations.

Note: It could be argued that the **Stub_bind()** operation (§5.3.3) does this since it takes an ASCII URL and converts it to something useable by libwww.

Operational semantics specifications are ignored. Semantics are built on top of TCP (which is what the implementation of HTTP in libwww uses). The semantics are: exactly once if no exception is raised; at most once if an exception is raised.

There is a limit on how much data can be sent and received in a single invocation. If you try and send more than about 24KBytes you are likely to see the error message “Broken pipe”. We have not had time to investigate this problem.

The value of the stub_browser variable is driven off the accept headers sent to the server by the client. If the accept header indicates that the client can accept media of type `application/x-corba`, stub_browser is set to false and the server stubs will marshal the results for the client. If it is set to true the server stubs will make no attempt to marshal the results. Instead they will output an HTML wrapper (media type `text/html`). Everything the programmer writes using `printf` is wrapped inside `<PRE> .. </PRE>`.

You cannot output your own `<HEAD> .. </HEAD>` tags — the stubs do it for you, generating default headers. There is no easy way to override this.

This means that support for two media types is “hardwired” in and makes it more difficult to support different media types.

We have not explored alternative, more flexible, strategies such as generating no HTML at all by default, but generating functions would could be called to output reasonable defaults for the interface e.g. `<TITLE>Result for SimpleBank</TITLE>` as the default title.

4 The Stub Compiler

The purpose of this section is to describe the architecture and the main concepts of the stub compiler, so that you can customise the back end for your requirements. If you wish to modify the front end, you are referred to SunSoft's documentation in directory **./docs/** — please take note of the conditions listed in the file **./docs/README**.

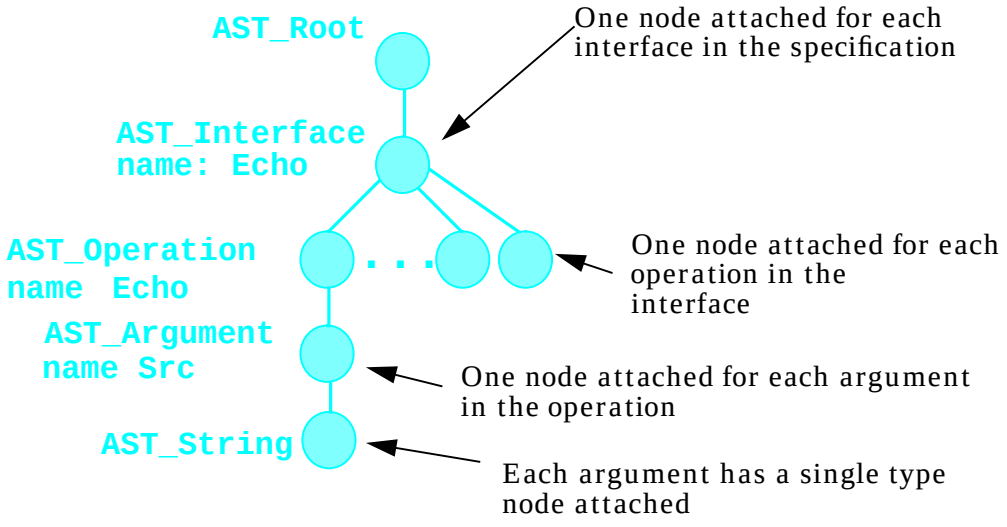
4.1 The main concepts

Consider the IDL specification of the Echo interface:

```
interface Echo{
 string Echo(in string Src);
 void Sink(in string Src);
 string Source(in long Length);
 string Reverse(in string Src);
};
```

From this definition the stub compiler uses the UNIX utilities lex and yacc to generate an Abstract Syntax Tree or AST as shown in figure 4.1. Each node in the abstract syntax tree is an instance of a C++ class.

Figure 4.1: Part of the abstract syntax tree for the Echo interface



4.2 The Compiler back end

The front end of the compiler generates the AST. The job of the back end is to walk the AST generated by the front end dumping the procedures and declarations required by each node. Its implementation is in the directory

./be. This directory contains a number of subclasses of the classes defined by the front end (e.g. **be_interface** for **AST_Interface**, **be_operation** for **AST_Operation** and **be_argument** for **AST_Argument**).

If the back end has a subclass for one of the front end's classes, the front end attaches an instance of the subclass into the generated AST rather than an instance of the superclass (hence **be_interface** appears in the AST rather than **AST_Interface**). The subclasses contain routines to dump the header, stub and HTML files for that class and every node attached in the abstract syntax tree below it. For example, the implementation of **be_interface** contains the following routines:

```
void be_interface::dumpServerStub(ostream &o);
void be_interface::dumpClientStub(ostream &o);
void be_interface::dumpIncludeFile(ostream &o, idl_bool server);
void be_interface::dumpHTML(ostream &o, char *name);
```

The method **dumpServerStub()** will invoke the **dumpServerStub()** method on each **be_operation** node attached to the interface. Thus by invoking **dumpServerStub()** on each node attached to the root of the AST, the server stub file is created (e.g. SEcho.c).

The back end routines which initiate the dump of each file generated by the stub compiler are contained in **./be/be_produce.cc**:

```
BE_mk_server_side()
BE_mk_header()
BE_mk_client_side()
BE_mk_html_side()
```

These are all called from the routine **BE_produce()**.

Finally, the file **./be/be_util.cc** contains many general purpose routines used throughout the back end. In particular it contains the routines which determine the type of a declaration and dump the appropriate calls to unmarshal or marshal that type:

```
BE_Smarshall();
BE_Sunmarshall();
BE_Cmarshall();
BE_Cunmarshall();
```

5 Stub Library

This chapter gives a brief description of the routines in the stub library. The sources for the stub library can be found in the directory **./ansa_stubs/src** and the include files in **./ansa_stubs/include**.

5.1 Smarshall.c, Cmarshall.c, Sunmarshall.c and Cunmarshall.c

These files contain the marshalling and unmarshalling routines for the primitive data types.

Clients and servers use different marshalling routines because the data encoding for requests and responses is different. The format of a request is exactly the same as one which would be generated by an ordinary browser invoking the POST method from an HTML form.

A more human readable format is used for the reply, so that humans can read and understand the contents of buffers without having them unmarshalled.

Note: This is of some benefit in debugging (§5.6.6), but the cost is efficiency. We are still debating whether the costs outweigh the benefits.

5.2 exceptions.c

Variables of type **CORBA_Environment** are used to carry exceptions information:

```
typedef struct CORBA_Environment{
 CORBA_unsigned_long _major;
 CORBA_void *_exception;
 char *_id;
 jmp_buf env;
}CORBA_Environment;
```

The **_major** field identifies whether the structure contains a valid exception, and, if it does contain an exception, whether it is a system exception or a user exception. It can take one of the following values: **CORBA_NO_EXCEPTION**, **CORBA_SYSTEM_EXCEPTION** and **CORBA_USER_EXCEPTION**. The **_id** field can be thought of the exception identifier string: it identifies the type of the exception stored in the **_exception** field.

5.2.1 exception_sys()

CORBA_boolean CORBA_exeception_sys(char *id);

This routine returns TRUE if the supplied string corresponds to an exception identifier string for a standard or system exception as defined [OMG 93]. Otherwise it returns FALSE.

The file **./ansa_stubs/include/exceptions.h** contains the definitions of all the system or standard CORBA exceptions.

5.2.2 exception_id()

```
CORBA_char *CORBA_exception_id(CORBA_Environment *ev);
```

This is a data accessor function for **CORBA_Environment**, if an exception has occurred it will return the exception identifier string (the **_id** field of **CORBA_Environment**). Otherwise it returns a NULL pointer.

5.2.3 exception_value()

```
CORBA_void *CORBA_exception_value(CORBA_Environment *ev);
```

This is another data accessor function for CORBA_Environment. If an exception has occurred it returns a pointer to the region of memory containing the exception (the **_exception** field of **CORBA_Environment**). Otherwise a NULL pointer is returned.

5.2.4 exception_free()

```
CORBA_void CORBA_exception_free(CORBA_Environment *ev);
```

This routine frees the memory used in the **_id** and **_exception** field of **ev**.

5.2.5 exception_throw()

```
CORBA_void CORBA_exception_throw(CORBA_Environment *ev,
                                 CORBA_void *exception,
                                 char *id,
                                 CORBA_boolean immediate);
```

This routine attaches the supplied **exception** and **id** parameters to the **_exception** and **_id** field of **ev**. The **_major** field is set to indicate whether it is a system or user exception. Finally if immediate is set to TRUE the routine executes **setjmp**(). This has the effect of ending the execution of the request immediately.

5.2.6 throw_immediate()

```
CORBA_void CORBA_throw_immediate(CORBA_Environment *ev,
                                 CORBA_void *exception,
                                 char *id);
```

Calls **CORBA_exception_throw()** with immediate set to TRUE.

5.2.7 throw_delayed()

```
CORBA_void CORBA_throw_delayed(CORBA_Environment *ev,
                               CORBA_void *exception,
                               char *id);
```

Calls **CORBA_exception_throw()** with immediate set to FALSE.

5.2.8 exception_continue()

```
CORBA_boolean CORBA_exception_continue(CORBA_Environment *ev);
```

Returns TRUE if no exception has occurred or if a system exception has occurred with _done set to **CORBA_COMPLETED_YES** or **CORBA_COMPLETED_MAYBE**. This is used in the client stubs to determine whether or not it makes sense to continue unmarshalling after they have unmarshalled an exception.

5.2.9 exception_print()

```
CORBA_void CORBA_exception_print(CORBA_Environment *ev);
```

If the **_major** field of **ev** indicates that an exception has occurred this routine will try to print relevant information. For a system exception it will print the exception identifier string, the value of the **_done** field and the integer returned in the **_minor** field. For a user defined exception it is only possible to print the exception identifier string (the stub library does not know about the structure of user defined types).

5.2.10 handle()

```
CORBA_void handle(CORBA_Environment *ev);
```

This is a convenience routine to allow application programmers to handle exceptions without writing code of their own. This routine uses **CORBA_exception_print()** to print the exception. If **CORBA_exception_continue()** returns false it calls **exit()**, otherwise it calls **CORBA_exception_free()** and returns.

5.3 util.c

This module provides several utility functions used by client and server stubs.

5.3.1 Stub_allocate() and Stub_free()

```
#define Stub_allocate(n) malloc(n)
#define Stub_free(p) free(p)
```

These definitions are hooks for a future release which will provide better memory management.

5.3.2 Stub_escape ()

```
char *Stub_escape (char *part, CORBA_Environment *ev);
```

This routine is adapted from an early version of Mosaic. It takes the supplied string and escapes any unsafe characters. Unsafe characters are defined in [BERNERS-LEE 95].

Strictly, this routine is no longer necessary, as “unsafe” characters are only unsafe if they appear in the HTTP headers. The client stubs use the HTTP POST method and send parameters for the CGI program in the body of the HTTP request, rather than in the header (where they would need to be escaped).

However, browsers will escape all data whether or not they using POST to send the parameters from a form to a server. So if a client did not escape the data it would no longer be compatible with current browser practice and we would have to test explicitly whether or not the data was escaped or not. Both

ServerStub_subplus() and **ServerStub_subhex()** (§5.5) are needed to undo the effects of **Stub_escape()**.

5.3.3 Stub_bind()

```
HTParentAnchor* Stub_bind(char* address,
                          CORBA_Environment *ev);
```

This routine takes the URL supplied as **address** and converts it into a reference to the resource which can be used by libwww.

5.3.4 Stub_request_new()

```
HTRequest * Stub_request_new(CORBA_buffer* buffer,
                          CORBA_Environment *ev);
```

This routine takes a buffer and returns an HTTP request, it is used by client stubs. In particular this routine includes the media type **application/x-corba** in the HTTP accept headers and registers the routine **HTCORBAConvert()** (§5.4.1) with libwww. If libwww receives a response of media type **application/x-corba**, it will call **HTCORBAConvert()** to create a stream object in which to write response (see **HTCorba.c** — §5.4).

The type definition for HTRequest is in the following file:
./ansa_stubs/WWW/Library/Implementation/HTAccess.h

5.3.5 Stub_request_free()

```
void Stub_request_free(HTRequest* request,
                    CORBA_Environment *ev);
```

This routine is called by client stubs after sending an HTTP request to free any memory used by it.

5.4 HTCorba.c

This is an implementation of a stream class into which libwww writes responses of type **application/x-corba**.

5.4.1 HTCORBAConvert()

This is called by libwww to create and initialise the stream object. The returned stream consists of a buffer and a dispatch table containing routines which libwww uses to write into this buffer. These routines are **HTCORBA_free()**, **HTCORBA_abort()**, **HTCORBA_put_character()**, **HTCORBA_put_string()**, **HTCORBA_write()**.

5.4.2 HTCORBA_put_character(), HTCORBA_put_string(), HTCORBA_write()

These routines use **Stub_buffer_write()** defined in **buffer.c** (§5.6) to write into the **HTStream** object's buffer respectively a character, a string and a block of text.

5.4.3 HTCORBA_free()

This routine is called by libwww when it has finished writing the HTTP response into the **HTStream** object. It frees memory associated with the **HTStream** object and attaches the object's buffer to the global variable **HTCorbaResult.buff**. Thus the buffer is made accessible to the client stubs for unmarshalling.

5.4.4 HTCORBA_abort()

This routine frees all memory used by HTStream including the buffer and sets the global variable **HTCorbaResult.buff** to NULL. The client stubs will raise an exception when they encounter this variable set to NULL.

5.5 ServerStub.c

This module contains the routine **ServerStub_split()** which is used by server unmarshalling routines to preprocess the buffer before unmarshalling. The other routines in this module are used by **ServerStub_split()**.

5.5.1 ServerStub_subhex ();

```
char * ServerStub_subhex (char *value, CORBA_Environment *ev);
```

This routine returns a string in which certain characters from the supplied string, **value**, are "unescaped". For example, the string fragment "%3D" is replaced by the character "=". This restores characters which were escaped by the client using the routine **Stub_escape()** (§5.3.2). The caller is responsible for freeing the returned string. Both **ServerStub_subhex()** and **ServerStub_subplus()** are needed by to undo the effects of **Stub_escape().**

5.5.2 ServerStub_subplus()

```
extern void ServerStub_subplus(char *s2);
```

This routine substitutes all occurrences of '+' in **s2** with ' '. This is the inverse of what occurs in **Stub_escape()**.

5.5.3 ServerStub_split()

```
int ServerStub_split(char *input, const unsigned long length,
                     unsigned long *i, char **s1, char **s2,
                     CORBA_Environment *ev);
```

The client marshals data in the following format:

an_1=av_1&an_2=av_2&....&an_n=av_n

Where an_m is the name of the m^{th} argument and av_m is its value.

The arguments to this routine are: the buffer containing the data to be unmarshalled, **input**; the length of the buffer, **length**; and the current position in the buffer, **i**. It splits off the next argument name, value pair an_m, av_m and returns unescaped copies (using **ServerStub_subhex()** and **ServerStub_subplus()**) in the strings **s1**, **s2**.

The caller is responsible for freeing the memory allocated to **s1** and **s2**.

5.6 buffer.c

This module provides the buffer management routines. Buffers used by the client and server stubs have the following structure.

```
struct _CORBA_buffer{
     char * data; /*The start of the buffer*/
     char * next; /*Next free data slot*/
     long len; /*Length of buffer*/
     long free; /*Free space in buffer*/
};

typedef struct _CORBA_buffer CORBA_buffer;
```

Note: The buffer received by a server stub to be unmarshalled prior to making an invocation has a slightly different structure. This is handled as a pointer to an area of memory, a length and index variable. A future release could remove this anomaly.

5.6.1 Stub_buffer_new()

```
CORBA_buffer* Stub_buffer_new(CORBA_Environment *ev);
```

This returns a new buffer. The data storage area (**data** field) is set to NULL, the buffer will be extended the first time it is used.

5.6.2 Stub_buffer_write ()

```
void Stub_buffer_write (CORBA_buffer* buff, char* s, int m,
                        CORBA_Environment *ev);
```

This copies **m** bytes starting at **s** into the buffer. If there is insufficient room in the buffer it is extended.

5.6.3 Stub_buffer_extend ()

```
void Stub_buffer_extend (CORBA_buffer* buff, int m,
                         CORBA_Environment *ev);
```

This extends the buffer by at least **m** bytes.

5.6.4 Stub_buffer_free()

```
void Stub_buffer_free(CORBA_buffer* buff,
                      CORBA_Environment *ev);
```

This reclaims all memory used by the buffer.

5.6.5 Stub_dump_buffer()

```
void Stub_dump_buffer(CORBA_buffer* buff,
                      CORBA_Environment *ev);
```

This is a utility function for dumping the contents of the buffer into **stdout**. It is not used in the stub library, but is sometimes useful for debugging. For example, a call to it could be inserted in a client stub after an operation has been invoked. This would dump the contents of the buffer on the screen before

it is unmarshalled by the client stubs. Since everything is marshalled as ASCII, the contents should be understandable (with some effort).

5.6.6 Stub_dump_data()

```
Stub_dump_data(CORBA_char *p, CORBA_long l,
               CORBA_Environment *ev);
```

This does the same job as **Stub_dump_buffer()**, but for an arbitrary area of memory. It can be used by in the server stub to dump the contents of the received buffer. Since the output is dumped into **stdout** it will be returned to the client and can be viewed on the screen by calling **Stub_dump_buffer()** at the client.

The data dumped by **Stub_dump_data()** will be at the beginning of the buffer received by a client in response to an invocation. Assuming it is used to dump the contents of the buffer received by the server stubs, it will not affect the clients ability to unmarshal the results. This is because the client unmarshalling routines expect data fields to be separated by ':' characters. The data dumped by **Stub_dump_data()** will not contain a ':' — such characters are defined to be unsafe in [BERNERS-LEE 95] and are removed by the client stubs using the **Stub_escape()** routine (§5.3.2). Thus any data dumped by **Stub_dump_data()** will be ignored by the client's unmarshalling routines.

Note: This is a benefit of using different formats for request and response data.

5.6.7 Stub_buffer_endline(), Stub_buffer_terminate(), Stub_buffer_string()

```
#define Stub_buffer_endline(b, ev)\
     Stub_buffer_write(b, "\n", 1, ev)
#define Stub_buffer_terminate(b, ev)\
     Stub_buffer_write(b, "\n\0", 2, ev)
#define Stub_buffer_string(b, s, ev)\
     Stub_buffer_write(b, s, strlen(s), ev)
```

These are macros used by server stub routines.

5.7 GridText.c and DefaultStyles.c

To learn how to write a client program using libwww, we started by studying the CERN line-mode browser. These files are inherited from the CERN line mode browser. Much of the code they contain is no longer used, but time has prevented removing the unnecessary code.

The file GridText.c contains the callback routines required by libwww to create a hypertext object. This is not used in any of the examples in this distribution, but removing it would cause the linker to complain about undefined symbols.

6 Conclusions

Deploying new and customised applications and services in the Internet is difficult. Yet many organisations require more than the basic functionality provided by standard applications such as HTTP servers (WWW servers) (see [EDWARDS 95]).

The tool kit described in this document demonstrates how it can be made relatively easy to program applications in WWW which use libwww and, in particular, CGI and HTTP. By using the concept of client and server stubs we have provided a layer of abstraction which protects the programmer from the underlying details of the protocols. This means that programmers have less work to do to write their applications and because they have less code to write, they are less likely to make mistakes.

The programming language we used was C, this because when we started the work C was more widely used in WWW than alternatives such as C++. In the future it is possible that C++ will displace C as the most widely used WWW programming language.

So far we have not had time to try the tools described in this document on a significant example. Further application experience is needed to determine how useful these tools really are and what changes could be made to make them more useful.We note that the concepts of stub compilers and stubs have proved extremely useful in other platforms (see [EDWARDS 95]).

There are a number of possible directions for future work of which the most interesting would be to turn HTTP into a genuine extensible RPC protocol. As noted in §3.4, we believe it would be relatively easy to adapt libwww to support the notion of extension methods and to change an HTTP server to execute these extension methods as CGI programs.

The new HTTP methods could still be driven by HTML form technology, but care will be needed to make sure that the return type is something which a browser understands how to display.

The above does not solve the problem of stateful interaction which is sometimes required. The only way of handling this at present is storing state externally from the HTTP server and CGI program, usually on the local file system. This could be solved by incorporating the extension methods into the HTTP server itself. This would involve redesigning an HTTP server to make it look like a conventional CORBA application in which each method is implemented by a separate operation in the server.

To add a new method a programmer would merely need to add a new operation to the HTTP server. A stub compiler would need to be built to generate IDL stubs to marshal and unmarshal parameters, otherwise the programmer would need to write all the marhsalling and unmarshalling code by hand (see §3.3). Technology such as Java [JAVA] or Tcl [OUSTERHOUT 94] could be used to build a server in which it was possible to install new HTTP methods

dynamically. We already have a prototype which demonstrates this [McCLENAGHAN 95].

The implementors of new methods would then have the option of storing state internally, inside the HTTP process, rather than writing it to the local file system. This would only work if the same HTTP process is used to handle all requests. This requires threads. Currently it is usual (at least in the public domain servers) to fork a new HTTP server to handle each request. This is slow and expensive in terms of computational resources; it is also a performance bottleneck.

Building an HTTP server and stub compiler as described above is not a trivial task. It would require an IDL mapping of HTTP (and possibly a new stub compiler). However, it would have major benefits in terms of:

- performance
- extensibility
- modularity

The easiest way of building such a server would be to plug the HTTP protocol into an ORB and implement the HTTP methods inside an ORB object; we hope to explore this in the near future.

6.1 Acknowledgements

The author would like to acknowledge to helpful comments on this work by the following members of the ANSA team: Mike Beasley, Andrew Herbert, Mark Madsen and Owen Rees.

References

[APM 93]

ANSAware Programming Manual, APM Ltd., Cambridge, U.K., 1993.

[BERNERS-LEE 95]

T. Berners-Lee, R.T. Fielding, H. Frystyk Nielsen, “Hypertest Transfer Protocol — HTTP/1.0”. Internet-Draft (work in progress) , March 1995, <URL:ftp://nic.nordu.net/internet-drafts/draft-ietf-http-v10-spec-00.ps>.

[CERN 95]

“Status of the WWW Library of Common Code”, <URL:http://info.cern.ch/hypertext/WWW/Library/>

[CONNOLLY]

Daniel Connolly, “HyperText Markup Language (HTML)”, <URL:http://www.w3.org/hypertext/WWW/MarkUp/MarkUp.html>.

[EDWARDS 95]

Nigel Edwards, “Object Wrapping (for WWW) - The Key to Integrated Services?”, APM.1464, APM Ltd., Cambridge U.K. April, 1995. Also available as:<URL:http://www.ansa.co.uk/phase3-activities/1464/1464prt1.html>.

[JAVA]

“Hot Java Home Page”, Sun Microsystems, 1995. <URL:http://java.sun.com/>

[McCLENAGHAN 95]

“The Changeling Web Server”, APM Ltd., Cambridge, U.K., 1995. See also: <URL:http://www.ansa.co.uk/phase3-doc-root/sponsors/APM.1453.00.02.html>

[McCOOL]

Rob McCool, “The Common Gateway Interface”, <URL:http://hoohoo.ncsa.uiuc.edu/cgi/overview.html>

[OMG 93]

“The Common Object Request Broker: Architecture and Specification”, Revision 1.2, Draft, December 1993, OMG Document Number 93-12-43. (Available by anonymous ftp from ftp.omg.org, file: pub/docs/93-12-14.ps.z).

[OUSTERHOUT 94]

John S. Ousterhout, “Tcl and the TK Toolkit”, Addison-Wesley, 1994.