CONTENTS

UKC ANSAWARE SURVIVAL GUIDE

1 INTRODUCTON

So you need to write an ANSAware application but don't know where to start? This document will try to break you in as gently as possible by explaining the concepts you need to understand and then showing you how ANSAware implements them. You will not get enough information from this document to fully understand ANSAware so you should read [APM93] as well. Although the concepts are general, the parts of this document that explain how to compile code are specific to UKC.

2 OVERVIEW OF DISTRIBUTED SYSTEMS

2.1 What is a Distributed System?

Ok, so you're used to one program which starts, runs and then stops. It contains procedures that perform some function and these are called in turn so that the program performs some required task.

In a "distributed system" we take a problem and we build a solution from several different programs that all talk to one another, when we can make these programs communicate over some network we are able to have each one run on a different machine. It is this that makes a distributed systems solution so useful; these programs may run on machines on other side of the room or the other side of the world.

Another important aspect of distributed systems is "relocation" — if one of our machines stops working then we might be able to start that part of the system on another machine somewhere else and carry on.

2.2 How Does ANSAware Help Build One?

ANSAware provides support for building client/server applications. A server capsule provides some "service" to the rest of the ANSA world, an example might be "I can tell you the temperature outside"; a client capsule will make use of this service by making a request to the server.

Here are two terms you will use a lot when talking about ANSA programs:

- An ANSA program is called a **capsule**.
- The request that a client makes to a server is implemented via a **Remote Procedure Call** (RPC).

Let's look at a normal piece of C code; your program calls a procedure with some parameters and you may get some results back. An RPC is exactly the same except that the procedure is executed by the server which may be on another machine, ie. remote. You can pass parameters and get results back in just the same way, remember that passing pointers to data structures across machines isn't going to work!

This leaves us with two BIG issues to deal with:

- We now have all these server capsules that we can talk with and they can be running on different machines in several locations so how do we know what services are out there and how do we contact them?
- What do we do when our call to a server fails?

The second question will be answered later. Let's answer the first by considering the Yellow Pages.

I am a plumber, a good plumber who works at a reasonable rate and I know that everyone would like to use my plumbing service but first I need to tell the world that I exist. What do I do? I take out an advertisement in the Yellow Pages, the advertisement says I am a plumber and identifies a method of contact. Now someone with a leaking water pipe knows they need the services of a plumber so they can find me in the Yellow Pages and call me.

The ANSAware Trader is similar to our Yellow Pages. The Trader is an ANSA capsule that knows about all the visible services in our little world and how to contact the servers that provide them.

- When a server wishes to tell the world that it is around it **exports** a service offer to the trader. In our plumber example this occurs when I place my advertisement in the Yellow Pages.
- When a client wants o use a service it **imports** the location of the service from the Trader. What it gets back is a contact point for the service and can use this to talk to the server directly and perform the RPC.

Wildcards can be used so that the Trader performs a search and returns a list of matching services.

3 IMPLEMENTING THE CONCEPTS

When you write code for an ANSA capsule you write one or more `.dpl` files. These contain standard C code and ANSAware calls to perform the interactions between capsules. All ANSAware statements begin with a `!` in the first column, the preprocessor expands these to pieces of C and this is then compiled to produce an executable program:

`example.dpl` → preprocessor → `example.c` → C compiler → `example`

You will have one or more DPL files for each capsule; these are linked together to produce one executable. The best way to run several capsules is to run them in different windows on an X-Terminal.

Before we continue we need a little more terminology:

- Each server capsule has at least one **interface**. This is the point of contact with the client capsules. An **interface reference** uniquely identifies each interface in the system; when your client imports a service from the Trader what you get back is the interface reference to the service.
- Each interface can support some number of **service operations**. Operations are usually grouped together using some criteria and each group appears on a different interface, one capsule can have several interfaces.

4 BUILDING AN APPLICATION ONE STEP AT A TIME

The easiest way to explain how to build an application is to go through a simple example one step at a time. We will build a plumber service who can give quotes for jobs and can be called out by a customer capsule which we shall also build[1].

The stages that we go through to build our application can be summarised by this list:

1. Configure our shell for an ANSA session.
2. Write an interface definition for our service.
3. Copy the default Imakefile and modify it for our interface.

[1] The code for this example can be found in /proj/ansa/examples/plumber.

4. Generate service routine templates and signal handlers.
5. Flesh out the service routines.
6. Write the server body.
7. Write the client.
8. Build our application.

There is a logical order to these steps, use the above list as a reference when you come to writing your own programs. The following sections will explain each of these steps in detail.

4.1 Configuring the Shell for an ANSA Session

The ANSA system relies on certain environment variables being set so you must configure the shell for an ANSAware session before starting. If you use the csh shell then add the first line to your `.cshrc` file, add the second line to your `.bashrc` if you use bash:

```
alias ansa source /proj/ansa/.ANSAwarerc
alias ansa="source /proj/ansa/.ANSAwarerc.sh"
```

Once this has been sourced you can type `ansa` to set up all the library paths and other things that are necessary but which you really don't want to know about. This setup automatically defaults to using the latest version installed on the current platform; if you wish to use another version then set the `ANSA_VERSION` environment variable accordingly. [2]

4.2 Defining the Interface for our Service

We need to tell ANSAware what operations are available on each interface, for this we use an Interface Description Language (IDL). There is at least one IDL file for each interface - if you are writing a client, read the IDL file for the interface you need to talk to to see what operations there are.

If you are writing a server then you must write an IDL file to describe the operations it will implement. You can also create your own types to make the operation signatures clearer.

Here is the IDL file for our plumber service (`plumber.idl`):

```
plumber :       INTERFACE =
BEGIN
-- Defines a simple plumber service, Ian Buckner - 13/03/95

  -- client supplies the problem to the plumber who sends a
  -- quote back
  GetQuote : OPERATION [ problem : STRING ]
               RETURNS [ quote : CARDINAL ];

  -- supply your address to the plumber, he tells you if
  -- he can come out
  CallOut  : OPERATION [ address : STRING ]
               RETURNS [  booked_ok : BOOLEAN ];

  -- put the plumber out of business
  Sack : OPERATION [ ] RETURNS [ ];
END.
```

We see that there are three operations, two of which take one parameter and return one result, see [APM93], section 3.1, for details of the IDL syntax. It is a good idea to comment what each operation will do to avoid searching through the server source when you forget!

[2] Supported versions are 4.0 and 4.1 on the research workstations, 4.1 on the fish and 4.1.1 on the snakes.

4.3 Copying the Default Imakefile

The Imakefile explains how to build your ANSAware capsules, it specifies which files will be used in the build and describes how to construct a list of dependencies so that the correct files are re-built when something changes. Take a copy of `/proj/ansa/templates/Imakefile` which we will modify for our application.

Now we have written the interface description for our service we update the `IDLFILES` entry in the Imakefile:

```
IDLFILES = plumber.idl
```

We must also add an `IDLDepend()` entry:

```
IDLDepend(plumber)
```

More complex applications will use more than one interface. There must be an `IDLDepend()` line for each one and we must add them to the `IDLFILES` list too.

4.4 Generate Service Routine Templates and Signal Handlers

The first thing we need to do now is to generate the service routine template from the IDL file we wrote. This template will contain the skeleton code for the service routines which we will flesh out later.

Firstly, we must process the Imakefile to get a Makefile, do this with `ansamkmf` then `make templates` which produces several files:

```
plumberServ.dpl
mplumber.h
tplumber.h
eplumber.h
plumberServ.h
plumberSignal.h
plumber.sif
```

The DPL file is the service routine template, the other files are needed by the ANSAware system.

The next thing to do is `make signals` which generates signal handling code for each operation automatically, these are routines that are called by a client capsule when an RPC fails (see later). This code is contained in `plumberSignal.dpl` in our example.

We must now update the Imakefile to include these newly generated files, firstly we list the header files:

```
HDRS = mplumber.h tplumber.h eplumber.h plumberServ.h plumberSignal.h
```

An entry for the SIF files:

```
SIFFILES = plumber.sif
```

and dependency lines for the new DPL files:

```
DPLDepend(plumberServ)
DPLDepend(plumberSignal)
```

As the Makefile is generated from the Imakefile we must run `ansamkmf` whenever the Imakefile is changed - if we don't then our changes will not take effect.

REMEMBER, `make templates` and `make signals` generate template code by examining the IDL files you have specified in the Imakefile. These operations can only be done once to prevent any changes you make to the templates being overwritten by fresh versions. If you do want to generate the templates again then you will have to delete the generated DPL files first.

4.5 Fleshing-out the Service Routines

We now implement the behaviour of each of the operations we defined in `plumber.idl`. The stub compiler has produced a skeleton template called `plumberServ.dpl`; this file currently contains a procedure definition for each of the operations we defined in out IDL file which simply return failure. The naming convention used is:

```
<interface name>_<operation name>( ansa_InterfaceAttr *_attr
                                   [, parameter] )
```

It is important to note that the first parameter to each service routine is always an `ansa_InterfaceAttr` structure - this has been added by ANSA and can, for simple applications, be ignored but not removed.

The other important thing to note is the way in which return parameters are specified. The parameters are always ordered with the parameters supplied when the operation is invoked first, followed by the result parameters which are always passed as pointers. The return value of the function call indicates whether the operation was successful and takes one of two possible values, `SuccessfulInvocation` or `UnsuccessfulInvocation`. Here is the fleshed-out routine for the `GetQuote` operation:

```
#if AW_PROTOS
int plumber_GetQuote(
        ansa_InterfaceAttr      *_attr,
        ansa_String             problem,
        ansa_Cardinal           *quote
)
#else
int plumber_GetQuote( _attr, problem, quote )
        ansa_InterfaceAttr      *_attr;
        ansa_String             problem;
        ansa_Cardinal           *quote;
#endif
{
  /*
   * the quote is completely unrelated to the work that needs doing
   */

  *quote = (ansa_Cardinal) ((rand() / (pow(2,31)-1)) * MAX_BILL);
  fprintf( stdout,
    "Plumber: asked for a quote on %s, reckon it will cost %$%lu\n"
    , problem, *quote);

  return SuccessfulInvocation;
}
```

Lastly, a `GetMgmtInterface` service function is generated for each interface. You will not usually be using this and so it can be left unchanged.

4.6 Writing the Server Body

We now need to write the body of the server, this is in `plumberBody.dpl`.

We indicate which interfaces we will be using and that we will offer a plumber service.

```
/* Interface declarations */
! USE Trader
! USE plumber

/* Interface reference declarations */
! DECLARE { plumberServerRef } : plumber SERVER
```

One important thing to remember is that Ansa capsules start running from the `body` function, NOT from `main` as usual C programs do. Most servers will execute along the following lines:

Create a service interface and specify the number of simultaneous requests that can be made to it:

```
! {plumberServerRef} :: plumber$Create(1)
```

Tell the rest of the world about the service they provide by exporting it to the Trader:

```
! {} <- traderRef$Export( "plumber", CONTEXT, propbuf, \
                          plumberServerRef)
```

The first parameter to the export request is the service type. This must have already been added to the trader or a message similar to this will appear when you run your server:

```
(./plumber) :: warning, file 'trading.c': line 104
(./plumber) :: capsule 15286 WARNING: binder_export - trader error
'unknownType' : 1027 (bindFailure)
```

You can use Trcontrol (see section 7) to add service types to the Trader.

`CONTEXT` is the trading context for this offer. You should have your own context space to work in and only import from other spaces if you are talking to servers not owned by you. Now we just provide the service until close down when we call:

```
! capsule$Terminate()
```

which will remove the offer from the trader and close the capsule down cleanly.

The plumber example shows how the server can simply loop until it is asked to terminate. The application is prevented from using excessive CPU by making calls to

```
timer_Sleep(TSeconds, (ansa_TimeDelay) DAYLENGTH);
```

The ANSA sceduler is non-preemptive so any capsule that has more than one thread of execution must sleep at some point in the main thread to ensure that the body is descheduled and give time to allow other threads to execute. The exception is if your capsule is a pure server and does nothing else other than service RPC's; if this is the case then the main body can be left to exit leaving just the RPC service routines to be started when required.

You must also be carefull not to use blocking services like keyboard input unless there is data to read or you will lock up your capsule.

4.7 Writing the Client

So now let's take a step by step look at how a client capsule would use this service. Lets look at the ANSAware parts of `customer.dpl` first:

We need to say that we shall be using the Plumber interface:

```
! USE Plumber
```

and we need to save an interface reference so that we can talk to it, remembering that we will be a client. The use of the second temporary reference will be explained in a moment.

```
! DECLARE { plumberClient } : Plumber CLIENT
! DECLARE { tempRef } : Plumber CLIENT
```

In the body we must declare the variable for the interface references we will use, ANSAware generates a type from the interface name by adding a `Ref` suffix to the interface name:

```
plumberRef tempRef, plumberClient;
```

NOTE: always use the ANSA generated interface types rather than the generic `ansa_InterfaceRef` so that the compiler can enforce type checking.

Now let's talk to the Trader and try to import the plumber service:

```
! {tempRef} <- traderRef$Import("plumber", CONTEXT, "")
```

The Import operation on the Trader takes three parameters, the first of these is the type of the service we want to contact, we get back the interface reference for the service. We now copy the returned interface reference into the variable we will use in the RPC calls using the interface reference copying function.

```
ifref_copyRef(&plumberClient, &tempRef);
```

ANSA places return parameters in its own private data space, This data space changes whenever ANSAware performs an RPC so you are likely to loose your data unless you copy it out explicitly. This is one of the most common problems with peoples programmes.

Now we have a handle on the service we can talk to it directly:

```
! {cost} <- plumberClient$GetQuote(problem)
```

When we have finished we can free up the space used by the interface reference with:

```
! plumberClient$Discard
```

4.8 Building the Application

We are now ready to build our plumber and customer capsules. So far we have just used the `make` mechanism to generate template code, we must now modify our Imakefile to reflect the programs we want to build:

We have written two new DPL files so we add:

```
DPLFILES = plumberServ.dpl plumberBody.dpl customer.dpl \
           plumberSignal.dpl
```

and two new dependency lines:

```
DPLDepend(plumberBody)
DPLDepend(customer)
```

We wish to build two separate capsules so we define a list of objects needed to build the customer and a list for the plumber. When an IDL is compiled by stubc three files of stub routines are generated:

- `m<interface name>.c` - marshalling code
- `s<interface name>.c` - server stub code
- `c<interface name>.c` - client stub code

A capsule which is a client of this interface will require the client stubs and marshalling code, a server will need the server stubs and marshalling code. The customer uses the signal handling routines and the plumber requires the service routines so we now know which objects will be needed for each of our capsules:

```
CUSTOMER_OBJS = customer.o plumberSignal.o mplumber.o cplumber.o
PLUMBER_OBJS = plumberServ.o plumberBody.o mplumber.o splumber.o
```

It is also necessary to list the names of the capsule executables in the `PROG` variable:

```
PROGS = plumber customer
```

The last thing to add is an entry to actually build our capsules; `SingleProgramTarget()` takes the name of the executable, followed by the list of objects and then libraries:

```
SingleProgramTarget(customer,$(CUSTOMER_OBJS),$(LOCALLIB), \
                    $(LIBPATH)$(LIBS))
SingleProgramTarget(plumber,$(PLUMBER_OBJS),$(LOCALLIB), \
                    $(LIBPATH) $(LIBS))
```

Now rebuild the makefile with `ansamkmf`, do a `make depend` to work out file dependencies and finally do `make` to build the applications.

NOTE, `make depend` informs `make` of the relationships between source files so that it knows what to recompile when you change things. These dependencies are added to the end of the Makefile so you must do this after every `ansamkmf`.

5 Debugging

One thing to note: when the compilation fails and gives you a line number that contains an error this is the line in the DPL file, NOT in the C file.

Running an ANSA capsule through a debugger can be a real problem because of the macro-expansion; PREPC inserts comments into the C files it produces to show where it has expanded code in the DPL file, this makes debuggers get confused about line numbering. You can make the compiler drop code with debugging information and convince PREPC not to insert comments by adding these lines to your Imakefile:

```
CCFLAGS = -g
DPLFLAGS = -n
```

`gdb` does a pretty good job at handling ANSA code. When you debug a capsule you will be presented with the source code from the C file so you will have to look at the DPL file to work out where code has been expanded. You will notice that most of the expanded code will call routines in the marshalling code or in the client/server stubs. Gdb should be able to find this source as it will be in the same directory as the executable. You can find a gdbinit file for each ANSA version in `/proj/ansa/<version>/config/gdbinit` [3] , copy this to your home directory as .gdbinit to point gdb at the ANSA library source.

6 Other Things to Know

The Trader supports more than just simple Import and Export operations. The Lookup operation allows a client to obtain a list of services which match a particular criteria, here is an example:

```
  LResult lookup_result;
  LPolicy policy = Lookup_All;

! {lookup_result} <- traderRef$Lookup("plumber", CONTEXT, "", policy)
```

Here we ask the Trader to give us all the interface references to plumber services in the given context. What we get back is quite a complex structure and needs some explaining:

- `lookup_result.designator` is either `F_S` or `S_S` indicating failure or success.
- On failure, `lookup_result.u_F_S` indicates why the Lookup failed.
- On success, `lookup_result.u_S_S.length` is how many interface references we got back.
- `lookup_result.u_S_S.data[0].ifr_ref` is the first in the list of interface references.

See [APM93], section 3.11, for more details on the Trader.

It is often necessary to see if two interface references are identical. This can be done with

```
ansa_Boolean ifref_cmpIdentity( ansa_InterfaceRef *a,
                                ansa_InterfaceRef *b);
```

[3] `<version>` is `ANSAware.v4.0` or `ANSAware.v4.1` on the research workstations, `ANSAware.4.1.1` on the snakes and `ANSAware.v4.1` on the fish.

which returns `ansa_TRUE` if both interface references refer to the same interface instance.

The last point to make is regarding premeture termination of server capsules. If a server capsule dies or is killed without closing down properly, it's offer can remain in the Trader. A client may subsequently import the interface reference associated with this offer and will then terminate with a bind failure when it tries to invoke an operation on the interface.

This can be overcome by using Trcontrol (see later) to remove stale offers from the Trader after a server has terminated abnormally.

7 Trcontrol - a Useful Tool

Trcontrol is an X-based application that allows you to visually see the services registered with the Trader, as well as add and delete service types and trading contexts [4]. The application is started by typing `trcontrol`; Figure 1 shows the application window.

The main screen of Trcontrol is divided into seven areas. At the top is the message window showing the previous operations executed by Trcontrol, below this is an input bar into which you can type. Next are two large buttons which refresh the display and show the details of the Trader being managed.

The next row of buttons perform operations on the trading contexts that are shown in the window below. The trading contexts are hierarchical and ones ending in "/" can be expanded to reveal lower levels. All normal ANSAware users will have a trading context below "/ansa/users/" or "/ansa/groups/", followed by their login — contact the ANSAware administrator if you do not have one.

Once a trading context has been expanded, you will see a list of the service offers which have been exported in this context. Once you have highlighted an offer, the operations on the "Offers" button can be used to delete it or view it's details. Deleting services is useful for removing stale offers - when a server has died but has not withdrawn it's offer from the trader. Importing and using this service will fail as the server is no longer around.

The bottom part of the Trcontrol display is a button bar and a hierarchical display of the types registered with the trader. You can expand and move around the type hierarchy in the same way as you do in the context window. The "Add Type" option on the "Type" menu can be used to insert a type in the part of the hierarchy that is currently highlighted.

One word of warning, there is no sense of ownership on offers, contexts or types which means you are free to delete any ones you want. Be sensible and do not go around deleting other peoples offers or trading contexts as this is VERY annoying!

8 Exception Handling

The call that a client makes to a server may not always succeed; this may be because the server is no longer running or because the server is unreachable due to a network problem. We don't want our client to fall over in a heap when this happens and would like to, at the very least, be told that a failure has occured and have the client close down gracefully.

This is implemented using ANSA exception handling - we add an exception clause to each RPC invocation, ie:

```
!  {quote} <- plumberClient$GetQuote(problem) Continue ok Signal *
```

This will let us know if the request fails for ANY reason - [APM93], Section 3.7 explains how this exception syntax can be used to indicate certain types of failure and not others ,though, in general, you will find that you will use the above form more often than not.

The UKC stub compiler generates standard signal handlers when it compiles with the `-g` option (ie. when you do a `make signals`). The default behaviour of these handlers is to try to relocate the service

[4] Trcontrol is not yet available on the snakes, you will have to use the standard ANSA "trclient" program to query the Trader.