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**

Training

ANSAwise - Building Applications with ANSAware Tools

Chris Mayers

Abstract

Organizations wish to gain practical experience with distributed systems technology. ANSAware is an excellent vehicle for this purpose.

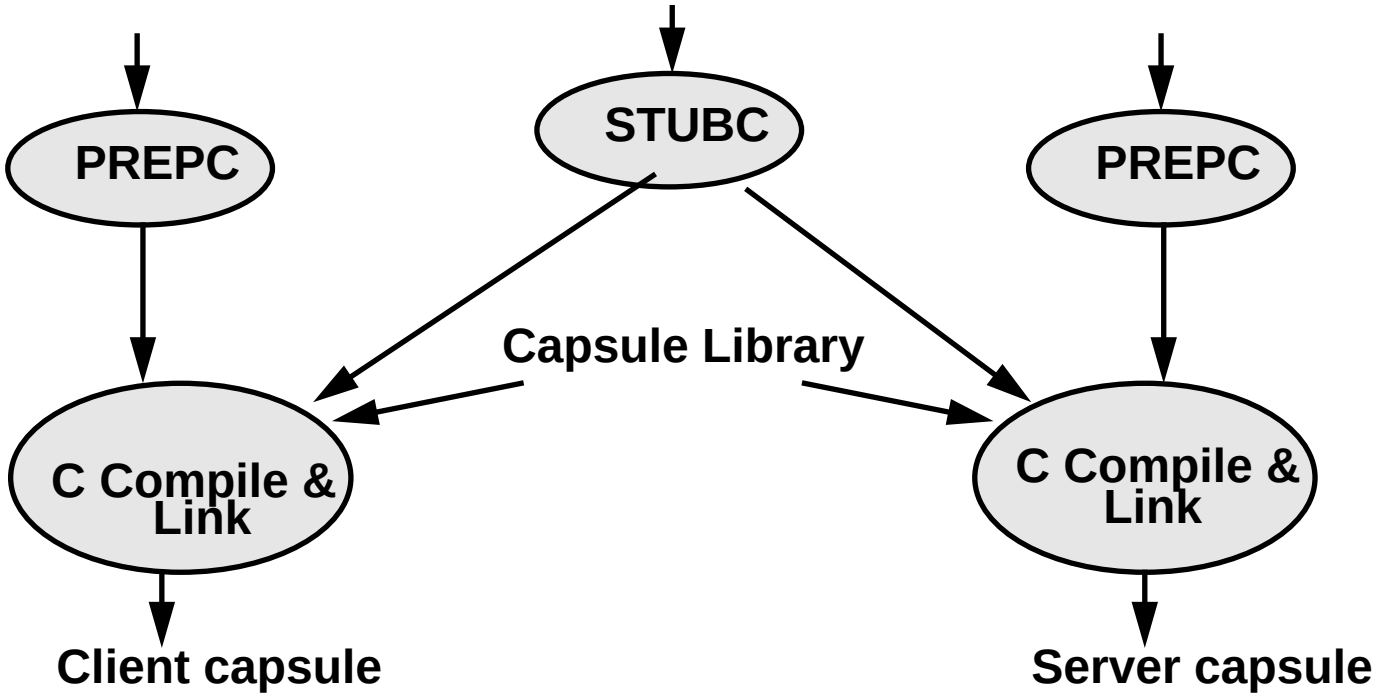
This module of the ANSAwise training programme is a hands-on module with ANSAware 4.1; it is the first such session. It briefly describes ANSAware applications, how ANSAware compares with CORBA and DCE, and describes ANSAware IDL. It then shows the steps to build an ANSAware client and server, their structure, and how to run them.

APM.1584.01 **Approved** 2nd October 1995

Briefing Note

Distribution:
Supersedes:
Superseded by:

ANSAware 4.1
Building Services with ANSAware Tools



In this session

- **Show the steps needed to build a simple service (Echo) using ANSAware tools**
- **Show the IDL and PREPC languages, and how to use their preprocessors (STUBC and PREPC) to generate C**
- **Build, modify, and try out a simple example service (Simple Bank)**

What is ANSAware?

- A toolkit for building distributed applications
  - based on the ANSA architecture
  - supporting diverse operating systems and hardware platforms

Distinctive ANSAware features

- Based on long-term research; a mature toolkit
- Implements ANSA transparencies
- Supports concurrency and light-weight implementations
- Supports group execution
- Third-party extensions available
- Ported to many environments
- Not a shrink-wrapped product - you can experiment with it

ANSAware 4.1 - Infrastructure and Tools

- **Basic infrastructure support**
  - **the ANSAware run-time library**
  - **the ANSAware RPC protocol (REX/GEX)**
- **Tools**
  - **STUBC: Compiler for Interface Definition Language (IDL)**
  - **PREPC: Preprocessor for service calls embedded in C**

How do applications interface with ANSAware?

- **Toolkits can provide two general approaches**
  - **Application Programming Interface (API): the procedural approach**
  - **Language Extension**
- **ANSAware provides Language Extension**

The procedural approach

- **Other toolkits provide Application Programming Interfaces (APIs) between applications and systems**

Applications and Applications Programmers

Application Programming Interface

Systems and Systems Programmers

  - **the *procedural* approach**

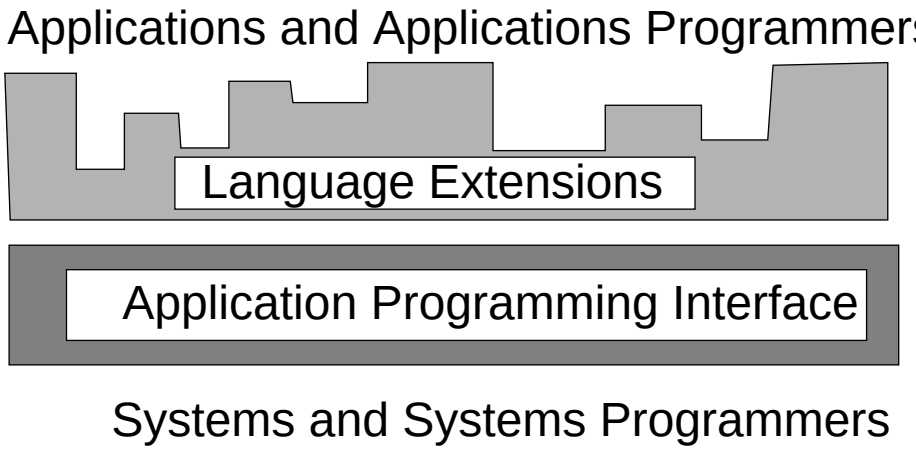
- **This approach causes problems...**

Problems with the API/procedural approach

- Large APIs are difficult to comprehend
  - you often need to understand *all* the API to use *any* of it
- Large APIs grow larger
  - the API widens as application requirements grow...
  - ... it gets harder and harder to comprehend
- APIs do not adequately hide system detail
  - the detail shows through to the applications
- Mistakes are easy to make, and are not detected early enough
  - there is a lot of code to write
  - most checks are done at run time

The ANSA approach - Language Extension

- **ANSA takes a more powerful programming language view:**



- **The interface seen by the application programmer is expressed as a series of Language Extensions**
- **Simpler to understand and use than a procedural API**

Language Extension - Advantages

- **Advantages**
  - a simple, system-independent programming model
  - Independence between application and system designers
  - easy migration of software between platforms
  - compile-time checking
- **Benefits**
  - increased confidence in software
  - more robust, error-free and dependable software
  - system evolution
  - applications unaffected by system re-engineering
  - system unaffected by application re-design

ANSAware Node and Nucleus

- **Node: a single machine or a tightly-coupled set of machines**
  - **a node allows for the creation/destruction of processes with a unique process id**
- **Nucleus: an engineering object which manages the resources of a node**
  - **the Nucleus is implemented as a set of libraries that get linked together with the user's application to form a *capsule***
  - **the Nucleus is also called the *infrastructure* or *capsule library***

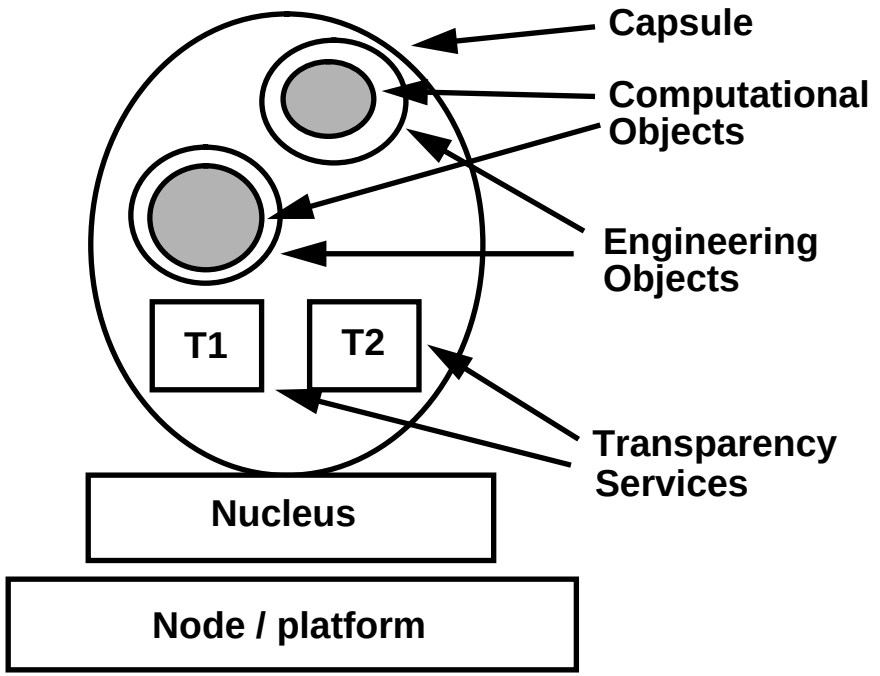
ANSAware Capsule

- A capsule is
  - the unit of autonomous execution within ANSAware
  - an address-space supporting a single instance of the run-time system...
  - ... providing a memory protection boundary
- A capsule provides:
  - an efficient, transport-independent and portable RPC protocol
  - light-weight threads
  - synchronisation operations
  - timer handling
  - support for interworking with other systems - e.g. X11 graphical user interface

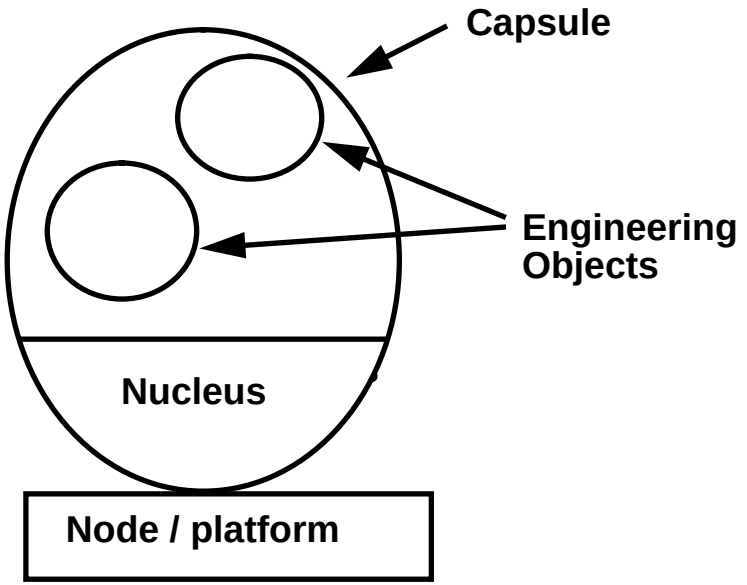
Node support for capsules

- **The node support for capsules depends on the underlying operating system**
  - **Unix supports multiple capsules per node**
  - **DOS only supports one**
- **Capsules can be created and destroyed dynamically**

ANSA Node, Nucleus, and Capsule structure

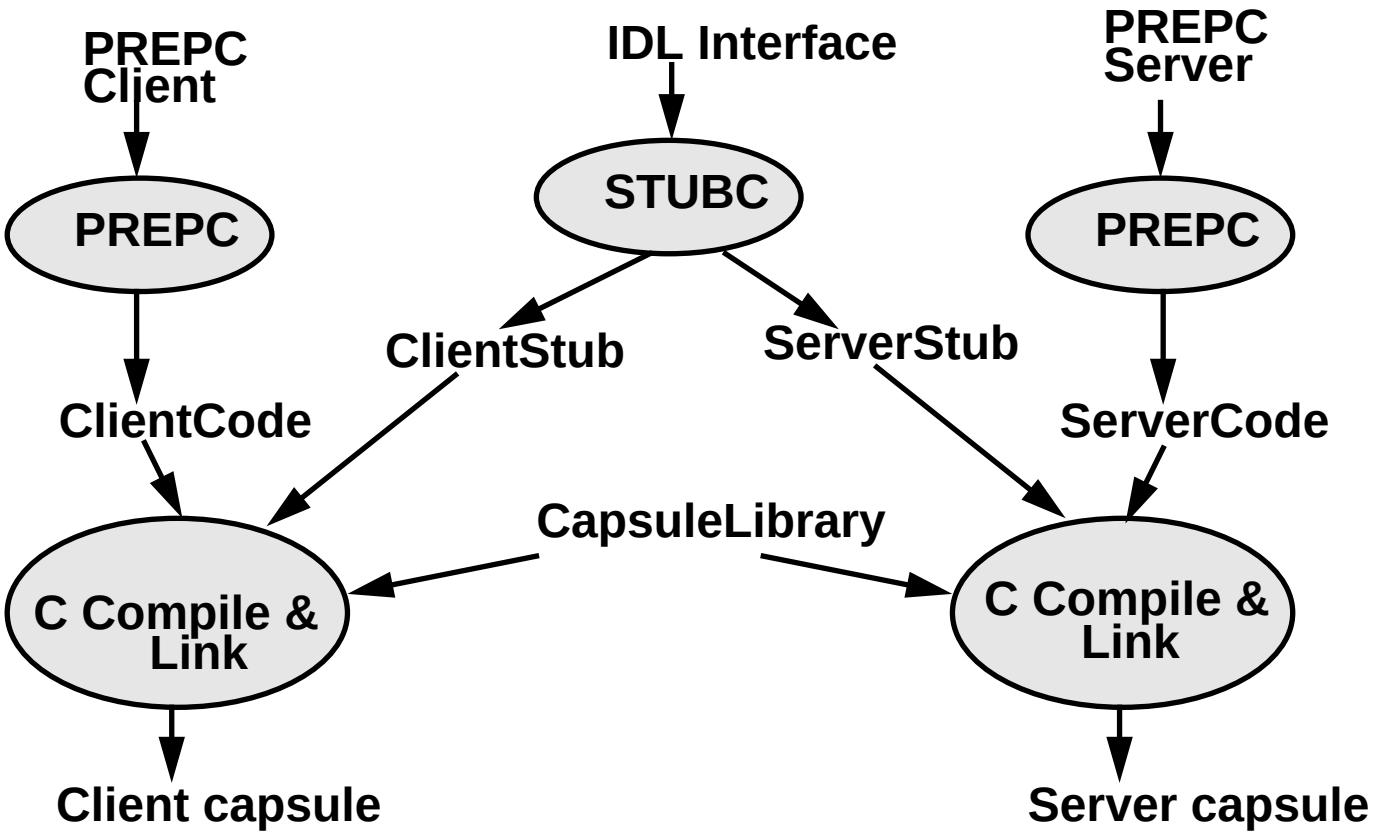


ANSAware Node, Nucleus, and Capsule - Implementation

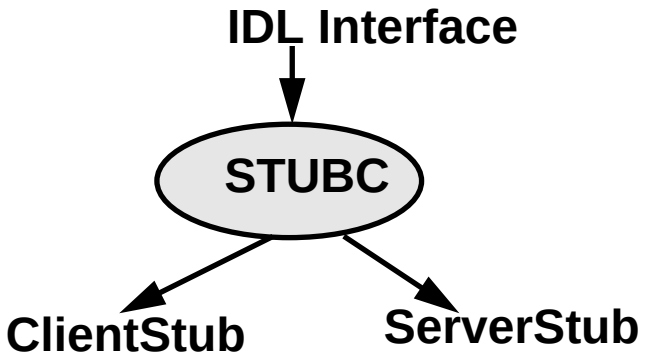


- the Nucleus is linked into the capsule (as the capsule library)
- computational objects disappear

ANSAware Components



IDL - the Interface Definition Language



- Application writers specify service interfaces in IDL
- STUBC (the IDL stub compiler) compiles these specifications to generate the stub code for the defined operations
- Stub code uses a data exchange format that is identical for all clients and servers

IDL - Defining operations


```
Echo : INTERFACE =

-- Comment lines start with two dashes

BEGIN

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

Sink : OPERATION [ Src: STRING ] RETURNS [];

Source : OPERATION [ Length: CARDINAL ] RETURNS [ STRING ];

Reverse: OPERATION[ Src: STRING ] RETURNS [ STRING ];

END.
```

IDL Types

- You can use the primitive types (for instance, STRING) directly
- You can define new types with constructors (for instance, ARRAY)
- Interface Reference types are generated automatically

IDL Primitive Types

- **IDL provides the primitive types:**
  - **BOOLEAN**
  - **[SHORT] CARDINAL**
  - **[SHORT] INTEGER**
  - **[LONG] REAL**
  - **OCTET**
  - **CHAR**
  - **STRING**

IDL Type Constructors

- These allow you to define new types constructed from existing types
- The type constructors are:
  - Alias: TYPE = type
  - Enumeration {identifiers}
  - ARRAY OF Type (fixed size)
  - SEQUENCE OF Type (variable size)
  - RECORD [Types]
  - CHOICE (discriminated union)

IDL - Defining new types


```
Pt: TYPE = RECORD [
  x: INTEGER,
  y; INTEGER
];
Plot: TYPE = SEQUENCE OF Pt;
Box: TYPE = ARRAY 4 OF Pt;
Hexagon: TYPE = ARRAY 6 OF Pt;
Which: TYPE = { B, H };
Polygon: TYPE = CHOICE Which OF {
   B => Box,
   H => Hexagon
};
```

IDL - built-in types

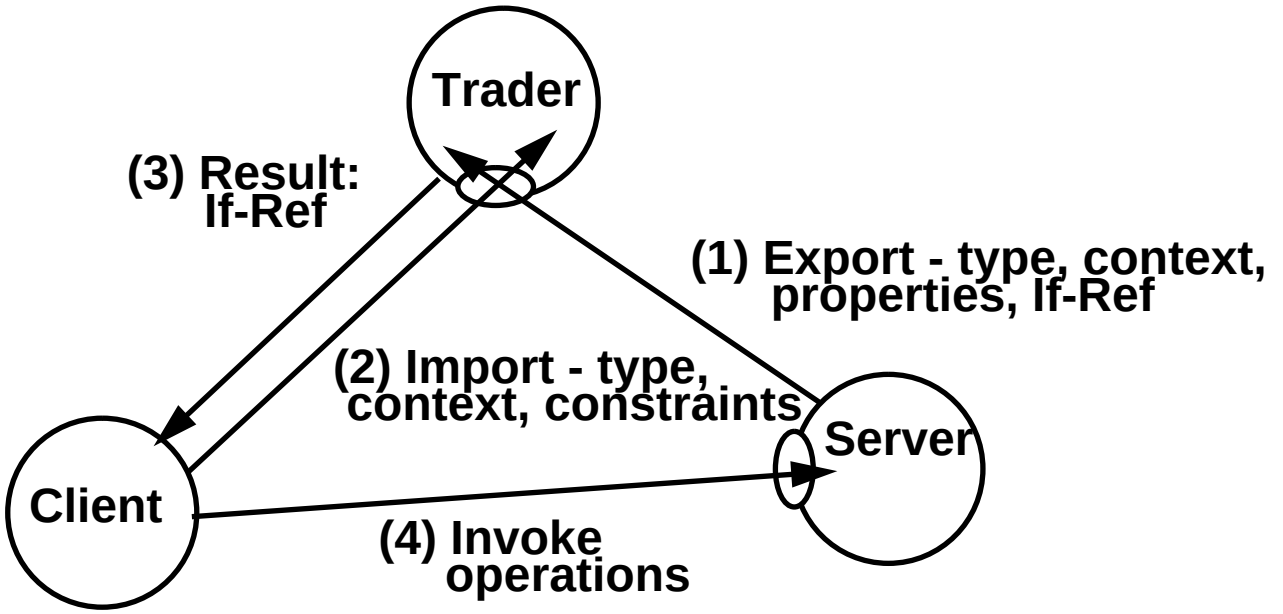
- An abstract type definition is provided for `ansa_InterfaceRef`
- Every interface has an interface reference type defined for it automatically:

```
X: INTERFACE =

BEGIN

   XRef: TYPE = ansa_InterfaceRef;

END.
```

PREPC

- C preprocessor
- Write clients and servers in C
- Embed PREPC statements in C
- PREPC with C provides implementation for clients and servers
- PREPC allows operations provided by a service interface to be invoked by clients
- PREPC provides access to all services: the trader, factory, node manager, as well as the services you define

Using the Trader from PREPC statements



- **Import, Export are special PREPC statements for using the Trader**

PREPC statements - Echo client structure

- **Client:**

```
! USE Echo
! DECLARE { intRef } : Echo CLIENT
...
ansa_InterfaceRef intRef; /* or EchoRef intRef; */
  /* import service */
! { intRef } <- traderRef$Import ( “Echo”, “/”, “”)
...
  /* invoke operations */
! { obuf } <- intref$Echo(ibuf)
...
  /* discard service */
! intRef$Discard
```


PREPC statements - Echo server structure

- **Server:**

```
! USE Echo
! DECLARE { ir } : Echo SERVER
...
ansa_InterfaceRef ir;
...
  /* create & export interface-instance */
! {ir} :: Echo$Create(16)
! {} <- traderRef$Export( “Echo”, “/ansa/testservices”, \
                          propbuf, ir )
...
  /* operations invoked */
```

PREPC statements - Echo server operations

- **Server Operations:**

```
Interface_Operation (
  _attr, /* always required */
  args,  /* arguments */
  results /* results (pointers) */
)
{
/*  do the operation */

return SuccessfulInvocation;
/* return UnSuccessfulInvocation; would indicate a failure */
}
```

Understanding the Echo service

- **The Echo example is simple...**

- **... but is worth examining in detail, to understand how the parts fit together**

Echo service description

- A service that, when sent a string, simply echoes back a string
- It has 4 ways of echoing; each is a separate operation
  - Echo: echoes the same string it was sent
  - Sink: echoes nothing
  - Source: echoes a string of random characters, of a given length
  - Reverse: echoes the string it was sent, but spelt backwards

Echo service interface (IDL)


```
Echo : INTERFACE =

-- Comment lines start with two dashes

BEGIN

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

Sink : OPERATION [ Src: STRING ] RETURNS [];

Source : OPERATION [ Length: CARDINAL ] RETURNS [ STRING ];

Reverse: OPERATION[ Src: STRING ] RETURNS [ STRING ];

END.
```

Echo Client


```
! USE Echo
! DECLARE { intRef } : Echo CLIENT
char ibuf[1024], *obuf;
void body( int argc, char *argv[], char *envp[] )
{  ansa_InterfaceRef intRef;
!  {intRef} <- traderRef$Import( "Echo","/", "")
   printf(">");
   while( fgets(ibuf, sizeof(ibuf), stdin) != (char*)0 ) {
!     {obuf} <- intRef$Echo(ibuf)
      printf("%s", obuf);
      printf("> ");
   }
!  intRef$Discard
}
```

Echo Server


```
! USE Echo
! USE Trader
! DECLARE { ir } : Echo SERVER
void body(int argc, char *argv[], char *envp[] )
{
   ansa_InterfaceRef ir;
!  {ir} :: Echo$Create( 16 )
   (void) system_init_properties( propbuf, PROPSIZE, \
                                  argc, argv );
!  {} <- traderRef$Export( "Echo", "/ansa/testservices", \
                           propbuf, ir )
}
```

Echo Server - Echo operation


```
int Echo_Echo( _attr, src, result)
  /* All server operations have this first argument */
  ansa_InterfaceAttr *_attr;
  /* Arguments */
  ansa_String src;
  /* Results (pointers) */
  ansa_String *result;
{
  *result = src;
  return 1;
  /* return 0; would indicate a failure */
}
```

Using Imakefiles

- Standard Unix Makefiles are hard to maintain
- Imakefiles are ‘higher-level’ Makefiles
  - using higher-level rules
  - generating a standard Makefile from the Imakefile
- The ansamkmf tool generates a Makefile from the Imakefile
- The make depend rule updates the Makefile to include the dependencies
  - it uses a tool that scans the source files

Imakefile for Echo client and server


```
IDLFILES = Echo.idl
SIFFILES = Echo.sif
DPLFILES = client.dpl server.dpl
RCSFILES = Imakefile $(IDLFILES) $(DPLFILES)
PROGS = client server

# compile idl files
all:: $(SIFFILES)

# dpl and idl file dependencies
DPLDepend(client)
DPLDepend(server)
IDLDepend(Echo)

SingleProgramTarget(server,server.o sEcho.o,$(LOCALLIB),)
SingleProgramTarget(client,client.o cEcho.o,$(LOCALLIB),)
```

Building the Echo service

$ansamkmf

to create a Makefile from the Imakefile

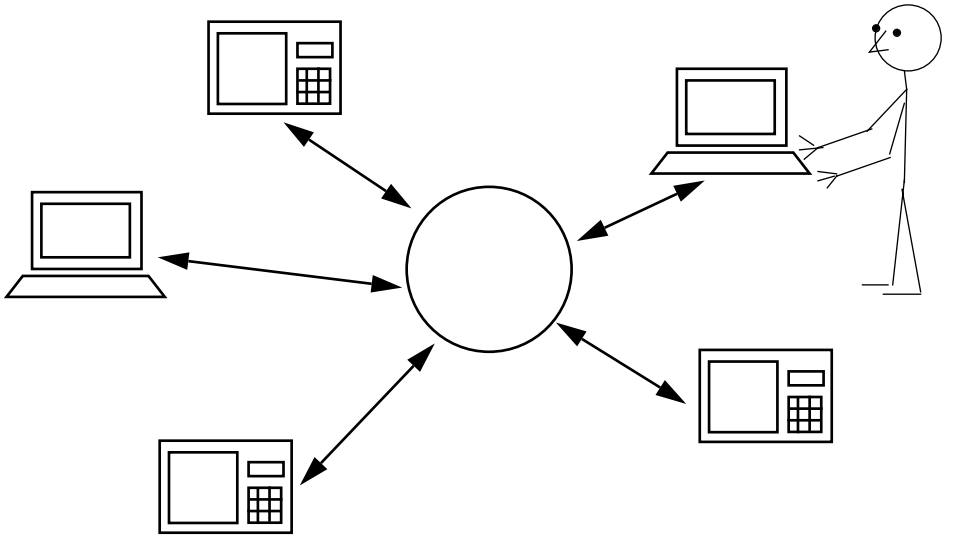
$make depend

to create the required dependencies

$make

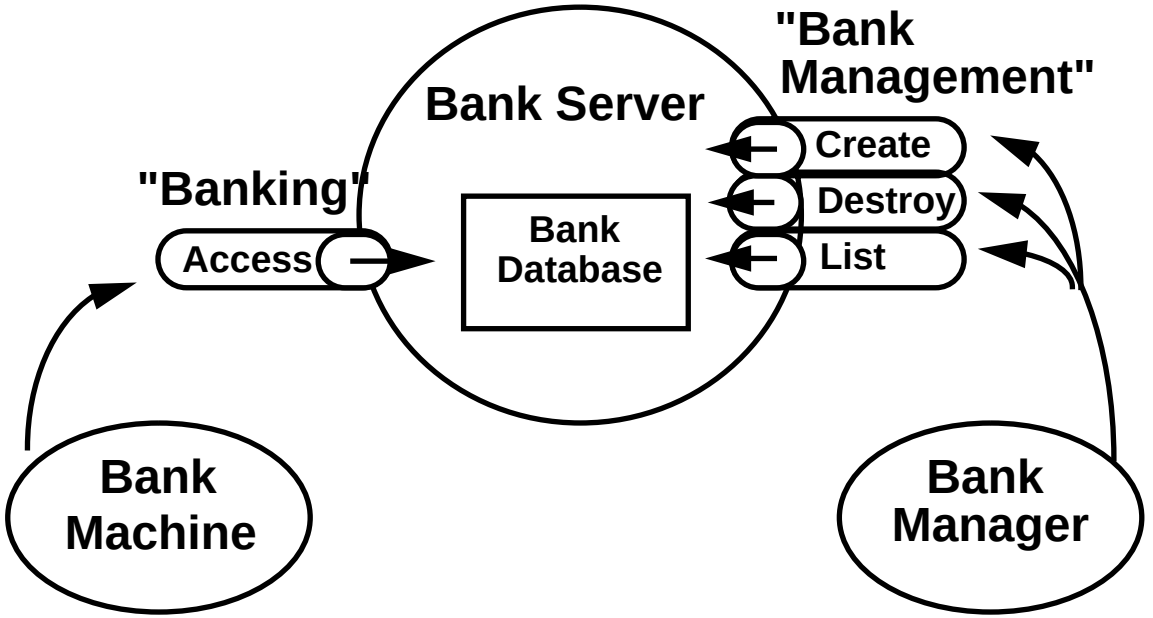
to create client and server programs

Simple Bank Example



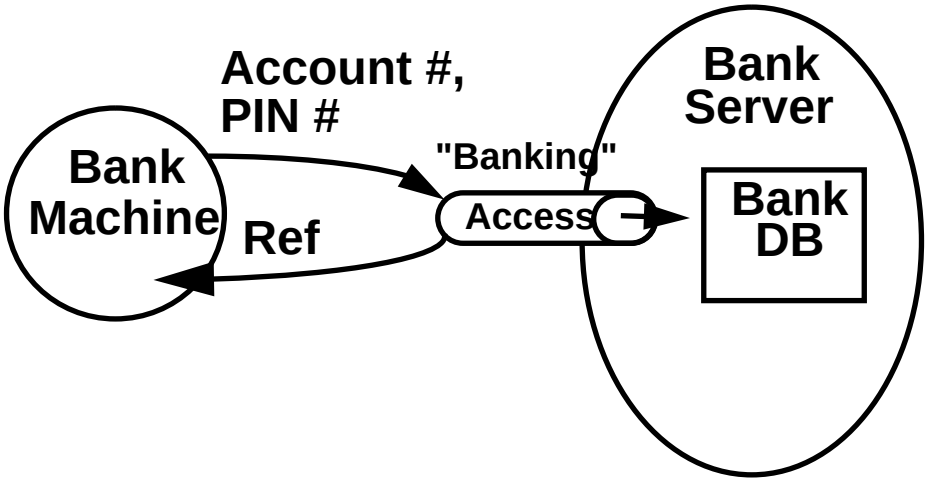
- A Bank Machine (Automatic Teller Machine) system
  - Users accessing their accounts
  - Bank managers administering these accounts

Bank Server - Interfaces

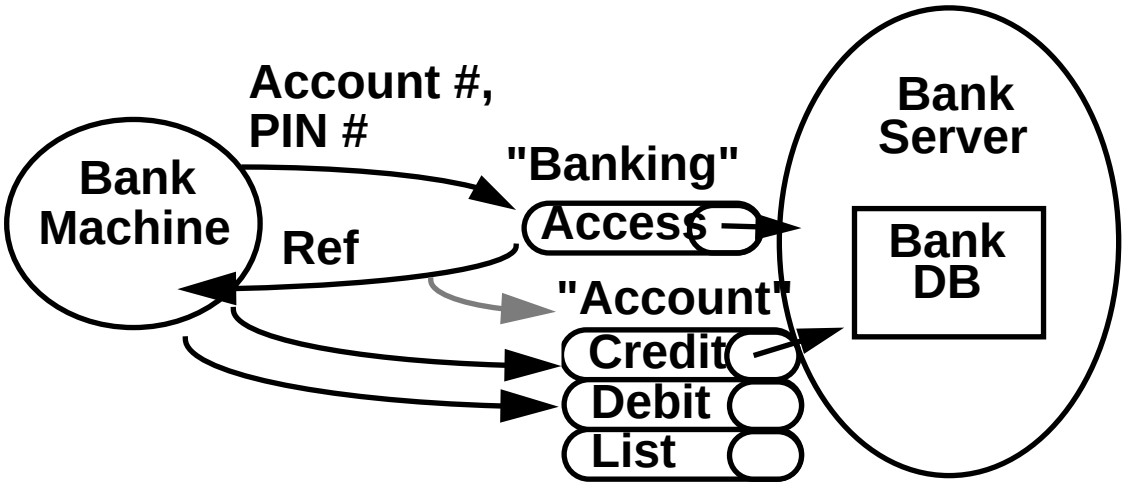


Banking Interface

- User should only be able to interact with account if PIN is valid
- Do not want Accounts to be public
  - customers should only access their own accounts
- If "Access" succeeds, "Account" is created and its object reference returned



Banking and Account interfaces



- If "Access" succeeds, "Account" is created, and its object reference returned
- "Account" interface provides operations on one particular account
- Only that client knows the "Account" object reference

Simple Bank Interfaces

- **Banking (SBank)**
  - for access with PIN by customer via Bank Machine, with PIN
- **Account (Account)**
  - for credit/debit/list by customer via Bank Machine
- **Bank Management (SBankMgmt)**
  - for create/destroy/list by manager

Account Interface

- **Account - credit/debit/list account**
- **An Account object is created (via the SBank interface's Access operation) for each account being accessed**
- **So there is no need to pass account numbers as arguments to the operations in the Account interface**

Account Interface - IDL Definition


```
Account: INTERFACE =
NEEDS SBankTypes FROM SBTypes;
BEGIN
ListResult: TYPE = CHOICE OpStatus OF {
   OpSuccess => AccountRecord,
   OpFailure => OpReason
};
Credit: OPERATION [ Amount: REAL ]  RETURNS [ OpResult ];
Debit: OPERATION [ Amount: REAL ]   RETURNS [ OpResult ];
List: OPERATION []  RETURNS [ ListResult ];
Destroy: OPERATION [ ] RETURNS [ OpStatus ];
END.
```

Banking (SBank) Interface

- **SBank - PIN based access to Account interfaces**
- **Returns an Account interface for the account identified by an account number and corresponding PIN**
- **Equivalent to the user interface offered by a real bank machine to a human user**

Simple Bank - interface SBank


```
SBank: INTERFACE =
NEEDS SBankTypes FROM SBTypes;
NEEDS Account;
BEGIN
AccessResult: TYPE = CHOICE OpStatus OF {
   OpSuccess => AccountRef,
   OpFailure => OpReason
};

Access:
   -- You must complete the specification of this operation
   -- adding the PIN argument
END.
```

Exercises

- **Fill in the specification of the Access operation**
- **Add the logic to the implementation of the Access operation to validate the PIN against the account number**
- **Build the server and check that it is operating as expected**

About the exercises

- **CAUTION - you will find the Simple Bank example does not handle errors well so far**
- **This is intentional!**
  - **a later session will show how to modify Simple Bank to solve this problem**
- **Some operations take time to execute**
  - **allow at least 1 minute for output to be displayed**

Your notes

Using Simple Bank - Creating accounts


```
% ./server &
% manager create "Jane Dunlop" 123.45
New account particulars:
      owner:  Jane Dunlop
      accno:  1
      pin:    6263
      balance:123.45
% manager create "Fred Bloggs"
New account particulars:
      owner:  Fred Bloggs
      accno:  2
      pin:    2507
      balance:0.00
```

Using Simple Bank - Listing accounts


```
% manager listall
1     6263     123.45 Thu Mar 14 16:14:55 1992 Jane Dunlop
2     2507       0.00 Thu Mar 14 16:15:50 1992 Fred Bloggs


% teller 1 6263 list
Details for account
      owner:  Jane Dunlop
      balance:123.45
      lastaccess:  Thu Mar 14 16:14:55 1992
```

Using Simple Bank - Error handling


```
% teller 1 6263 debit 400.00

Debit(400.00) failed, reason: InsufficientFunds

% teller 1 6264 list

Access(1, 6264) failed, reason: InvalidPin

% teller 3 6263 list

Access(3, 6263) failed, reason: NoSuchAccount
```

Using Simple Bank - Destroying accounts


```
% manager destroy 2
```

```
% manager listall
```

```
1     6263     123.45 Thu Mar 14 16:14:55 1992 Jane Dunlop
```

Using Simple Bank - Shutting down the server


```
% sbshut.sh

% ./server &

% manager listall

1     6263     123.45 Thu Mar 14 16:14:55 1992 Jane Dunlop
```

Summary

- For more information, see *Application Programming in ANSAware*
  - For the IDL language, see Appendix A
  - For examples of IDL, see Appendix D
  - For the PREPC language, see Appendix B
  - For the STUBC and PREPC commands, see Appendix C
  - For the example Echo service, see Chapter 5