Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

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

APM Business Unit

Writing Distributed Applications using ANSA and ANSAware 4.1

Chris Mayers

Abstract

A training course in ANSA and ANSAware 4.1. Based on a course written by Jane Dunlop presented in May 1992, using ANSAware 4.0. Version 00.01 of this document is the lightly reformatted, but uncorrected course,

APM.1261.00.01	**Draft**	23 June 1994
	External Paper	

Distribution:
Supersedes:
Superseded by:

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

Writing
Distributed Applications
Using ANSA

Tutorial Overview

DAY 1

- **Distributed Systems - benefits and drawbacks.**
- **ANSA overview**
  - **History**
  - **ANSA & ANSAware**
  - **The ANSA approach to distributed systems**
    - **problems, approach, design philosophy**
- **The ANSA computational model**
- **Engineering model & implementing the ANSA architecture**
- **Overview of using ANSAware**
- **ANSAware in detail, with hands-on examples**
  - **Using ANSAware services & tools**
  - **How to write a simple application**

Tutorial Overview (cont'd)

- DAY 2
- ANSAware in detail (cont'd)
  - ANSAware capsule-library features & constructs
  - Writing more complicated applications
  - Exception Handling
  - Dynamic service creation - Factory & Node Managers
- Practical concerns - Installing & setting up ANSAware on your system
- Future developments of ANSA and ANSAware
  - Interface Groups
  - Storage, migration

Motivation

- Technology - multiple workstations offer better CPU performance than a single large machine.
- Administrative control decentralised - greater autonomy of operation.
- Evolution & Interworking of Existing Systems - in most organizations, information is spread across different, incompatible systems
- Faster Communication and Travel - international and global information systems already exist - distributed applications and requirements exist all around you!

Advantages

- Exploit multiple machines to improve performance.
- Increased productivity through greater machine availability.
- Fault tolerance.
- Evolutionary system growth and expansion.

Drawbacks

- Harder to design, build, maintain and administer than a centralised system.

- Heterogeneity in hardware, networks, protocols, operating systems, data representations, expertise requirements etc, must be managed.

- Communication delays and errors make it impossible to have a 100% consistent view of a distributed system.

ANSA Project: 1985-1988

- **Alvey Programme - UK Government.**
- **Collaborators: BT, DEC, GEC, GPT, HP, ICL, ITL, Racal, Olivetti, STL.**
- **Developed the initial ANSA architecture, published as the ANSA Reference Manual.**
- **Produced a prototype realisation of that architecture - ANSA Testbench 2.5.**

ISA Project

- Esprit II - CEC.
- Collaborators: AEG, BT, Case, Chorus, CNET, CTI, DEC, Ellemtel, GEC, GESI, GPT, HP, ICL, Newcastle Univ, Origin, SEPT, Siemens, STL, Syseca, Televerket.
- Continuing development of the ANSA architecture and an example prototype implementation. Testbench 2.5 -> ANSAware 3.0 -> ANSAware 4.0.
- To play a leading role in the ODP area and influence relevant standards (e.g. ODP, ISO RPC, ECMA RPC, UI, OSF, OMG).
- Distributed computing areas convered: groups, transactions, concurrency control, migration, storage, trading, naming.

Advanced

Networked Systems

Architecture

Overview

ANSA - Design Principles

- ANSA is an architecture for building distributed systems
- Distribution can be made selectively transparent to application writers & users
- Different approach from networking collections of single systems
- Integrated approach to separation, heterogeneity, interworking
- Systems viewed as coordinated sets of subsystems, appropriate to the enterprise they serve - not as random collections of boxes.

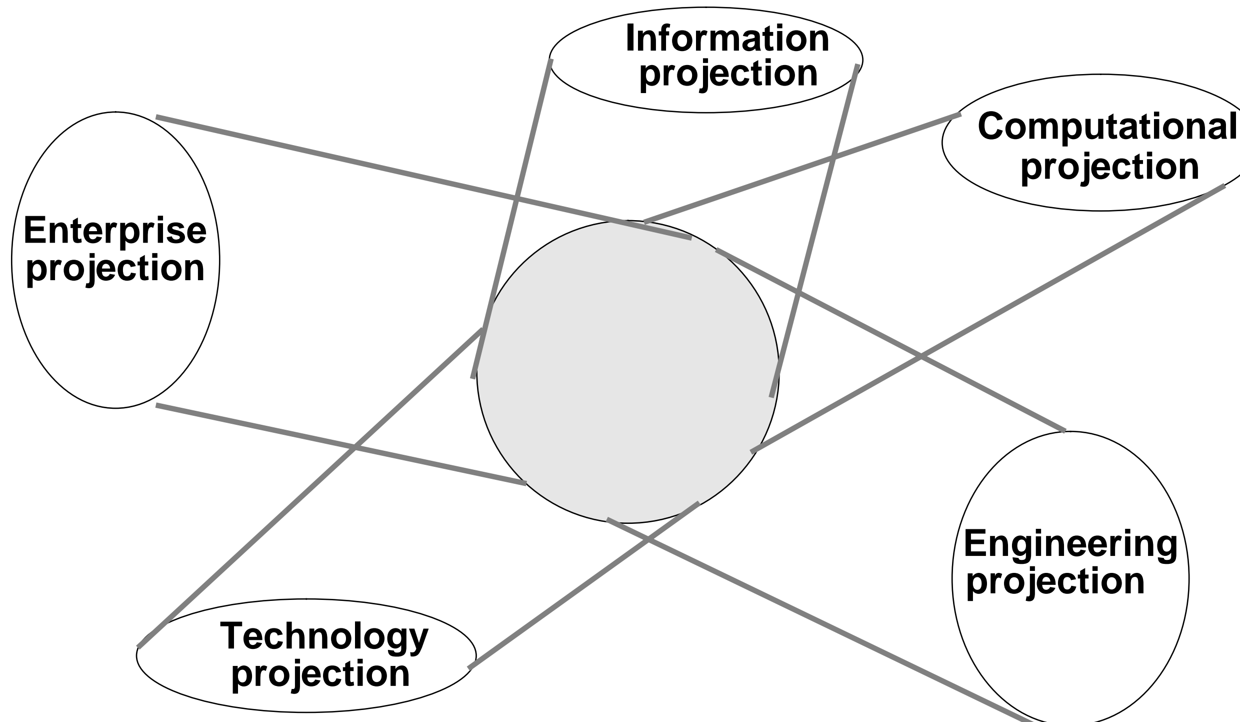
Integrated Approach - five projections

- ANSA identifies five viewpoints, called "Projections", on distributed systems:
- *Enterprise* - describes the enterprise and hence the overall objectives of the system.
- *Information* - information requirements within the system.
  Computational - abstract model of distributed processing.
  Engineering - a design for realising the computational model.
  Technology - implementations of hardware, operating systems, compilers etc.

ANSA

The five projections

- **ANSA is defined with reference to five related models of distributed systems. These are not layered.**



ANSA & ANSAware

- **Most work to date (and this tutorial) concentrates on the ANSA Computational and Engineering Models.**
- **ANSAware design is an *example* implementation of the Engineering Model - Note: not a *reference* implementation**
- **ANSAware is a set of tools, libraries, and run-time services to make developing distributed applications easier**
- **ANSAware is portable - it can run in conjunction with many different technologies.**

The Problem Space

- Systems are required to operate in "open" distributed environments
  Systems are heterogeneous and under different administrative domains
- Two important viewpoints and models:
  - ANSA computational model (application designer’s viewpoint)
  - ANSA engineering model (system builder’s viewpoint)
- Concept of service from two perspectives:
  - an abstract speciifcation of a set of system or application functions
  - an engineering entity that implements these functions

Distributed Systems - Properties/Issues

- **Separation**
- **Heterogeneity**
- **Federation**
- **Concurrency**
- **Scaling**

Distributed Systems - Assumptions to avoid

- Single global name space
- Synchronous interaction
- Homogeneous environment
- Sequential execution
- Fixed location

- Global consistency
- Locality of interaction
- Global shared memory
- Total failures
  - Direct binding

Approach to Separation

- Assume all services are remote, allowing co-location as an optimization
- Require each service to be entirely responsible for operating on its encapsulated data
- Perform all interactions with services via instances of (pre-defined) interfaces
- Allow propagation of interface-references as the means of acquiring access to services
- Name and report all detected interaction faults and failures

Approach to Heterogeneity

- **Assume heterogeneity and identify unnecessary diversity**
- **Abstract away from unnecessary diversity, while still retaining the benefit of specializations**
- **Request remote services to manipulate their encapsulated data through interface-instances**
- **Pass only interface-references, rather than having some data-presentation syntax.**

Approach to Federation

- Allow each system to control its own policies and services locally
- Allow cooperating systems to negotiate the sharing of services
- Require cooperating systems to identify all available services via a context-relative naming scheme
- Provide a trading facility through which federated cooperating systems can organize and control sharing of services

Approach to concurrency

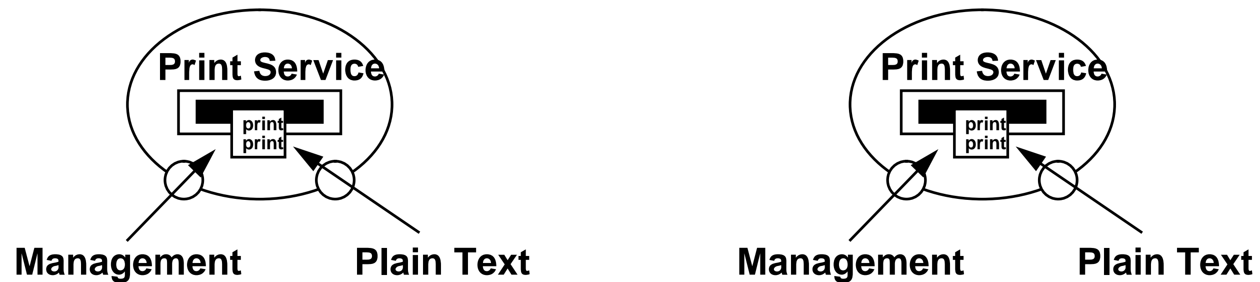
- Distinguish between the computational and engineering views of concurrency
  - i.e. specifying requirements for concurrency is kept separate from specifying the mechanisms to provide it
- Require declarative expression of parallel execution and concurrency control in the computational model
- Provide programmers with suitable linguistic tools for building distributed applications
- Provide engineering tools to map computational specifications to engineering mechanisms

Approach to Scaling

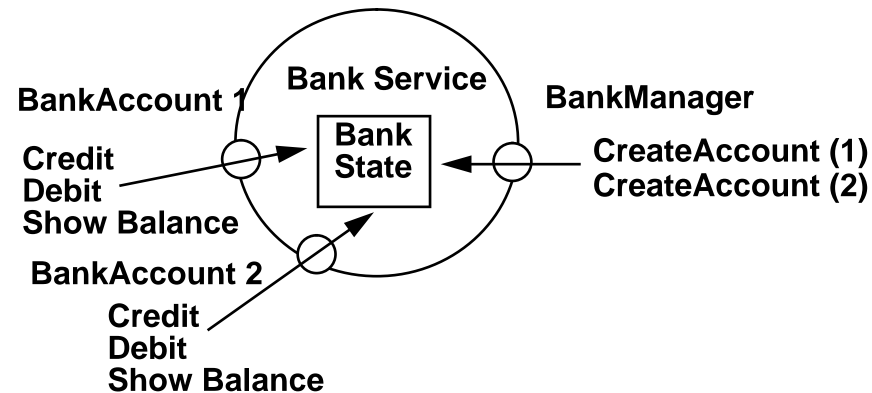
- Allow for scaling variability by building expansion capability into the architecture
- Provide extensible naming & trading facilities
- Federate through negotiable, cooperating, remote services
- Do not assume global mutable knowledge

Instances of the same object

- Instances of an object



- Instances of an interface



ANSA

Computational Model

Overview

Computational Projection

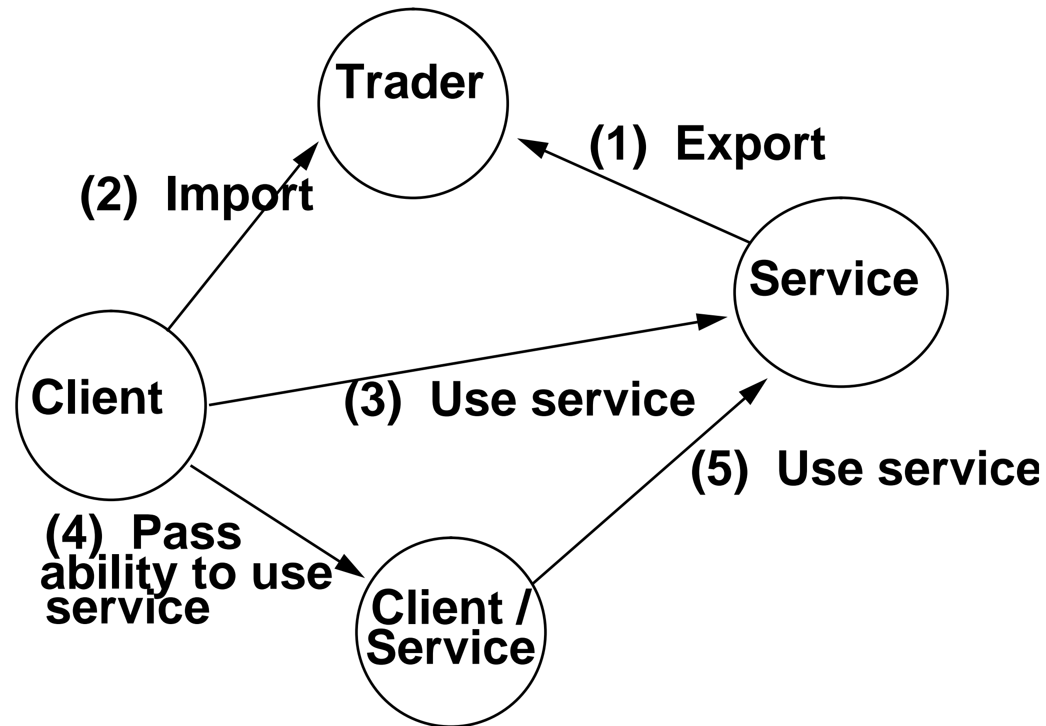
- View system as interacting computational entities.
- Logical partitioning of components.
- Independent of physical and logical configuration.
- An entity specifies its behaviour and not resource management - this is private (encapsulated).

ANSA Computational Model (in one slide)

- **Based upon service provision and use.**
  - **One entity/object may simultaneously provide and/or use multiple services.**
  - **Services can be dynamically created and destroyed.**
- **Ability to use a service may be passed from one object to another.**
- **Providers and users are typically put in touch via a "Trading Service" or *Trader***
  - **Dynamically adding and removing services allows dynamic re-configuration**

ANSA

ANSA Computational Model (cont'd)



Summary of Restrictions on Interaction

- Must assume everything is remote.
- Can’t manipulate external state.
- Indirect procedural access.
- Diversity of data representations.
- Don’t know the encoding of remote data.
- Can’t manipulate data representations directly.
- Can’t transfer data representations.
- Approach: Use references to Abstract Data Types (ADTs).

Using references to ADTs

- **ADTs - all access to external data is procedural**
- **Service describes how it can be used in terms of ADTs**
- **Can only distinguish between entities by interaction. Cannot examine their representations to determine equality.**
- **Can only transport references to ADTs**
- **Call them *Interfaces***

Interfaces

- *Interface* = Point-of-access to a service

- A computational entity (service) has one or more interfaces

  - Service encapsulates state
  - Interface is service provision point

- An ADT describes an interface to a service

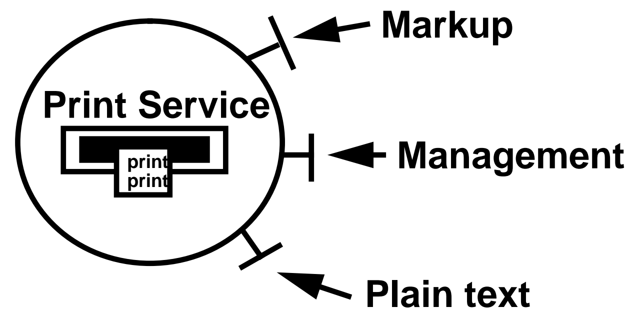
Services & Interfaces - Example

- Considering everything as services & interfaces allows implementation details to be hidden from the service user
- Consider a set of printing services providing capabilities:
  - printing plain text documents
  - printing documents in some markup language e.g. PostScript (TM)
  - a management interface - providing capacity to start & stop print queue, delete print jobs, other administrative tasks relating to printer
- Show four different configurations for implementing this

ANSA

Possible Structures for printing services (1)

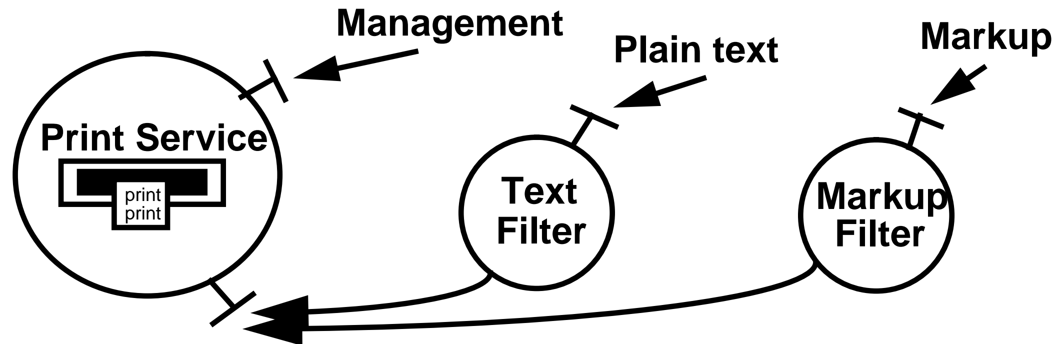
- **One object/entity presents plain text, markup & management interfaces**



ANSA

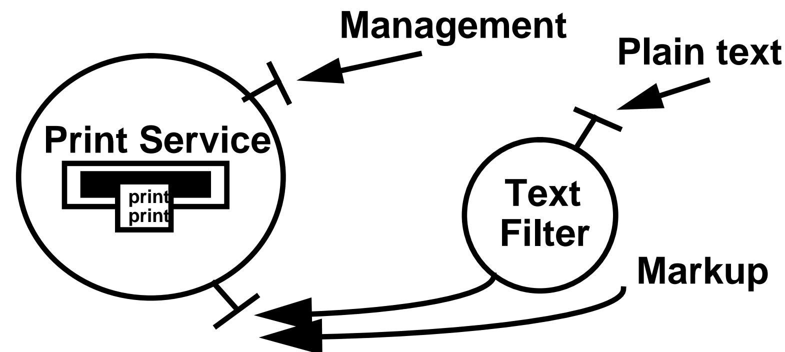
Possible Structure for printing services (2)

- **Raw printer presents management & control interfaces - filter services use these**



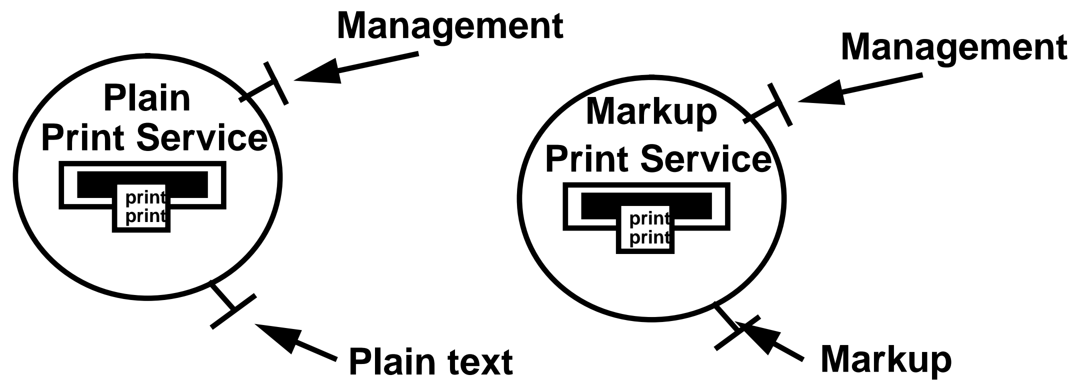
Possible Structures for printing services (3)

- **Printer supports markup-language - raw text converted to markup language for printing**



Possible Structures for printing services (4)

- **Two physically separate printers, one for plain text, one for markup-language, encapsulated as separate objects/entities**



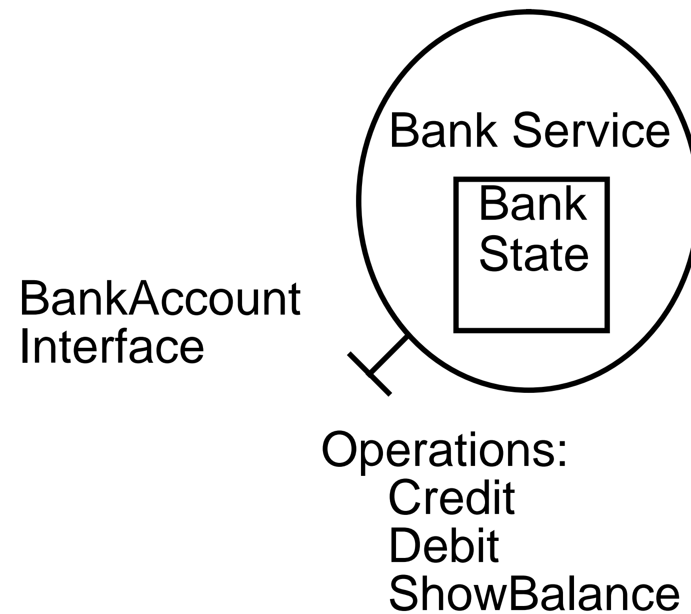
Interfaces, Operations

- An interface has one or more operations.
- Operations are the defined ways of using the interface.
- Like "methods" in traditional object-oriented vocabulary

Service, Interfaces, Operations - Example

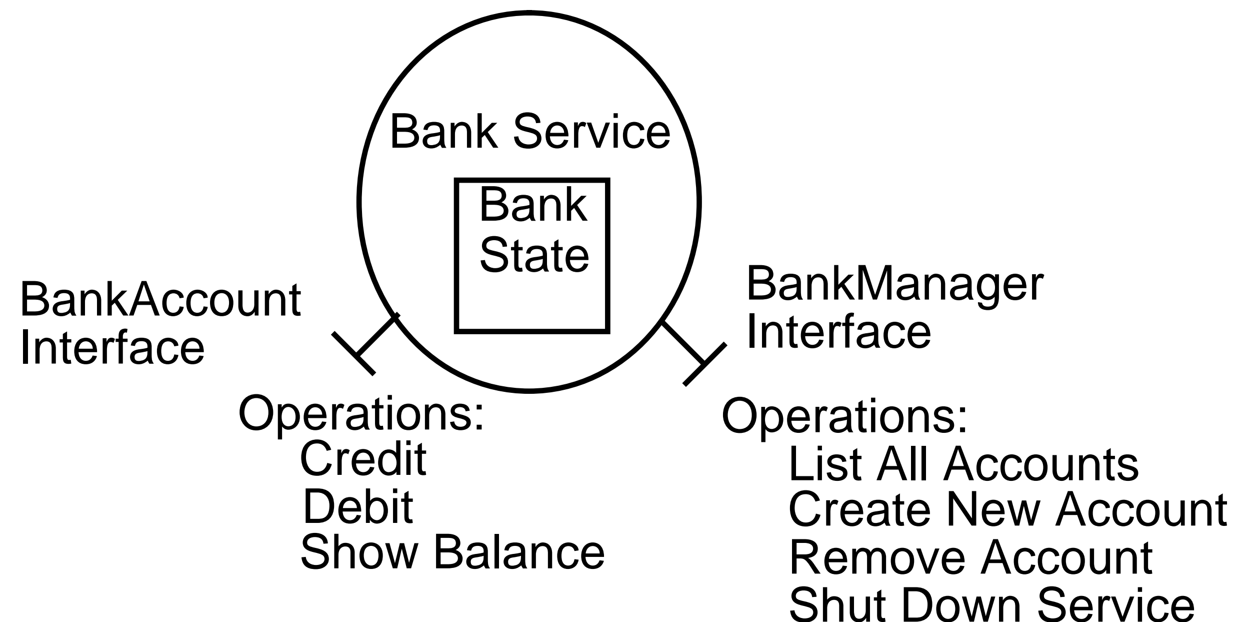
- Consider a bank account as service
- Has some state, such as account balance, name of customer etc.
- Has some defined set of actions that can be carried out on this state
  - Decreasing balance by writing a cheque
  - Increasing balance by depositing money
  - Inquiring account balance
- These actions are the only way of manipulating the bank account
- In ANSA terminology, these are called *Operations*

Service, Interfaces, Operations - Example (cont'd)



Service, Interfaces, Operations - Example (con't)

- **A service can have more than one interface**



Terminology - Interfaces & Interface References

- An interface is just the definition of possible interactions with the service
  - To be strict, we should probably say "interface-type"
- The actual things are *instances* of these interfaces
  - But we usually just say "interfaces"
- A service can present one or more instances of the same interface-type
- Instance means interface-type, together with its data
  - e.g. BankAccount - particular data (account number, name balance) is what distinguishes one instance from another
  - type is still same, though

Terminology - Interfaces & IfRefs (cont'd)

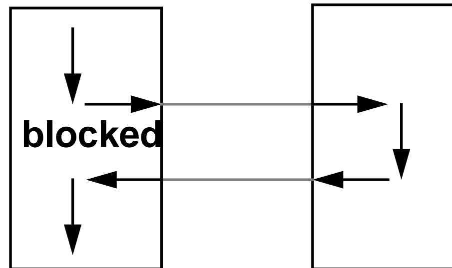
- Refer to an interface via an interface-reference (*if-ref*)
- Operations are invoked on interface references.
- Computationally, everything referred to is an interface-reference
  - Arguments and results are interface references

Operations

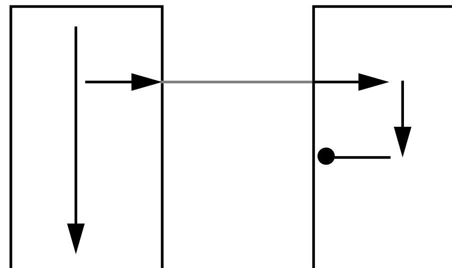
- **Operations manipulate local data.**
- **Operations can be described by four parts**
  - **Name, parameters, results, outcome (or *termination*)**
- ***Invoke* operations by name and arguments.**
- **Two kinds of operations: synchronous or asynchronous.**
  - ***Interrogation*: synchronous, reliable - return one of a set of named terminations with results.**
  - ***Announcement*: asynchronous, unreliable - do not return.**

Invocations

- **Interrogation -** synchronous:



- **Announcement -** asynchronous:



Terminations

- **An interrogation can have different possible outcomes**
- **Consider operations on the BankAccount interface described earlier:**

  Deposit (amount : Integer) : (newBalance : Integer)

  Withdraw (amount : Integer) : (newBalance : Integer)

  OR

  Withdraw (amount : Integer) : () -> InsufficientFunds

- **The latter outcome is called a *Named Termination*, and the normal outcome is an unnamed (anonymous) termination**

Terminations (cont’d)

- One termination may be anonymous.
- Each termination may return multiple results.
- The invoker of an interrogation distinguishes between different terminations by name.

Argument (and Result) Passing

- **Can only pass interface references (if-refs).**
- **Only the reference is copied, not the interface.**
- **Each end has a reference to a shared service.**
- **This is the semantic model - the implementation can make optimisations - i.e. use & transmit copies of immutable state, e.g. integer constants.**
- **Integer variables are mutable, may not be copied and interface-references to them are used instead.**

Shared State and Encapsulation

- **Operation state only has invocation scope.**
- **For state maintained across invocations**
  ⇒ interface state.
- **For separate copies of interface state**
  ⇒ interface instances.
- **For**
  - **sharing state between interfaces**
  - **defining granularity of distribution**
  - **defining scope boundaries for attributes (fault propagation, security, checkpoints etc).**

    ⇒ *Objects* (and object instances).

Objects - Summary

- **Objects enforce strict encapsulation (mutable state can't be shared between objects).**
- **Only objects can be re-configured or migrated (i.e. not just *part* of an object).**
- **Objects may have multiple interfaces.**
- **Objects may have concurrency.**

Encapsulation and Service Provision - Difference of ANSA from OOLs

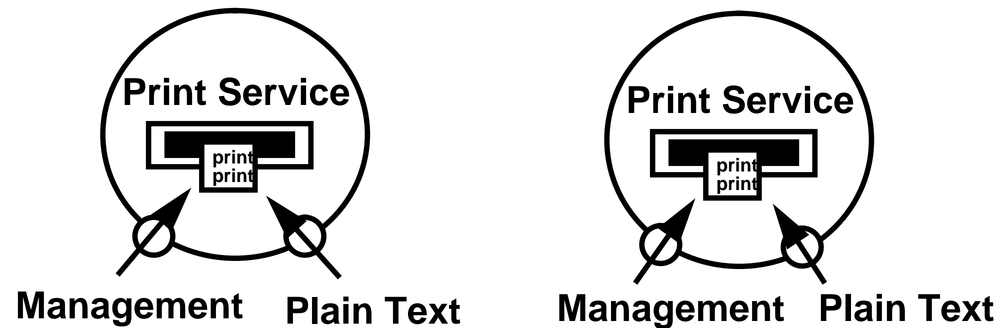
- **ANSA separates the notions of encapsulation and service provision.**
- **Objects provide encapsulation.**
- **Interfaces are the points of service provision.**
- **OOLs use one mechanism for both encapsulation and service provision, whereas ANSA has separated these functions.**

Instances

- **No interface or object classes.**
- **Interface instance constructors generate new interface instances when executed.**
- **Object instance constructors generate new object instances when executed**
  - **resulting object: 0 or more interfaces**

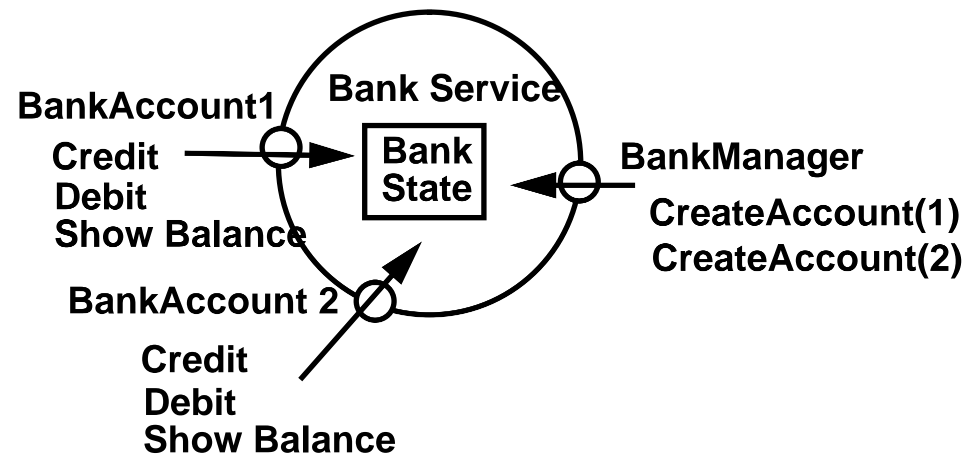
Instances of the same object

- **Two instances of a Print Service object**



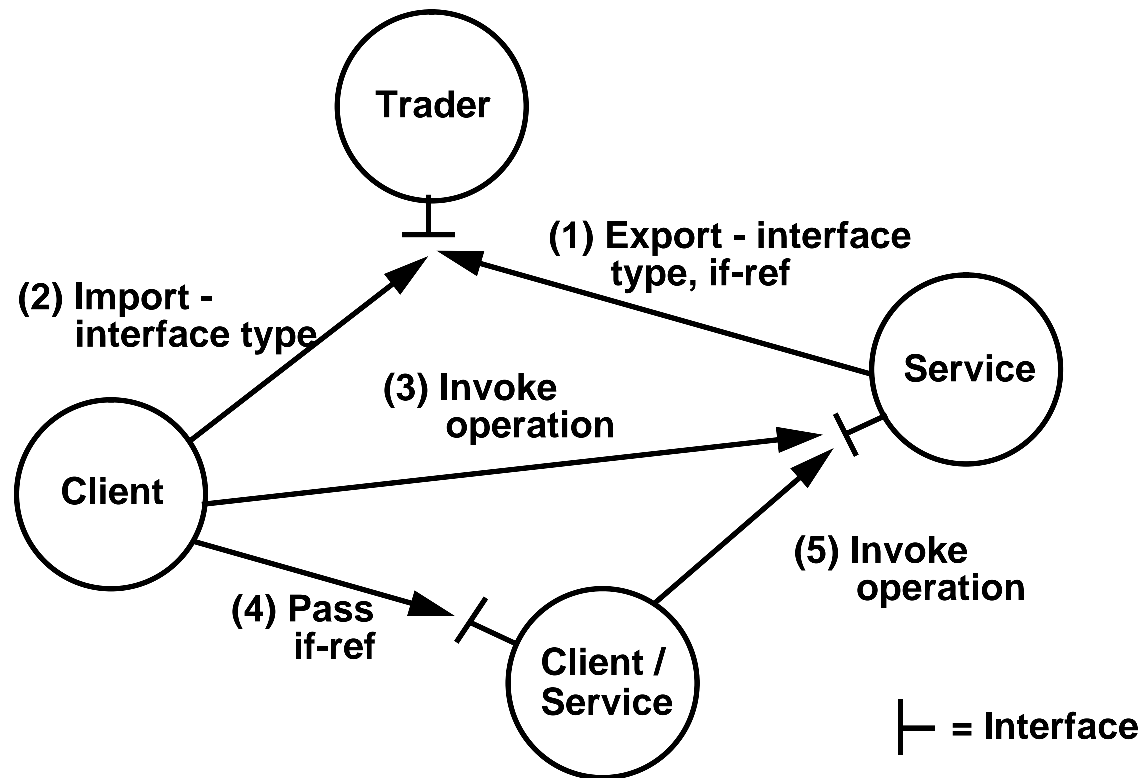
Instances of the same interface

- **Two instances of BankAccount interface**



ANSA

Using types in ANSA



Types

- Interface types are based on operation names and signatures.
- An operation signature consists of:
  - the number and type of its arguments.
  - the names and signatures of its terminations.
- A termination signature consists of the number and types of its results.

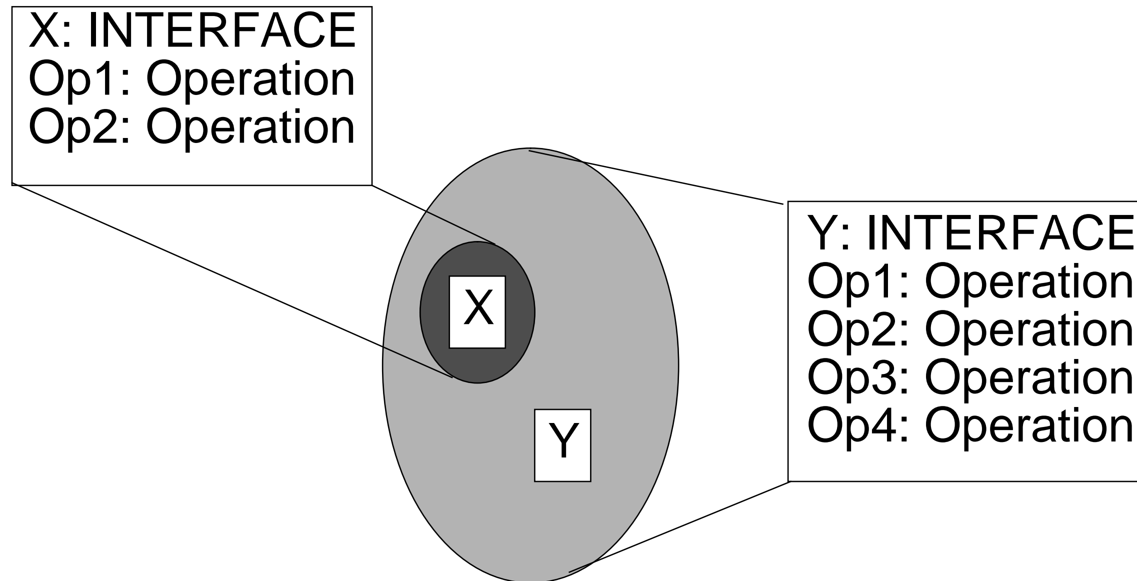
Type Checking

- The type provided must be checked against the type required before a binding is made.
- This can be done any time before binding.
- Dynamic binding requires dynamic type checking.
- Use conformance check rather than exact matching.

ANSA

Type Checking - Type Conformance

- **Type Y *conforms to* type X**



Conformance Rules

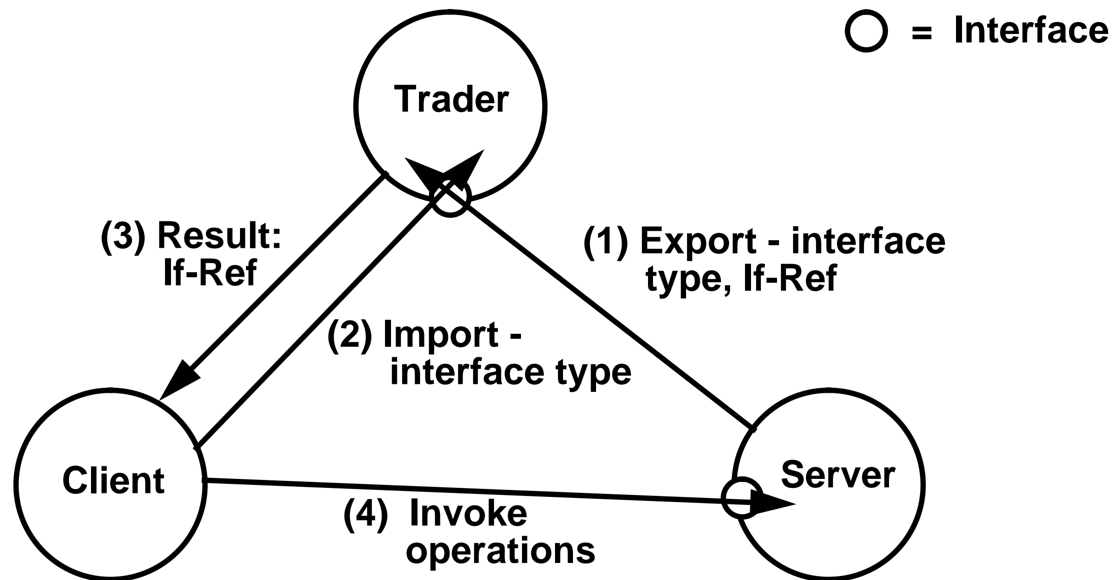
- Conformance of offer type to requested type assures that server & client can exchange information
  - Server does not receive unknown operations.
  - Client does not receive unknown terminations.
- All arguments and results of operations must also conform to defined types.

Trading

- Trading is the activity of choosing a service offer that matches the service requirement.
- The Trading service is provided as any other service
- A trader manages a database of service offers and matches requirements to offers.
- This matching is done on the basis of type conformance
- Once a trader has introduced a server to a client, it plays no further part in the interaction.

ANSA

Trading (cont'd)



- **A trader has no special privileges - it is a service like any other**
- **Interface references received from a trader are invoked like any others**

Implementing

ANSA

API's and Language Extensions

- Many existing systems provide a procedural separation between applications and systems:

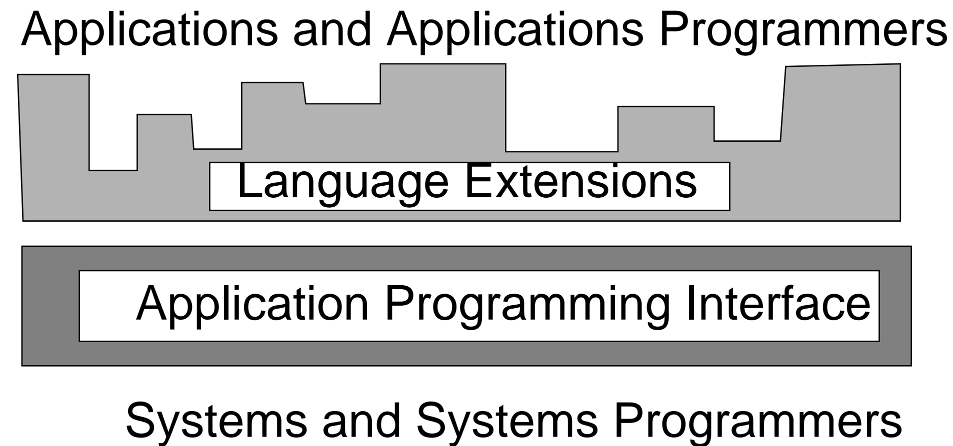
Applications and Applications Programmers

Application Programming Interface

Systems and Systems Programmers

- Allows too much of the system detail to show through
- Requires run-time checks to ensure correct use of API
- API widens as application requirements grow

- **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.**

- **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 4.0

Overview

ANSAware: Purpose

- To try out and validate the ideas and concepts described in the architecture.
- To allow users to try out the architectural ideas.
- To provide feedback into the architectural design.
- To be sufficiently well engineered for use in practical situations and products.

ANSAware: Components

- A run-time system providing the engineering components required to realise an instance of the architecture.
- A set of libraries used by all application programs.
- A set of tools for constructing applications.
- Tools (programming constructs) specifically targeted at aspects of distribution.

Components & Terminology

- ***Node***: a single machine or a cluster of machines which allows for the creation/destruction of processes with a unique process id.

- ***Nucleus***: an engineering object which manages the resources of a node

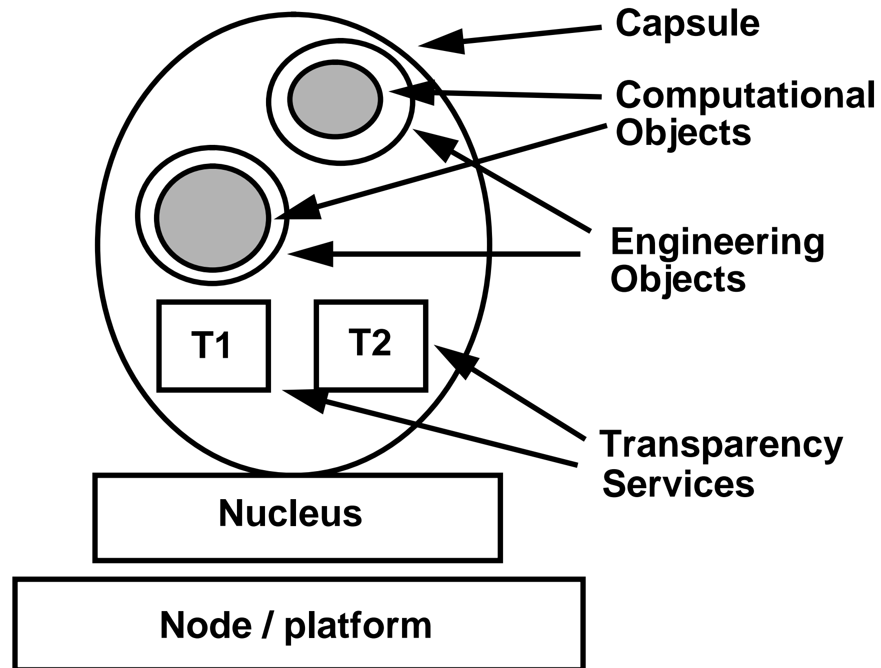
  - **This is implemented as a set of libraries that get linked together with the user's application to form a *capsule***

  - **Nucleus also referred to as *infrastructure* or *capsule library***

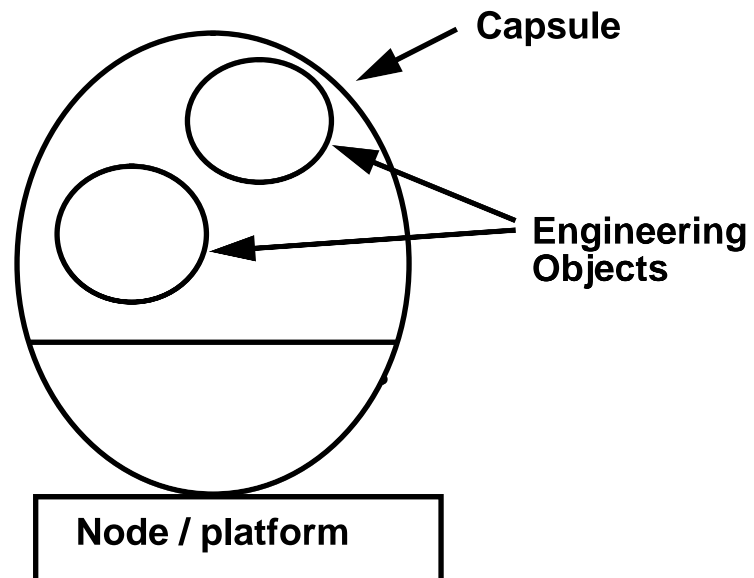
Components & Terminology - Capsule

- Capsule
  - is the unit of autonomous operation within ANSAware
  - is an address-space supporting a single instance of the run-time system.
- Includes:
  - efficient, transport-independent and portable RPC protocol.
  - light-weight threads.
  - synchronisation operations.
  - timer handling.
  - support for interworking with other systems - e.g. X11.
- On a multi-tasking operating system such as Unix, one node may support several capsules

Node, Nucleus, Capsule



Node, Nucleus, Capsule - Implementation



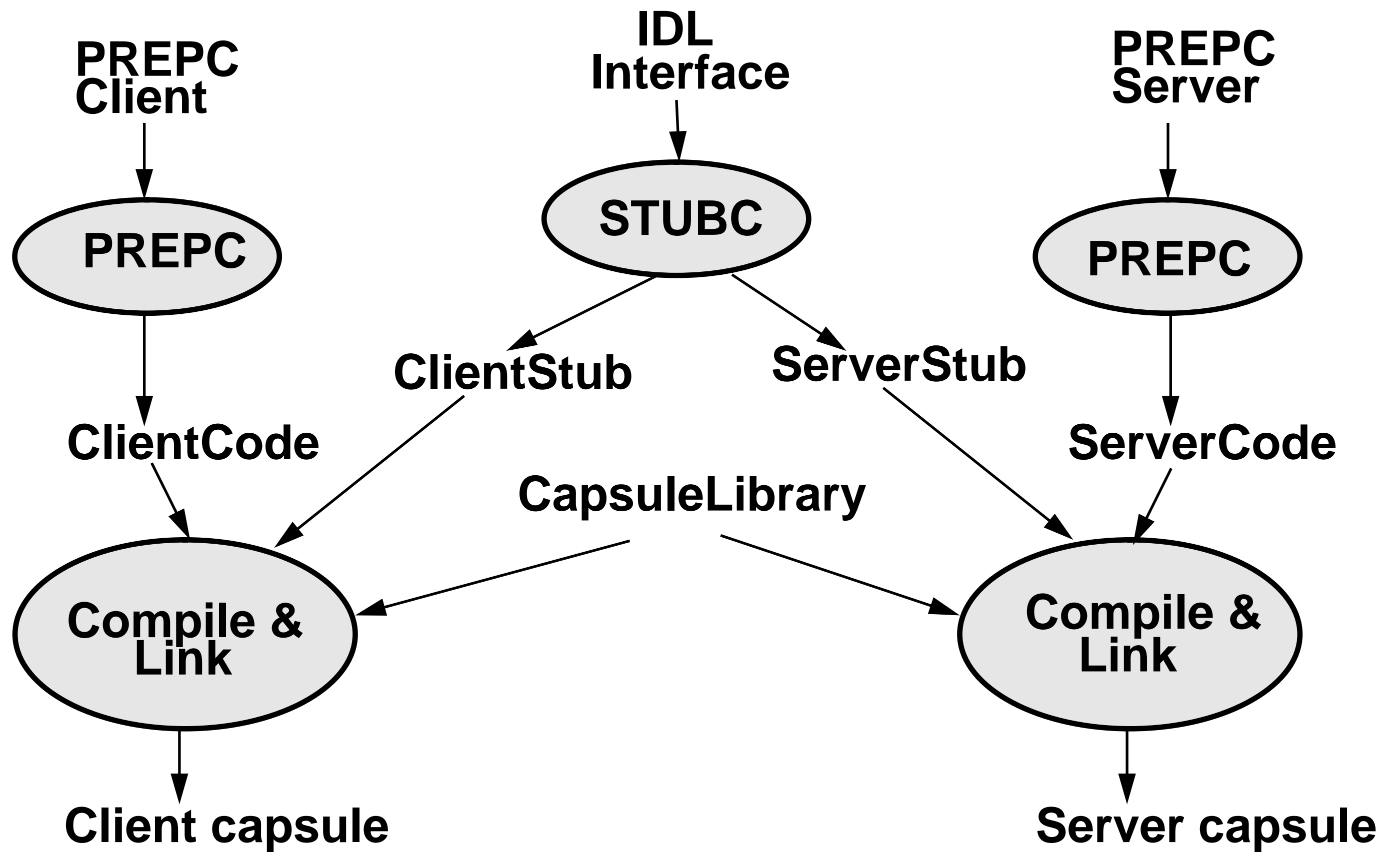
ANSAware Components (cont'd)

- *IDL*: language for defining service interfaces.
- *STUBC*: IDL compiler:
  - Generates marshalling/unmarshalling code (called *stubs*)
  - Interfaces with the underlying Capsule libraries.

ANSAware Components (cont'd)

- ***PREPC Language*: statements which can be embedded in C programs to access and implement services.**
- ***PREPC Pre-processor*: translates PREPC statements into C code which interfaces to:**
  - **Capsule libraries.**
  - **STUBC stubs.**

Components (cont'd)



ANSAware Services

- *Trader*: provides run-time matching of service requests to available services.
- *Factory*: dynamic creation and destruction of services on a single node.
- *Node Manager*: per-node service management.

ANSAware is Portable

- ANSAware runs on the following operating systems:
  - UNIX
  - VMS
  - MS-DOS
  - Chorus
  - Wanda (68K, ARM, DEC Firefly)

ANSAware is Portable (cont'd)

- **UNIX variants:**
  - **HP 300 and 800 series - HP/UX version 7.0 and 8.0.**
  - **DEC VAX - ULTRIX 3.2.**
  - **DECStation 3100 - ULTRIX 3.2.**
  - **Sun3, Sun4 - SunOS 4.0 and 4.1.**
  - **Acorn R140 - RISCIX 1.13.**
  - **Apple A/UX.**

Intro to Using ANSAware

- **Purpose:**
  - **Show basics of building a distributed application.**
  - **Familiarise yourselves with how ANSAware works in practice.**
- **Outline:**
  1. **ANSAware trader - what it does, how to use it**
  2. **Exporting & Importing offers**
  3. **Example service - "Echo" service**
  4. **Building & using Echo client.**

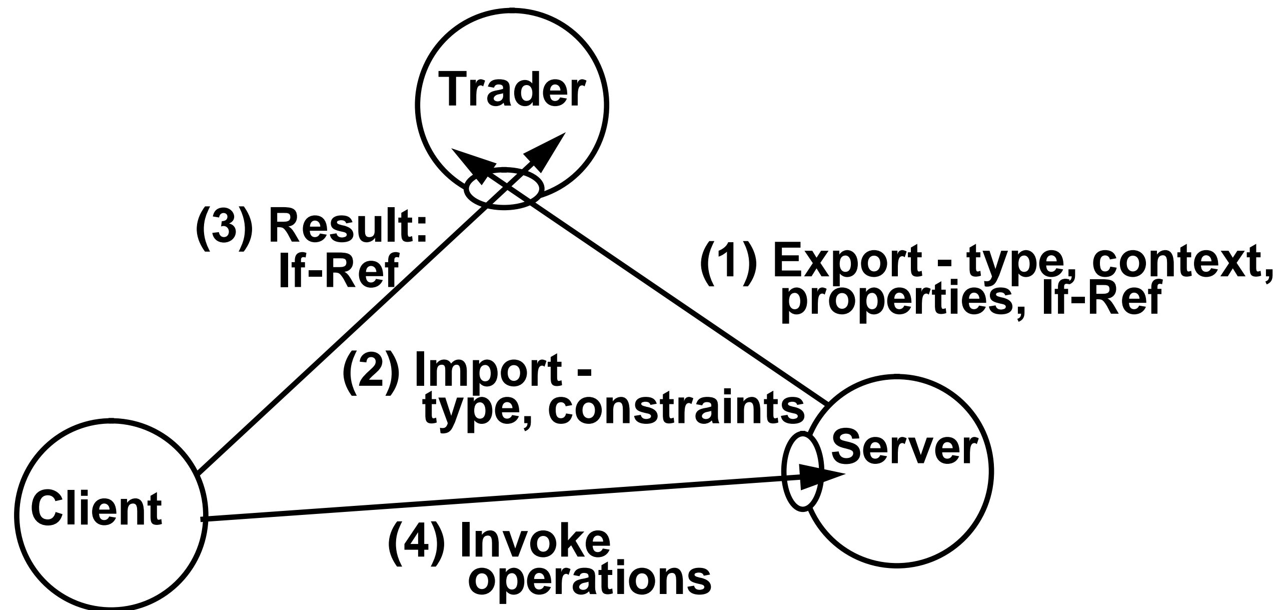
ANSAware Trader

- Trading is the activity of choosing a service offer that matches the service requirement.
- A trader manages a database of service offers and matches requirements to offers.
- *Type*, *context* and various (optional) *properties* are associated with each offer
- *type* is just the name of the type
  - e.g. Printer, Trader
- *context* is the (unix-like) pathname of the offer in the trader's database
  - provides a way of organizing offers

ANSAware Trader (cont'd)

- properties: (Name Value) pairs
  - describe the offer in more detail
  - could be anything e.g. (Cost-per-page 2p), (NodeName basilisk)
- A trader is just an ordinary service
  - but clients automagically receive interface-reference for local trading service on creation - this is built-in to the infrastructure
- Once a trader has introduced a server to a client, it plays no further part in the interaction.

ANSAware Trader (cont'd)



- **Service offers are traded via *Export* & *Import* statements**

Export

- **A server exports an offer of a service to trader specifying:**
  - ***TypeRef*: a type specification.**
  - ***Context*: path in the context graph.**
  - ***Properties* (optional): a list of (name, value) pairs.**
  - ***Interface Reference*: reference to the interface of the service**
    - contains addressing information used by ANSAware infrastructure to determine where messages should be directed

Export (cont’d)

- Consider an example service called "Echo", which is passed a string, and returns that string
- It might give these parameters to the Export call:
  - type - Echo
  - context - /ansa/testservices
  - properties - User Jane Node basilisk

Import

- A client imports a service from the trader specifying (TypeRef, Context, Constraints).
  - constraints specify allowable property values - this is optional
- The trader searches the Context (and any below it) for all offers which conform to TypeRef and which satisfy constraints - it then chooses one at random and returns the InterfaceRef to it.
  - client could choose to look up all matching offers, rather than just one, then choose one or more itself

Hands-on Example

Check that the trader is working:

```
prompt> trtest
59.16 calls/second, 16.902 ms/call, \
1000 Lookup calls completed
prompt>
```

- **Check what offers are in the trader:**

```
trclient search ansa /
```

  - **This displays all offers in the trader**

- **Build the Makefile for the Echo service for your machine-type:**
  - `ansakmf` **- command to create Makefile from Imakefile**
  - `make depend` **- adds dependencies to Makefile**

Hands-on Example (cont'd)

- Build client and server
  - `make server` - builds "server" capsule
  - `make client` - builds "client" capsule
- start Echo service
  - `server &`
- check that offer has been posted to Trader
  - the context for these examples is /ansa/testservices

```
trclient search Echo /ansa/testservices

trclient search Echo /ansa/testservices "Node == 'machine'"
```

Hands-on Example (cont'd)

- start the Echo client

```
prompt> client
echo> a string to be echoed
mach-name:  a string to be echoed
echo>
```

- CTRL-C to kill
- Run the client several times & see what is different
- Kill the server & note that it disappears from Trader's list
- run the client when there are no matching services & see what happens

ANSAware 4.0
In Depth

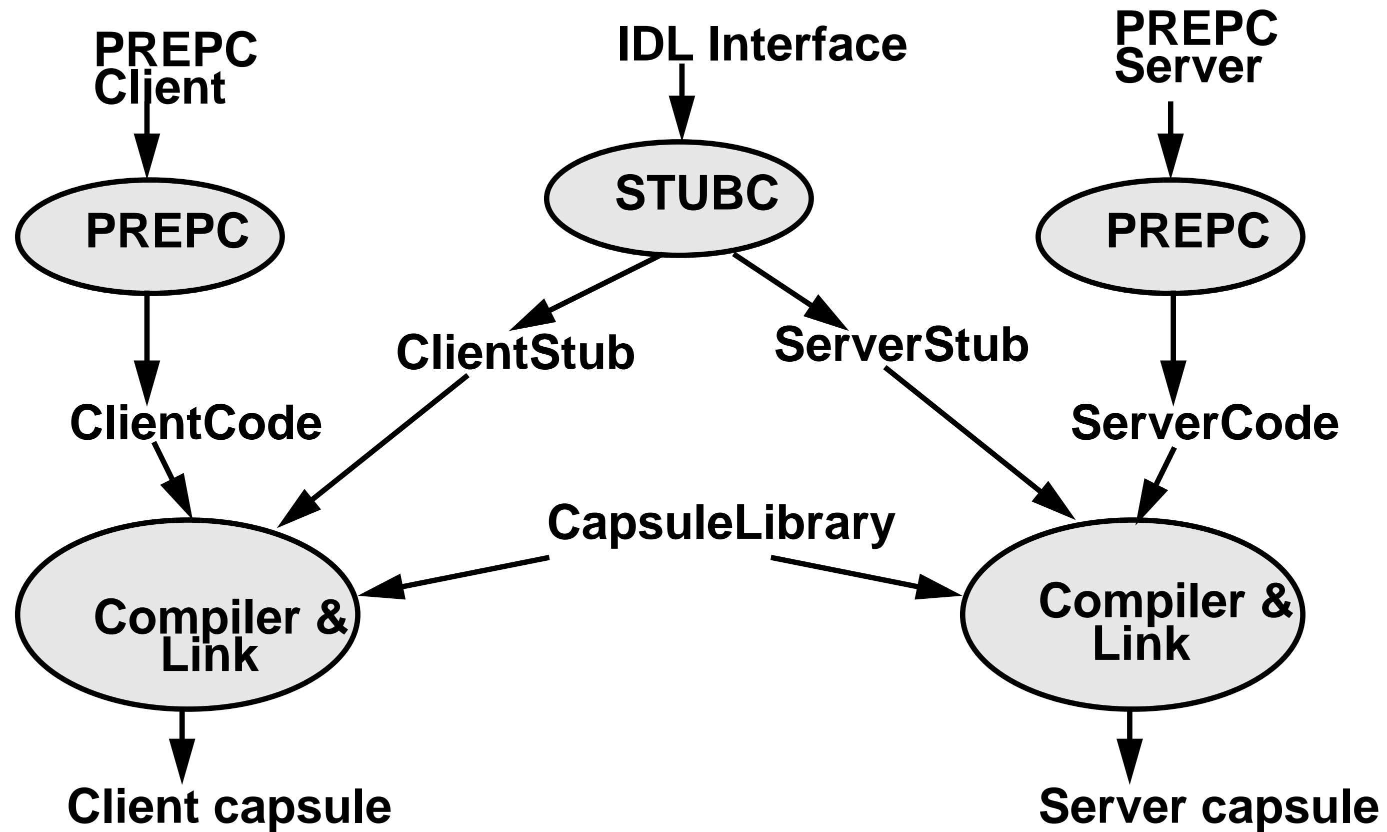
Speaker Notes

ANSAware 4.0 in Depth - Outline

- Examine ANSAware components in greater detail
- Hands-on: Extend *Echo* example
- Describe ANSAware trader in greater detail
- Describe ANSAware built-in facilities with an example
- A more complicated application-program
- Factory & Node Manager
- Other ANSAware features
- Installation hints

ANSA

Components - Review



IDL

- Specifies a common data exchange format
- STUBC (the IDL stub compiler) provides a programming language interface to the data format
- Application writers specify their service interfaces using IDL
- STUBC compiles these specifications to generate the appropriate stub code for the defined operations

IDL (cont'd)

```
Echo : INTERFACE =
-- Comment lines start with two dashes
BEGIN

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

Reverse: ANNOUNCEMENT OPERATION
  [ Src: STRING ]
  RETURNS [ Res: STRING ];
```
Note: names optional for results, could just have [STRING]

```
END.
```

Have mentioned interrogation/announcement - interrogation is the default, can specify announcement if desired

ANSA

- **IDL provides a set of built-in concrete types:**
  - **BOOLEAN**
  - **[SHORT] CARDINAL**
  - **[SHORT] INTERGER**
  - **[LONG] REAL**
  - **OCTET (signed)**
  - **CHAR**
  - **ENUMERATION**
  - **STRING**
- **IDL provides concrete type constructors:**
  - **ARRAY OF Type (fixed size)**
  - **SEQUENCE OF Type (variable size)**
  - **RECORD [Types]**
  - **CHOICE (discriminated union)**

More IDL

```
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-ins

- **An abstract type definition is provided for `ansa_InterfaceRef`.**
- **Every interface has an automatically defined type as follows:**

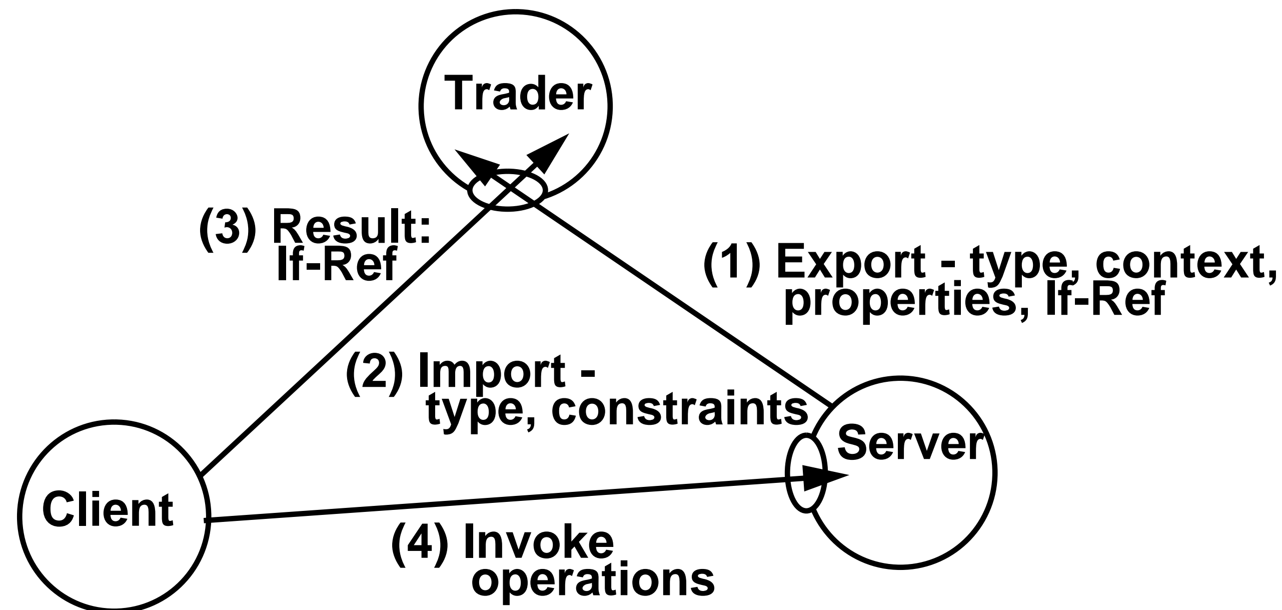
```
X: INTERFACE =
BEGIN
   XRef: TYPE = ansa_InterfaceRef;
END.
```

PREPC

- C preprocessor.
- Write clients and servers in C.
- Embed PREPC statements in C.
- PREPC/C provides implementation for clients and services.
- PREPC allows operations provided by a service interface to be invoked by clients.
- PREPC provides access to all services: i.e. the trader, factory, node manager, in addition to user written services.

ANSA

PREPC (cont'd)



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

PREPC code

- **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 code (cont'd)

- **Server:**

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

Export statement attempts to ensure that exported offer is removed from Trader on capsule termination

PREPC code (cont'd)

- **Server Operations:**

```
Interface_Operation (
  _attr, /* always required */
  args,  /* arguments */
  results /* results (pointers) )

{
/*  do the operation */

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

PREPC & IDL - future

- Currently PREPC and IDL is the only way.
- Support C, looking at C++, other projects are already using C++, Objective C and other proprietary languages.
- Eventually PREPC and IDL will be merged into a single language - thus avoiding the need for two compilers/preprocessors etc.
- Enhancements will include support for group execution protocols, atomicity, transactions and multimedia.

Echo example in detail

- **Looking at the syntax of the components of the *Echo* example**
- **Motivation: Adding the operation *Reverse* to its specification**

Echo Interface

```
Echo : INTERFACE =
BEGIN
Echo: OPERATION [ Src: STRING ]
      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", \
                              "", ir )
}
```

Note here that body() fn just ends, but does not mean capsule dies - capsule hangs around, waiting for ops to be invoked - capsule won’t die while interfaces to it exist

Echo Server (cont'd)

```
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 */
}
```

Imakefile

```
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),)
```

Modifying and Extending Echo

- Changing the implementation of the server consists of simply changing the implementation of the server procedures.
- New operations can be added by:

  1. Adding the new operation to the IDL file (and compiling).
  2. Adding the new server procedure to the server implementation.

- Modify client, or write new one, to use new operation

Building & Running Echo

$ansamkmf

creates a Makefile from the Imakefile.

$make depend

create the required dependencies.

$make

create client and server programs.

$server &

to run the server.

$trclient search Echo /ansa/testservice

to find all of the available servers.

$trclient search Echo /ansa/testservices "Node == 'yourmachine'"

to find the server on your node.

Building & Running Echo (cont'd)

$client < Imakefile

to run the client and echo the contents of the Imakefile onto your screen. A random server will be used.

$client "Node == 'yourmachine'" < Imakefile

to use the server on your node.

Syntax Note

- **Import and Export PREPC statements use different syntax for specifying properties:**
  - `Export( type, context, "Node machine")`
  - `Import( type, context, "Node=='machine'")`
    specify constraint-expression (more on this later)
  - `Export( type, context, "Node machine Name Jane ...")`
    space-separated list of name-value pairs

Now go off & try adding Reverse operation to the echo service.
Don't confuse them by talking about a conforming service now, don't think - mention after

Creating a conforming service

- If an interface-type *StringOps* conforms to type *Echo*, it provides at least the functionality of type *Echo*, & possibly more
- A new interface, which is an extension of an existing one, can be created using the IS COMPATIBLE WITH statement.

```
StringOps: INTERFACE =
IS COMPATIBLE WITH Echo;
BEGIN
new operations

END.
```

Conforming service (cont'd)

- StringOps conforms to Echo and can be used in its place - a client that imports type Echo may obtain an IfRef for either Echo or StringOps type of interface.
- code for *StringOps* needs to provide at least all operations of Echo
- *StringOps* type would also need to be registered with the trader (more on this later)

ANSAware
Trader
In Depth

ANSAware Trader - Details

- **An implementation of the trading service, including federation.**
- **The ANSAware Trader presents 4 distinct interfaces:**
  - **Service interface - of type *Trader* - for adding, removing and searching for offers**
  - **3 management interfaces - types *TrFed, TrCtxt, TrType* - for federation, shutting down, managing type & context-space**

Trader's *Trader* Interface

- Operations: Register, Lookup, Delete, Search
- *Register* - causes new service offer to be put in trader's database
- *Delete* - removes offers having the given Nonce
- *Lookup* - causes Trader to search its database for one or all matching offer(s) - depends on specified *Policy* - `Lookup_Random` or `Lookup_All`

  e.g. `traderRef$Lookup(...)`
- These are distinct from $Import, $Export PREPC statements
  - pseudo-operations which make use of these Trader operations
- user-program `trtest` does 1000 Lookup operations and prints time taken

Trader Interface (cont'd)

- **user-program `trclient` does Search on given type, context & constraints**

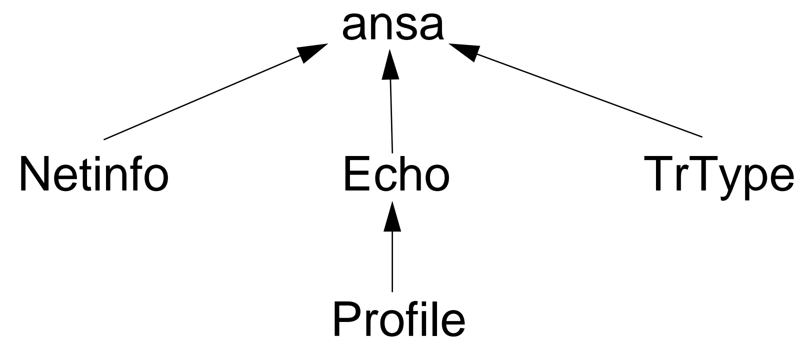
  trclient search ansa /

- **user-program `offerMgr` does *Lookup* every few seconds & displays updated results**

Trader notion of Type

- Trader uses simplification of computational model type system
- All types have names
- Each type has one or more immediate supertypes
  - Exception is root type, conventionally called *ansa*
  - Relationship of new type to others is given when registering type with Trader
  - Trader maintains DAG (directed acyclic graph) representing type relationships

Trader's type graph



- **conformance of offer type to requested type assures that server & client can exchange information**
  - **Whether the exchange actually means anything is up to the implementations of client & server**
  - **Type relationships are asserted by user, and checked at invocation time**

Trader - Type simplifications

- **Only offers of known types (i.e. in graph) can register with Trader**
- **Type conformance represented in DAG is based on what a user has asserted**
  - **sanity check at invocation time**
- **Type checking is done by name**

type space manually maintained at present - no automatic conformance-checker
- to assert type conforms to another type, tell Trader in advance

-future: automatic conformance checker so that relationship of your type to other types can be deduced or specified in offer-posting rather than all this manual stuff

TrType - managing the type-space

- Trader’s TrType interface provides operations for managing Trader’s type-space
- user-program `typecl` uses this interface to manipulate trader’s type-graph
- `typecl` - manipulate type graph
  - add a new type (optionally specifying types to which it conforms)
  - delete an existing type
  - mask an existing type - i.e. prevent its use in anticipation of subsequent deletion
  - unmask a masked type
  - list existing types

Examples of using typecl

```
$typecl add Newtype ansa

$typecl del NewType

$typecl add NewType ansa/SBank
add failed - status = TNoSuchSuperType
```
wrong syntax: / not allowed

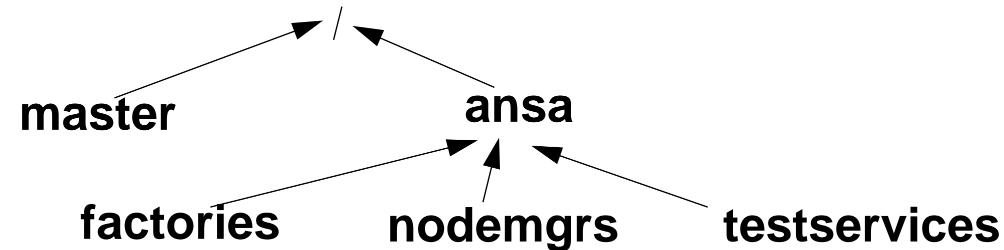
```
$typecl add SBank ansa
add failed - status = TAlreadyExists

$typecl add NewType t1 t2
```
[in this case, NewType conforms to both t1 & t2]

Trader notion of Context

- **Trader maintains hierarchy of contexts**
  - **Each contexts hold a number of service offers, and can have other contexts below it**
  - **Each offer is posted in a context**
  - **Use for administrative categorization of offers posted to the trader**
- **Trader uses Unix-like pathnames to name contexts in hierarchy**
  - **Root of the context-tree is"/"**

Managing Context-space

- **Context or name graph:**



- **offers have both type and context**
- **in Trader's offer-base, keeps track of all relationships:**
  - offers
  - types
  - contexts

Using ctxtcl

- `ctxtcl` - manipuate context graph
- `ctxtcl`
  - add a new context.
  - delete an existing context.
  - list existing contexts.

Using ctxtcl

```
$ ctxtcl list
/master
/ansa
/ansa/factories
/ansa/nodemgrs
/ansa/testservices
$ ctxtcl add /ansa/testservices/course
```

Trader Notion of Property

- Properties hold application-specific information about a service offer
  - e.g. for a print service: LinesPerMinute, PaperSize, costPerPage, Turnaround
- Property Name/value pairs specified when offer posted
  - e.g. Trader also automatically attaches (Type, TypeName) property to every offer posted
- Clients requesting services may specify
  - Acceptable property constraints (e.g. PaperSize=='A4')
  - Selection of offer with maximum (minimum) value of some property

Trader Properties (cont'd)

- **Property List (Export):**
  Node basilisk Price 100
  - name-value pairs, separated by spaces
- **Constraint Expression (Import):**
  (Node == 'basilisk') and (Price <= 200)
  - property constraint language (`==`, `!=`, `and`, `or`, `not`, ...)
  - full syntax of constraint language in manual (section 3.11.4)
- **properties passed (as strings) as parameters to trader-operations**

trclient usage

- `trclient`

    `lookup type context [constraints]`

    - finds all offers matching type, context and constraints, randomly selects one of these and returns its interface reference. Directly equivalent to PREPC Import.

    `search type context [constraints]`

    - finds and returns all offers matching type, context and constraints.

trclient (cont'd)

- `register type context [constraints]`

  - registers an offer.

- `delete nonce [constraints]`

  - deletes an offer given its nonce (end-to-end check data, used as an offer-identifier by the trader here) Note: trader should assign offer #s or sthg to avoid using nonce as id

  - normally services automatically remove the offers they have posted, so this is only used if something has gone wrong (e.g. service that posted offer has terminated incorrectly)

Federation

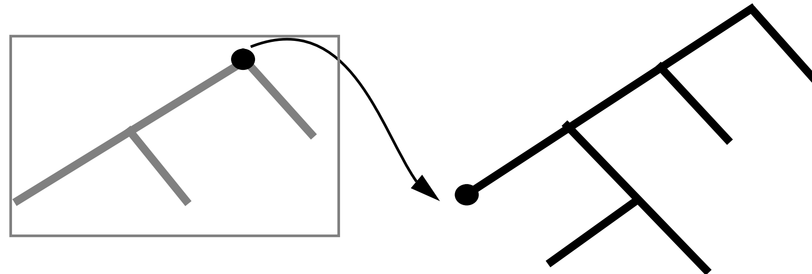
- Trading systems developed in isolation will eventually need to interwork.
- Federation allows interworking without resorting to a global context space.
- Need separate administrative domains.
- Domains can only be linked at negotiated points.
- Federation transparency means administrators can see the joins but users can’t.

Federation (cont’d)

- **Two types of trading federation:**
  - **bind a context in one space to a context in another**
    - ⇒ **requests targeted at bound context are forwarded to the federated trader which maintains the federated context.**
  - **bind an offer in one space to another trader’s space**
    - ⇒ **Lookup requests on the offer are forwarded to the associated trader for resolution. Such offers are referred to as "proxy offers".**

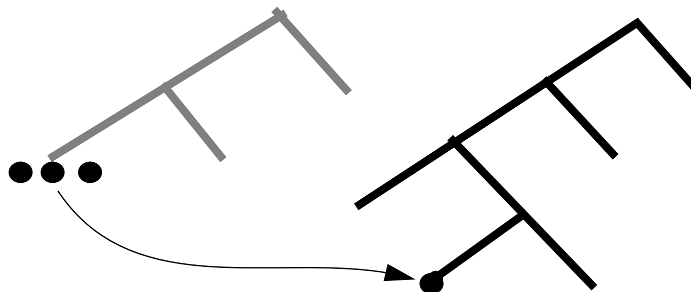
Federation (cont'd)

- **Federation via naming context:**



with current federation capabilities in AW, can only federate root of context to another -like UNIX mount
- computationally, sb able to do anything -mount any part of tree to anywhere, but this is implementation limitation

- **Federation via offers:** post one particular offer to another trader's space - can post anywhere



Federation (cont'd)

- Requirements for federation:
  - mutual agreement on common interface types.
  - appropriate mapping functions to maintain InterfaceRef guarantees (InterfaceRef's are guaranteed to be unique within a trading domain).

Trader - Current Implementation

- **Types: No run-time conformance checking by the trader. Conformance relationships are specified explicitly when types are added to the graph (DAG).** this means that you can fool trader by stating that a type conforms to another via typecl - checking is done at invocation time by the server whose operation is being invoked
- **Context: name space is hierarchical rather than DAG. Context name components are separated by a "/".**
- **Properties: All properties are optional, i.e. no way of forcing an offer of type X to specify properties (Name, x), (Host, y).**
- **Searching is implemented in a shallow manner, this is, federated traders are only consulted if the current trader cannot satisfy the import request.**

Removing service offers from Trader

- **When a service ends normally, it can explicitly withdraw any offers it has `Export`'ed via PREPC `Withdraw` instruction**
- **If a service is killed by a signal (CTRL-C), ANSAware infrastructure withdraws all `Export`'ed offers from trader before exiting the capsule** signal handler intercepts signal only done for offers posted via PREPC Export stmt (as opposed to other mechanisms that have not yet been mentioned
- **Trader will automatically remove offers it considers "stale"**
- **If a client cannot invoke operations on an interface-reference received from the trader, its infrastructure notifies the trader**

infrastructure provides following: if client's operation invocation times out (or some other failures that will be shown later) then will by default (default exception-handler - exception handler's explained later) attempt to inform Trader. Trader will attempt to contact service & if it can't, will remove from offer list

ANSAware Constructs In Depth

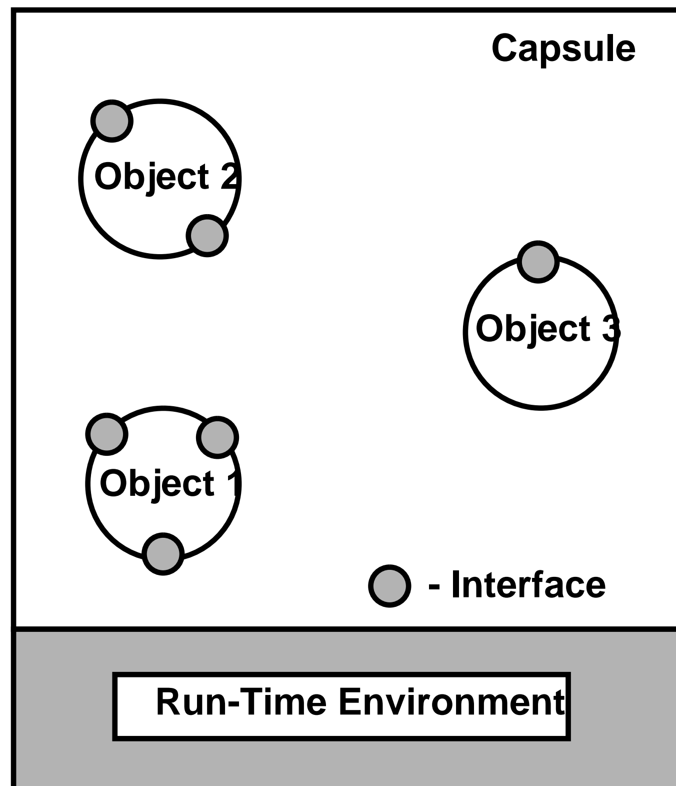
Capsules

- Capsule:
  - the ANSAware unit of autonomous operation.
  - represents an instance of the run-time environment.
  - a single point of failure for that environment.
- On UNIX, a capsule ≡ UNIX process.
- On MS-DOS, a capsule ≡ the entire machine.

Capsules (cont'd)

- A single capsule may support multiple objects.
- A single object may support multiple interfaces.
- Objects represent templates for the creation of one or more interfaces.
    - multiple, different, templates within the same capsule are distinguished by name.

Capsules (cont'd)



Capsule Library

- **Primary functions are:**
    - **RPC protocol implementation (REX)**
    - **Tasks and Threads**
    - **Event Counts and Sequencers (for synchronisation & ordering)**
    - **Timers**
- **Capsule library itself uses `main`, ANSAware applications must use**

```
body( ac, av, envp )
int ac;
char **av;
char **envp;
```

Binding

- Interface references are created by the server's capsule infrastructure and include all protocols and addresses of the service.
- Client's capsule infrastructure binds (i.e. end-to-end communication channel established) on first use, choosing the most efficient common protocol.
- Neither client nor server application knows its own or the other's address (or protocol used). This is all handled by the infrastructure.

Tasks and Threads

- **Tasks are the unit of concurrency.**
- **Tasks have a stack and save area for their CPU state.**
- **Threads are the unit of *potential* concurrency.**

thread is an independent execution path through a sequence of operations/statements/steps

- **Threads must be assigned to a task in order to execute.**

Task is like virtual processor that provides thread with the resources it requires - task is unit of actual concurrency

- **Tasks cannot be shared among threads - once a thread has been allocated to a task, that thread stays with that task until the thread is done.**

Tasks & Threads (cont’d)

- **Tasks may be pre-emptively scheduled and therefore threads cannot assume that they execute to completion.**
  - **cannot assume that they have sole access to a critical section - must use locking to be sure**
- **Tasks may also be non-pre-emptively scheduled and therefore threads that are in a tight loop, should pause (`instruct_Pause()`) to allow other threads a chance to execute** otherwise deadlock may result if no other tasks can be swapped in while thread blocked
- **Threads are much cheaper in terms of memory than tasks, because a task has all that stack space.**

If actual concurrency required for correct operation of cpt, then at least 2 tasks req’d, o/w cpt will deadlock

Tasks and Threads (cont'd)

- **Task creation is controlled by the application programmer:**
  - **static:**

```
GLOBAL ansa_Cardinal Ansa_InitialTasks = NUM_INITIAL_TASKS;
```

  - **dynamic:**

```
nucleus_tasks( (ansa_Cardinal) NUM_NEW_TASKS,
               (ansa_Cardinal)stack_size );
```

- **Calling `nucleus_tasks()` , with** 0 **(or a below-minimum size) stack size causes it to use the default stack size** which is fairly big, but depends on arch/O/S

Tasks and Threads (cont'd)

- **Threads are created by the capsule library as required**

- **Thread Fork, Join and Spawn primitives are supported for explicit dynamic creation of threads**

  - **These are `instruct_Fork()`, `instruct_Spawn()` & `instruct_Join()`**

Event Counts and Sequencers

- **Sequencers provide a sequencing mechanism amoung concurrent activities; library functions provided for using them**
- `ecs_ticket ( sequencer )`
  an atomic operation which returns the current value of `sequencer` and then increments `sequencer` by one.
- **Eventcounts provide a synchronization mechanism among concurrent activities.**
- `ecs_await ( eventcount, value)`
  blocks until the value of `eventcount` is at least `value`.
- `ecs_advance ( eventcount )`
  increments the value of `eventcount` by 1.
- **can use these for e.g. producer-consumer synchronization**
- **reference: Reed & Kanodia CACM 22(2) Feb. '79.**

Mutual Exclusion

- Mutual exclusion can be implemented using eventcounts & sequencers
- Macros are provided as follows:

```
ansa_InitMutex( ansa_Mutex *m )

ansa_AcquireMutex( ansa_Mutex *m )
```

blocks current thread until it can acquire mutex lock for crit section

```
ansa_ReleaseMutex( ansa_Mutex *m )

ansa_FreeMutex( ansa_Mutex *m )
```

Timers

- **Two forms of timers supported:**

  1. `timer_Sleep( unit, delay )`
     the calling thread is suspended for `delay` `TSeconds`, `TMilliSeconds`, or `TMicroSeconds` as specified by the `unit` argument.

  2. `timer_setTimer ( unit, delay, action, data, owner )`
     will cause the function `action` to be called `delay` units in the future with `data` as an argument. `owner` is for debugging purposes only

"action" is a function-pointer.

Concurrency: Initiating & Redeeming

- **Rather than invoking an operation, can *initiate* it**

Invoking:

```
! {result} <- IfRef$Operation( args )
```

Initiating:

```
!{voucher} := IfRef$Operation( args )
```

note syntax of := rather than normal <-

. . .other code. . .

```
! {result} <- IfRef$Redeem ( voucher )
```

Initiate & Redeem (cont'd)

- **With an interrogation operation, the thread will block until interrogation returns**
- **With an initiation, another thread is automatically started up to carry on with intervening code**
- **When `Redeem` is executed, thread blocks until operation returns**
- **`Redeem` is not a real operation (although it has the same syntax)**
  - **it is a PREPC statement**

execution continues after an initiate operation - thread will only block if results are not available when they are collected/redeemed

shorthand for explicitly creating thread via Fork which only does invocation & joining it again later, (see next slide)

Initiate & Redeem (cont'd)

- Doing an initiate & redeem is equivalent to:

```
dispatcher_fn ( arg )
{
! { result } <- IfRef$Operation ( args )
}
thread_id = instruct_Fork ( dispatch_fn, arg );

...other code...

instruct_Join ( thread_id );
```

- Easier to write & read - hides thread-management details from the user

Trying out Initiate & Redeem

- **Modify the *Echo* client from inoking operations to initiating several operations on same or different Echo servers**
- **You can *Redeem* in a different order from *Initiating***
- **Vouchers are declared:**

```
ansa_Voucher voucher;
```

- **Are extra tasks needed? May have to increase number of tasks.**

break here for doing exercise

Creating New Interface-Instances

Creating an Interface-Instance

- A service may may instantiate any number of interface instances
- The `$Create` statement explicitly creates an interface instance and creates an interface reference data structure.
- interface reference can be exported to Trader, or not
- Syntax for creating an instance of an interface

```
IfTypeName$Create( socket-concurrency )
```

e.g.

```
! {acct_ifref} :: Account$Create( 1 )
```

- Note special syntax `::` which goes with *Create* construct

Interface-Instances (cont'd)

```
! {acct_ifref} :: Account$Create( 1 )
```

- ***1* is the number of invocations the interface-instance agrees to simultaneously support**
- **Any more than this will be ignored, & they will have to retry (the infrastructure does this automatically)**
- **This is often called the *concurrency* of the interface's *socket***

Of socket, not interface. Note, however, that true concurrency of interface is determined by both this and number of tasks avail for executing poss || threads, and whether or not threads block

Instance-specific State

- **When multiple instances of the same interface exist, may need instance-specific state**
- **When the instance is created, some state can be associated with that instance**

```
! {ir} :: BankAccount$Create (1, acct_num)
```

- **If an interface has state, creation & destruction functions are needed to properly allocate and initialize that state, and properly free it when the interface is destroyed:**

```
ansa_StatePtr Iftype__Create ( [state_init_args] )

void Iftype__Destroy ( ansa_StatePtr *ptr )
```

this function needed to free any data structures that were allocated by `Create` fn to make sure unused memory doesn't hang around causing capsule to grow unneccesarily if no longer needed.

Initializing Instance State

- State stored in some locally-defined data structure (which gets cast to `ansa_StatePtr`)
- e.g.:

```
typedef struct acstate {
                                int acct_num;
                              } AcctState
...
ansa_StatePtr Account__Create ( int acc_num )
{
     AcctState *pAS;

     pAS = (AcctState *)system_allocate(sizeof(AcctState));
     pAS->acct_num = acc_num;
     return (ansa_StatePtr *) pAS ;
}
```

Accessing Instance State

- The state initialzed by the `IfTypeName__Create()` function is then stored by the infrastructure, and can be accessed by special calls
- To get state for the interface instance within which the current thread is executing:

```
ansa_StatePtr thread_getInterfaceState()
```

- To get state for any interface-instance within the same capsule,

```
ansa_StatePtr
     awifref_getInterfaceState( ansa_InterfaceRef *ref)
```

Instance State (cont'd)

- For example, in Bank Account example, need instance-state to determine account details for the current interface-instance, before the operation can be performed
- When creating interface-instance:
  `! {ir} :: Bankaccount$Create (1, acct_num)`
- accessing instance state within operation:

```
int acct_num;
AcctState *p_state;
...
p_state = (AcctState *)thread_getInterfaceState()
acct_num = p_state->acct_num;
...
```

Interface Operations

```
IfTypeName_OpName( _attr, args, results )
   ansa_InterfaceAttr *_attr;
   argTypeN                                    argN;
   resTypeM                                    *resultM;
```

- By definition, every operation has the first parameter `_attr`
  - contains a pointer to ifref for interface-instance within which current operation is executing:

    ```
    (ansa_InterfaceRef) _attr->attr_dst_ref ;
    ```

- Can be used within any operation if if-ref of operation's enclosing interface is needed

Interface References

- These are complex data structures, but the application-programmer should not normally have to look inside them
- To make a copy of an interface-reference, copying function is provided:

```
ansa_Status ifref_copyRef(
                          ansa_InterfaceRef *toRef,
                          ansa_InterfaceRef *fromRef );
```

- to free an interface-reference allocated in this way:

```
void ifref_freeRef( ansa_InterfaceRef *ref );
```

Destroying an Interface Reference

```
typedef struct acstate { int acct_num;} AcctState
ansa_StatePtr Account__Create ( int acc_num )
```
execution continues when remote activity complete

```
{
      AcctState *pAS;

      pAS = (AcctState *)system_allocate(sizeof(AcctState));
      pAS->acct_num = acc_num;
      return( (ansa_StatePtr *) pAS );
}
void Account__Destroy(ansa_StatePtr state)
{
         free((char *)state);
}
```

so the destroy function frees any state allocated by Create function

Destroying an Interface Reference (cont'd)

- **Creating (instantiating) an interface-instance**

```
! {ifRef} :: IfTypeName$Create( args )
```

this statement will result in this function getting called:

```
ansa_StatePtr IfTypeName__Create( args )
```

AND, and interface-instance is created, and an interface-reference to it is allocated and returned

- **Destroying an interface instance**

```
! {} :: IfTypeName$Destroy( ifRef )
```

this statement will result in this function getting called:

```
void IfTypeName__Destroy(ansa_StatePtr state)
```

AND, and interface-instance is automatically destroyed, and the interface-reference to it is freed

- **Attempting to use an interface reference variable before it is instantiated or after it is destroyed will lead to errors, as the variable does not contain any meaningful information**

Destroying an Interface Reference (cont'd)

- **A thread is not necessarily prevented from destroying the interface-instance within which it is executing**
  - **but this will lead to the operation not returning**
  - **the client that invoked the operation will time out**
  - **server fails to reply because the interface executing the operation has been destroyed**
- **Solution to this is to spawn a background thread to destroy the interface once the operation has completed**
  - **current method is to make the background thread delay for some length of time, then call `IfTypeName$Destroy`**

Simple Bank Example

Speaker Notes

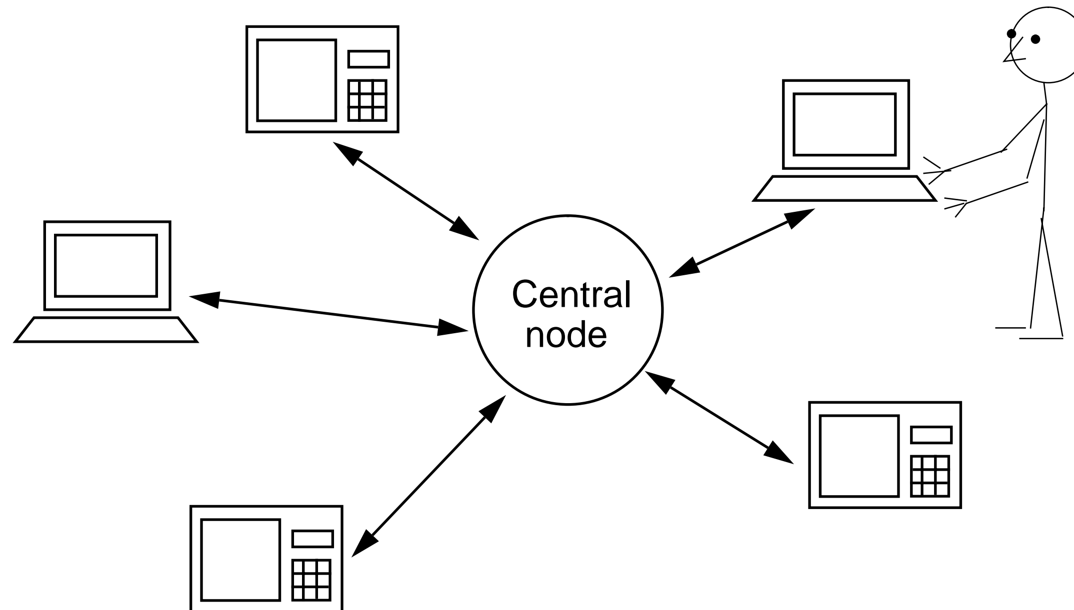
describe example service, then we will be trying it out - example is included with AW dist’n, but have fooled around with it to take out bits, and these bits will be for you to try out as example.

Simple Bank Example

- Design and implementation of a simple banking service
- Design a centralised Automatic Teller Machine (ATM) management system
- Manage a network of ATM’s all of which are connected to a central node
- Bank managers and their assistants administer bank accounts, they too are distributed around various local branches

ANSA

Bank Example (cont'd)



Bank Example (cont'd)

Q) What data structures are required?

- Personal accounts: customer name, account number, PIN, balance, time of last change

Q) Where is this data accessed from and what form does this access take?

- ATM access (guarded by PIN): read balance, credit and debit account.
- Manager access: create, destroy and list accounts, shut down bank service

Bank Example (cont'd)

Q) What failure modes must be handled?

- Failure of the central node

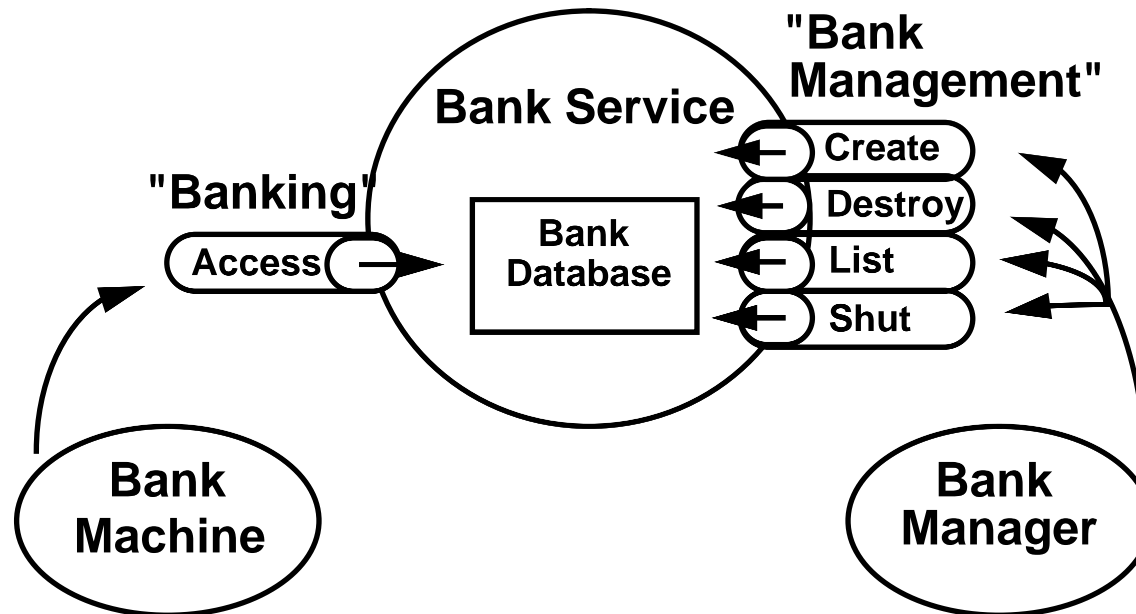
- Failure of individual ATM's (when in the middle of some transaction)

Q) How do we cope with these failures?

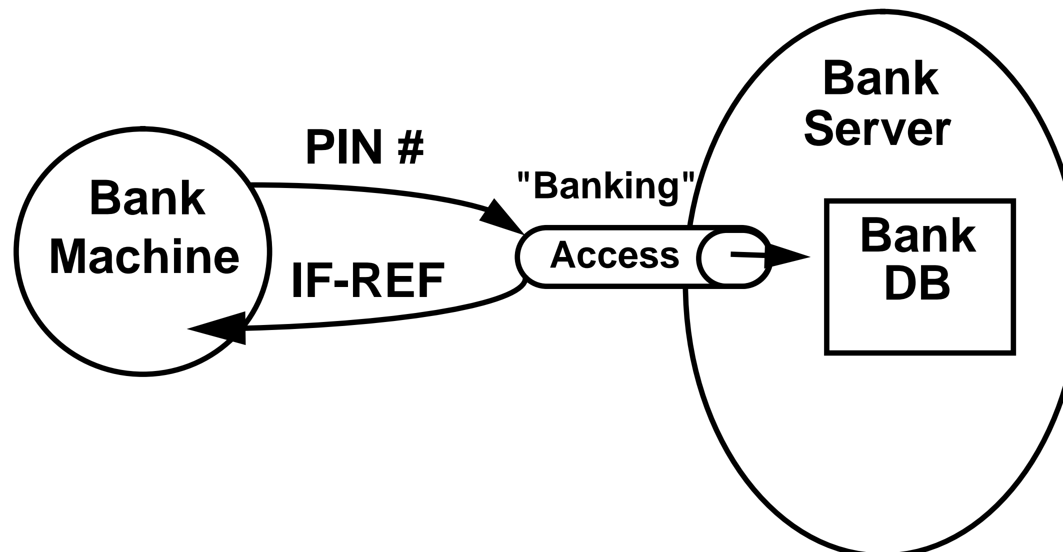
- Central node must checkpoint its state to persistent storage

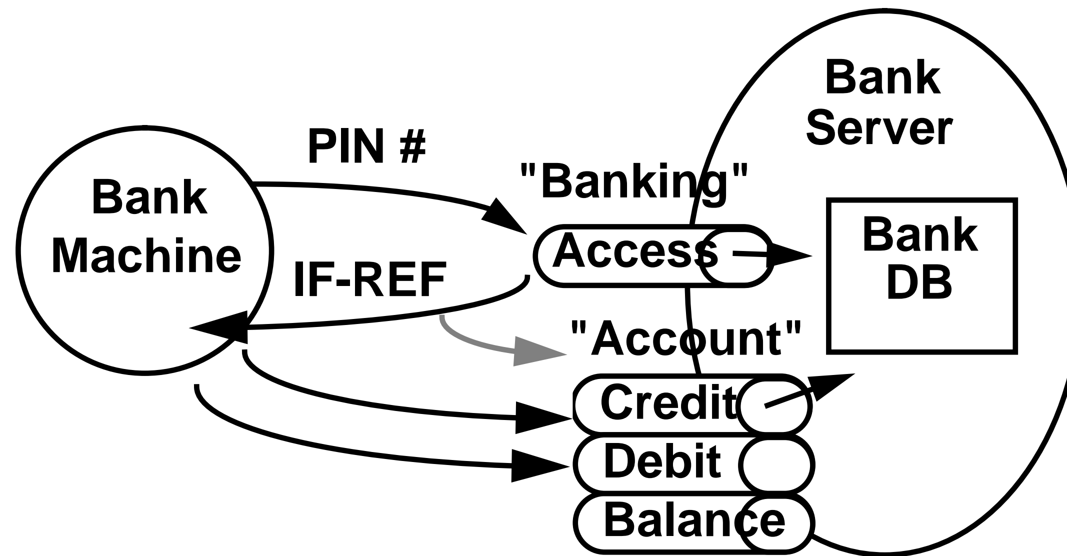
- ATM failures should be detected and any state associated with that ATM recovered or reconstructed. Handling this kind of failure not implemented by this example, but will explain how to do it.

Bank Service



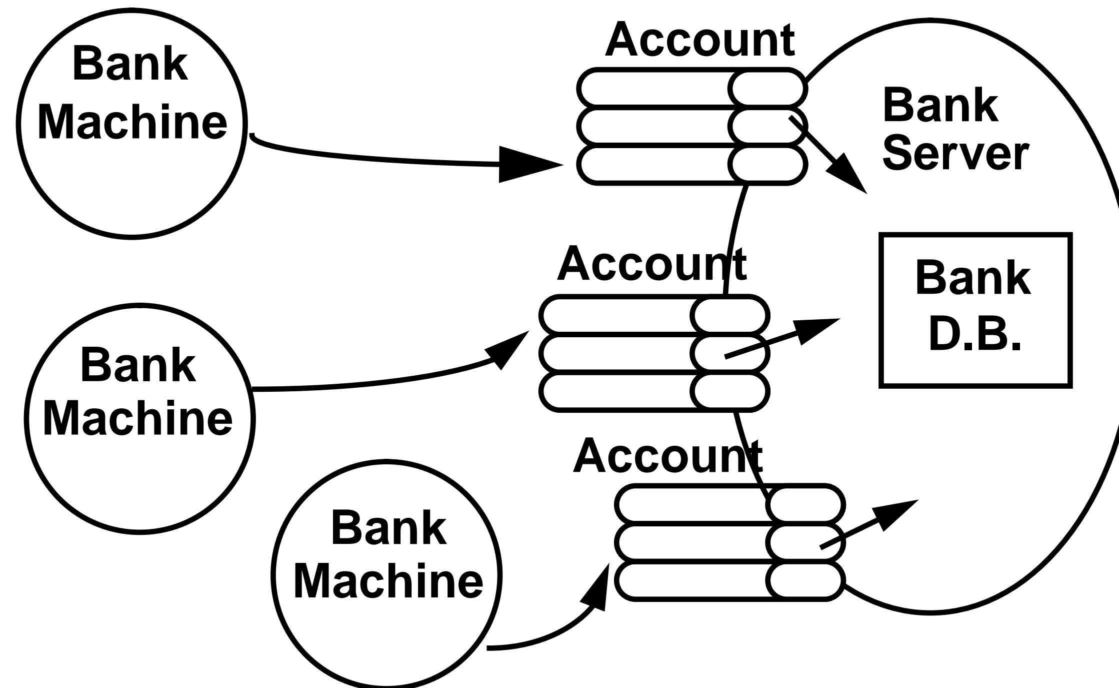
- **User should only be able to interact with account if PIN is valid**
- **Do not want Account interface to be public**
- **If "Access" succeeds, "Account" i/f is created & its *if-ref* returned**





- If "Access" succeeds, "Account" i/f is created & its *if-ref* returned
- "Account" interface provides operations on one particular account
- Only that client knows the "Account" i/f - it is not registered with the Trader

ANSA



- **Many clients can interact simultaneously with data via separate interfaces**
- **Each can have its own view of the data**

Bank Database

- Keep one array of account-information
- Keep global counter representing next account-number to be used
- Never re-issue account numbers, but re-use array elements if they are no longer in use
- Checkpoint this to a master file periodically
  - also write each transaction to an update-file to recover from crash
  - on normal termination (shutdown), write out master file, and clear update-file

SBank interface

- SBank - PIN based access to Account interfaces.
- Returns an Account interface for the account identified by an account number and corresponding PIN.
- Directly analogous to the interface offered to a human user by an ATM.

SBank i/f (cont'd)

```
SBank: INTERFACE =
NEEDS SBankTypes FROM SBTypes;
NEEDS Account;
BEGIN
AccessResult: TYPE = CHOICE OpStatus OF
{
   OpSuccess => AccountRef,
   OpFailure => OpReason
};
Access: OPERATION [
   acc: AccountNumber;
   pin: PersonalIdentificationNumber
] RETURNS [ AccessResult ];
END.
```

- typical use of choice IDL data str - for returning diff thing on success/failure

Account interface

- Account - credit/debit/list account.
- An instance of the Account interface is created (via the SBank interface's Access operation) for each account being accessed.
- Therefore there is no need to quote account numbers as arguments to the operations in the Account interface.

Account i/f (cont’d)

```
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 ];
```

Destroy used to destroy if-ref & dispose of instance-state when client done with i/f instance

```
END.
```

SBankMgmt i/f

- **SBankMgmt - management operations for creating, destroying and listing accounts.**
- **Note the use of a sequence to return a variable amount of data.**

SBMgmt.idl

```
SBankMgmt: INTERFACE =
NEEDS SBankTypes FROM SBTypes;
BEGIN
CreateRecord: TYPE = RECORD [
      acct: AccountNumber,
      pin: PersonalIdentificationNumber ];
CreateResult: TYPE = CHOICE OpStatus OF {
      OpSuccess => CreateRecord,
      OpFailure => OpReason } ;
```

```
FullRecord: TYPE = RECORD [
      acct: AccountNumber,
      pin: PersonalIdentificationNumber,
      owner: STRING,
      balance: REAL
      lastaccess: STRING ];

ListOneResult: TYPE = CHOICE OpStatus OF {
   OpSuccess => FullRecord,
   opFailure => OpReason
};

ListAllResult: TYPE = SEQUENCE OF FullRecord;
```

```
Create: OPERATION [
      owner: STRING ; balance: REAL
   ]  RETURNS [ CreateResult ];
Destroy: OPERATION[ acct: AccountNumber] RETURNS [ OpResult ];
ListOne: OPERATION [ acct: AccountNumber ]
     RETURNS [ ListOneResult ];
ListAll: OPERATION [] RETURNS [ ListAllResult ];

END.
```

Useful Data Types

- **SBankTypes - data types shared by all other interfaces.**
- **Included in all other interfaces using the `! NEEDS` statement.**

SBTypes.idl

```
SBankTypes: INTERFACE =
BEGIN
OpStatus: TYPE = {OpSuccess, OpFailure};

OpReason: TYPE = {NoSuchAccount, InvalidPin, Credited,
   Debited, InsufficientFunds, Created, Destroyed, Initiated,
   ResourcesExhausted, StaleAccountReference};

OpResult: TYPE = RECORD [
   status: OpStatus,
   reason: OpReason ];

AccountNumber: TYPE = CARDINAL;

PersonalIdentificationNumber: TYPE = CARDINAL;
```

```
AccountRecord: TYPE = RECORD [
   owner: STRING
   balance: REAL,
   lastaccess: STRING ];
END.
```

used by (result of) List operation of Account interface

Internal interface

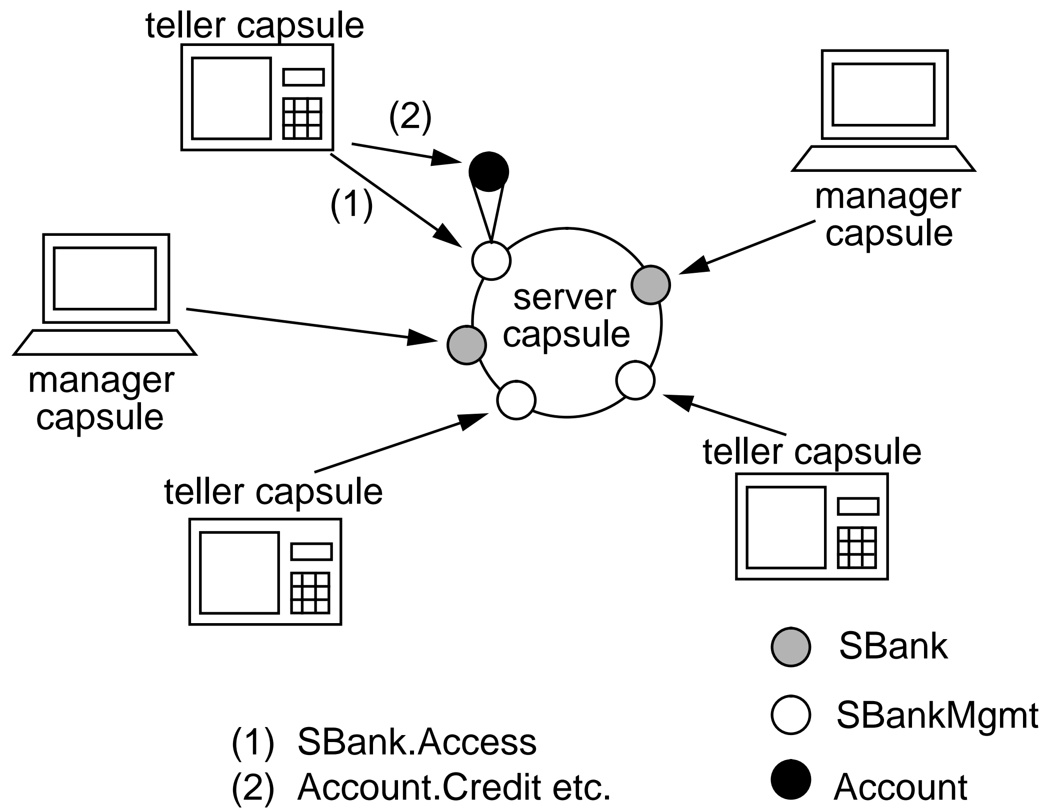
- `Internal` - an internal interface used by the server.
- Operations of this interface are Announcements
  - used to spawn background activities
- `StartTimer` starts up thread that periodically checkpoints the server's state.
- `DestroyAccount` - needed for destroying Account i/f instances
  - as mentioned earlier, a thread cannot destroy the i/f instance within which it is executing can, but clients will time out waiting for reply bcs i/f is destroyed
  - `Account` i/f's `Destroy` operation calls this (via announcement)
  - destroys the specified `Account` interface-instance once it is no longer in use not really true - waits 3 seconds, then destroys it, hoping that this is enough time for reply to be finished.

Internal i/f (cont'd)

```
Internal: INTERFACE =
BEGIN
   StartTimer: ANNOUNCEMENT OPERATION [] RETURNS [];

   DestroyAccount : ANNOUNCEMENT OPERATION [
               ref: AccountRef    ]
         RETURNS [];

END.
```

ANSA

Capsule Structure



Code Structure

- **One IDL file for each interface previously described**
- **server capsule:**

  `server.dpl` : **server initialisation code (**`body()`**)**

  `Internal.dpl` : **implementation of Internal interface**

  `SBank.dpl` : **implementation of SBank interface**

  `SBankMgmt.dpl` : **implementation of SBankMgmt interface.**

  `Account.dpl` : **implementation of Account interface operations.**

  `state.c` : **routines for initializing and modifying the server's global state.**

  `random.c` : **random number generator used for generating PINS.**

  `types.h` : **type definitions and macros for mutual exclusion and debugging.**

  `glbldefs.h` : **global variable definitions.**

  `glblrefs.h` : **global variable declarations.**

- **teller client capsule -** `teller.dpl.`
- **manager client capsule -** `manager.dpl.`
- **Can compile server with** `-DDEBUG` **to enable some debugging output**

Data Structures

```
typedef struct accrec {
        /* account number - if 0, cell can be reused */
    unsigned long accno;
    unsigned long pin;                     /* PIN for account */
    char *owner;                           /* owner's name */
    float balance;                         /* current balance */
    char accesstime[50];
                /* time of last access, in ctime( ) format */
} AccRec;

/* array of all account information */
AccRec accounts[NUMBER_OF_ACCOUNTS];
```

Data Structures (cont'd)

```
typedef struct ifrec {
    int index;                      /* index into accounts */
    unsigned long accno;            /* account number */
} IFRec;
```

allows us to associate index with acct # as state for current if-ref - guards against old ifrefs accessing account-recs that have been re-used for a new account # in the mean time

Server

- **Main functions are:**
  - **create extra tasks**
  - **initialise global event counter and sequencer for critical section management.**
  - **initialise the random number generator used for PIN's.**
  - **initialise the volatile account information from the most recent checkpoint - i.e. read in checkpoint file.**
  - **instantiate the Internal interface**
  - **start the timer thread by invoking the `Internal$StartTimer` operation**

Server (cont’d)

- instantiate the SBank and SBankMgmt interfaces.
- register these interface-instances with the Trader.
- set up the termination-handler - this function will be called when the capsule is terminated - calls checkpointing functions - more on these later
- wait for service requests.

Creating New Accounts

- SBankMgmt interface
- check through account-array for unused cell (account-number = 0)
- assign new account-number by using global variable ACNumber
  - increment this value by 1 each time acct nums never re-used, but array cells are
- fill in fields of the array element with client-provided arguments
  - customer name, account balance, etc.

Creating `Account` Interface Instances

- **`SBank$Access` (`SBank.dpl`) creates and returns Account interfaces.**

this is interface to particular account, issued to particular client, if client knows correct PIN

- **The following statement creates an Account interface.**

```
! {acct_ifref} :: Account$Create(1, acct_index, acct)
```

- **`1` is concurrency - only want 1 invocation serviced at once for mutual exclusion reasons** is not enough in itself, though - as manager interface fns may also be accessing?
- **`acct_index` is the index into the array of account records for the specified account.**
- **`acct` is the account number for the specified account.**

- **The following function creates the state required to associate an Account interface instance with an `IFRec` (account-number and array-index)**

```
ansa_StatePtr Account_Create( index, accno )
int index, accno;
{
IFRec *p;
char *malloc();
   p = (IFRec *)malloc(sizeof(IFRec));
   p->index = index;
   p->accno = accno;
   return (ansa_StatePtr)p;
}
```

- **the account-number and index is this interface’s instance-specific state**

Building SimpleBank

1. `ansamkmf`
   to create a Makefile from the Imakefile.

2. `make depend`
   to create the required dependencies.

3. `make`
   to create the server, teller and manager programs.

Using SimpleBank

```
% ./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 SimpleBank (cont'd)

```
% 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

% teller 1 6263 debit 400.00
Debit(400.00) failed, reason: InsufficientFunds
```

Using SimpleBank

```
% teller 1 6264 list
Access(1, 6264) failed, reason: InvalidPin
% teller 3 6263 list
Access(3, 6263) failed, reason: NoSuchAccount
% manager destroy 2
% manager listall
1     6263    123.45 Thu Mar 14 16:14:55 1992 Jane Dunlop

% sbshut.sh
```
uses trclient terminate to kill service with approp checkpointing

```
% ./server &
% manager listall
1     6263    123.45 Thu Mar 14 16:14:55 1992 Jane Dunlop
```

Exercises

- **Customise the bank server**
  - **Export with distinctive property, e.g**
    `"BankName 'MyBank PLC'"`
- **Implement the Account interface**
  - **i.e. its operations**
- **Change the teller and manager client programs to Import your bank, rather than other peoples'**
  - **Add constraints to Import statement, e.g**
    `"BankName == 'MyBank PLC' "`

Checkpointing

- Checkpointing functions are in the file *state.c*
- two checkpoint files: masterfile & updatefile are set to:

```
<your-path>/SBMaster
<your-path>/SBUpdate
```

- `SBMaster` must exist for the server to work, even if it is just empty

Implementing Account Interface

- Partially complete `Account.dpl`
- Has been generated using `stubc -t`
  - generates a template containing function skeletons.
- `List` operation definition is already written
- Need to implement `Credit`, `Debit` operations

Implementing Account

- **The implementation of each operation in the Account interface has the same generic structure:**

1. **grab mutex**
2. **locate interface specific state :** `thread_getInterfaceState()`
3. **validate request**
4. **invoke appropriate account operation from state.c**
   e.g. `credit_account(index, amount)`
5. **update access time (local function)**
6. **free mutex**
7. **return** `SuccessfulInvocation` currently set up to return unsuccessfulInv, bcs not written

Now go off & try examples

Handling Client Failure

- **This SimpleBank example doesn't handle failure of teller-machines**
  - **if an individual client fails, do not want to waste server resources storing state-information associated with failed client**
- **Could be done through "dead-man's handle"**
  - **client passes server interface-reference to be called on**
  - **server could call client back on this interface-insatnce periodically**
  - **if invocation times out, then client can be considered dead, and state can be released**

Exception Handling
in
ANSAware

Handling Exceptions

- **Many programming languages ignore the possibility of errors from the run-time support-system - these simply result in "system error"**
- **PREPC - invocation syntax used so far:**

  `! { results } <- ifref$OpName ( args )` status value results from any invoc'n

- **PREPC - Exception Syntax:** (have prob seen transmitTimeout, bindfailure); - user can specify what to do

  `! { results } <- ifref$OpName ( args ) exception-list`

  where `exception-list` **has the following syntax:**

  `Continue statuslist Abort statuslist Signal statuslist`

- **e.g.**

  `!{ results } <- ifref$OpName ( args ) Continue ok Abort *`

- **exception-list is optional - , PREPC default behaviour will be used****(page 216)**

PREPC Exception Syntax

- Can use * to indicate "any other status values" e.g.

```
! { res } <- ifref$OpName( args ) \
                                   Continue ok \
                                   Signal   bindFailure \
                                   Abort    *
```

- `Continue`: program goes on to next statement
- `Abort`: program aborts with appropriate error msg
- `Signal`: program calls an exception-handling function, passing it status, invocation-args, and invocation-results
    - based on these, the exception handler can take appropriate action

Signal function could print out message like: "Failed to Import Bank Service" in graphical teller, if fails to import bank service, displays msg in window “Bank temporarily out of service”

Exception Handling

- **Signal function-name has specific construction:**
  If operation-call is as follows:

```
   ! DECLARE {ifref} : IfName CLIENT
...
   ! { results } <- ifref$OpName( args ) Continue ok Signal *
```

Then signal-function must be:

```
Signal_OpName_IfName( status, arg1,...argN, res1,...resN )
ansa_Status status;
argType1   arg1;
...
resultType1  result1;
{ ... }
```

all args & results passed - same types as defined in IDL file of if-definition

{ ... } whatever signal-handling stuff you want in here

Signal Function (cont'd)

- **Signal function can return `ExceptionAbort`, `ExceptionContinue`, or `ExceptionRetry` status**
- **Code generated by PREPC will check this return-value, to see how to proceed on returning from the Signal function** if Continue, then next point important
- **Code that follows any operation invocation may need to know whether exception has occurred or not - macro-definitions provided for this purpose:**
  - `thread_setExceptionCode(ansa_Cardinal code)`
  - `ansa_Cardinal thread_getExceptionCode()`
- **these allow exception handler to communicate with the calling thread**
  - `thread_setExceptionCode` **sets a value, stored within the current thread's data record**
  - **this can be examined when the operation which generated the exception has completed** guaranteed that signal-handler will be exec'd on same thread as op that gens excepn

Signal Function - Example

```
/* signal handler */
Signal_Foo_Op( ... )
{
   thread_setExceptionCode( 1 );
   return ExceptionContinue;
}
body()
{
!  {} <- foo$Op() Continue ok Signal *
   if( thread_ExceptionCode() == 1 )
      /* exception has been raised */

}
```

PREPC invocation-stmts automatically set exceptioncode to 0 before op is actually invoked, so this works

Signal Function - Example 2

```
...
! {access_result} <- bank_ifref$Access(account_number, pin) \
                        Continue ok Signal *

   /* Check that no exception occurred on Access operation. */
   if ( thread_getExceptionCode() == 1 )
```

tells if exception (unexpected status) occurred

```
   {
      bank_state == BANK_OUT_OF_SERVICE
      return( OpFailure );
   }
```

if sthg went wrong (not success/failure, but transmit timeout, etc), then assume sthg is wrong - might then start again, try to import ifref over again, etc - keep trying while out-of-service, etc

```
   /* check whether access operation succeeded... etc */
```

Signal Function (cont'd)

- For communicating more complicated information between signal-handler and calling thread:
  - `thread_setExceptionState(ansa_StatePtr state)`
  - `ansa_StatePtr thread_getExceptionState()`
- Can pass a pointer to any sort of data structure, but must coerce to be an `ansa_StatePtr`
- `thread_setExceptionCode`, `threadSetExceptionState` both automatically called with a value of 0, ((`ansa_StatePtr`) 0) by PREPC-generated code before actually making any invocations that could generate an exception

Exceptions - Default behaviour

- **The default behaviour if no exception-list is given is roughly equivalent to:**

```
! { results } <- ifref$op ( arguments ) \
    Continue ok \
    Signal transmitTimeout invalidNonce illegalOperation \
          illegalInterface abnormalReturn \
    Abort *
```

but, rather than a user-supplied signal-handler, ANSAware default signal-handler will be called if any of "Signal" status-values occur

- `Signal_binder_relocate()` **- provided by infrastructure**

Default signal-handler

- `Signal_binder_relocate()` tries to relocate the interface
  - does this by contacting any locator services it knows about
- interfaces registered with Trader will have information about Trader's `Relocator` interface, so this will get called
  - this will check whether it knows of a new location for given interface
- if relocation succeeds, interface-ref will be updated to new one, and `ExceptionRetry` returned
  - otherwise, `ExceptionAbort`:

```
Abort: Prepc.Relocate: 1282 (transmitTimeout)
```

- No built-in support in ANSAware yet for migrating a service
- Full list of status-values given in section 3.7 of Application Programmer's Manual

Exception Handling (cont'd)

- signal-handlers for PREPC pseudo-operations such as $Import and $Export are called Signal_Prepc_Import and Signal_Prepc_Export, respectively

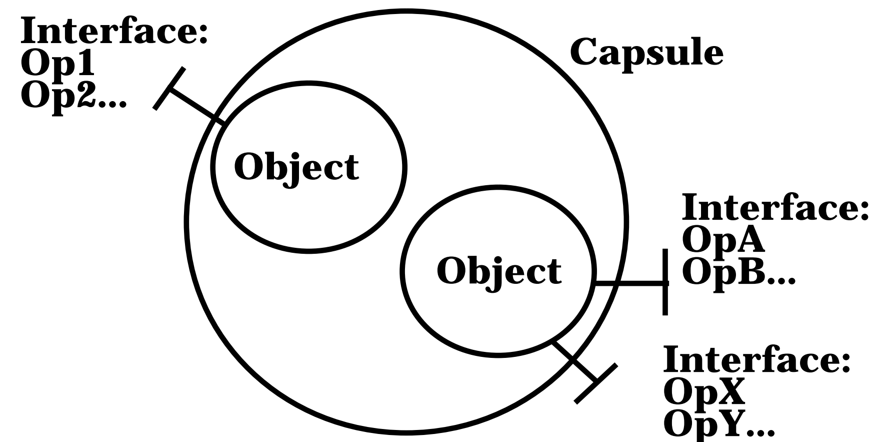
- If user provides own signal-handler, default relocation will not automatically take place - if user wants this, will have to call Signal_binder_relocate explicitly

Dynamic
Service Creation

- will show how to dynamically instantiate services from other programs - exercise at end, so lots of details included for referring to when doing exercise

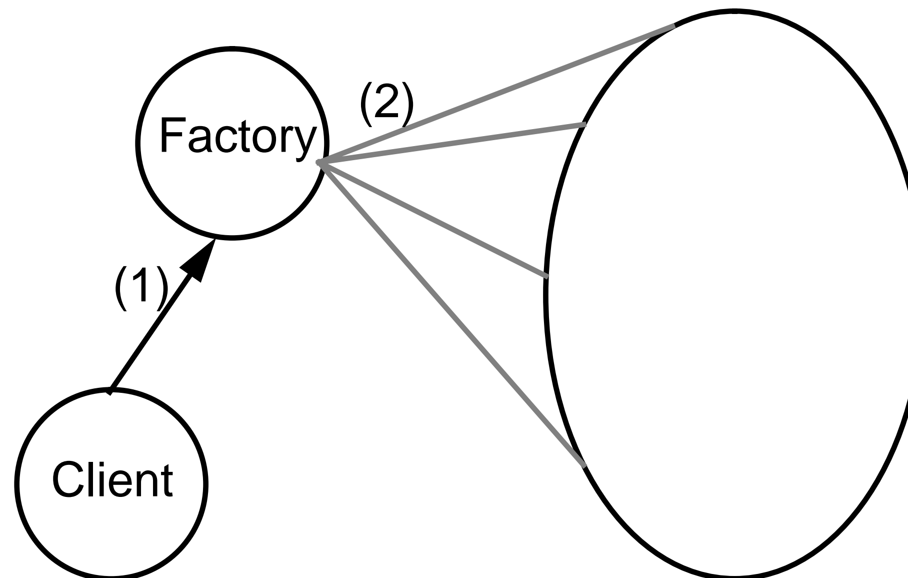
Factories

- Factories provide a service for :
  - Creation/destruction of capsules
  - Simple monitoring of capsules it creates
- Once created capsules provide a service for :
  - Creation/destruction of objects in a capsule.
  - Creation/destruction of interfaces in an object

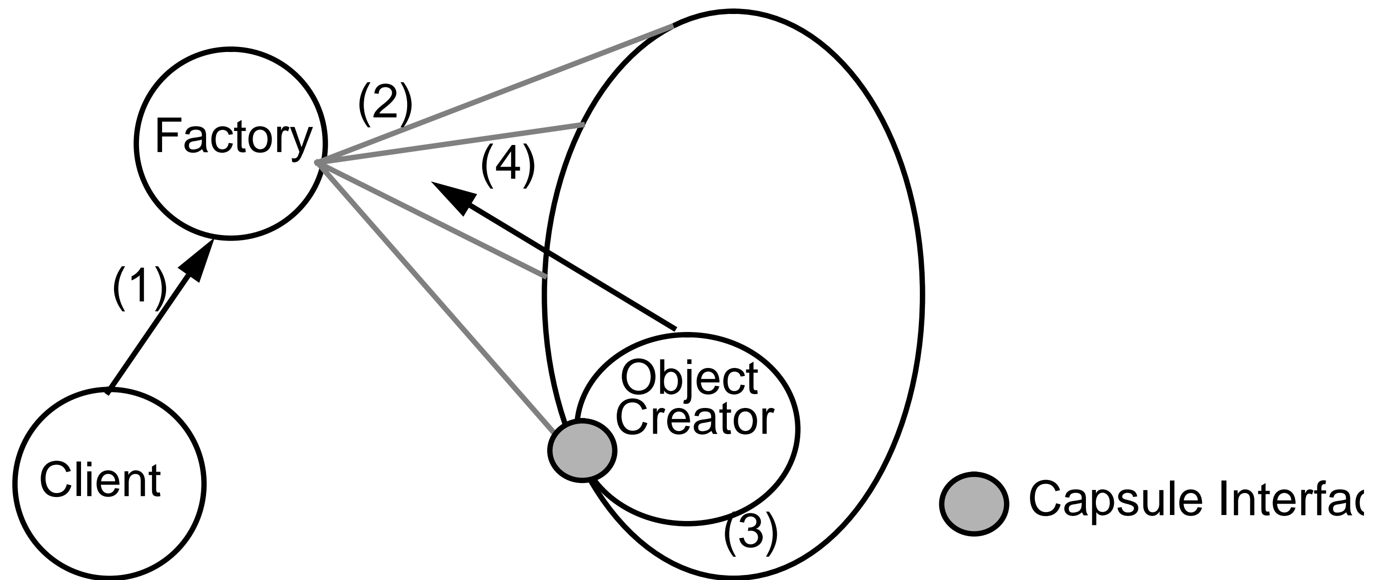


ANSA

Factory Interactions



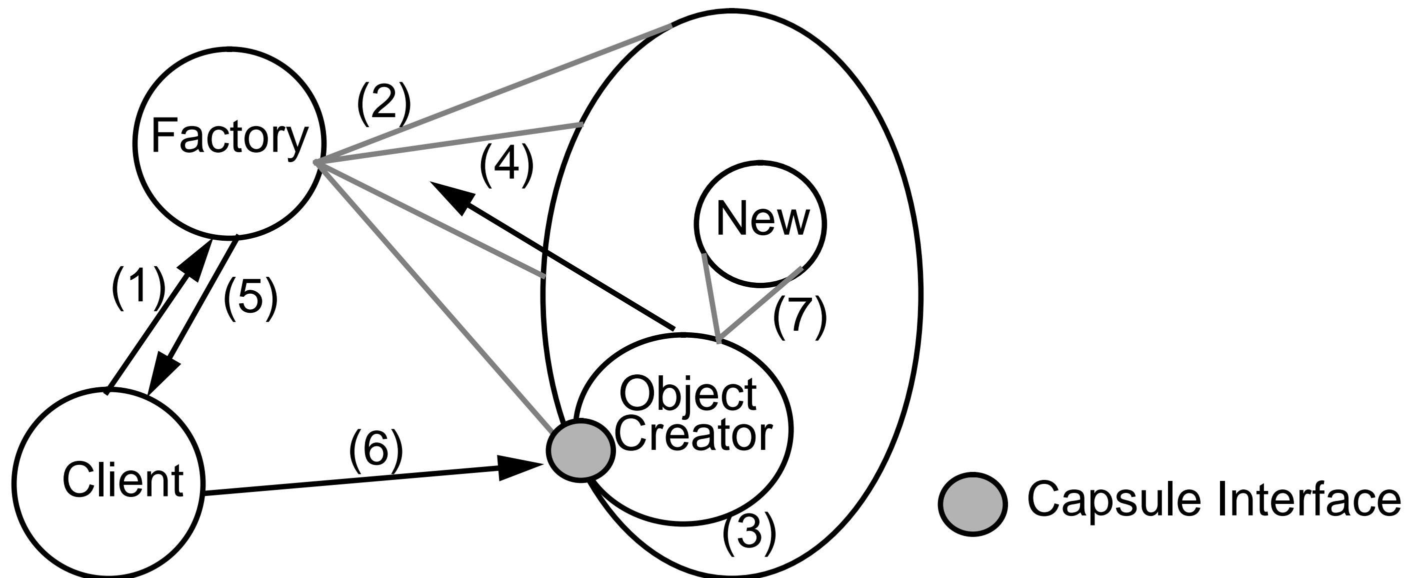
1. **Client asks factory to Instantiate a capsule.**
2. **Factory instantiates a new capsule.**



3. **New capsule creates a single object with at least a single Capsule interface. Any number of other interfaces may also be created.**

4. **References to the Capsule and any other interfaces are returned to the**

factory.



5. **These are in turn returned to the client.**
6. **The client may then use the Capsule interface reference to create yet more objects.**
7. **A new object is created.**

Factory Interface

```
Factory : INTERFACE =

NEEDS Capsule;
NEEDS Terminated FROM Term;

BEGIN
    Instantiate: OPERATION [
        Path: STRING; if no path given, default path built in to factory will be used (set at build-time)
        Template: STRING; capsule executable
        Arguments: STRING;rest of these args are all optional
        Environment: STRING;
        Terminated: TerminatedRef;interface-ref to be invoked when this capsule dies
        Data: CallerData  data to be passed - factory monitors continued existence of capsules
    ] RETURNS [            it has created & can notify creator of demise via this mechanism -more later
        Ref  : CapsuleRef;ifref of Capsule i/f of created capsule - more shortly
        Cid  : ansa_CapsuleId ]; process id of capsule
```

```
ReRegister: OPERATION [
      Cid: ansa_CapsuleId;
      Terminated: TerminatedRef;
      Data: CallerData
   ] RETURNS [ BOOLEAN ];
```

this op not important at the moment

```
IsAlive: OPERATION [ Cid: ansa_CapsuleId ]
   RETURNS [ BOOLEAN ];
```

ditto this one

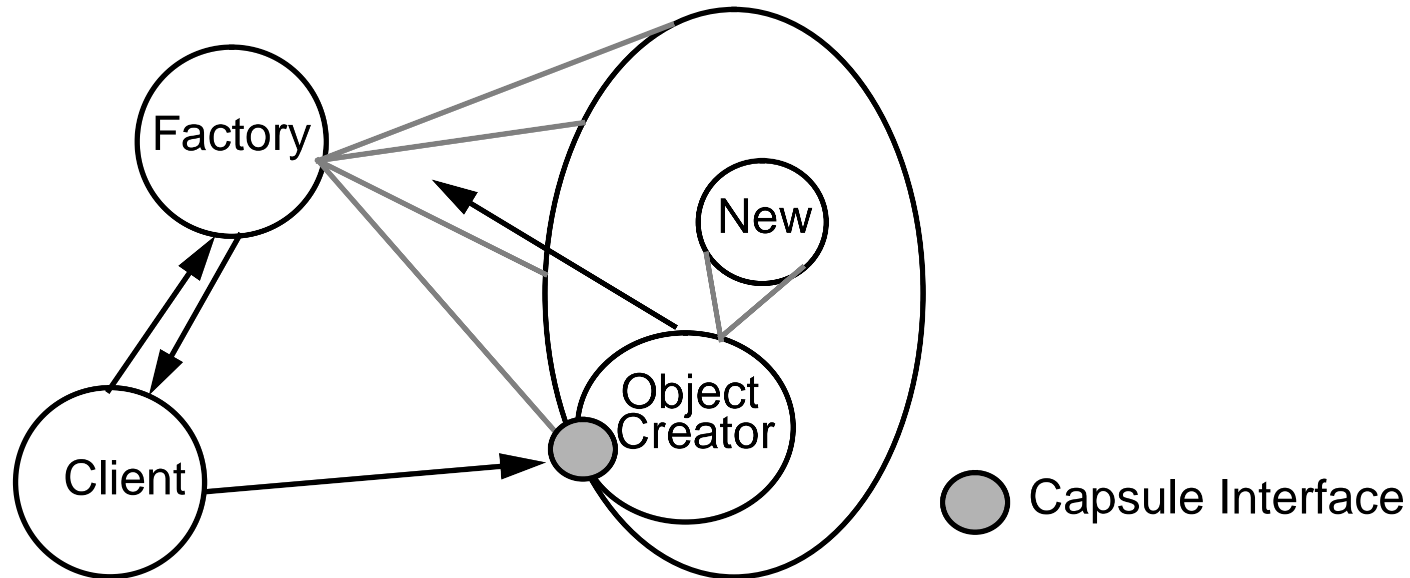
```
Terminate: OPERATION [ Cid :ansa_CapsuleId ]
   RETURNS [ BOOLEAN ];
```

used to kill capsules created by Instantiate above

```
END.
```

ANSA

Capsule Interface



- **Every capsule contains a Capsule interface, provided by the ANSAware infrastructure**
- **Used by factory to instantiate objects within that capsule**

```
Capsule : INTERFACE =
NEEDS Notify;
NEEDS Object;
BEGIN
   InstantiateResult : TYPE = SEQUENCE OF ansa_InterfaceRef;

   Instantiate : OPERATION [
            MyRef        : NotifyRef; these 2 args both just historical
            Capsule      : CapsuleRef;historical
            Template     : STRING; object-template
            Arguments    : STRING;optional
            Environment  : STRING  optional
          ] RETURNS [  Object : ObjectRef;inst of Object i/f - will describe this shortly
                       Interfaces : InstantiateResult ];
   Terminate : OPERATION [] RETURNS [ BOOLEAN ];
END.
```

Using the Factory - simple client program

```
! USE Factory
! USE Capsule
! USE Object
! USE Foo
! DECLARE { factRef } : Factory CLIENT
! DECLARE { capRef } : Capsule CLIENT
! DECLARE { objRef } : Capsule CLIENT
! DECLARE { res.data[0] } : Echo CLIENT
void body( int argc, char *argv[], char *envp[] )
{
ansa_InterfaceRef factRef, capRef, objRef;
InstantiateResult res; this data structure is returned by Capsule$Instantiate (prev slide)
ansa_CapsuleId cid;
ansa_Boolean r;
```

```
stub_setFreeCltMem( ansa_FALSE );
```
- needed to make sure results not freed by next call
default freeing behaviour needs overriding when later invocns use earlier results

```
! {factRef} <- traderRef$Import( "Factory", "/", "")
! {capRef, cid} <- factRef$Instantiate("", "fooserver",\
                                        "", "", nullRef, 0)
! {objRef, res} <- capRef$Instantiate( nullRef, nullRef,\
                                        "Foo", "", "")
...
/* Can now call operations of Foo interface */
!  { fooOpRes } <- res.data[0]$FooOp( fooOpargs )
...
```
note: res.data[0] contains if-ref - only 1 in this case, but could E seq of ifrefs - just made-up results, ops & args

```
/* When done, should terminate created capsules/objects,
   otherwise service will remain active */
!  {} <- objRef$Terminate()
!  {r} <- factRef$Terminate( cid )
```
returns boolean success/failure

```
}
```

Dynamically Creatable Service

- In order for a server to be able to be started by the factory, it needs to provide certain functions.
- Such servers do not require a `body()` function,
  - but if a `body()`function is provided, the service can be started from the command-line, or from the factory.
- This line in the client:

```
! {objRef, res} <- capRef$Instantiate( nullRef, nullRef,\
                                        "Foo", "", "")
```

invokes Object creation function for given object-template ("Foo")

Capsule structure for Object Creation

- A `! MANAGED` PREPC statement is required, specifying name of object templates supported, as well as Create and Destroy functions for each such object.

  `! MANAGED ObjName` ObjName is just some name you choose to give template

- The Capsule interface supported by all capsules invokes `Create_`*`ObjName`*`_Object` when its `Instantiate` operation is invoked with an object name of *`ObjName`*.

- `Destroy_`*`ObjName`*`_Object` is invoked when the object is to be destroyed.

Object Creation (cont'd)

- **Create_... & Destroy... function signatures:**

```
ansa_StatePtr Create_ObjName_Object( argc, argv, envp,
                                     results )
   int argc;
   char *argv[], *envp[];
   InstantiateResult *results;

ansa_Boolean Destroy_ObjName_Object ( state )
   ansa-StatePtr state;
```

Creating Objects (cont'd)

- Create_*ObjName*_Object:
  - *ObjName* is just any name you choose for referring to this template
  - must be same as *ObjName* used in `!MANAGED` statement

- Steps performed by Create_*ObjName*_Object:
  - 1. use arguments and environment (if any) to decide precisely what action is required.
  - 2. instantiate any interfaces required.
  - 3. allocate and initialise any object state required (more on this later).
  - 4. return any state, set up results (of type `InstantiateResult`) - a sequence of interface references to instantiated interfaces

    InstantiateResult type defined in Capsule i/f

Creating Objects (cont'd)

- **Capsules can support multiple objects (i.e. multiple templates for creating interfaces) - the *ObjName* specifies the object to be used.**
- **Capsule$Instantiate operation:**

```
! {objRef, res} <- capRef$Instantiate( nullRef, nullRef, \
                                        "Foo", "", "")
```

automatically creates an instance of the Object interface, and returns it as the first result :

```
Object : INTERFACE =
BEGIN
    Terminate : OPERATION [] RETURNS [];
END.
```

- **Object interface has only this one operation**

Creating and Destroying Objects

- **When this Terminate operation is invoked, e.g.**
  `!  {} <- objRef$Terminate()` as shown in example client, earlier (page 229)
  user-provided Destroy_*ObjName*_Object function is called

- **Destroy_*ObjName*_Object should destroy any interfaces and free any object state set up by Create_*ObjName*_Object (more on object-state shortly).**

Simple factory-startable service

fooserver.dpl:

```
#include “ansa.h”     generally useful header file
#include “tFoo.h”     header file generated from interface specification by stubc

! MANAGED FooObj      Note this word must be same as in Create...() fn below -also called template name

! USE Foo
! DECLARE { ifref } : Foo SERVER

   ansa_StatePtr Create_FooObj_Object( ac, av, envp, results )
   int ac;
   char *av[], *envp[];
   InstantiateResult *results;
   {
      ansa_InterfaceRef ifref;
```

$Create stmt: create i/f instance & put resulting if-ref into result to be returned

```
!      {ifref} :: Foo$Create(1)
```
will become 2nd result of Capsule$Instantiate op (1st is ObjRef)

```
       results->length = 1;  /* Assign results - seq. of ifrefs */
       results->data = &(ifref);
       return (ansa_StatePtr)NULL; /* No obj-state in this ex. */
    }
ansa_Boolean Destroy_Echo_Object(state)
     ansa_StatePtr state;
{
     return ansa_TRUE;
}
```

should destroy any created if-instances, but to be v. simple, we won't

- can't anyway in this case, cos we haven't got any reference to it (was local above)
- will show in next example how to keep track of all this via object state

```
...
/* Functions implementing Operations of "Foo" interface,
   body() function */
...
```

Object State

- **Objects have optional state**
  - **can be initialised by Create_*ObjName*_Object function**
  - **easily accessible from Destroy_*ObjName*_Object function**
  - **usually based on arguments and/or environment**
- **Object state can be anything - user-defined:**

```
typedef struct objstate { .../* any structure definition here */
                          } yourStateStruct;

Create_ObjName_Object( ...)
{
   p = system_allocate( sizeof( yourStateStruct) )
   ...
   /* ...store whatever info is necessary into this structure...*/
   ...
   return ( ansa_StatePtr )p;
}
```

Object State (cont'd)

- Could be used in earlier example for making interface-reference available to Destroy_... function
  - store interface-ref of created interface-instance in object-state
  - Destroy_...() function can then access if-ref and destroy it
- Commonly used for indicating whether interface-instance has been exported to Trader
  - If so, Destroy_...() function can Withdraw offer from Trader before destroying interface
- With factory-created services, ANSAware services often use environment variable "ANSA_MANAGED_EXPORT" to indicate whether interface should be exported to Trader
- Next example illustrates use of object-state as suggested above

Factory client program - 2

```
! USE Factory
! USE Capsule
! USE Object
! USE Foo
! DECLARE { factRef } : Factory CLIENT
! DECLARE { capRef } : Capsule CLIENT
! DECLARE { objRef } : Capsule CLIENT
! DECLARE { res.data[0] } : Echo CLIENT
void body( int argc, char *argv[], char *envp[] )
{
ansa_InterfaceRef factRef, capRef, objRef;
InstantiateResult res;
ansa_CapsuleId cid;
ansa_Boolean r;
```

```
stub_setFreeCltMem( ansa_FALSE );
```
- needed to make sure results not freed by next call default freeing behaviour needs overriding when later invocns use earlier results

```
! {factRef} <- traderRef$Import( "Factory", "/", "")
! {capRef, cid} <- factRef$Instantiate("", "fooserver",\
                                        "", "", nullRef, 0)
! {objRef, res} <- capRef$Instantiate( nullRef, nullRef,\
                    "Foo", "", "ANSA_MANAGED_EXPORT=yes")
/* Can now call operations of Foo interface */
!  { fooOpRes } <- res.data[0]$FooOp( fooOpargs )
...
/* When done, should terminate created capsules/objects */
!  {} <- objRef$Terminate()
!  {r} <- factRef$Terminate( cid )
}
```
Note: ANSA_MANAGED_EXPORT is only difference btwn this & prev example - could pass envp, or extract that var from envp and pass it, if you want to just use local env't.

Factory-startable server program - 2

```
#include "ansa.h"
#include "tFoo.h"

#define PROPSIZE 1024
char pbuf[PROPSIZE];

! MANAGED Foo

! USE Foo
! USE Trader
! DECLARE { ir, p->ref } : Foo SERVER

void body( argc, argv, envp )
int argc;
char *argv[], *envp[];
```

```
{
void body( argc, argv, envp )
{ ... definition of body ... }

...Foo Operation definitions ...

typedef struct objstate {
    ansa_InterfaceRef ref;
    ansa_Boolean export;
} ObjState;
```

user's own object-state storing type - inthis case we want to store an ifref & a boolean

```
ansa_StatePtr Create_String_Object(ac, av, envp, results)
int ac;
char *av[], *envp[];
InstantiateResult *results;
{
ObjState *p;
```

these params corresp to the last 2 args of the cap$Inst call

need pointer to our state-type

```
  p = (ObjState *)malloc(sizeof(ObjState));allocate mem for state-type

! {p->ref} :: Foo$Create(1) note that this is assigned into obj-state str - will be stored away

  if( system_getenv("ANSA_MANAGED_EXPORT", envp) != (char *)0)
  {                  based on this env var passed in as arg, will decide whether to export or not
     p->export = ansa_TRUE; will store away whether or not this is an exported offer
     (void)system_init_properties( pbuf, PROPSIZE, ac,av );
!    {}<- traderRef$Export("Foo", "/ansa/testservices", \
                                 pbuf, p->ref) give normal properties, p->ref is ifref
  }
  else
     p->export = ansa_FALSE; do nothing, offer is not to be exported to trader
  results->length = 1;
  results->data = &(p->ref);
  return (ansa_StatePtr)p;
}
```

```
ansa_Boolean Destroy_Echo_Object(state)
   ansa_StatePtr state;
{
ObjState *p;

   p = (ObjState *)state;

   if (p->export == ansa_TRUE)
!     traderRef$Withdraw( p->ref )

!  {} :: Foo$Destroy( p->ref )

   free ((char *)p);
   return ansa_TRUE;
}
```

gets called when objRef$Terminate called

state is automatically provided by infrastr as param to this fn.

check if was exported, & if so, withdraw -ensures stale offer doesn't remain in trader

-destroy i/f inst - couldn't do this in prev ex cos had no ref to it now ref is in obj-state (object instance), so can refer to it

-free the state-pointer - no longer needed, as this obj-instance will go away when this function ends (auto destroyed as part of call)

Note that this is object-instance state, so if there were multiple instances through multiple calls to cap$Inst, then each obj-inst woul dhave own state which it would point to

Terminated Interface

- **Factory.Instantiate operation:**

```
Instantiate: OPERATION [
        Path: STRING;
        Template: STRING;
        Arguments: STRING;
        Environment: STRING;
        Terminated: TerminatedRef;
        Data: CallerData
    ] RETURNS [
        Ref : CapsuleRef;
        Cid : ansa_CapsuleId ];
```

Terminated: interface-ref to be invoked when this capsule dies

Data: data to be passed - factory monitors continued existence of capsules it has created & can notify creator of demise via this mechanism -more later

- **A client can use the TerminatedRef arg to provide an interface-instance which is to be invoked if/when the capsule in question terminates**

Terminated Interface (cont'd)

```
Terminated : INTERFACE =
BEGIN

   CallerData: TYPE = CARDINAL;

   CapsuleTerminated: OPERATION [
         Cid: ansa_CapsuleId;
         Data: CallerData
         ] RETURNS [];

END.
```

- **Next example shows how this can be used in a client program, similar to earlier examples**

assume everything else same as in previous client examples, but this new stuff just added

```
GLOBAL ansa_Cardinal Ansa_InitialTasks = 2;
```
need 2 tasks, bcs of eventcount/seqs

```
...
! USE Terminated FROM Term
```
uses Terminated interface-definition, need to specify this

```
! DECLARE { notRef } : Terminated SERVER
ansa_EventCount ec;
```
this example uses event counts & sequencers - need to set them up

```
ansa_Sequencer sq;

void body( argc, argv, envp )
int argc;
char *argv[], *envp[];
{
ansa_InterfaceRef notRef;
```
notRef will be ifref for i/f of type Terminated, created by this capsule

```
...
```
initialize event count & seq to be used later.

```
    ec = ecs_makeEventCount((ansa_Cardinal)0);
    sq = ecs_makeSequencer((ansa_Cardinal)1);
```

```
!   {notRef} :: Terminated$Create(1)
```
create instance of Terminated i/f

```
...
!  {capRef, cid} <- factRef$Instantiate( "", "Foo", "", \
                                          "", notRef, 0 )
```
pass notRef (ref to inst of Term i/f that we have just created) to factory when creating new capsule

```
...
   /* do all object-creation, etc. etc. */
...
   /* When finished, terminate object & capsule, as before */
!  {} <- objRef$Terminate()
!  {r} <- facRef$Terminate(cid)
   /* Wait for capsule to really be dead */
   ecs_await( ec, ecs_ticket( sq ) );

!  {} :: Terminated$Destroy(notRef)
```
not good to destroy i/f inst while possibly still needed, as another capsule may still expect to use it (factory), but must destroy before ending, o/w cap won't end

```
}
```

```
int Terminated_CapsuleTerminated(_attr,cid,handle)
ansa_InterfaceRef *_attr;
ansa_CapsuleId cid;
CallerData handle;
{
   printf("%s: CapsuleTerminated( %lu, %lu ) invoked\n",
            nucleus_name,cid,handle);
   ecs_advance( ec );
   return successfulInvocation;
}
```

this function doesn't actually do anything except print out a message, but you can imagine that a capsule that creates another capsule might want to keep track of whether it still exists or not

Note that the CallerData originally passed in the fact$Inst op is given here as the 3rd param - not used in this case, but
- capsule can use this to use same Terminated i/f inst for callbacks from a whole bunch of instantiated capsules, and distinguish btwn them by this parameter

Factory client programs

- **frun node template object arguments environment [client args ...]**

arguments, environment are passed to the object constructor/template

- **e.g.:**

  `frun "" myserver ObjName "" ""` will use current node
  `15307`

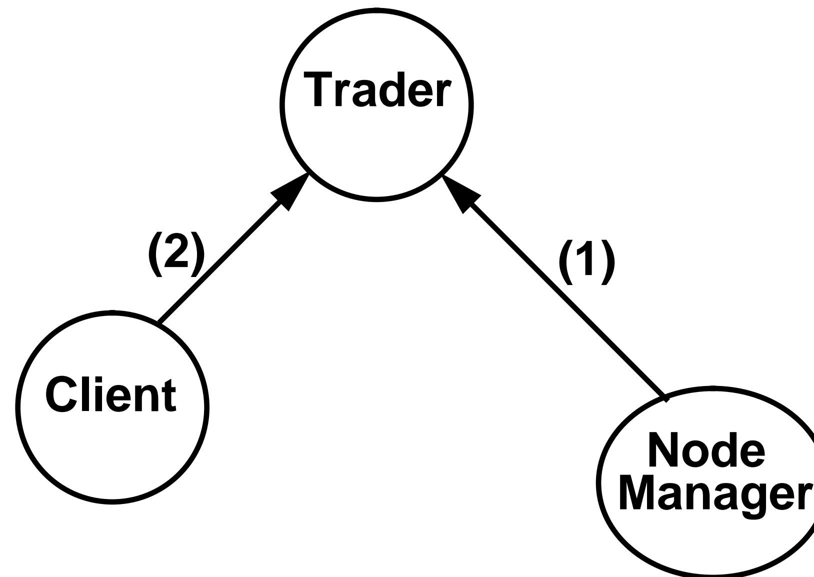
- **frun sets `ANSA_MANAGED_EXPORT` before instantiating capsule to make sure any interface-instances will be exported**
- **if `client` arg is given, it will be started as a sub-process, and `args` will be passed to it - when `client` terminates, `frun` will terminate the object, then terminate the capsule**

- **fkill host cid**
  - **e.g.:**
    `fkill "" 15307` means run on current node (no host param means use default)

Node Manager

- Provides an architectural interface for the creation, simple monitoring and destruction of services.
- Provides a database for describing services. Each description is identified by an alias.
- Aliases may be run to create static services, which may be automatically restarted if they terminate.
- Dynamic service creation is provided by the federated trader interface's proxy export facility.
- Service activations may be destroyed.
- Node Manager state is persistent. (checkpointed)

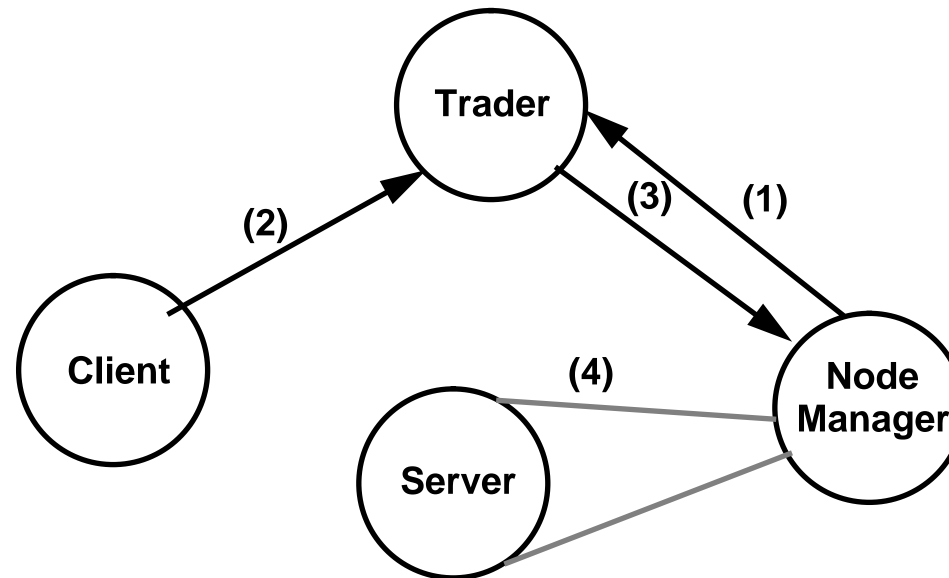
Dynamic Service Creation



1. Node manager registers a proxy offer with the trader.

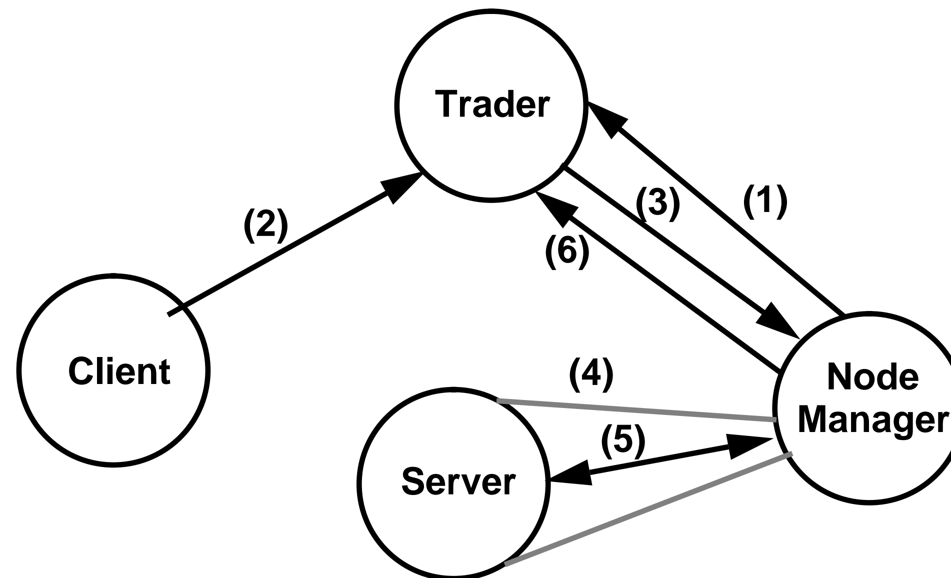
2. Client performs an import.

ANSA



3. **Trader, recognising that the offer is federated, forwards the import to the node manager.**
4. **Node manager creates the server capsule (using the factory).**

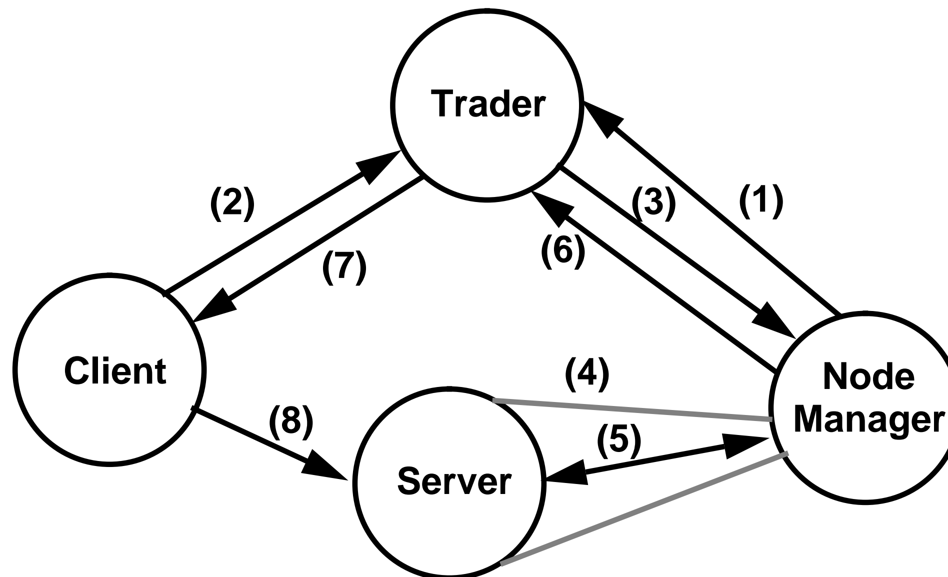
ANSA



5. Node manager invokes the Instantiate operation on the newly created capsule's Capsule interface.

6. Node manager returns the InterfaceRef (result of Instantiate operation) to

the trader.



7. Trader returns this InterfaceRef to the original client.
8. Client can now invoke operations on the server.

nmclient

- `nmclient` **provides a way of using the node manager (NM)**

```
nmclient postproxy Echo
```

posts offer to trader for specified service, but does not start up - when client import svc, trader detects offer is proxy, fwds to NM & NM starts up svc

- **In order to post a proxy offer for a service, the NM must know about that offer**
- `nmclient install`

```
nmclient install alias max_activations
                 interface-type context properties
                 capsule object arguments environment
```

nmclient (cont'd)

- **e.g.**

```
nmclient install "MyEcho" 1 "Echo" "/ansa/testservices" \
                 "" "myserver" "Echo" "" ""
```

- **To list all of Node Manager's aliases:**

```
nmclient listall
```

- **To list single alias:**

```
nmclient showalias AliasName
```

nmclient (cont'd)

- **By default, `nmclient` uses the NM on your same node, but can specify properties:**

```
nmclient -p "Node == 'crippen'" listall
```

any constraint can be specified - does not need to be node name

NM will have this prop by default though, & generally you do want to give node

- **Can run NM aliases**

```
nmclient run alias arguments environment
```

  - **an activation of that alias is started - activation id for the new activation is printed out**
  - **activation can later be killed:**

```
nmclient kill alias id
```
many activations can exist for partic alias - must distinguish

```
nmclient [-p properties] install alias max_activations interface
          context properties capsule object arguments environment
   [-p properties] remove alias
   [-p properties] mask alias
   [-p properties] postproxy alias
   [-p properties] deleteproxy alias
   [-p properties] run alias args env
   [-p properties] runforever alias args env
   [-p properties] kill alias id
   [-p properties] showalias alias
   [-p properties] showactive alias
   [-p properties] showid alias
   [-p properties] listall
   [-p properties] listactive
   [-p properties] allaliases
   [-p properties] allactive
```

Node Manager (cont’d)

- **Proxy offers posted via:**

  `nmclient postproxy Echo`

  can be withdrawn by:

  `nmclient deleteproxy alias`

- **Node Manager proxy offers cannot be removed from the trader via** `trclient delete`
  - **if necessary, can use** `trclient proxydelete`
  - `trclient proxydelete` **can not delete non-proxy offers.**
  - **as with** `trclient delete`**, this should not normally need to be used, as the Node Manager should correctly manage the state of posted proxy offers**

Hands-on Exercise

- **Make Echo a managed service. You will need to add the following**
  - `!MANAGED Echo`
  - `Create_Echo_Object()`
  - `Destroy_Echo_Object()`
- **Test out service by first using frun to make sure it works**
- **Use nmclient to install an Alias for your service with the Node Manager**
- **nmclient will pass your path environment to the NM & it will pass this path to Factory to use when searching for the executable**
  - **if not found in this path, Factory will look for templates for services it manages in the default directory:** configurable at build time, but this is default

    `<ANSAware4-path>/install/<platform>/etc/templates`
  - **executable must either be in your search path or in this directory**

Exercise (cont'd)

- Test new server first by doing `nmclient run NewAlias`, then trying client
- kill service started in this way by: `nmclient kill Alias <activation-#>`
  - Can find out activation-# by: `nmclient showactive Alias`
- post a NM proxy offer so it will be run automatically when needed
- No need to change client program - proxy offer is same as normal offer, as far as client is concerned
- Write a new client that uses the Factory to explicitly instantiate your new Echo service, as shown in previous examples
  - can do this without Node Manager, proxy offers, etc
  - try out using ANSA_MANAGED_EXPORT as shown in the examples

More ANSAware Features

Capsule$Terminate

- **Capsule interface Terminate operation:**
  `Terminate : OPERATION [] RETURNS [ BOOLEAN ];`

this op not shown before - in prev examples, used Fact$Term & Obj$Term

- **Factory$Terminate kills specified capsule by issuing SIGTERM**
- **Object$Terminate kills object by invoking `Destroy_ObjName_Object`() fn**
- **Capsule$Terminate requests that the capsule terminate itself**
  - **will return TRUE if request is accepted, otherwise FALSE**
  - **if accepted, capsule will terminate itself by spawning a thread to perform the actual termination, allowing the `Terminate` invocation to return.**
  - **request will only be accepted if application has supplied a terminator fn**

terminator fn will normally do things to shut down service cleanly, e.g checkpoint state, close open files, withdraw offers from trader, from other svcs has passed them to, destory i/f-instances, any appl'n specific stuff

- **terminator function installed via function `Capsule_SetTerminator()`**

Capsule_SetTerminator function

- **function-signature of `Capsule_SetTerminator()`:**

  ```
  typedef ansa_Boolean (*ansa_CapsuleTerminator)( void );
  ```
  i.e. a function that returns an ansa_Boolean
  ```
  void
  Capsule_SetTerminator(ansa_CapsuleTerminatorterminator);
  ```
  i.e. a function pointer to that function

- **`trclient terminate` causes this terminator function to be called**

  ```
  trclient terminate type context [constraints]
  ```
  output: `trclient: terminating` if terminator function is installed

- **Capsule$Terminate invocations will succeed if a terminator function has been supplied in the capsule in question**

- **but the capsule will only terminate if terminator returns ansa_TRUE**

  this tells infrastructure that it's ok to die

Capsule$Terminate Example

this example is taken from SBank server.dpl

```
ansa_Boolean terminate( void )
{
   checkpoint(ansa_TRUE);
   return ansa_TRUE;   /* allow this capsule to be terminated */
}
void body(int argc, char * argv[], char *envp[])
{
   ...
   /* do all initialisation, instantiate interfaces, etc */
   ....
   /* Set up the Capsule$Terminate handler */
   Capsule_SetTerminator( terminate );
}
```

Capsule$Terminate (cont'd)

- Capsule$Terminate invocation checks if terminator fn installed
  - if not, returns failure
  - if so spawns new thread to call terminator fn, and returns success (indicates terminate request accepted)
- Invocation can only terminate capsule within which it is executing in this manner (spawned thread) because:
  - A "Commit Suicide" interrogation cannot return (the server will commit suicide before it can reply) so the client will time out
  - A "Commit Suicide" announcement is not guaranteed to reach the server
- Spawning thread to do actual termination allows `Terminate` invocation to return

Management interface

- all interfaces automatically conform to `Management` interface-type:
  - has one operation: `GetMgmtInterface`

```
GetMgmtInterface: OPERATION [ domain: ansa_MgmtDomain ]
      RETURNS [ ansa_MgmtTermination ];
```

- support for this operation (function `Management_GetMgmtInterface`) is provided by the ANSAware infrastructure.
- because all interfaces have this operation, trader uses the `GetMgmtInterface` operation to “ping” interfaces of suspect offers

trader does this when client trying to use offer has trouble (client’s infrastr automatically informs trader via Relocate, etc, as described earlier) -operation should succeed (regardless of result). If operation times out, the trader assumes the interface cannot be contacted.

- GetMgmtInterface operation used to obtain interface references for interface’s enclosing Object or Capsule interfaces -management interfaces

Management Interface Definition

```
Management: INTERFACE =
NEEDS BaseTypes FROM BTypes;

BEGIN
```
-Note: need this bcs must explicitly define the InterfaceRef type in base type interfaces (not automatic as normally would be)
```
ManagementRef: INTERFACEREF OFTYPE Management;

ansa_MgmtDomain: TYPE = { union type
      ansa_InterfaceDomain, ansa_ObjectDomain,
      ansa_ClusterDomain, ansa_CapsuleDomain,
      ansa_NodeDomain   };

ansa_MgmtTag: TYPE = { ansa_UnsupportedDomain,
                        ansa_SupportedDomain };
```

Management Interface Definition(cont'd)

```
ansa_MgmtTermination: TYPE = CHOICE ansa_MgmtTag OF
{
   ansa_UnsupportedDomain => ansa_MgmtDomain,
   ansa_SupportedDomain => ansa_InterfaceRef
};

GetMgmtInterface: OPERATION [ domain: ansa_MgmtDomain ]
      RETURNS [ ansa_MgmtTermination ];

END.
```

- `GetMgmtInterface` **operation returns** `ansa_UnsupportedDomain` **for all domains except** `ansa_ObjectDomain` **and** `ansa_CapsuleDomain` **e.g:**

```
! { mres } <- ifRef$GetMgmtInterface(ansa_CapsuleDomain)
```

other domains provided for future mgmt fns at different levels (cluster, node), eg. migration - later

Stub Memory Management

- **when stubc compiles IDL files, generates stub code for each operation of interface**
  - **stubs contain marshalling and unmarshalling functions for all arguments and results of the operation** host/netwk byte-order -exaplain all this - all hidden from user
- **if these args are of types of variable size (contain sequences, e.g. interface-references) storage has to be allocated for marshalling arguments or unmarshalling results**
- **if storage is never freed, a component that makes many invocations will consume more and more memory** each time makes invocations mem allocated & not freed
- **ANSAware provides mechanisms for applications to control how this stub memory is managed**

i.e. when it should be released - app progr'r can decide on most suitable policy for partic appl'n

- **default policy is to free memory quite aggressively; this can be easily overridden** can lead to unexpected results, in earlier examples, we turned off this policy for simplicity.

Freeing Stub Memory

- **server operation:**
  - **args unmarshalled - results marshalled**
  - **all memory freed on operation completion**

i.e. when results have been successfully received by caller (retries, etc - infrastructure knows when finally done) or gives up -could not be any further need for these, so this is fine

- **client: `stub_setFreeCltMem(ansa_FALSE)`** have done this so far in examples
  - **overrides default - causes results to *not* be automatically freed**
- **client default behaviour:**
  - **results of invocation freed next time thread makes any invocation**
  - **freed after args have been marshalled, and before results have been unmarshalled**

Default Stub Memory Management

```
! { res1, new_ir, res3 } <- ir$OneOp( args )
! { results } <- new_ir$Op( res1, res3) /* works fine */
! { results } <- new_ir$Op( res1, res3) /* this will fail */
```

- **Override default policy in three ways: per-thread, -capsule, -all capsules**
- **`stub_setFreeCltMem(ansa_TRUE / ansa_FALSE)`**

  if false, no freeing, app must do explicitly if at all (via stub_free...)

  - **only affects *current thread***
  - **other threads in capsule not affected**
  - **`ansa_Boolean stub_freeCltMem()` - inquires current setting**
- **`Ansa_FreeClientStubMem` global variable**
  - **set to `ansa_TRUE` (default) or `ansa_FALSE` (override)**

will change the policy for the entire capsule, but can be overridden on a per-thread basis by stub_FreeCltMem()

Stub Memory Management (cont'd)

- `ANSA_FREERESULTS` - environment variable
  - `YES` (default) or `NO` (override)
  - all capsules run from shell start up with this initial setting
  - can be overridden for each capsule, and/or thread within capsule
- One exception to all this:
  - results of a PREPC `Import` operation are always kept until the thread finishes
  - can be explicitly freed via `Discard` statement
- for getting started, may be easiest to set:

```
Ansa_FreeClientStubMem = ansa_FALSE;
```

or do this if run into problems, then if fixes them, know what was going on & can analyse a bit more carefully to see what intention was, etc.

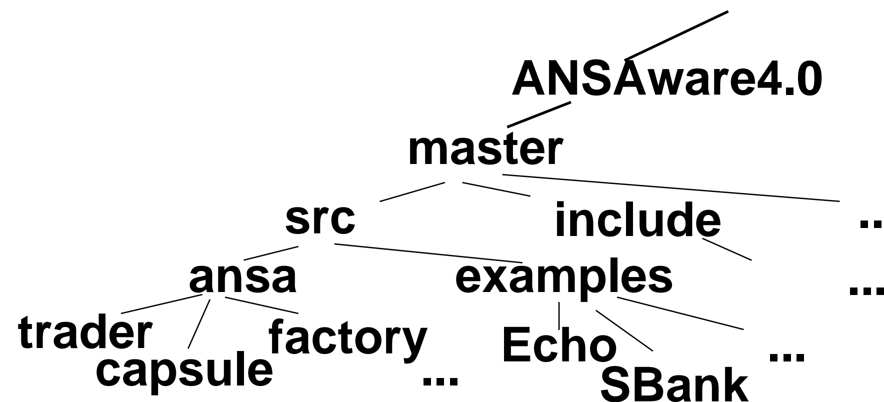
Summary

- Used nearly all of the ANSAware tools & services.
- Seen how to build distributed applications.
- Seen how to use the Factory service to dynamically create and destroy services.
- Seen how to use the Node Manager to manage this.

Installing and Configuring ANSAware 4.0

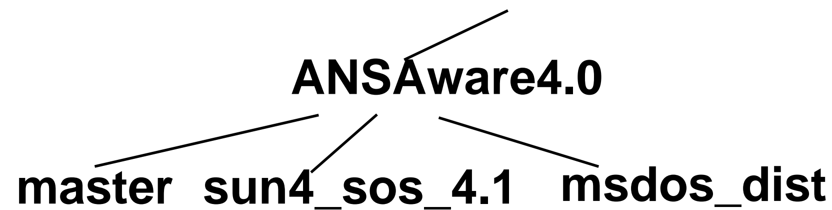
Installing

- ANSAware4.0 versions available for Unix, VAX/VMS, MS-DOS
- Unix release contains "master" copy - can build any version from it
- Others contain platform-specific code only
- Once you have read the files onto your system, you will have a master-tree
- All ANSAware source code is there, so you can look at it, modify it, etc



Different Versions

- **Use script `ANSAware.sh` to generate platform-specific distribution e.g. sun4_sos_4.1, hp300_hpux_8.0, msdos_dist etc.**



Configuring

- **For Unix systems,** `ANSAware.sh` **also builds all files automatically via series of scripts & Imakefiles**
- **Must configure ANSAware system with info such as**
  - **directory-paths**
  - **machines to be running well-known services**
- **This allows ANSAware system to build *TraderRef* and into capsule-libraries, so every capsule can automatically contact trader**

Well Known Interfaces

- Trader.Trader interface is "well-known"
- Capsule library looks in 3 places for this info
  - environment-variables
  - files
  - compiled-in definitions
- env’t vars: MASTER_ADDRESS, TRADER_ADDRESS

wkifref

- **Strings for redefining these env't vars can be created using the program** `wkifref`

  If machine is called "burgess" and address is 192.5.254.30, then:

  ```
  wkifref "" 192.5.254.30 11002 burgess 0 2 burgess 1 11002
  ```

  will produce:

  "[1: { 'c005fe1e2afac005fe1e2afa0000000000000000',0,1, [1: 2 :
  { [2: 0, 0] [2: [10 : '00066275726765737300'], [4 : '0000000f']] [] [] },
  { [2: 1, 0] [2: [6 : 'c005fe1e2afa'], [4 : '0000000f']] [] [] }]] }]"

  ```
  setenv TRADER_ADDRESS "..."
  ```

- **Files** `masterfile` **and** `traderfile` **can be used for this purpose as well**

ANSA:

New Developments

Speaker Notes

New Developments

- **Activation / Passivation**
- **Storage** — **of inactive objects/services**
- **Migration**
- **Location** — **of services that have moved / become passive**
- **Groups**
- **also Security, Transactions, Types** — **but I'm not going to talk about these**

computational model shown earlier was just basic - work going on in architecture on all of these topics to improve comp'l model to deal with a range of more complicated issues that arise in dist'd computing - this stuff not in ANSAware yet, but may be in future releases - some of it has been prototyped, some just architectural so far

Object persistance

- **Capsules & objects that are not busy could be stored away until needed**
  - **this way, save resources, but services can be called back into existance when needed**
- **Only activation/passivation/migration/storage of *objects* has been thoroughly designed** leaving aside issues of storing info abt executables for capsules, etc
- **Clearly, the capsules in which objects exist need to be dealt with as well**
  - **future work will address this**

Snapshots

- a *Snapshot* is a representation of object & its state
- Snapshot infrastructure provides machine- & O/S-independent means of
  - producing a snapshot of an object
  - installing a snapshot of an object (into a capsule that is capable of supporting that type of object)
- Snapshot-base service stores snapshots, separately from service's code
- Storage and management of capsules managed separately from that of objects & their snapshots
  - creation of appropriate capsule-types done via factory
  - also creation of appropriate object-type within capsule
  - object/service manager keeps track of appropriate capsule-executables for various machines, O/S's etc

Snapshots (cont'd)

- Snapshot consists of three components
  - object type
  - current state of object - bindings, references to interface-instances
  - representation of interface-types & instances currently supported by object
- Restrict snapshot to be of idle object onlybefore snapshot can be made or installed:
  - ongoing activities must finish
  - new activities prevented
- Programmer declares (via PREPC declarations) which components of state are to be put into snapshot
  - components must be of IDL-defined types so un/marshalling stubs can be generated for snapshot production / installation

Locators

- Locators keep track of old vs new locations of objects
- Object may have become passive, or moved to a new location
- When client's infrastructure calls locator, locator will hand out new interface-reference, replacing out-of-date one

Activation / Passivation

- **Objects are moved (swapped) between secondary and in-memory locations by passivation and activation functions similar to virtual memory mechanisms** objects could passivate selves according to various policies, eg. if inactive for certain time
- **Object must hold reference to SnapshotBase to be able to passivate**
- **Passivation steps:**
  - **decide to passivate or sanction external passivate request**
  - **wait until current activities have completed and prevent new activities**
  - **produce a snapshot**
  - **store the snapshot in a SnapshotBase**
  - **in appropriate Locators, register each interface instance as being mapped to the stored snapshot**
  - **terminate**

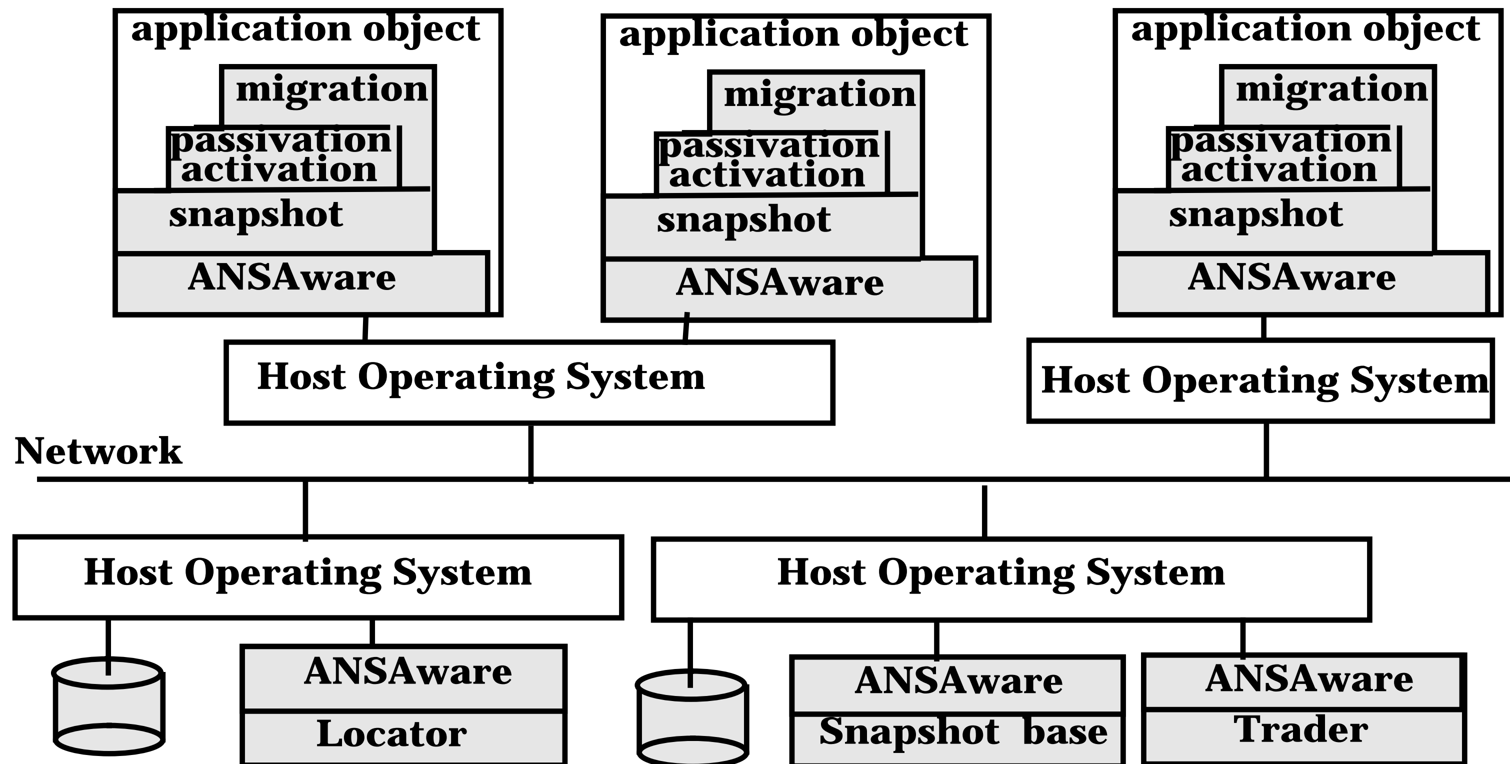
Activation / Passivation (cont'd)

- **When a client attempts to invoke a passivated object, it will time out**
  - **client infrastructure handles the timeout by contacting Locator or series of Locators until successful**
  - **Locator will give out reference to a Snapshot in a SnapshotBase**
- **Activation steps:**
  - **receive a reference to the Snapshot from a locator**
  - **instantiate a new object via a factory** of course something has to be done abt finding/creating capsule to support object - big hand-wave
    - **pass new object the snapshot reference from the SnapshotBase as args to Instantiate operation;**
  - **new object installs snapshot, creates interfaces, and updates interface mappings in the locator**
  - **client infrastructure can now rebind the failed reference to its replacement**

Migration

- **Migration is the process of moving an active object from one location in a distributed system to another** migration managed & requested, does not spontaneously happen
- **In case node has to be taken down for maintenance, etc**
- **Services of object temporarily unavailable during migration**
- **Migration also accomplished via snapshots**
- **Can migrate passive object by activating at new location**
- **Active object migrates by:**
  - **activating clone of object at new location**
  - **taking snapshot of existing object, and installing into new**
  - **update locators**
  - **terminate**

Persistent Object Infrastructure



yes - it's mostly all done in infrastructure

ANSA

Groups

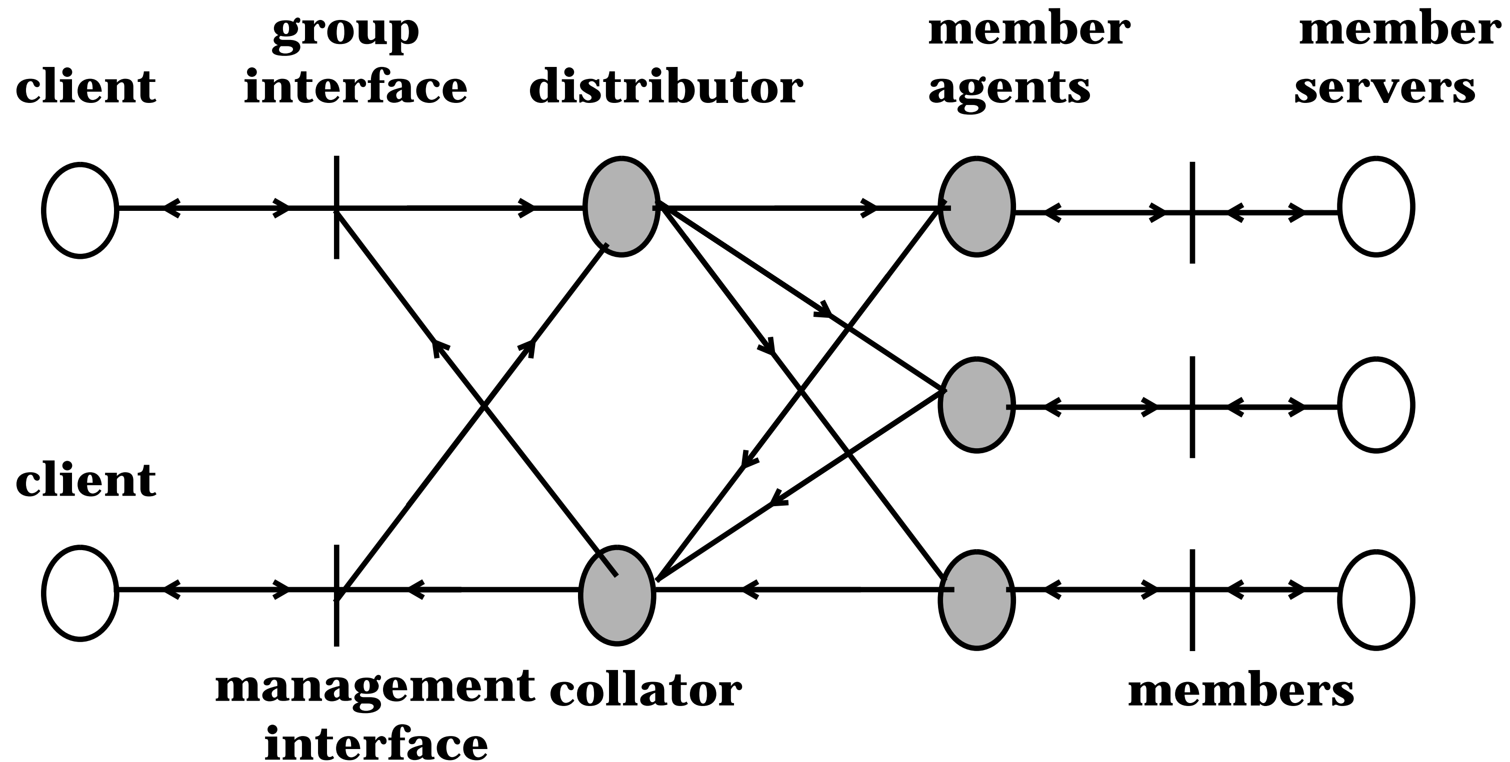
- What is an *interface-group*?
  - group of interfaces of the same type that act like a single interface
  - replicated for fault-tolerance some members could crash, but invoc'n still processed by remaining
  - reliability of results
- Distribution / existence of service as a group hidden from service user
- At application-level, membership of group hidden from group members
- All group members must conform to same interface-type
- Group management tool manages group membership
  - population control
  - initialise first group member

Groups (cont'd)

- **Create a group (each member) via a Factory. Two interfaces will result**
  - **group interface** - for comms btwn group members (synch'n, verifying other members still there etc)
  - **management interface** - for grp mgmt fns - start grp , change pop of grp
  - **By having two interfaces, make clear distinction between service of a group, and management of group's membership**
- **Logical components of a group:**
  - **distributor - broadcasts invocations on group i/f to all members**
  - **member agent - cooperate with other members to ensure**

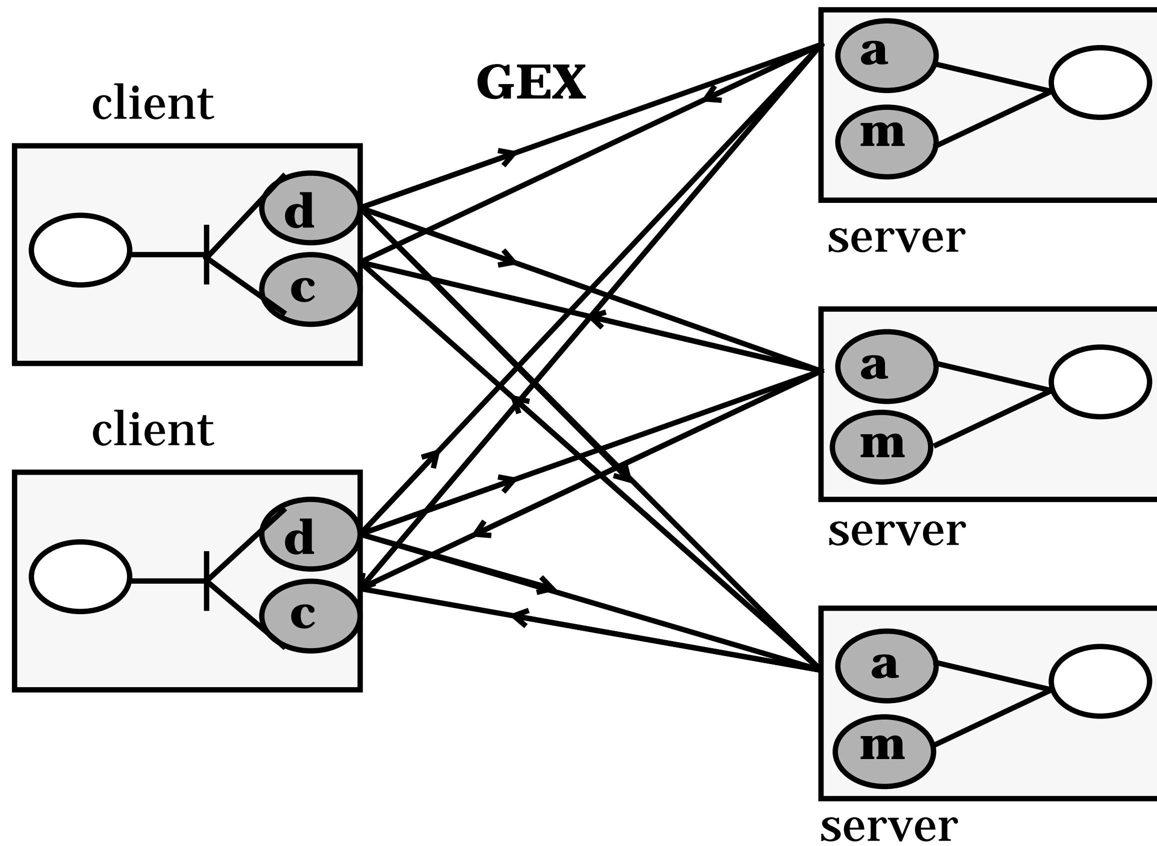
    cooperation - make sure no invocations missed, all must have same set of invoc'ns
    ordering - invoc'ns must be processed in same order in all mems
    failure detection - of other members
  - **collator - collects each members result to produce single result for client**

Logical Entities in a Group



ANSA

Groups: Implementation



Groups: Details

- clients:
  - after arguments marshalled by stubs, passed to dispatcher function
  - dispatcher passes to distributor in non-group operation this would just RPC the server i/f
  - distributor broadcasts invocation to each member (non-blocking RPC)
  - control passes to collator which awaits response
  - when all results received, one results returned to client invocation
- members:
  - invocation collator -ensures all invocations arrive
  - invocation sequencer - for agreeing order of processing
  - timer - expiry indicates member failure

ANSA

Groups: Client & Server Infrastructure