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. Version 00.02 is as delivered to Telefonica in July 1994, with several errors, and with some of the original material missing.

APM.1261.00.02 **Draft** 4 September 1995
External Paper

Distribution:
Supersedes:
Superseded by:

Copyright © 1995 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

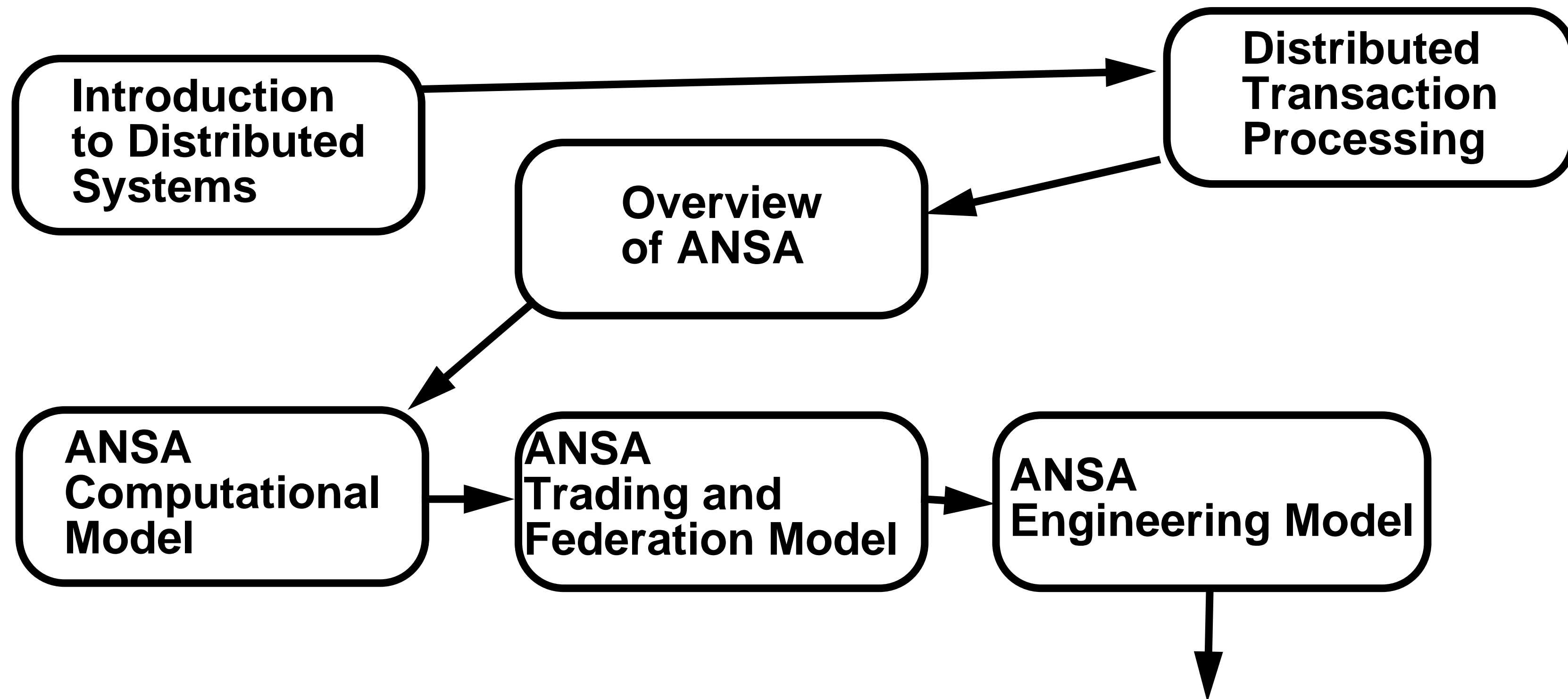
Welcome to ANSA

Chris Mayers (cmm@ansa.co.uk)

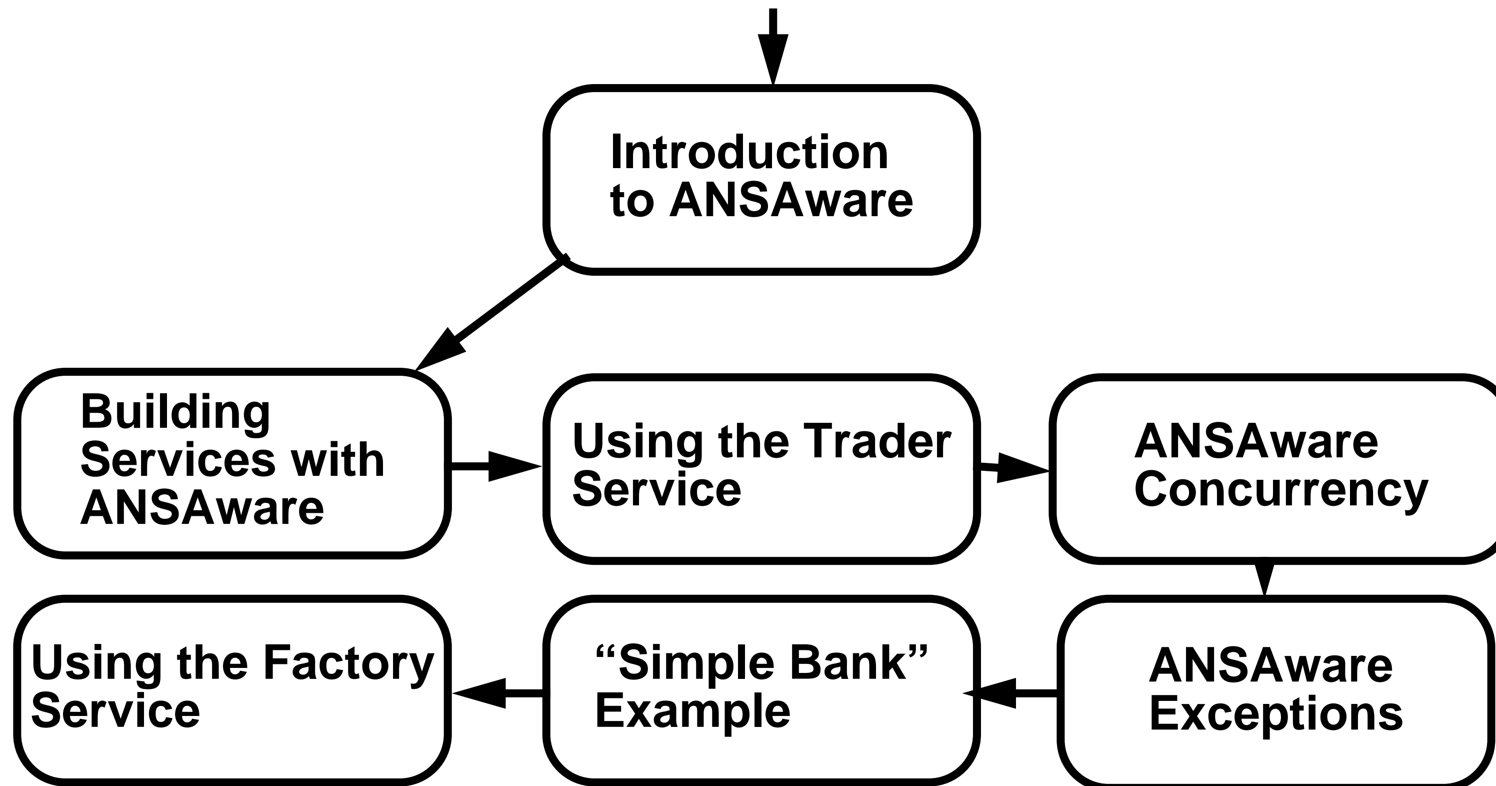
About this course

- It is an introduction to distributed processing, distributed systems, and distributed computing
- ... based around the ANSA concepts for building open distributed systems
- It includes hands-on sessions...
- ...based on ANSAware, our toolkit for building distributed applications

Course Outline - 1



Course Outline - 2

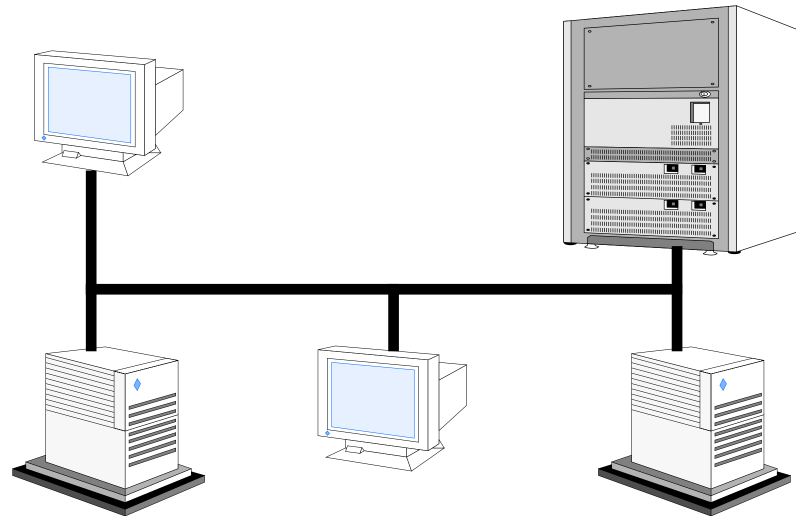


Enjoy the course!

- **... and ask questions whenever you wish**

ANSA

Introduction to Distributed Systems



Speaker Notes

In this session

- **Explain the business issues surrounding distributed systems**

- **Explain in what ways distributed systems are different**

Many people are confused, and have misconceptions

- **Explain what ANSA is, and how it helps you build distributed systems**

What's the real business challenge?

Coping with change

The pressures for change

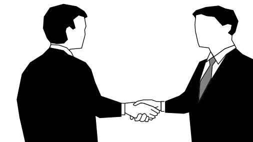
- **Political, economic, social, and technological...**

  - **Globalization**

Worldwide markets

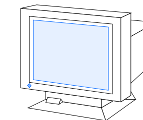
  - **Rapid organizational change**

Dynamic organizational boundaries. Everyone wants to do business with the best, anywhere in the world



  - **Increased customer expectations**

  - **Inexpensive computing and telecommunications**



But it is hard to make this a competitive advantage; your competitors can do this too

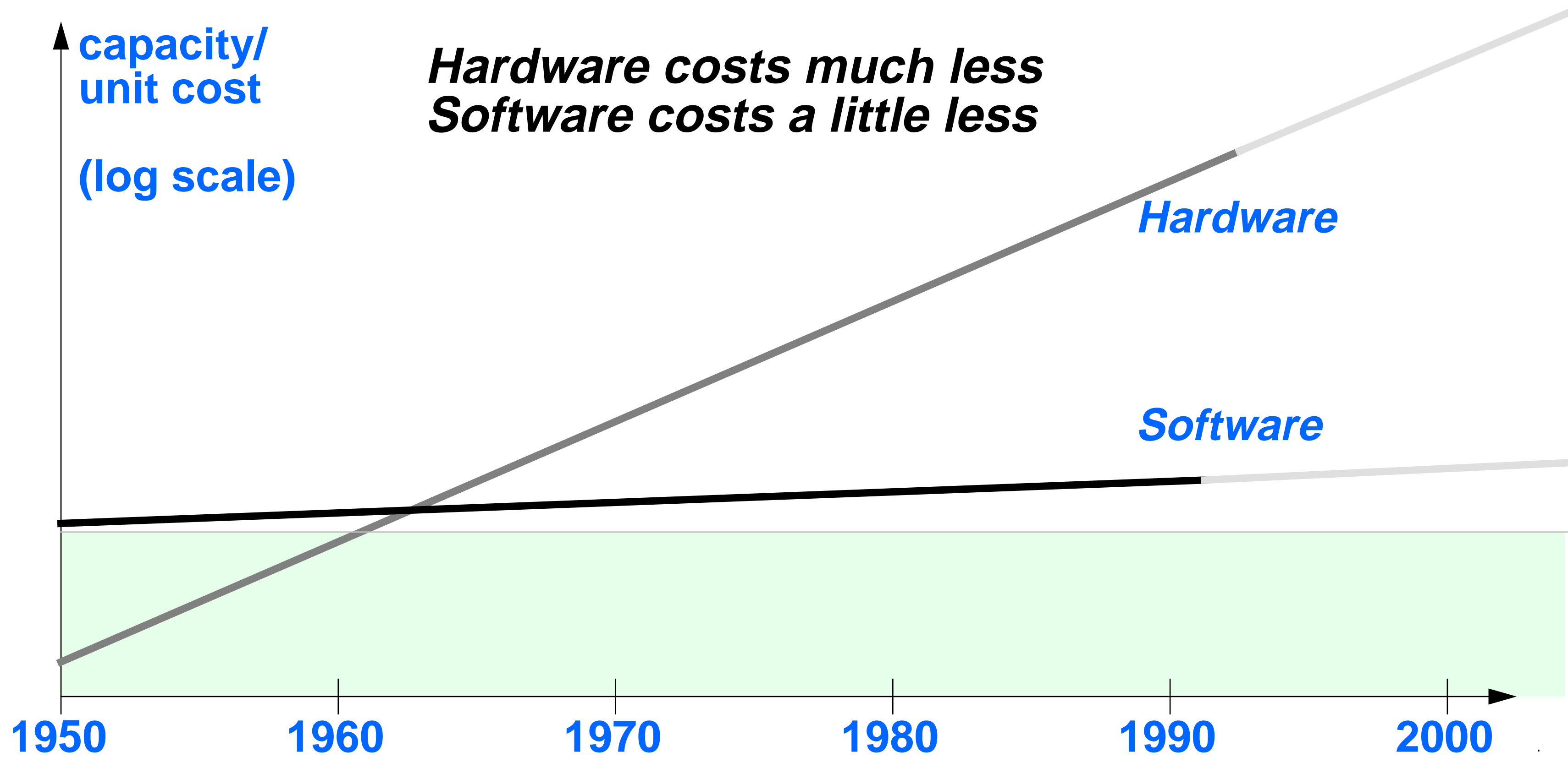
The business challenge for telecommunications

Meet the customer's service expectations...

- **timely:** **I want it immediately**
- **personalised:** **I want it to meet *my* needs**
- **competitive:** **I want to pay as little as necessary**
- **dependably predictable:** **I want it to be reliable**
- **integratable:** **I want it connected to my PABX, PC, …**

…. before your competitor does

Costs of providing service



Software cost in providing services – a new problem?

- **We have already tried these solutions....**
  - **Project management**
  - **Requirement analysis**
  - **Analysis and design methodologies**
  - **Informal and formal approaches**
- **....they work, but not for complex systems**
  - **they do not *scale***

The service provider's problem - Summary

- *Providing networked information services*
  - not simply the physical transport of data
- *Satisfying the Critical Success Factors*
  - services must be *developed rapidly*, to meet market windows
  - new services must *interwork* with existing services
  - services must be *easy to deploy*
  - services must be *easy to manage*
- *Meeting the customer's expectations*
  - before your competitors
  - at a price the customer will pay

About distributed systems

- **Distributed systems are those which consist of interconnected cooperating components**
    - **there is no central machine or group of machines**

The worldwide telephone network is a distributed system

- **Distributed applications are those written for a distributed system**

They are designed with distribution in mind

- **Distributed processing is the method for designing and building distributed applications**

- **Distributed computing is the technology we use in distributed systems**

Examples of distributed systems

- **Diverse business areas**
  - **Telecommunications**
  - **Airline reservations**
  - **Retail point-of-sale**
  - **Banking**
  - **Command and control**
  - **... and many more**

Naturally distributed systems

- **Built at the limits of the technology**

It has been hard to build these systems at all, but now we have learnt from these systems and can build better systems for the future. First, we must understand what makes them different...

Features of distributed systems

- **Diversity: many types of machines in the same system**
- **Legacy: evolution and interworking of existing systems**
- **Scalability: low cost of computing per machine**

PCs and workstations

- **Decentralization: no single point of control**
- **.... these differences are fundamental**

Distributed systems are fundamentally different - Separation

- **Separation**

The most obvious difference

- **remoteness**
- **migration**
- **no shared memory**
- **partial failure**
- **weak global consistency**

Distributed systems are fundamentally different - Diversity

- **Diversity**

Also known as heterogeneity

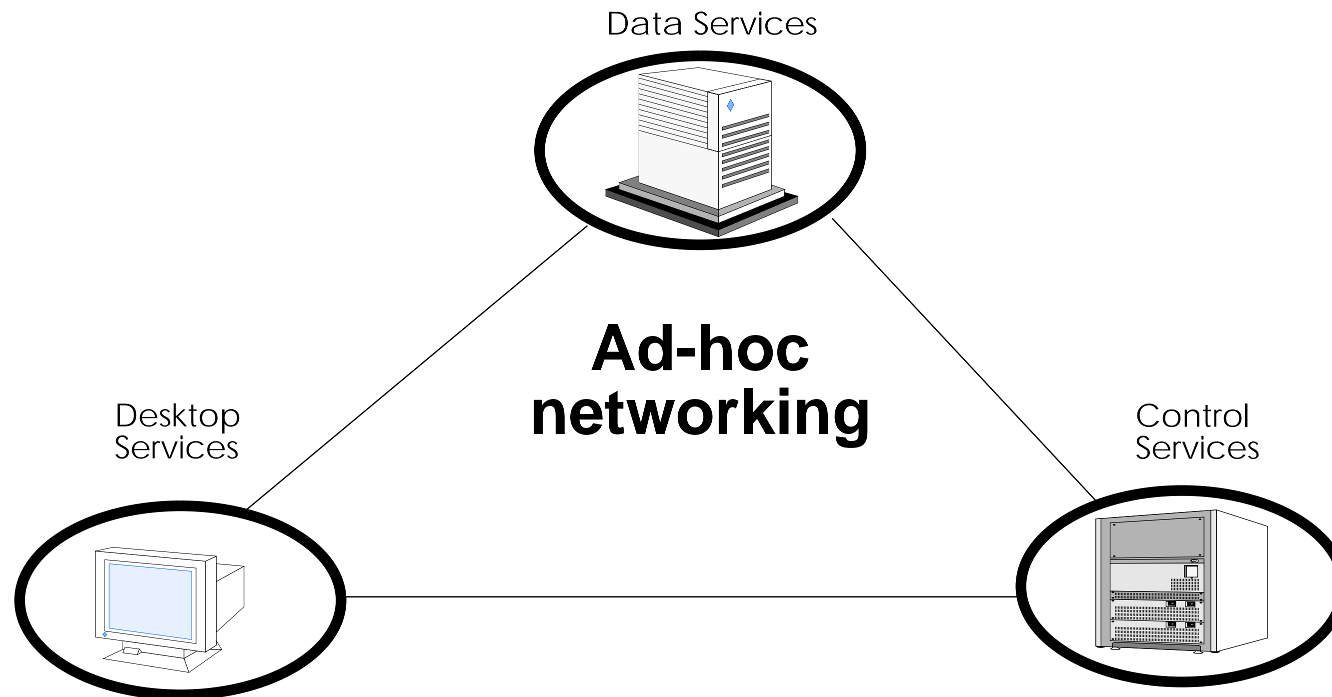
- **diversity of scale**
- **diverse data representations**
- **diverse naming schemes**
- **diverse hardware and software**
- **diverse communications mechanisms**

Distributed systems are fundamentally different - Federalism and Concurrency

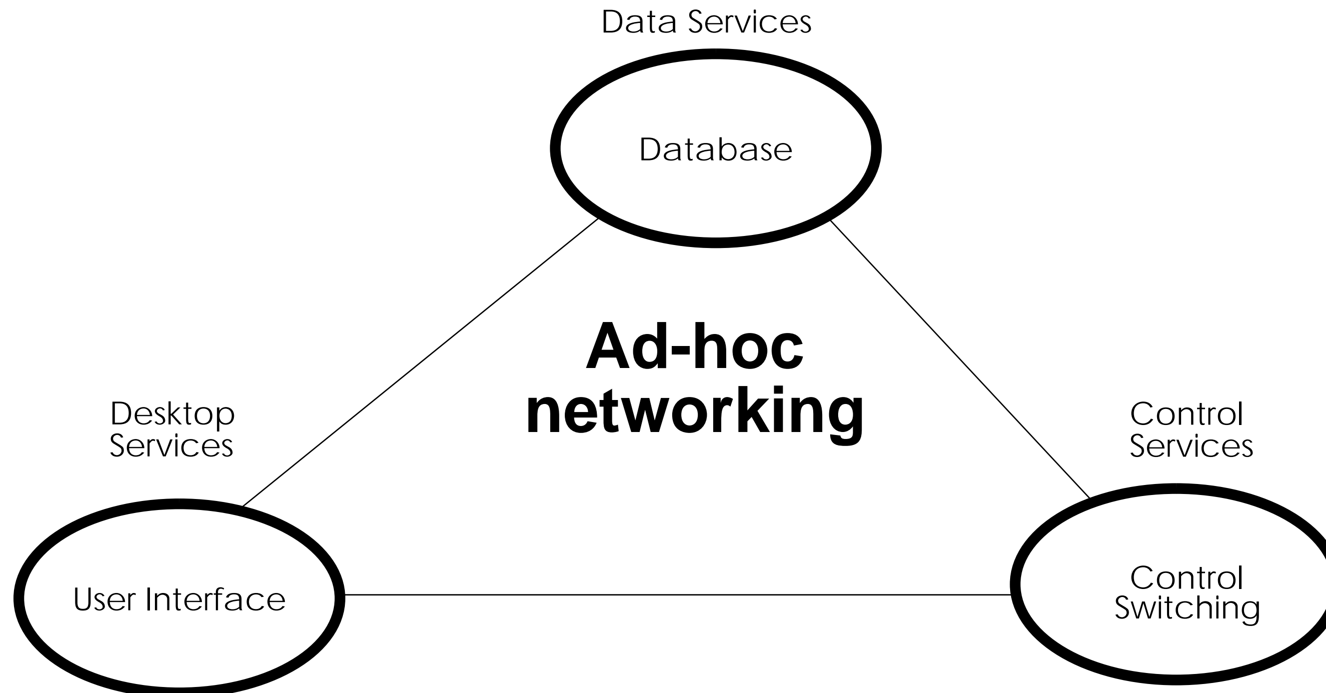
- **Federalism**
  - **no central authority**
- **Concurrency**
  - **simultaneous operation**
  - **multiple copies**

We cannot ignore these differences or wish them away. It is like the shift from menus to GUI. They are a fact of life, and are the rule rather than the exception. But there are ways of making the differences easier to handle

How distributed applications are built now



Typical skills needed to build them



Where might you place your background and your team's within this triangle? Not many are likely to be right in the middle.

ANSA

Skill cultures

Note that each of the cultures has at least one existing approach to distributed systems

- **Data Culture**
  - **Remote data access**
  - **Distributed databases**
  - **Stored procedures**
  - **Object repositories**
- **Desktop Culture**
  - **Individual PC productivity services**

Possibly linked via DDE/OLE

  - **File and printer sharing**
  - **Group productivity services**
  - **Mobile computers, universal personal digital communication**
- **Control Culture**
  - **Device control**
  - **Workflow**
  - **Robust messaging**
  - **Intelligent networking**

Possible solutions on offer?

- **Client-server**
- **Object-orientation**
- **Open systems**
- **Rightsizing**
- **... no single approach or technology will dominate**

These are not solutions, but they are useful

These are all good ideas as far as they go. But they are only ingredients. They do not address the fundamental differences that we just covered. Nor do they solve the ad-hoc networking problem. How do you use these good ideas as part of a solution?

Different policies for different applications

Also, there isn't going to be a single solution that meets the needs of every distributed system, because we are in a world of some very fundamental trade-offs, including.....

- ***Availability* versus *Consistency***

Availability means copies, increases risk of inconsistency

- ***Autonomy* versus *Uniformity***

Autonomy gives more freedom, but leads to differences which increases complexity

- ***Security* versus *Convenience***

Security makes things harder to do

- **... and many other unavoidable trade-offs**

Means that one size does not fit all.

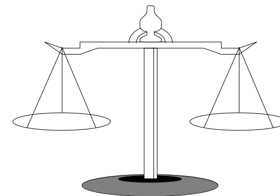
Distributed systems and coping with change

- **How do distributed systems help businesses cope with change?**

- **How do distributed systems balance**

  - **the demands of change...**

  - **...and the demands of continuity?**



The demands of change

- **Pressures for change make distributed computing *necessary*, as well as *possible***
  - **in the near future, most systems will be distributed**
  - **world-wide business requires world-wide systems**
- **Information networks are the starting point...**

LANs, WANs and the Internet are now core business technology

- **... how to build systems to coordinate information from many sources?**
  - **diverse sources: old systems, new systems, and other organizations' systems**
  - **separate sources: from different places at different times**

This is just 'plumbing' technology. Shipping data around is the easy part - although often not as easy as it should be!

The demands of continuity

- **Preserving investment**
  - **in people, and the legacy systems they use**
- **Bridging the old and the new**
  - **evolution not revolution**

The technical challenge

- **Provide a framework for systems that:**
  - **integrate products from many vendors**
  - **are owned and managed by many organizations**
  - **can grow larger than the international telephone network**
  - **can evolve gracefully**
  - **allow different kinds of applications to interwork**
  - **preserve the investment in existing technology**
  - **have lower development and operating costs**
- **... This framework is an *Architecture* for Open Distributed Processing**

Not in the sense of a product family “Intel 80x86 architecture”, or a post-hoc rationalization “SAA”.
It is Architecture as a common set of underlying principles.

It is a way of thinking about systems. Because it is abstract, it can be difficult to grasp.

Other demands on the Architecture

- **Must be easy to use and understand**
- **Must be widely applicable**
- **Must be durable and long-term**
- **Must be practical and proven**
- **Must be vendor-neutral**

Not enough just to be able to interoperate using one vendor's interoperability products

- **Must be backed by the authority of international standards**

meets these demands

Its principles are:

- **Distributed systems have different properties to centralized systems**
- **Different applications need different solutions**

Allow maximum freedom to vary and show how to cross boundaries where needs mismatch.

- **Object-orientation simplifies designs**

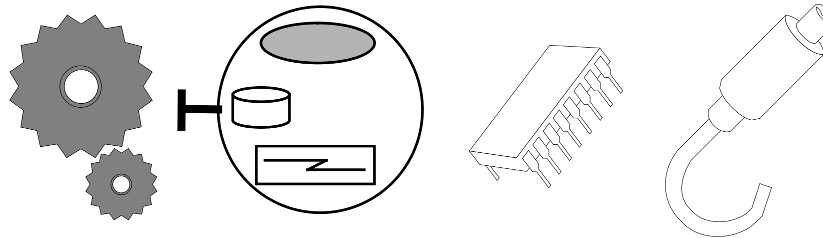
Need a careful balance between simplicity and expressive power; needs to be simple so that conversion at boundaries is easy, but also not so simple that using it is difficult and long-winded; general enough to apply to any component or interaction.

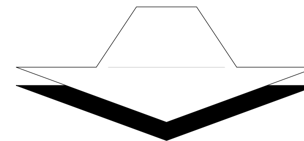
- **Unnecessary complexity should be masked from the applications**

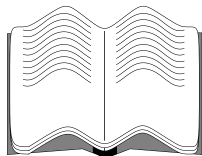
No doubt distribution is more complex - more errors, more information about system to manage. Ought to make them harder to use - but it needn't if all unnecessary complexity hidden. A question of finding good abstractions(i.e. the right things to want).

Architecture

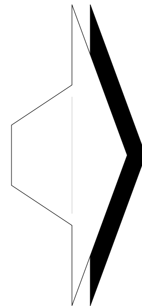
Basic building blocks

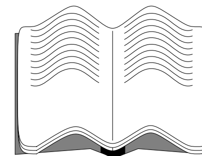




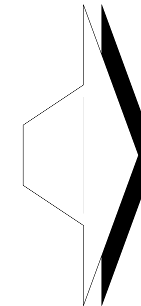


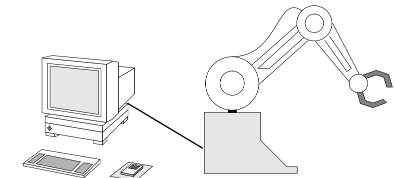
Combination rules





Recipes





In The Architecture there are...

- **Components**
  - standard functional building blocks
  - tools
- **Rules**
  - embodying principles and assumptions
  - deliberately prescriptive; to be enforced!
- **Recipes**
  - for satisfying commonly-occurring requirements
- **Guidelines**
  - for making design choices and trade-offs
- **Concepts**
  - clearly defined and delineated
  - as general as possible, and as few as possible

The Architecture leaves you to decide...

- **Which products to use**

The specific operating system types, and so on

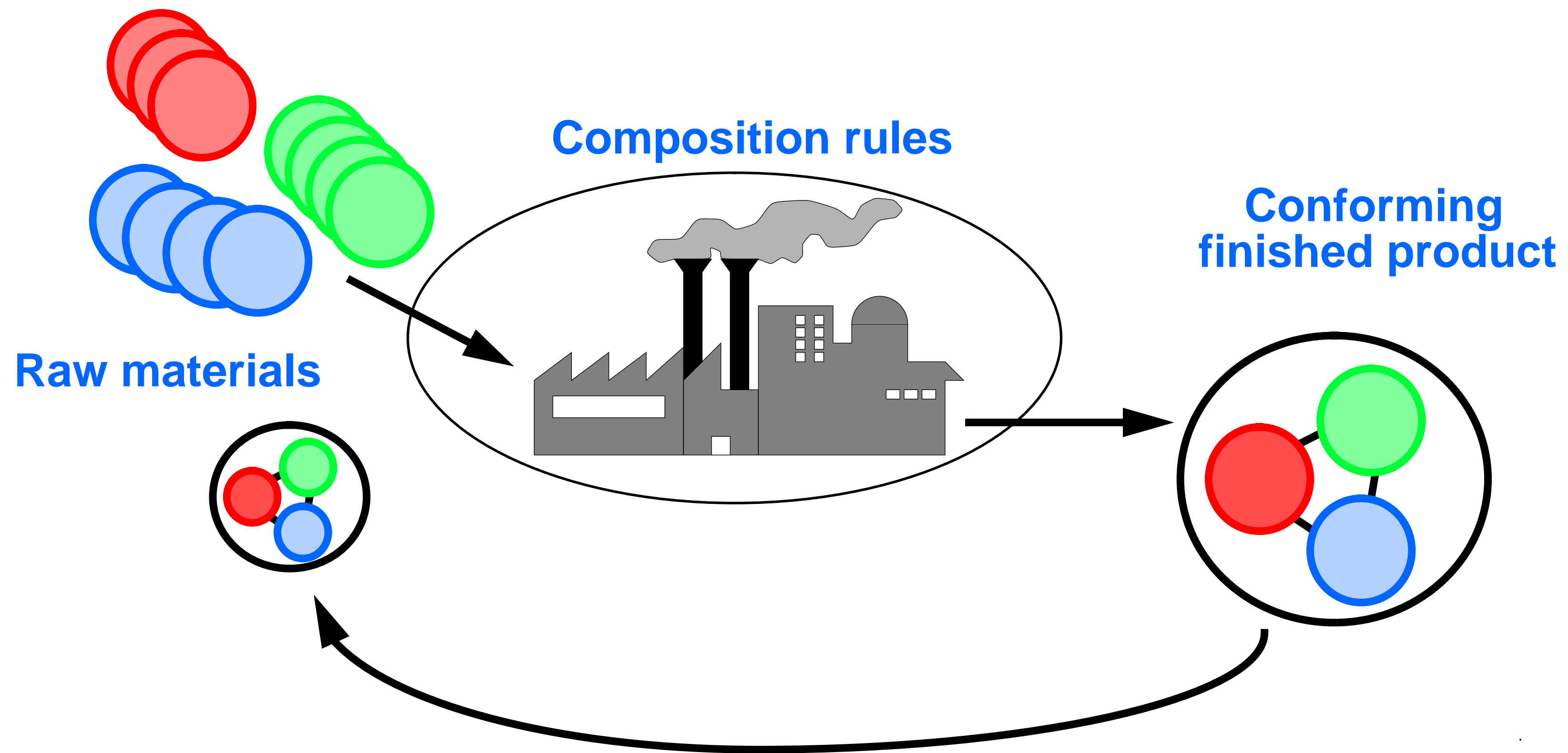
- **Which software development methods to use**

Not a step-by step approach, nor even a definition of notations.

- **Which user interfaces to provide**

The Architecture is only concerned with system aspects that are relevant to distribution

Architecture for reuse



About ANSA

- **By ANSA we mean two things:**
  - **a collaborative research programme into distributed systems. It started in 1985, and has been running continuously since then.**
  - **the results of the research programme; an architectural framework for building distributed systems**

Background to the ANSA research programme

Has expanded from the UK, to Europe, to the world

- **1985-1988: the ANSA project**
  - **Phase I sponsored under the UK Alvey programme**
  - **Focus on development of the architecture**
- **1989-1993: the ISA project**
  - **Phase II sponsored under the EC Esprit programme**
  - **Including partners from Austria, France, Germany, Greece, Holland, Italy, Sweden**
  - **Focus on completing the architecture**
- **1993-...: ANSA Phase III**
  - **Phase III sponsored by project partners**
  - **Focus on specific topics**

ANSA and ANSAware

- **ANSAware is a software toolkit for building distributed systems**

- **Produced from the ANSA research programme...**

- **... now in use at over 150 sites around the world**

Distributed systems - Summary

- Distributed systems have fundamentally different properties from centralized systems

- Coping with change is a fundamental business need

- The ANSA Architecture shows how to exploit distributed systems to cope with change

ANSA Phase III Topics

- **Federation**
- **Dependability**
- **Real-time**
- **Tool support**

This is research, and has only just got under way. For more information see your sponsor representative

Systems built with ANSA

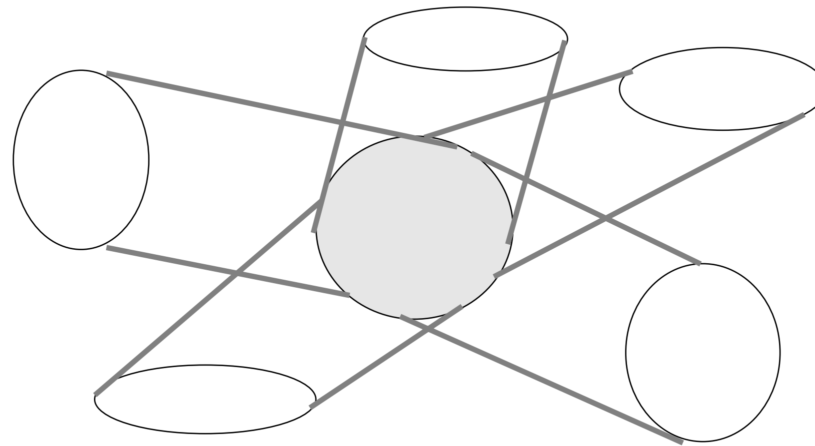
- **NASA Astrophysics Data System (ADS) - a distributed multi-database**

- **GESI Distributed Health-care Patient Management System**

- **CTI Distributed Newspaper “Hypertext” Composition**

- **AEG OSI resilient network management overlay**

An Overview of the Architecture



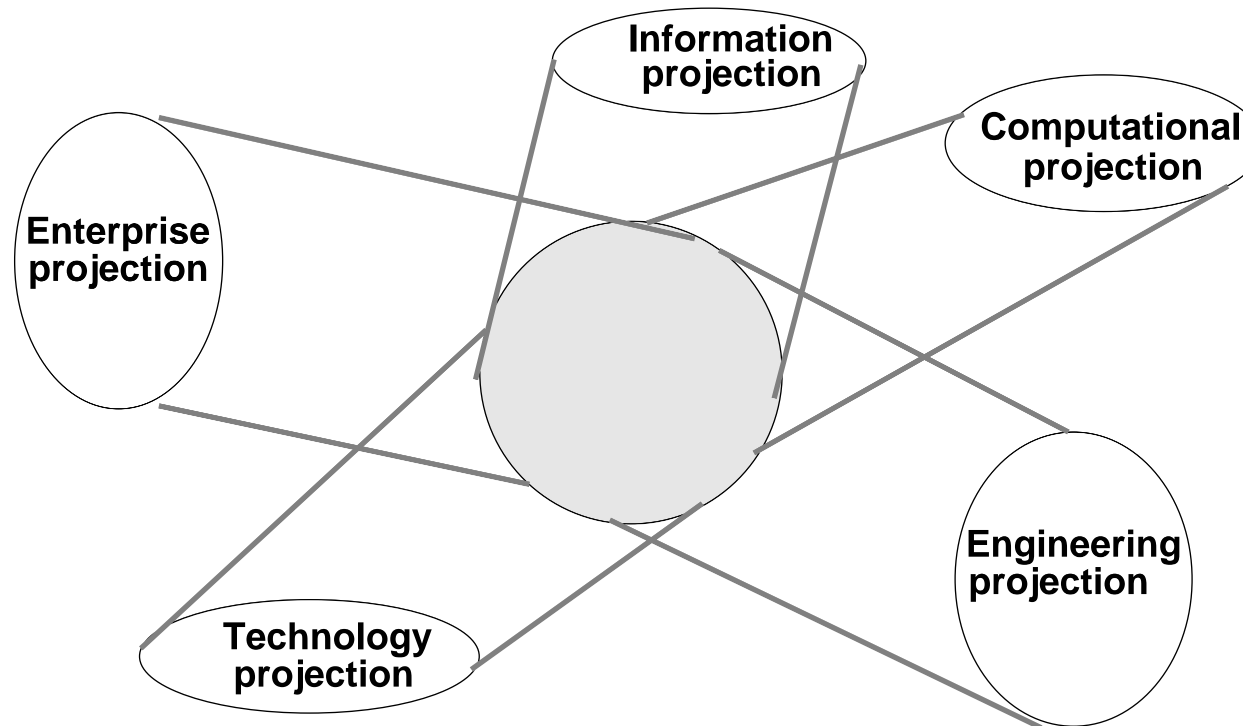
Speaker Notes
ANSA architecture and ANSAware software are distinct: ANSAware typically lags behind the arch by year or more

Distributed Systems have many aspects

- **Distributed systems involve many different people (the *stakeholders*)**
  - **business managers, users, IT managers, IT developers,...**
- **These people are concerned with different aspects of the system**
  - **they see the system from a different viewpoint**
  - **each viewpoint is important**
- **We need to be able to separate out these concerns when describing distributed systems**
  - **so that each stakeholder can see that their needs are satisfied...**
  - **... without being overwhelmed by descriptions of aspects that are irrelevant to them**

ANSA uses 5 different viewpoints (*projections*)

- These are of the same system and are not layered



Content of the 5 projections

- ***Enterprise*** **- the *purpose* of the enterprise and the system within it**

airline booking sys - access all co's DBs in single, straightfwd manner

- ***Information*** **- the *meaning* of the information within the enterprise**

flights, planes, passengers, sites, booking agents, who accesses it, how etc

- ***Computational*** **- the *execution* as a model of distributed processing**

break into objects/entities - flights DB svc -e.g. diff DBs but related for flights, prices, routes,- booking agents clients of this, what info given by whom

- ***Engineering*** **- the *mechanism* for realising the computational model**

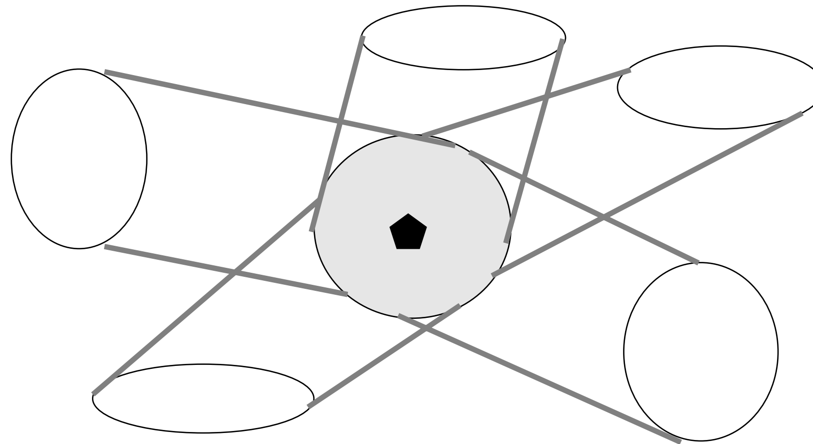
where are diff DBs kept, how many, what replication of objs req'd for desired reliability, what commit protocol, what retransmission timeout is acceptable/default - what are actual operations & interactions btwn these cpts

- ***Technology*** **- the *conformance* of hardware, operating systems, compilers,...**

what machines, where, what performance statistics, what MPS's

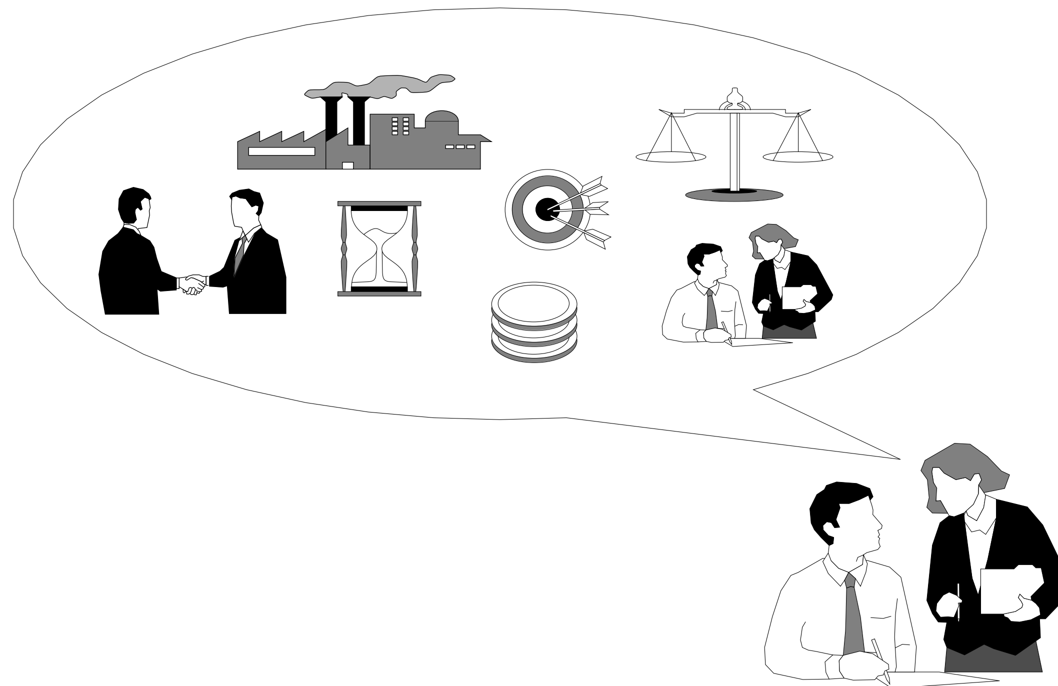
The projections are linked into a framework

- **Because the projections are views of the same system...**



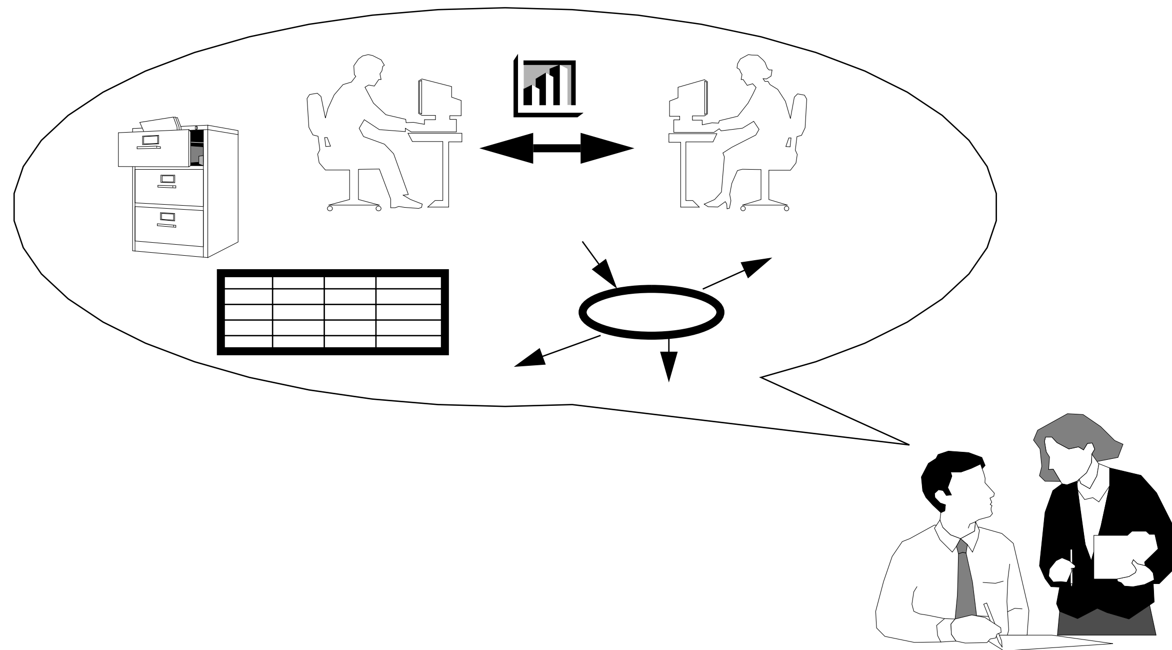
- **... there are cross-checks between them**
  - **to make sure the projections are consistent with each other**

The Enterprise projection



- **Describes agreements, targets, people, time, money,...**

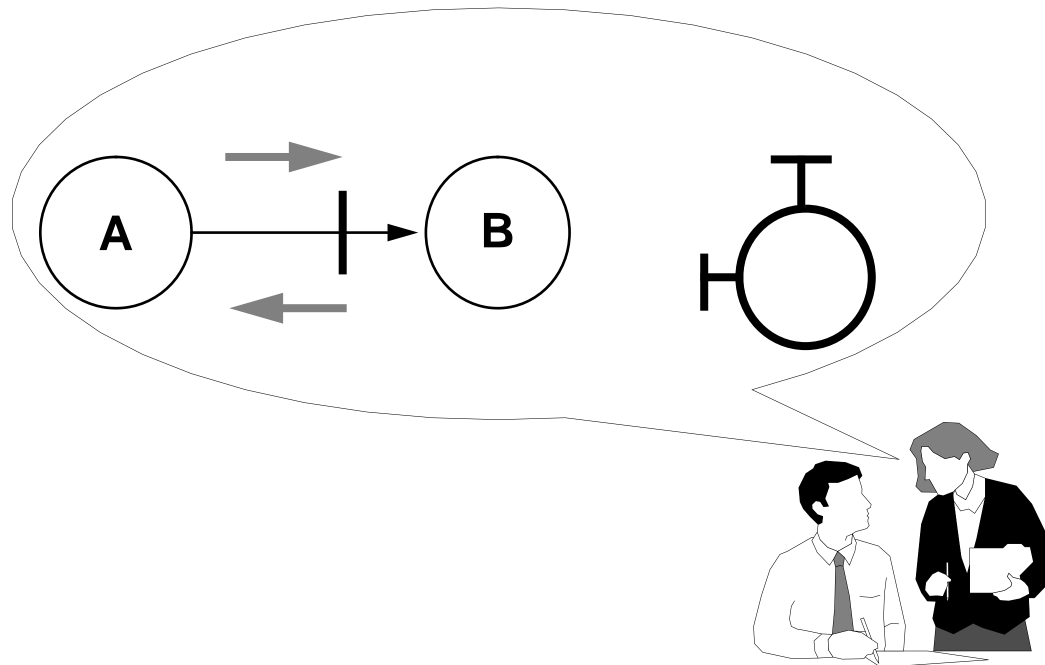
The Information projection



- **Describes information flows, information stores, information users,...**

ANSA

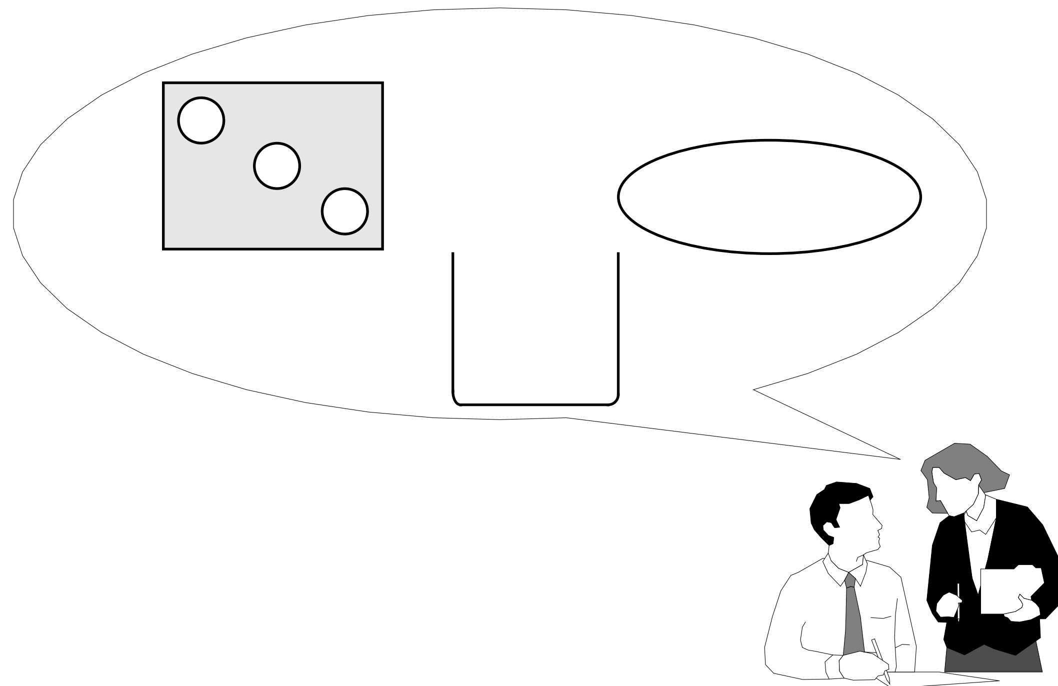
The Computational projection



- **Describes objects, interfaces, operations,...**

The application designer’s viewpoint

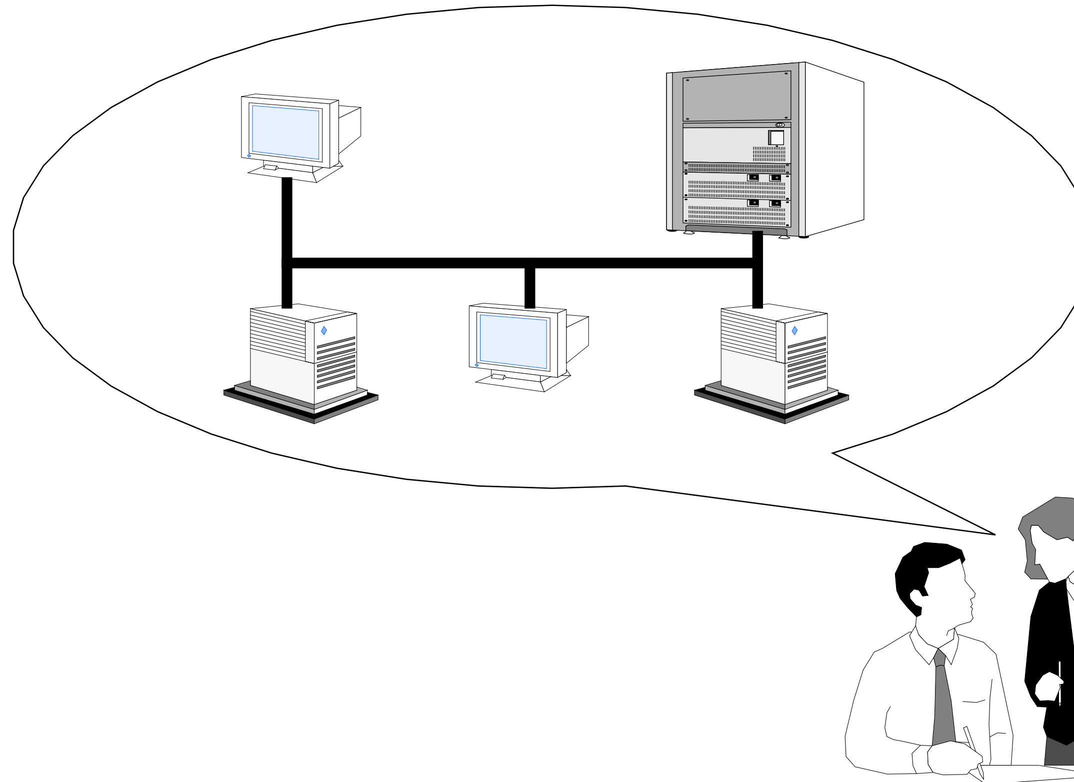
The Engineering projection



- **Describes clusters, nodes, channels,...**

The system builder's viewpoint

The Technology projection



- **Describes how the system design uses the actual technology**

Each projection has a corresponding model

- For example, the ANSA Computational Model
- The model describes
  - the concepts that appear in the projection
  - how to apply these concepts...
  - ... so you can describe your own system using the model
- We are continually refining these models
  - ... as ANSA users give us feedback
  - ... as new research ideas evolve
- Some specific topics have their own model

Currently, most new work is going into the Enterprise projection

A key concept - Service

- To separate a system into parts, each part must offer a coherent *service*
- The service must be explicitly specified
- Specifications are declarative; *what*, not *how*
- The provider of a service agrees to meet the specification
- The provider does not reveal how the service is provided
    - it could be via a mainframe legacy system
- ... in a word, *encapsulation*

Distributed systems are different

- **Many traditional system design assumptions must be reversed**

Distributed systems design is different: there are additional problems to grapple with, and new benefits to consider

Traditional	*Reversed*
Local	**Remote**
Sequential	**Concurrent**
Homogeneous Environment	**Diverse Environment**
Fixed Location	**Mobile**
Single Copy	**Multiple Copies**
Synchronous	**Asynchronous**
Direct	**Indirect**
Shared	**Separate**
Global	**Context Relative**
Complete Failures	**Partial Failures**
Early Binding	**Late Binding**

You must be able to handle these reversed assumptions in your system design

- **A systematic approach is needed to avoid these assumptions**

Exploiting the reversed assumptions

- **Exploit positive consequences**
  - **Consider, for example...**
  - **Late binding: Trading supports choice of Quality of Service**
  - **Multiple copies: Concurrency supports parallelism**
  - **Partial failure: Replication supports availability**
- **Mask negative consequences**
  - **Use selective *transparency* mechanisms, for example...**
  - **Migration transparency: Isolates client from service relocation**
  - **Replication transparency: Isolates client from multiple copies of service**

Note that replication appears in both lists

Handling the reversed assumptions - The Computational and Engineering projections

- **Isolate specification of transparencies from their design**
  - **Computational projection defines the transparencies**
  - **Engineering projection provides the mechanisms**
  - **Applications developers just state which transparencies they need**
- **Automate the building of transparencies**
  - **software tools can construct transparencies from the engineering mechanisms**

Characteristics of distributed systems

- **Federalism**
  - peer-to-peer and autonomous operation
  - within and between organizations
- **Concurrency**
- **Diversity**
  - of type (hardware, operating system, communications mechanism,...)
  - of scale (speed, capacity, bandwidth)
- **Separation**
  - in space
  - in time
- **...ANSA takes an integrated approach to these**

Approach to Federalism

- **Allow each system to control its own policies and services locally**
  - **centralized control is infeasible**
- **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**

- separate sources of authority e.g. diff dep'ts, companies, diff technical policies, etc.

- also, local systems usu. not built to assume non-local administration but need for interoperation arises when >=2 such systems want to interwork - interworking achieved in "federal" manner rather than dictatorship

Approach to Concurrency

Distributed systems are inherently concurrent

- **Separate concurrency requirements and policy from concurrency mechanisms**
  - **concurrency requirements in the Computational viewpoint**
  - **concurrency mechanisms in the Engineering viewpoint**
- **State computational requirements in declarative computational specifications**

Where parallelism is possible, and where synchronization is required

Concurrency is an application-level concern, because non-concurrent interfaces don’t scale with multiple clients

- **Provide tools to automate the use of concurrency**
  - **tools that extend their programming languages, for applications developments**
  - **tools to map computational specifications to engineering mechanisms**

Approach to Diversity

diff h/w & comm's sys designed diff'ly - problem for dist'd sys, but not desirable for all sys to be same bcs E benefits in diversity & spec'n - challenge: make diversity work harmoniously - need flexibility to cope w inevitable variation among cpts of dist'd sys.

- **Assume diversity (also called *heterogeneity*)**
- **Look for unnecessary diversity**
  - **Use abstraction techniques to identify what is in common**
  - **Discard undesirable diversity: it is wasteful**
  - **Retain desirable diversity: special solutions for special circumstances**

diff & incompat h/w & s/w data rep'ns, diff comms protocols - define level above all this stuff that hides it below. object-based comp'l model: one sys cannot directly manipulate data in another - remote info represented abstractly - avail svcs characterised w/o knowing how internal data is represented or manipulated

- **Manipulate encapsulated data via service interfaces**
- **Do not assume a common data representation**
  - **To share service interfaces, propagate only references to interfaces**

Approach to Scaling

Networks change over time: they grow, merge, diverge

- **Assume diversity of scale**
  - **Allow for it by building expansion capability into the architecture**
- **Provide extensible naming and trading facilities**
- **Federate through negotiable, cooperating, remote services**

not possible to assume global resource pool that is always known about -concept of trading - to find out about interface of desired svc type -the trading svc maintains changing knowledge abt available services

- **Do not assume global knowledge**

Of mutable state

also facilitated by data being manipulated where it is held - requires global negotiation of service agreements, but avoids assumptions of global knowledge

Approach to Separation

Because interacting comp'l entities are physically separate, a general comp'l model for interworking is needed

- **Assume all services are remote**
  - **co-location is a special case that can be optimized**
- **Require each service to be entirely responsible for operations on its encapsulated data**
- **Perform all interactions with services via instances of interfaces**
  - **the interfaces must be pre-defined**

ensure state & data of ea. remote svc can only be manip'd indirectly via inter'n w.>=1 i/fs supported & made avail by svc

- **Allow propagation of interface-references as the means of acquiring access to services**
- **Name and report all detected interaction faults and failures**

Communications can fail. Partial failure is possible-more things can go wrong than with loc proc call -can never be as transparent-to have equiv failure semantics to nondist'd sys, failures reported & processed, therefore svc interactions require multiple outcomes, each of which may comprose multiple results -> terminations

ANSA is an international standard - ODP

- The ANSA approach is now standardized as the *Basic Reference Model for Open Distributed Processing (ODP)*
  - by cooperation with the major international standardization bodies: ISO, IEC, ITU
- This is to be issued as ISO/IEC 10746, ITU-TS (CCITT) X.901-X.904
- ODP is very closely aligned with ANSA concepts and terminology
  - it has the same 5 projections/viewpoints
- Like ANSA, ODP is a framework into which you slot existing standards
- ODP is broader than ANSA in some respects...
  - for example, it tackles the issue of conformance
- ... but ANSA goes into more detail

ANSA - Summary

- ANSA is a framework for distributed systems
  - systems are viewed as coordinated sets of subsystems
  - ANSA takes an integrated approach
- It has 5 projections: Enterprise, Information, Computational, Engineering, Technology
  - each has its own model
- Services are encapsulated
- Transparency mechanisms simplify the construction process

ANSA - More information

- **For more on ANSA Fundamentals, see *An Overview of ANSA* (AR.000.00)**

- **For more on ANSA projections:**

  - **the Computational projection, see *The ANSA Computational Model* (AR.001.01)**

  - **the Enterprise projection, see *Architecture and Design Frameworks* (TR.38.00)**

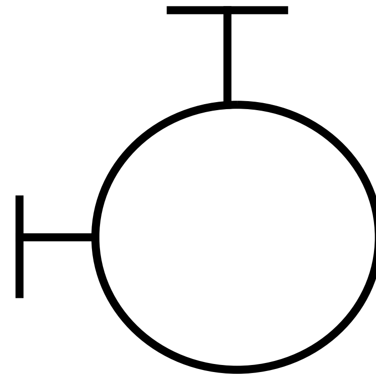
The description of the other projections is scattered across the other ARs and TRs

- **For more on transparency mechanisms, see *The Challenge of ODP* (TR.033.02)**

ANSA and ODP - Extra information

- **The ODP documentation is organized rather differently from ANSA**
- **The *Basic Reference Model for Open Distributed Processing* standard has 4 parts:**
  - **Part 1: Overview**
  - **Part 2: Descriptive model**
  - **Part 3: Prescriptive model**
  - **Part 4: Architectural semantics, specification techniques, and formalisms**
- **By contrast, the ANSA Architectural Reports describe each projection separately**

The ANSA Computational Model



In this session

- Explain the concepts of ANSA Computational Model
- Explain the rules that are imposed, and why they are important
- Explain how objects and interfaces are used in ANSA

What is the ANSA Computational Model?

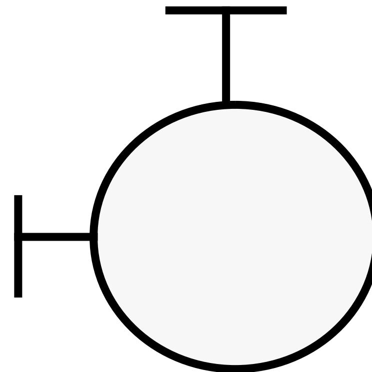
- **It is a model for specifying service provision and use**
  - **Objects may provide multiple services to other objects**
  - **Objects may use multiple services from other objects**
- **Objects and their services may be created and destroyed dynamically**
- **The ability to use a service may be passed from one object to another**
- **... The ANSA Computational Model is based on interacting *distributed objects***
  - **It makes no assumptions about where the objects are distributed**

What is meant by ‘object’?

- **Nowadays all software is supposed to be ‘object-oriented’**
  - **OOA/OOD (analysis and design methods)**
  - **OOL (programming languages)**
  - **OODBMS (databases)**
  - **... and many more**
- **All these have one key concept in common...**

Encapsulation - the key concept

- **Objects are encapsulated**
  - **every object provides a service via interfaces**
  - **the interface is public; the implementation is private and hidden**
  - **encapsulation forms a boundary; the only access to the object is via its interfaces**

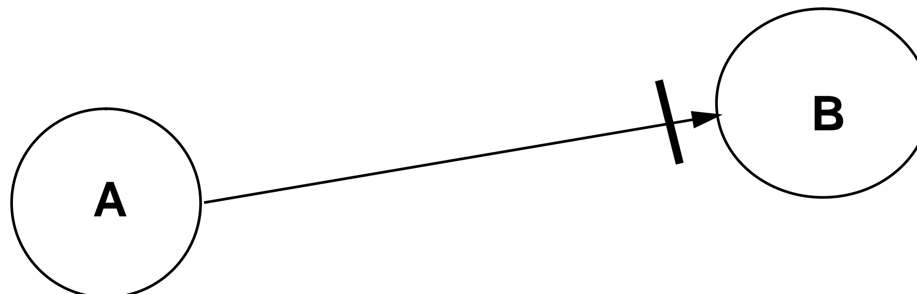


Encapsulation hides data representation

- **The boundary hides the data representation used by the object**
    - **Diversity: different objects use different representations**
    - **In a distributed system, different representations are used in different places**
    - **One object cannot manipulate another's data (only via the interface)**
    - **Data representations cannot be transferred between objects**

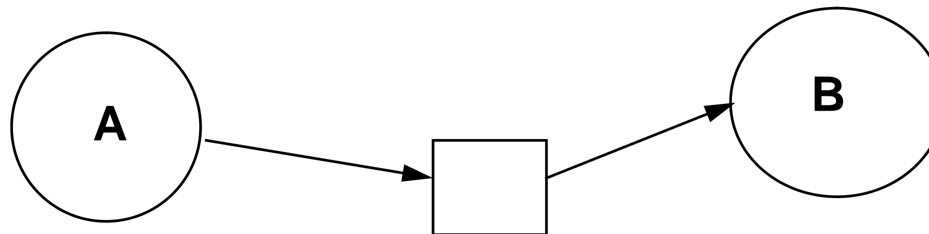
Encapsulation for distributed objects

- **We cannot assume anything about the location of distributed objects**
  - **Objects A and B could be on the same machine, or in different countries**

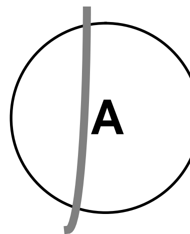


Encapsulation rules

- **Objects cannot share state information directly (only pass it via interfaces)**
  - ***this is not allowed***

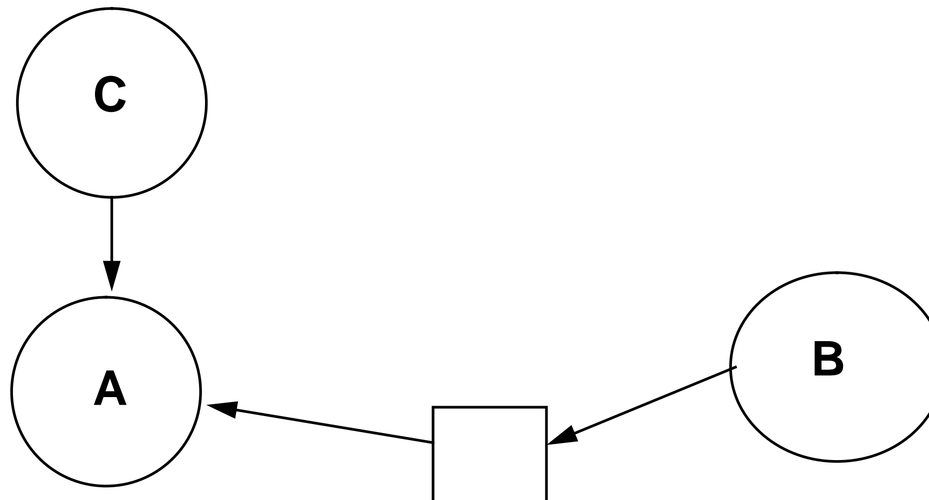


- **One object cannot be distributed in pieces; it must be entirely in one place**
  - ***this is not allowed***



Is shared state needed?

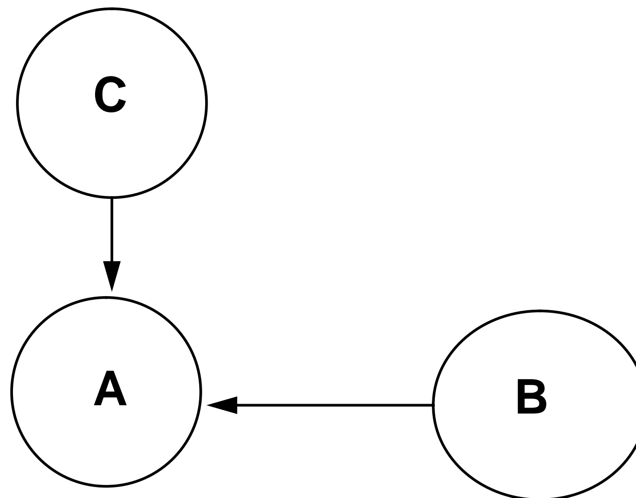
- **Suppose:**
  - objects A and B need to exchange some information with each other...
  - ... but A needs to exchange different information with C
  - the information flows must be separated; a “back door” between A and B?
  - *this is not allowed*



Multiple interfaces instead

- **No back door is needed; use multiple interfaces instead**

  - **A can have two interfaces**

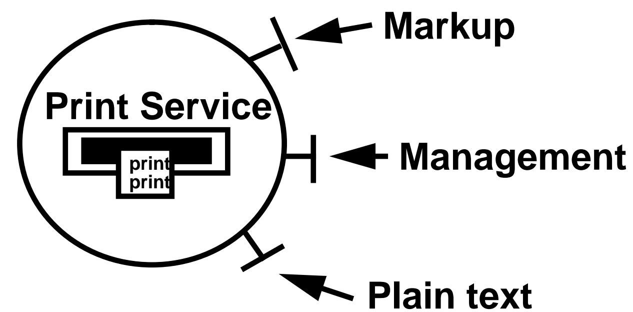


Objects and interfaces - an example

- Multiple interfaces allow flexible configurations...
- ... Consider a service for printing documents
- It must print two kinds of document
  - plain text documents
  - documents in some markup language, for example PostScript (TM)
- It must also have a management interface for
  - starting and stopping a print queue
  - delete print jobs
  - other administrative tasks
- Here are four possible different configurations

Configuration for a print service - (1)

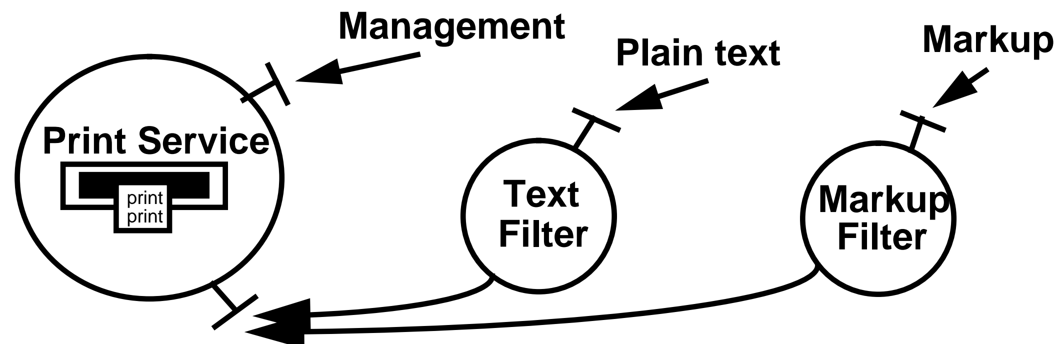
- **One object with plain text, markup, and management interfaces**



Configuration for a print service - (2)

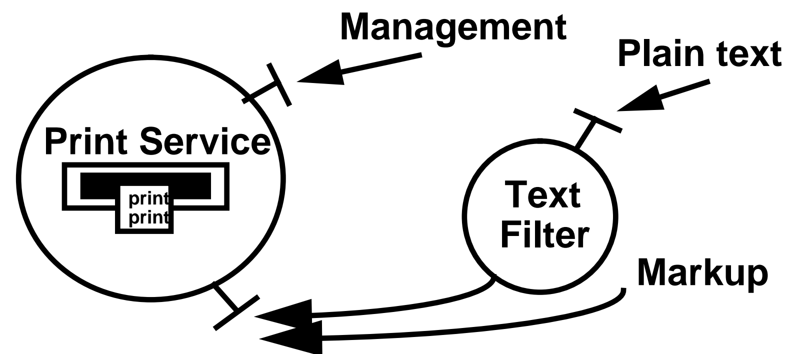
- **Printer object with management and control interfaces**

- **Filter objects use control interface**



Configuration for a print service - (3)

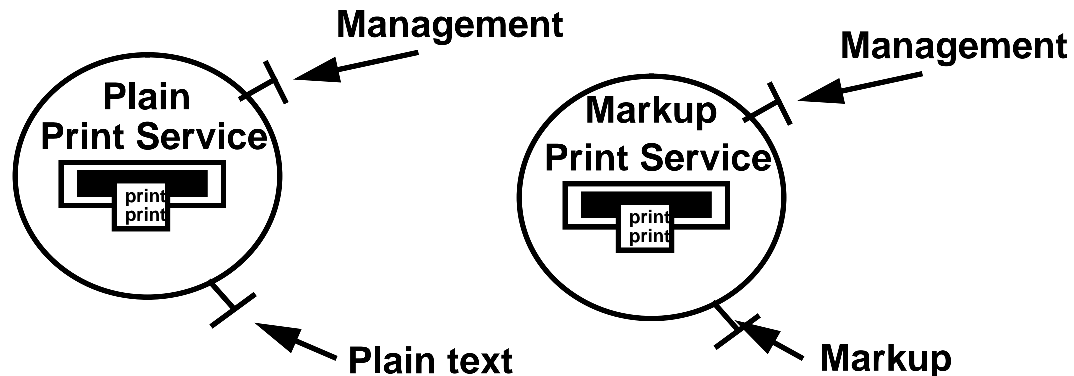
- **Printer supports markup language via the control interface**
- **Plain text converted to markup language for printing**



ANSA

Configuration for a print service -(4)

- **Two separate printers for plain text, and markup language**
- **Two separate objects**



Objects and Interfaces - summary so far

- Objects encapsulate services
  - encapsulating the state
- Objects interact only via interfaces
  - the point-of-access to the service
- Objects can have multiple interfaces
- Objects have the full responsibility for providing the specified service

Computational Model and Object-Oriented Programming

- **ANSA distinguishes between encapsulation and service provision**
  - **Objects for encapsulation**
  - **Interfaces for service provision**
- **Object-oriented programming languages use one construct for both encapsulation and service provision**
  - **for example, classes**

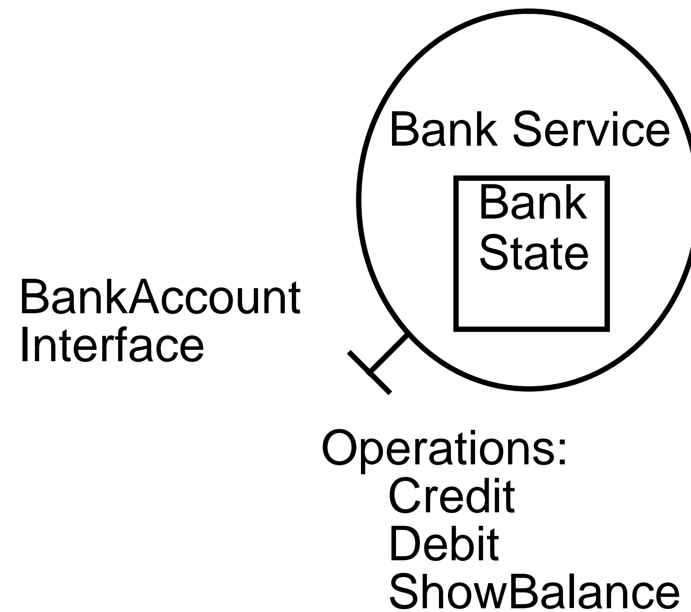
Operations - the actions in an interface

- An interface defines one or more *operations*
- Operations are the actions that the user of the interface can invoke
  - Like "methods" in object-oriented languages
- Because of encapsulation, the defined operations are only way to act on the object
  - No “back door”

Operations - a Simple Bank example

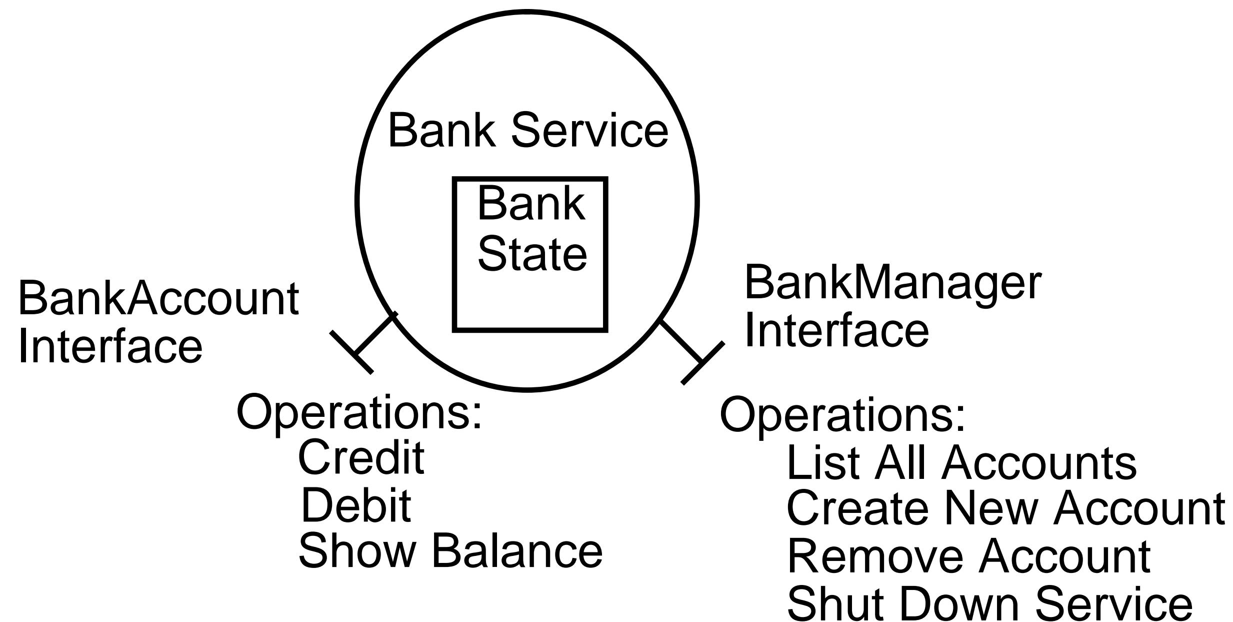
- Consider a bank account service...
- Each bank account has state
  - account balance
  - name of customer
  - ... and others
- Each bank account has actions that affect this state
  - Credit: depositing money, increases the balance
  - Debit: writing a cheque, decreases the balance
  - Show: enquires the account balance
- These actions are the only way of manipulating the bank account; these actions are the *operations*

Simple Bank - Service, interface, and operations



Simple Bank - Operations in multiple interfaces

- **A service can have more than one interface**
- **Each interface has its own operations**



Operations and Arguments

- **Each operation has a name, input arguments, and output arguments**
  - **for example, the Credit operation returns the new balance**

    Credit (amount : Integer) : (newBalance : Integer)
  - **amount is a input argument; newBalance is an output argument (result)**
- **Each operation can have multiple input and output arguments**
  - **this example only has one of each**

Operations and Terminations

- **An operation can have more than one possible outcome**
- **Consider operations on the BankAccount:**

    Credit (amount : Integer) : (newBalance : Integer)

    Debit (amount : Integer) : (newBalance : Integer)

    - ... but the balance may be insufficient!

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

- **There is one Debit operation with two possible outcomes; two separate *terminations***

Named Terminations

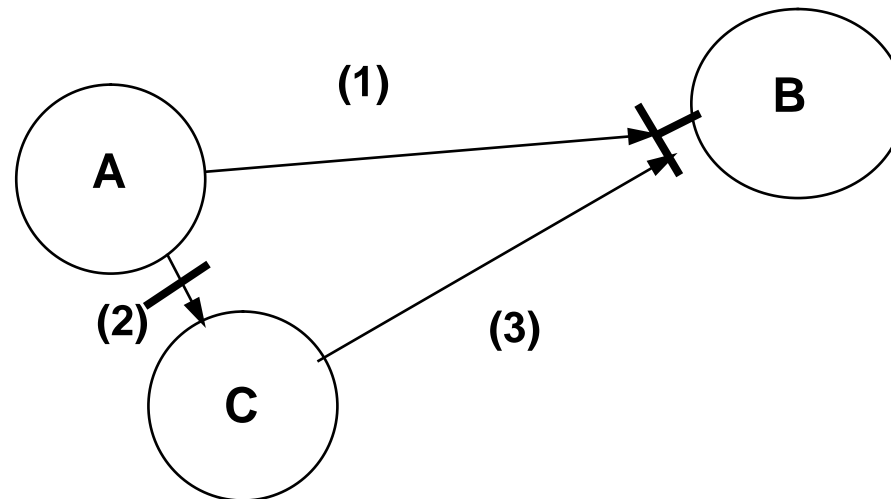
- The InsufficientFunds outcome is called a *Named Termination*
  - the normal outcome is an unnamed (anonymous) termination
  - one termination of an operation may be anonymous
- Each termination may return multiple results
- The invoker of an operation distinguishes between the different terminations by their names

Operations, Arguments, and Terminations - summary

- An *operation* is the unit of interaction. It has:
  - a name
  - a signature (list of input arguments)
  - a set of *terminations* (possible outcomes)
- A termination has:
  - a name
  - a signature (list of output arguments)
- Terminations make operations more powerful than ‘functions’ or ‘methods’

Using Interfaces

- Object A is using one of object B's interfaces
- Suppose it needs to tell C to use the same interface



- It must be possible to pass a reference to that interface between A and C

Interface References

- **Interfaces are identified by *interface references (if-refs)*; these can be passed as arguments to other interfaces**
  - **for example, when A passes B's reference to C, it calls...**

    C_op (interface: if-ref) : ()

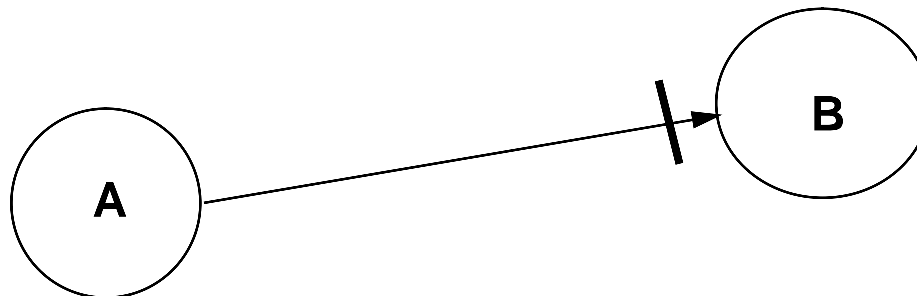
  - **... with B's interface-reference as the argument**

- **Only the reference is passed, not the interface...**

- **... In fact, all operations are invoked via interface references**

The right interface?

- **Suppose object A has an interface-reference for an interface of B...**
- **... how does object A know that B has the right kind of interface?**
  - **it can't inspect the object itself**
  - **objects A and B could be on the same machine, or in different countries**



- **It can tell, because interfaces have an *interface type***

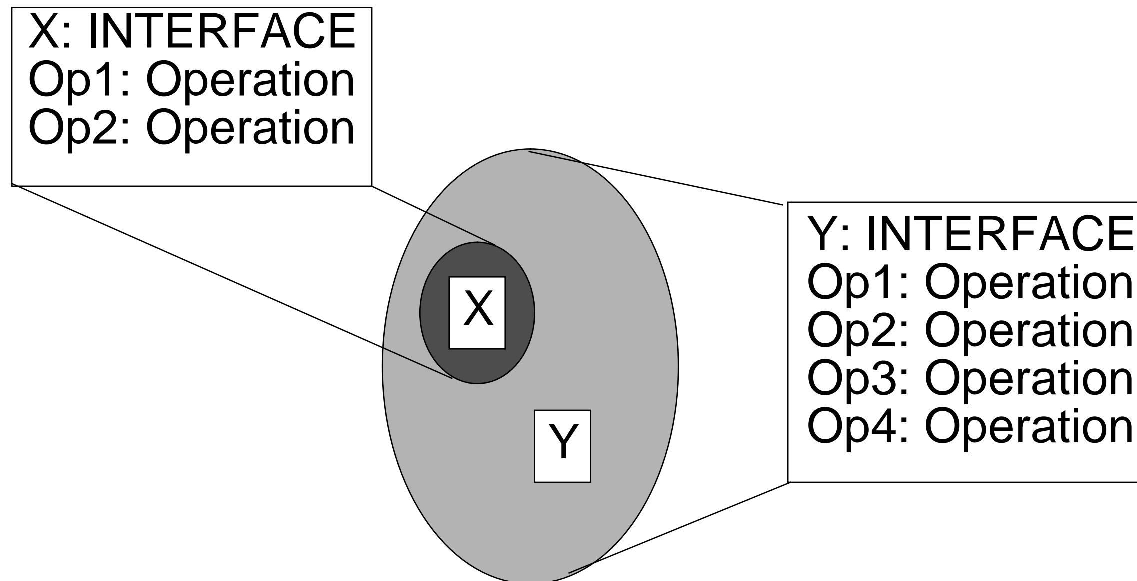
Interface types

- **The interface type simply consists of the set of its operations**
  - **All their signatures and terminations**
- **Object A can use an interface of object B, provided that the interface expected by A *conforms* with that provided by B**
- **This means that object A must state what interface type it expects**

ANSA

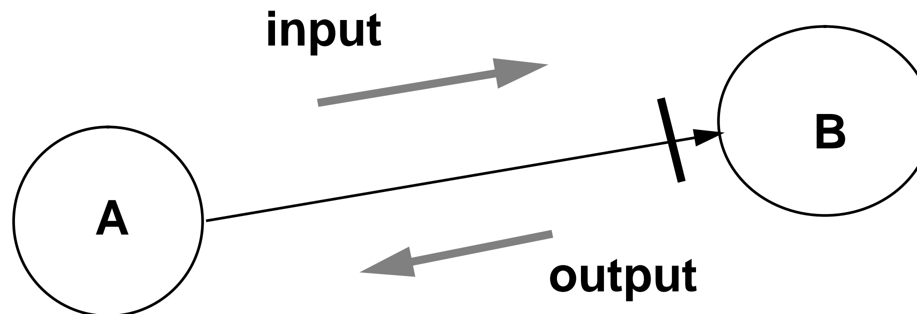
Type Conformance

- **Conforming interface types do not have to be identical...**
- **...Interface type Y *conforms to* interface type X**



Type Conformance Has Two Sides

- **To conform, Object B must provide at least the operations expected by A...**



- **...but also, Object A must handle at least the terminations given by B**
  - **the same idea, but the ‘at least’ rule is the other way round for terminations**
- **To conform, both of these must be checked...**
  - **B must not receive unknown operations: A must not receive unknown terminations**
- **.... this is the idea of a two-sided *contract* between A and B**

When is type conformance checked?

- Before the interface is used...
  - ... it may be at compile time (statically)
  - ... it may be at run time (dynamically)
  - or at some link, build, or installation stage between these extremes
- Dynamic type checking means that the type descriptions must be available at run time
- Dynamic type checking is done during *binding*

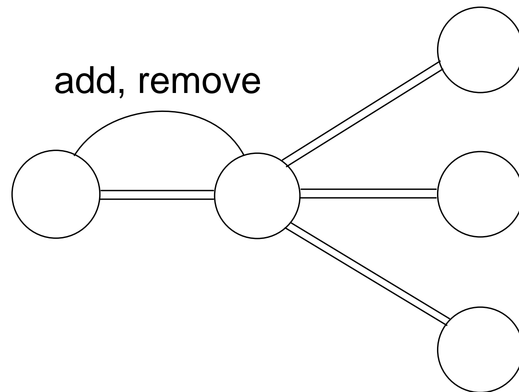
Bindings

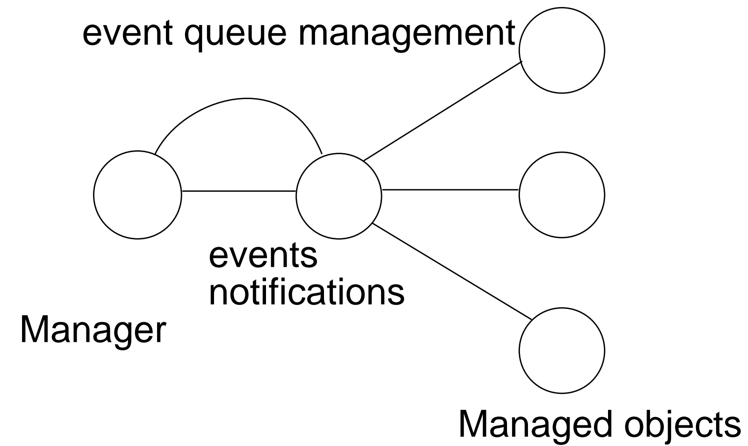
- When a object prepares to use an interface, a *binding* is established
- Type conformance is checked
- Binding may be implicit or explicit
  - implicit binding allocates resources automatically
  - explicit binding allows control over resource allocation
- Bindings can involve more than two interfaces

Binding objects

- A binding is itself a object with its own control interface for
  - adding/removing interfaces
  - stopping/starting information flows
  - changing quality of service
  - monitoring events
- The binding object does the necessary splitting, joining, filtering and control

Examples of bindings





Conferencing

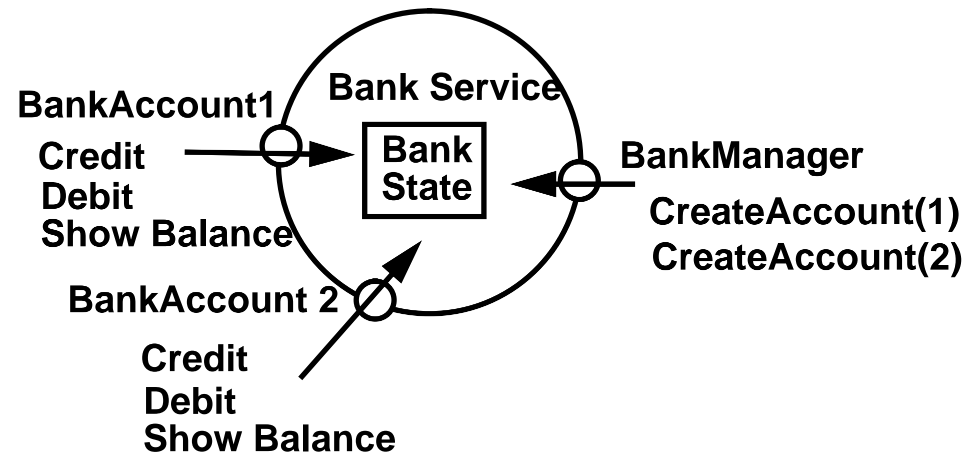
Object Management

Interface types and instances

- **An object can have more than one interface of the same type**
  - **these will have distinct interface-references**
  - **an interface-reference refers to an *instance* of an interface**
- **An instance of an interface comprises**
  - **an interface type**
  - **state information (data)**
- **For example...**
  - **all BankAccounts have the same interface type**
  - **state information (account number, name balance) is what distinguishes one instance from another**
- **Interface *instance constructors* generate new interface instances at run time**

Instances of the same interface type

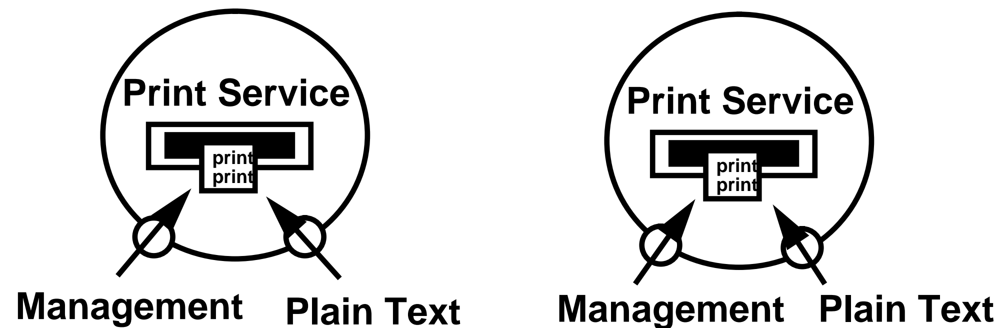
- **Two instances of BankAccount interface**



ANSA

Instances of the same object

- Similarly, objects can be created by an *object instance constructor*
- Two instances of a Print Service object

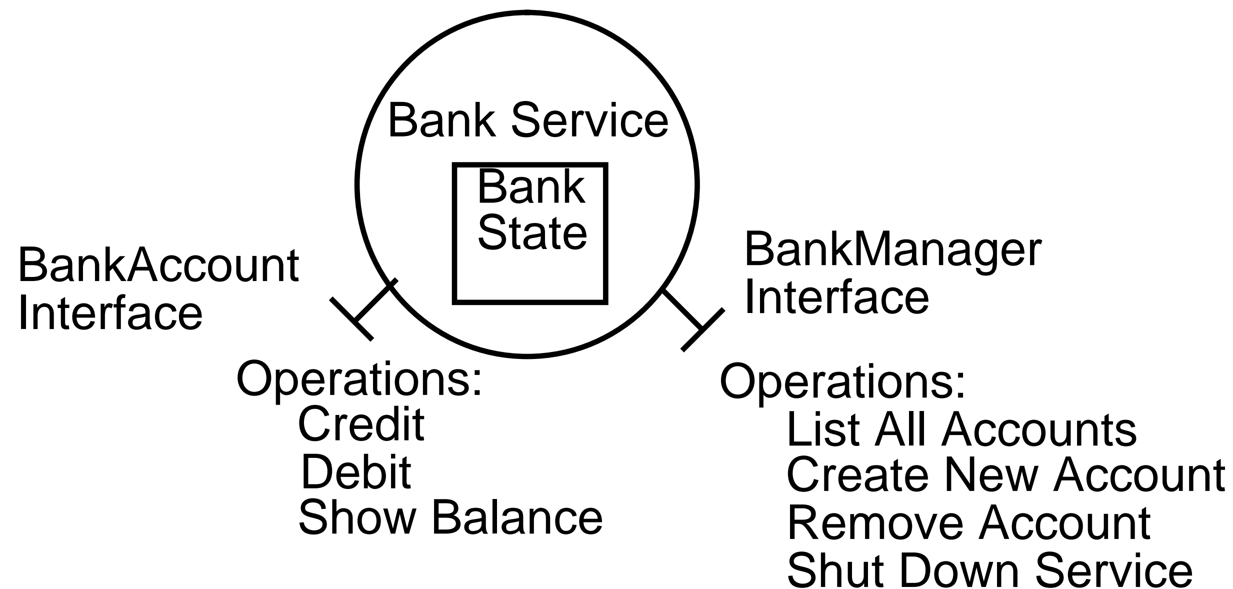


Object instances or interface instances?

- **To keep state information separate, use interface instances or object instances**
  - **they may be in the same object, but need not be**
- **To share state information, use interface instances**
  - **they must be in the *same* object**
- **To allow distribution, use object instances**
- **To enforce encapsulation boundaries, use object instances**
  - **for security**
  - **for reliability (fault containment)**
  - **... and other quality-of-service characteristics**

Interface Concurrency

- **What happens when the BankAccount and Bank Manager access the same account at the same time?**



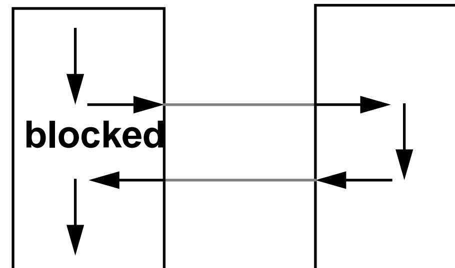
- **This topic is covered by ANSA Concurrency Model**

Operation Invocation

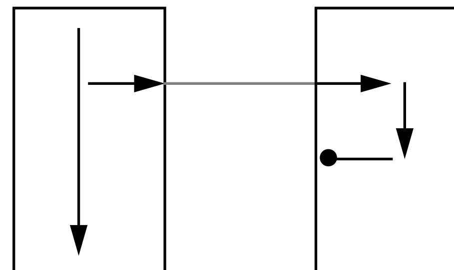
- **Operations are invoked giving**
  - **the interface reference**
  - **the operation name**
  - **the operation arguments**
- **Two styles of invocation:**
  - ***Interrogation*: synchronous; return one of a set of named terminations with results**
  - ***Announcement*: asynchronous; do not return**

Invocations

- **Interrogation -** synchronous:



- **Announcement -** asynchronous:



Objects - Summary

- **Objects enforce strict encapsulation**
  - **state cannot be shared between objects**
- **Only objects can be re-configured or migrated (*part* of an object cannot be)**
- **Objects may have multiple interfaces**
- **Objects may have concurrency**

Summary

- **Key concepts**
  - **objects**
  - **interfaces**
  - **operations**

- **For more information:**
  - **for object-oriented concepts, see *Distributing Objects* (TR.018.01)**
  - **for trading, see *The ANSA Model for Trading and Federation* (AR.005.00)**
  - **for a formal description, see *The ANSA Computational Model* (AR.001.01)**

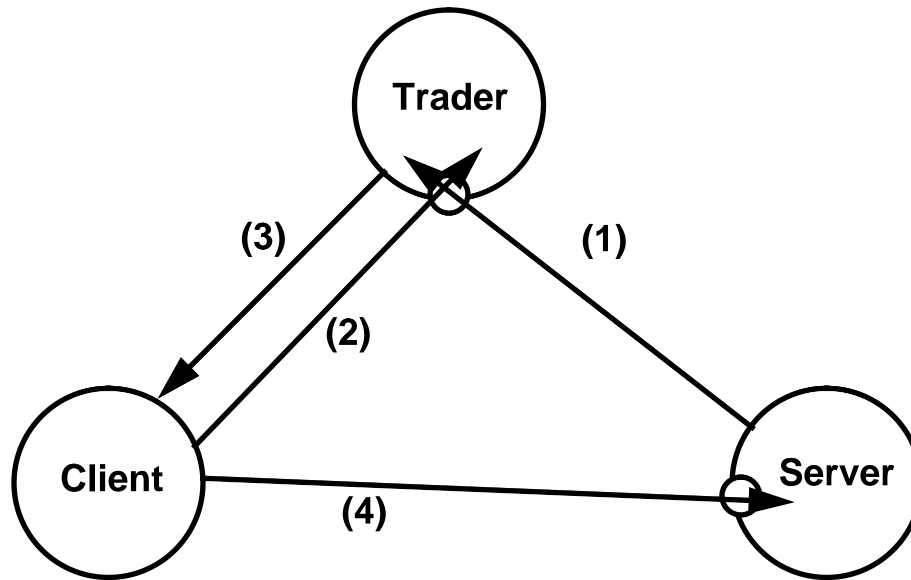
Computational Model - Extra information on Streams

- We have so assumed that actions only happen when invoked
- This model does not does allow for continuous information flows (streams), for example:
  - speech, in telephony and voice processing: an audio stream
  - video, in multimedia: a video stream
  - sensor data, in telemetry: a data stream
- The model has been extended to allow for streams
  - directions of flow must match for stream bindings

Computational Model - Extra information on Bindings

- *Implicit* binding can be used only for operational interfaces
  - Resources will be automatically allocated
- *Explicit* binding can be for all interfaces
  - For explicit control over resources
  - To form groups
  - To bind streams
- Explicit binding suits telecommunications and system management needs

The ANSA Trading and Federation Model

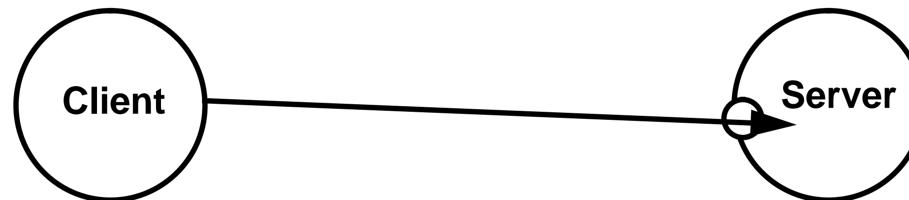


In this session

- **Explain the ANSA Trading Model**
  - **the roles of client, server, and trader**
- **Explain federated trading**

Client and Server are Roles, not Technology Types

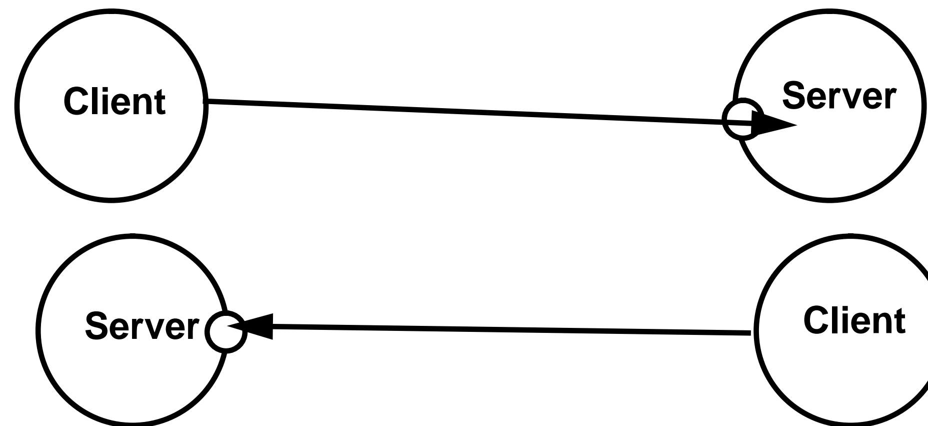
- **ANSA uses the terms *client* and *server* in a particular sense...**
- **... Client and Server are *roles* in a particular interaction between objects**



- **Nothing is implied about the technology that supports them**
  - **the client could be a mainframe...**
  - **...the server could be a PC**
- **Technology is not part of the ANSA Computational Model**

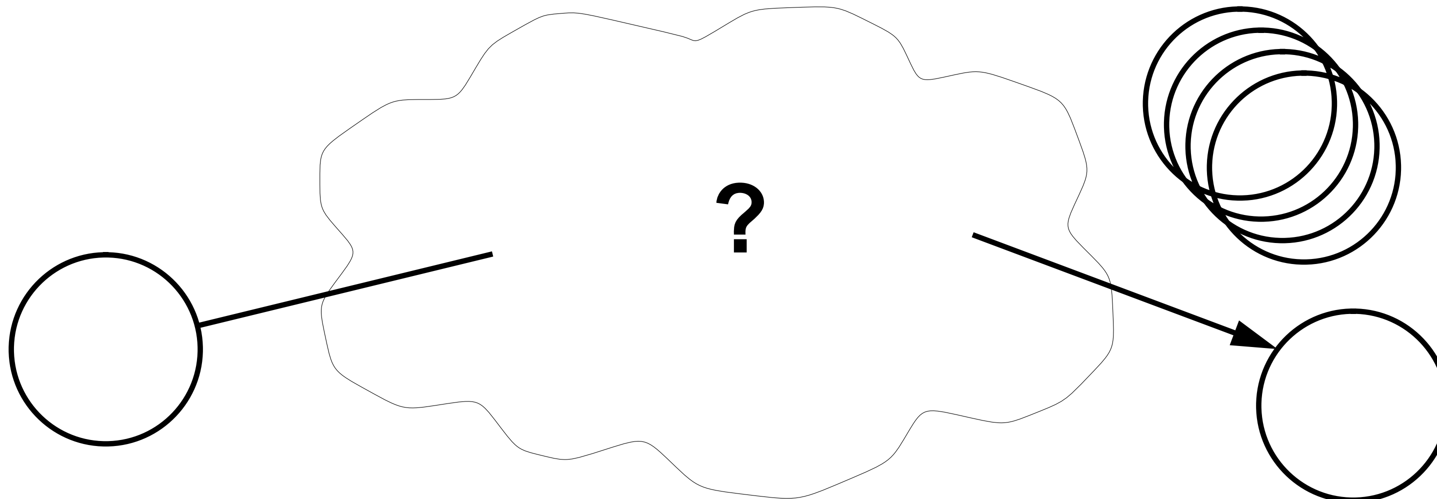
Client and Server Roles

- **The roles are determined by the objects themselves**



ANSA

The need for Trading



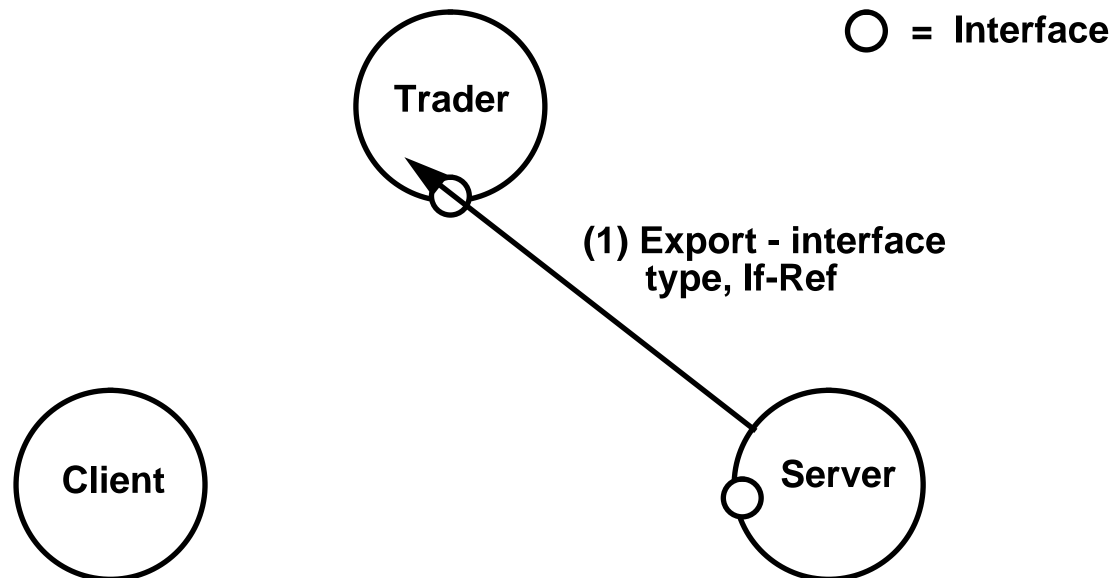
- **How can clients find servers that provide the services that they need?**
  - **in the future, there will be millions of interconnected servers around the world**
  - **clients will come and go dynamically**
  - **servers will come and go dynamically**

Trading - Basic needs

- **Server must state what it provides**
  - it must export a service offer
- **Client must state what it requires**
  - it must import a service request
- **Trading must find a service offer that matches the request**
  - there may be many such offers...
  - ...there may be none

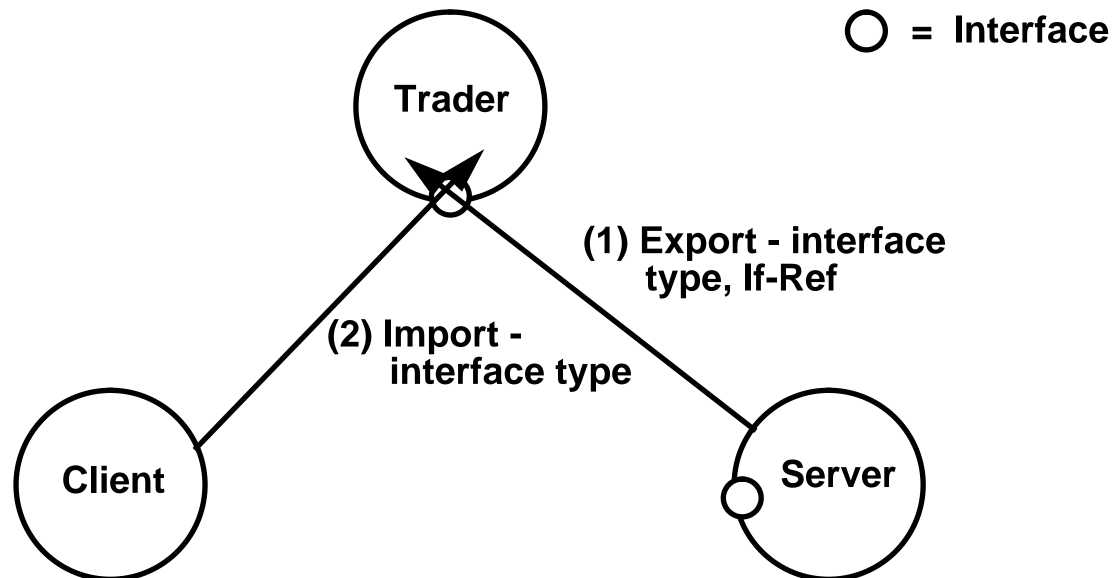
Steps in Trading - (1)

- (1) Server exports a service offer to the Trader



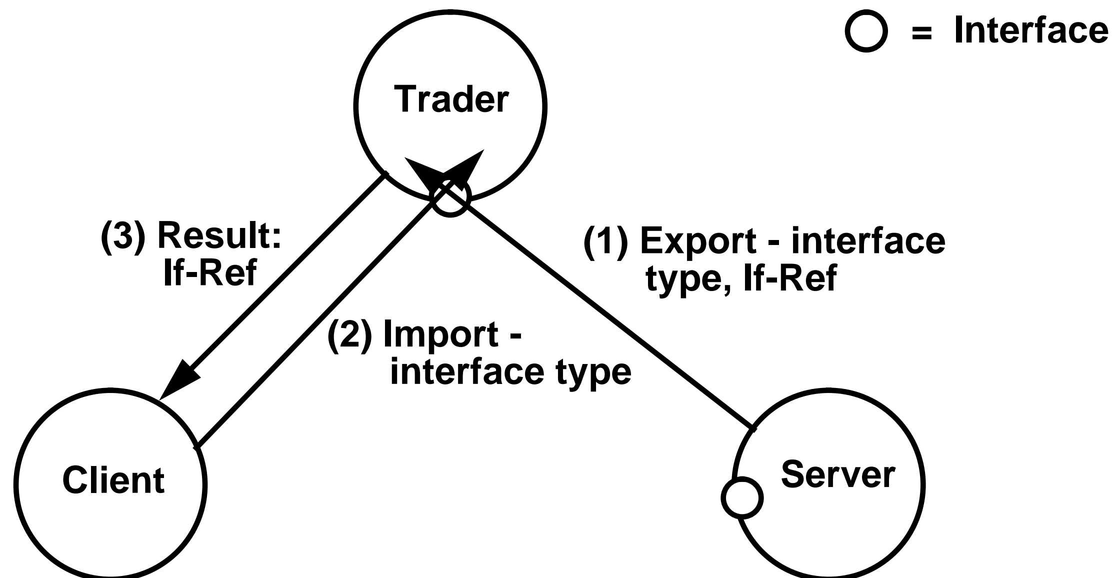
Steps in Trading - (2)

- (2) Client imports a service request from the Trader



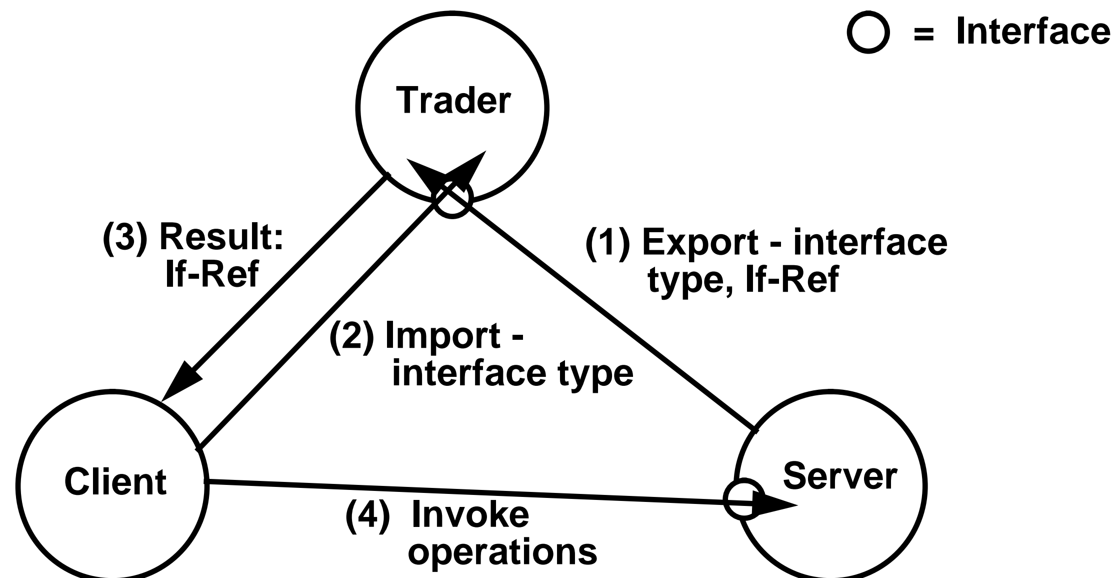
Steps in Trading - (3)

- (3) Trader returns a matching service offer to the client
  - it returns the interface-reference given by the server



Steps in Trading - (4)

- **(4) Client uses interface-reference to invoke the server's operations**
  - **Trader takes no further part in the interaction**
  - **an Interface-reference from the Trader is invoked just like any other**



Stating service requirements

- There are two potential ways to identify something
  - by naming it
  - by describing it
- Trading cannot rely on clients being able to name servers
  - the server may not even exist when the client was created
- Trading works by matching descriptions provided by clients and servers

Matching requests with offers - type conformance

- **How does Trading decide whether a client request matches a server offer?**
  - **it uses the interface *type conformance* concept from the Computational Model**
- **If the client request and server offer interface types do not conform, they are incompatible, and cannot match**

Matching requests with offers - other criteria

- **Type conformance is not the only criterion for matching**

  - **clients may wish to choose servers with given Quality of Service *properties***

  - **servers may wish to restrict the offer to a particular *trading context*, for administrative control**

Trading properties

- There may be many servers offering the same service...
  - conforming to the same type
- ... but with different quality-of-service characteristics, including:
  - Response time
  - Storage capacity
  - Availability
  - Cost
- ... these are sometimes called ‘non-functional characteristics’
- Trading represents these as *properties*
- Trading uses property constraints supplied by the server when matching
- Properties can be used for any similar reason
  - by mutual agreement and convention

Trading Contexts

- Each object has access to a *trading context* containing
  - service offers; type, properties, interface reference
  - links to other contexts; name, properties
- Trading contexts allow service offers to be grouped
  - to allow delegation of administration
- A service offer can be tied to an export policy
  - to monitor imports
  - to allocate resources

Different trading contexts for different needs

- A context may be optimized for
  - speed of look-up
  - number of offers stored
  - accuracy of offers
  - dependability
- There is no predefined naming structure for contexts - allows federation

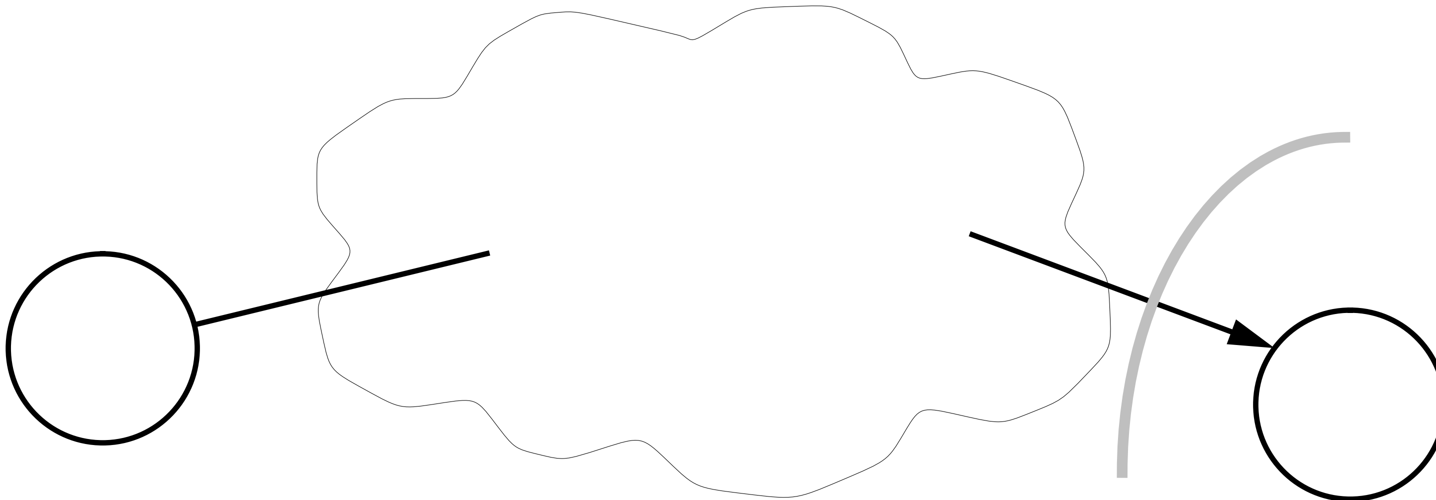
Trading in large distributed systems

- **Because there will be millions of servers in the world:**
  - **there will be many Traders providing the Trading service**
  - **... Traders must be interconnected**
  - **... the Trading service must itself be distributed for scalability**
  - **... and cannot be centralized**
  - **this is called *federated trading***
- **And also because organizations will wish to control their own Traders:**
  - **to determine who sees which services**

The need for federation

- Organizations want to offer electronic services to other organizations
  - services provided by applications in the usual way...
  - ...services provided by distributed objects
- Organizations want to make money
  - to *sell* services provided by distributed objects
- Organizations need to keep control of their objects
  - control over their security, for instance

Federation entails boundaries



- **Boundaries enforce control**
- **Distributed objects are ideal**
  - **their encapsulation provides a boundary**

Kinds of boundaries

- There may be many kinds of boundaries
  - Administrative
  - Judicial
  - Political
  - Technological
- We want these to be transparent to applications

Different policies for different organizations

- Federation raises policy issues including
  - Authority
  - Billing
  - Security
- Organizations that sell electronic services must be able to enforce their policies
  - they cannot rely on trust

Federated Trading

- **Distributed objects in the ANSA Computational Model already satisfy most of the needs of federation...**

- **... but there are new issues**

  - **federation of naming contexts**

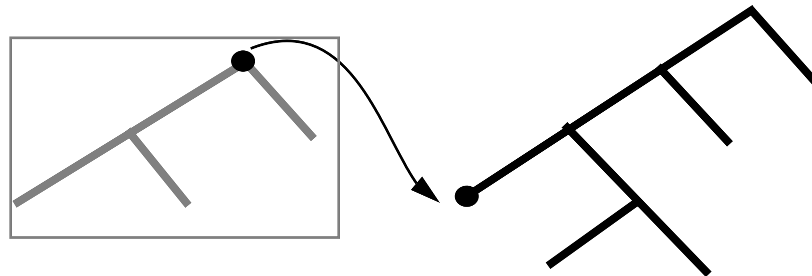
  - **technology boundaries**

  - **security**

Federation of trading contexts

- Two organizations are connected: how does each Trader find out which services are offered by the other?

- By federation of trading contexts:



- Organizations can control the context points at which federation is allowed

- Names are therefore context-relative

Summary

- The key concept is Trading: the activity of choosing a service offer that matches the service requirement
- The Trading service is provided just 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

Trading and Federation - more information

- For a general discussion trading and federation, see *The ANSA Model for Trading and Federation* (AR.005.00)

- For naming issues, see *The ANSA Naming Model* (AR.003.01)

- For security issues, see *A Framework for Federating Secure Systems* (AR.008.00)

- For a discussion of Quality of Service issues in multimedia applications, see *Integrating Multimedia into the ANSA Architecture* (TR.028.00)

Trading - Work in progress

- This session has described ANSA Trading as in *ANSA Model for Trading and Federation* (AR.005.00)

- ANSA Phase III is working on an Extended Trading Model with a more sophisticated treatment of federation

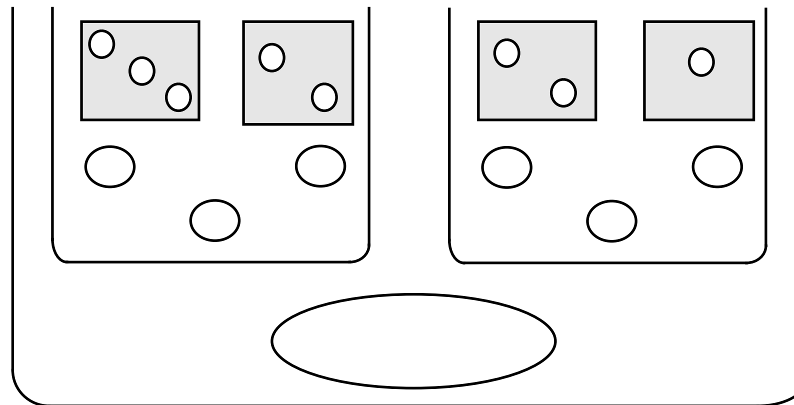
    - This work is available to sponsors in *Data Management for an Enhanced Trader* (APM.1162.01)

- Another project is extending Trader Security

Trading - Extra questions

- **If Trading is a service, how does a client find the Trading service to start with?**
  - **Yes, there has to be a way to bootstrap the client. This can be done by building a Trader interface-reference into the infrastructure, by broadcasting, or some other special means. This is really an Engineering issue**
- **How does Trading understand what the service offers mean?**
  - **It doesn't need to. It just needs to decide whether they match service requests**

The ANSA Engineering Model



Speaker Notes

In this session

- **Explain the concepts in the ANSA Engineering Model**

There are more new concepts than you might think

- **Explain the various types of transparency mechanisms**

- **Compare the ANSA Engineering and Computational Models**

What is the Engineering Model for?

- **To allow trade-offs**
  - **flexibility versus performance**
  - **time versus space**
  - **... and many others**
- **But without affecting the Computational Model**

A controversial though; engineering is the kind of work that a 1990’s systems programmer does.

An engineer is someone who can build for $5 what anyone else can build for $50.

Engineering is infrastructure

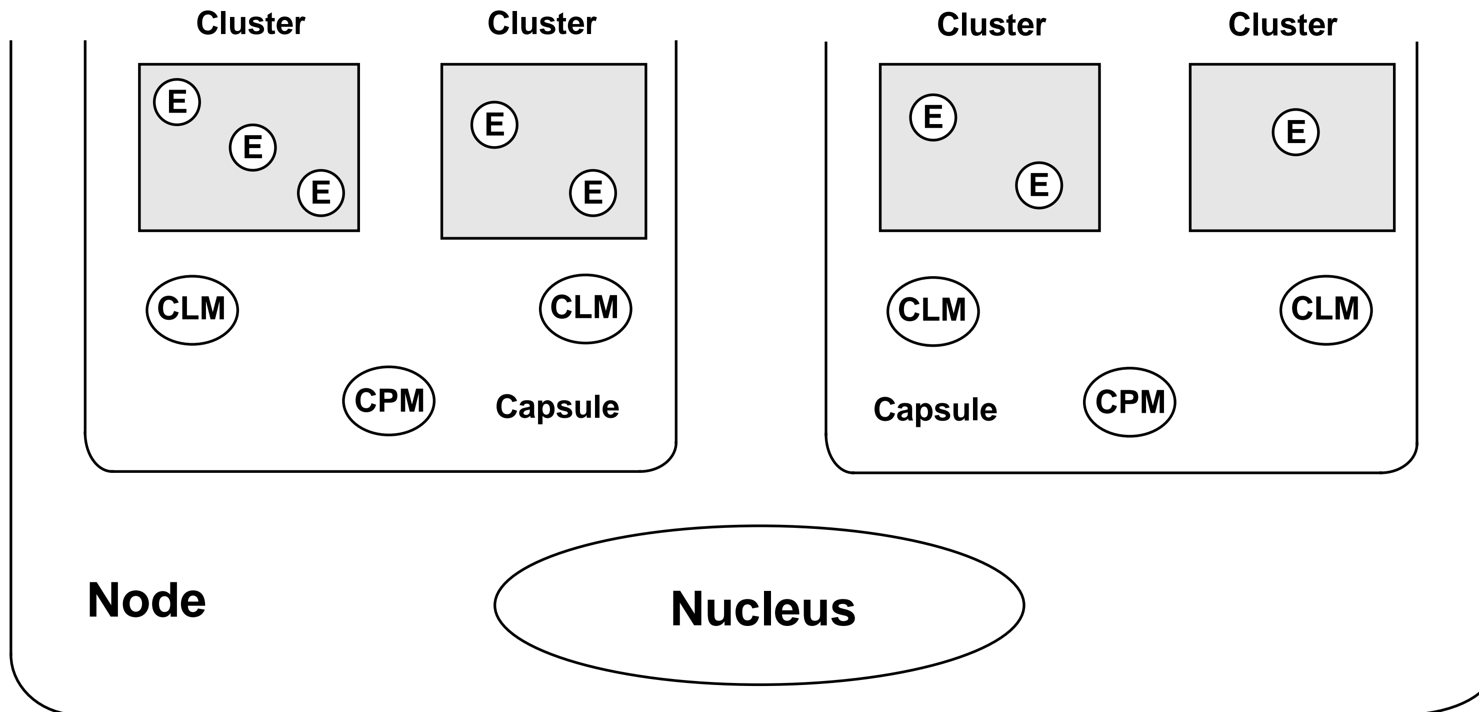
- **Provides infrastructure for encapsulation**
  - ***capsules* for resource allocation and protection**

Roughly speaking, an address space

  - ***clusters* for collective activation, deactivation, and migration**
  - ***nodes* for network addressing**
- **Provides *channels* for communication**
  - **point-to-point (simple client-server)**
  - **point-to-multipoint**
  - **streams**
- **Provides *concurrency* mechanisms**
  - ***threads* for concurrent execution**
- **Provides *transparency* mechanisms**

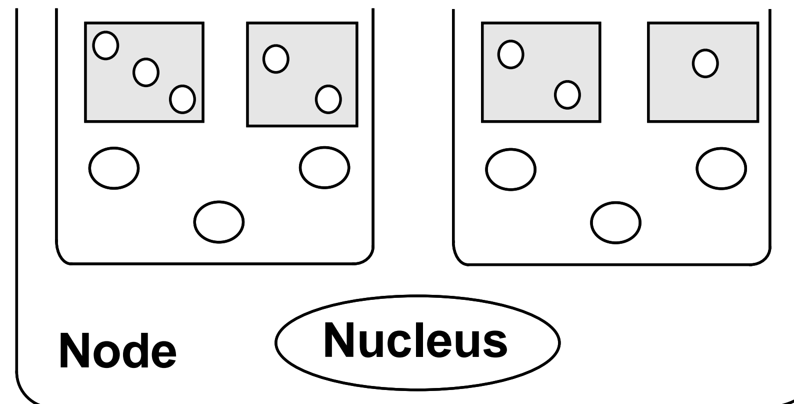
Note that all this is still logically above the level of the operating system

Engineering Encapsulation Infrastructure



ANSA

Node and Nucleus

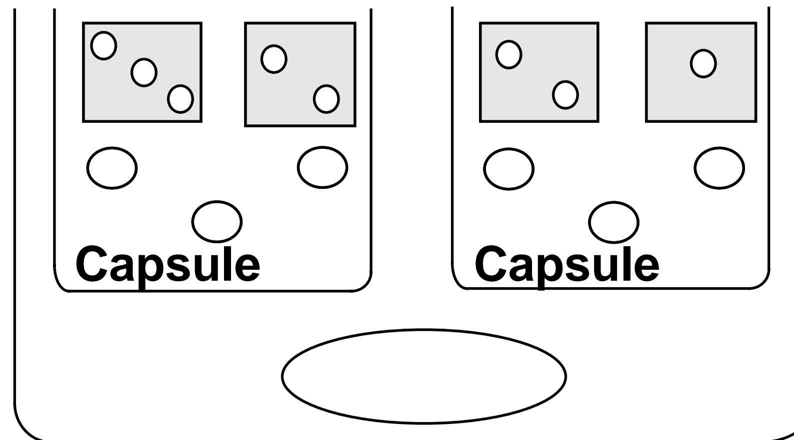


- **A node is the unit of network addressing**

Obviously the networking software (not in ANSA) has to know where to route packets. How this is done is not an issue for the Engineering Model; ultimately it just has to be part of an interface reference

- **The nucleus handles communications between nodes and capsules**
  - **it may be part of the operating system (if there is one), or a layer on top of it**

Capsules



- **Objects must be encapsulated**

As described in the ANSA Computational Model

- **every object is contained in a capsule**
- **different capsules separate the objects**

Capsules are single-level; capsules cannot be nested

Capsules can contain more than one object

- **Objects of the same type, or of different types**

... but must have the same security/reliability characteristics; see below

- **Put more than one object in a capsule to**

  - **share resources**

  - **share state information**

But doesn't the Computational Model say that objects can't share state? Yes, but this is the Engineering Model...

Capsules and Shared Memory

- **In a distributed system, two objects cannot usually share memory**
  - **the objects could be distributed anywhere in the world!**
  - **but if particular objects do, their implementation may be more efficient**
  - **... for example, you might keep objects of the same type together in the same capsule**
- **Objects in the *same* capsule can share memory for efficiency - but then there is no protection boundary between them**
  - **resources: they compete for the same resources (physical and virtual memory)**
  - **robustness: object failure within the capsule can cause the whole capsule to fail**
  - **security: there is no security between objects in the same capsule**
- **Sharing memory between *different* capsules is not permitted**

Even if the underlying operating system can support it, it would not make sense

Capsules and Objects

- **So, the Computational and Engineering Models of an object are different:**

  - **in the Computational Model, every object is separately encapsulated; objects cannot share state**

The Computational Model ignores resources, including memory (so therefore shared memory)

  - **in the Engineering Model, objects can share a capsule, and share state**

This is not a contradiction. It is an intentional separation of viewpoints

Capsules and Interfaces

- **Objects in the same capsule can still invoke each other's interfaces**
  - **you are not compelled to use shared memory within a capsule**
- **Object interfaces are invoked in the same way...**
  - **within a capsule**
  - **between two capsules on the same node**
  - **between two nodes**

This is just location transparency

- **...the infrastructure will optimize communications between objects on the same node**

Nodes and Capsules

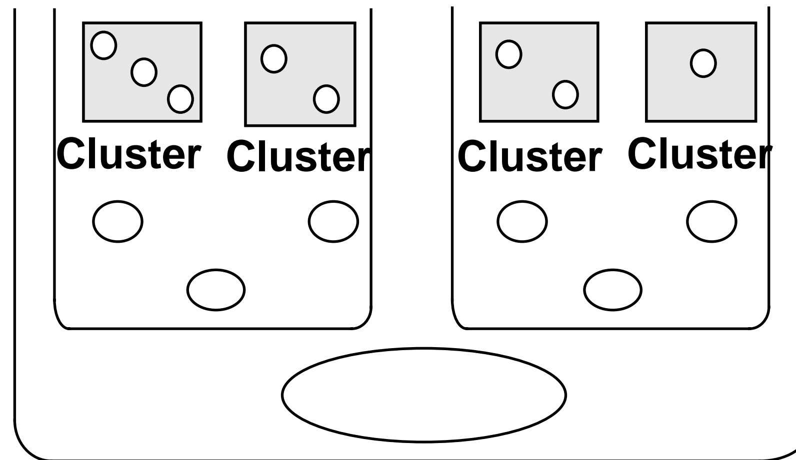
- **Most systems can support more than one capsule per node**

  - **using the multi-tasking and memory protection features of the operating system**

  - **Unix can...**

  - **...DOS can't (only one capsule per node)**

Clusters



- **A cluster is a collection of objects that must be 'kept together'**
  - **if an object moves (*migrates*) between capsules, objects that share memory must move together with that object**
  - **if an object fails, you might want to shut down other objects automatically**
  - **when starting up one object, you might want to start up other objects automatically**

Clusters and Capsules

- **A capsule can support multiple clusters**

- **A Cluster Manager handles this for each cluster**

  - **it coordinates *checkpoints* for the cluster (snapshots of its state)**

- **If there are no clusters, no Cluster Manager is needed**

Clusters are not tightly-coupled multiprocessors (as in the “DEC Vaxcluster”)

Channels

- **Channels are communication paths between client objects and server objects**
- **Channels may be:**
  - **1 to 1 (point-to-point)**
  - **1 to many (point-to-multipoint)**
- **Channels may be:**
  - **operational**

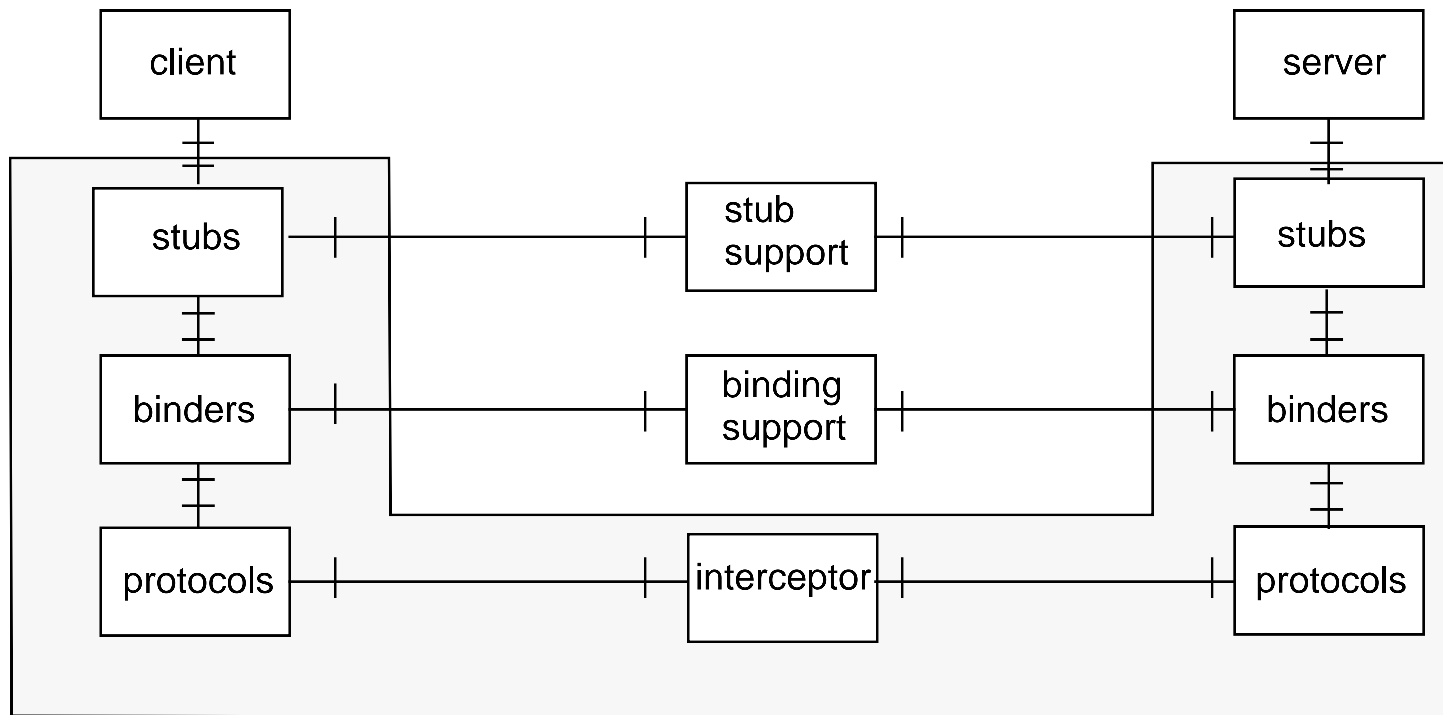
Ordinary RPC invocations

  - **stream**

For example, multimedia audio and video, telemetry and sensors

- **Channels are layered:**
  - **built from *stubs*, *binders*, and *protocol objects***
  - **there may be multiple protocols in a particular infrastructure...**
  - **...layering hides the diversity from the application**

Point-to-Point Client-Server Channel



The shaded portion is the channel proper. Stub support and binding use the control interfaces in the channel. For more details, see the ODP Prescriptive Model (Part 3)

Stubs, Binders, and Protocol Objects

- **Stubs provide data conversion**

Mainly marshalling/unmarshalling, but they can do data syntax/encoding conversion too

- **Binders manage end-to-end integrity and quality-of-service**

- **Protocol objects provide communication**

- **... Most application developers will only be aware of stubs**
  - **and even these will probably be generated automatically**

Concurrency

- **Capsules on the same node run concurrently**
- **Within a capsule, you can use *threads* to achieve concurrency**
  - **these may be supported by the operating system, or by the nucleus**
  - **... threads are always available, even under DOS!**
- **On a multi-processor system, different threads in the same capsule can run simultaneously on different CPUs**

There is a separate detailed module on concurrency

Transparencies - Simplifying distribution

- **Remember that in a distributed system, traditional design assumptions must be reversed**
  - **for example, mobility: objects do not stay in one place, they can migrate**
- **Must isolate the specification of transparencies from their design**
  - **Computational Model just assumes the transparencies are provided when needed**
  - **Engineering Model must show how transparencies are provided from engineering mechanisms**
- **Applications developers simply state which transparencies they require**
  - **software tools construct them automatically from the engineering mechanisms**

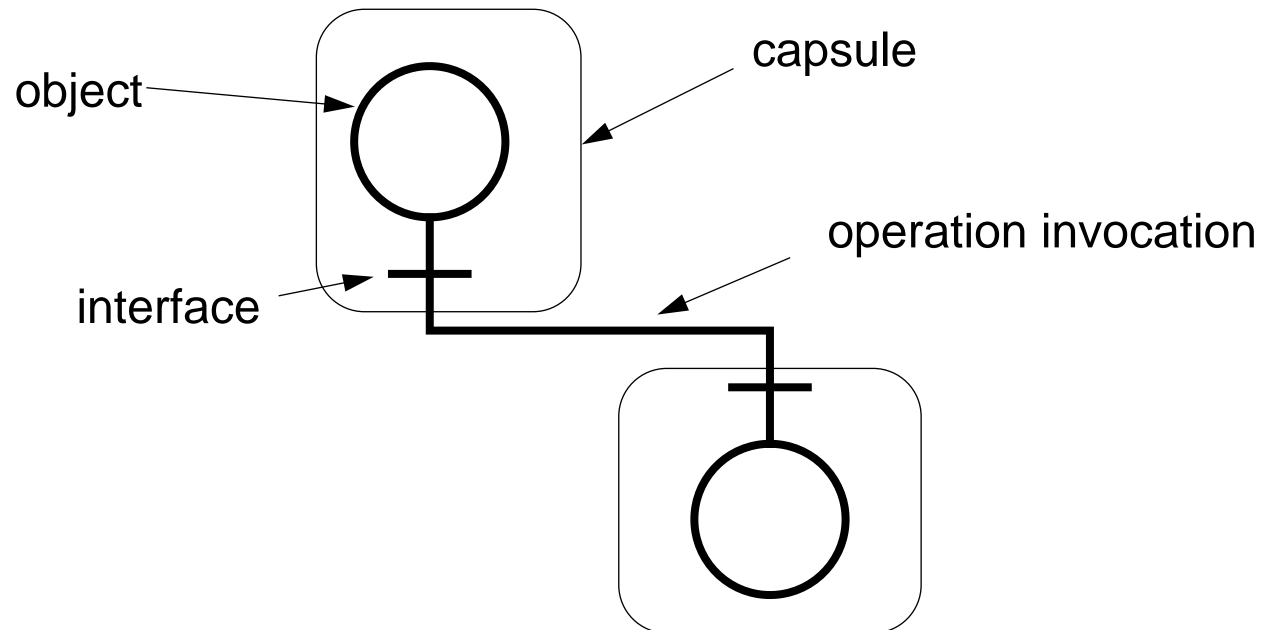
Ultimately derived from a QoS statement

Now let us look at the transparencies, one by one

ANSA

Transparency examples

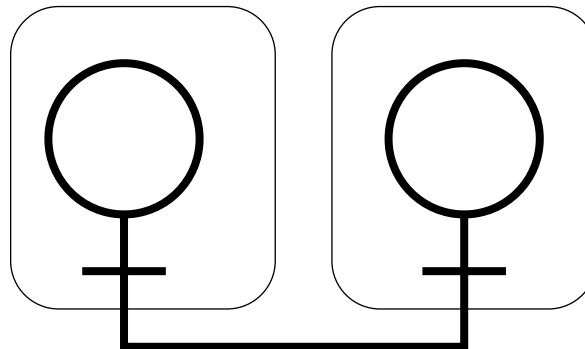
- **In these examples the diagrams are slightly simplified**
- **This shows an object invoking an operation from another capsule**



Selective Transparency Engineering - Location

- **Location Transparency**

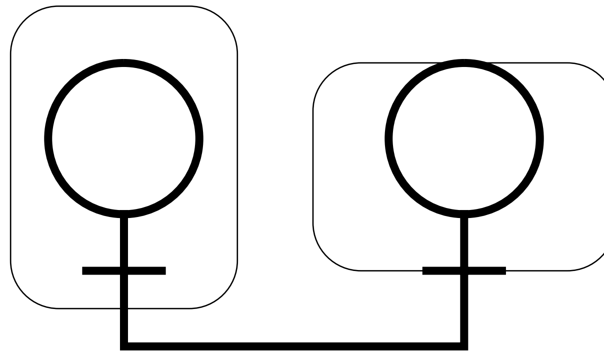
  - **application need not know where object is to use it**



  - **objects may be in the same capsule, different capsules, or different nodes**

Selective Transparency Engineering - Access

- **Access Transparency**
  - **application need not know the type of machine where the object is executing**

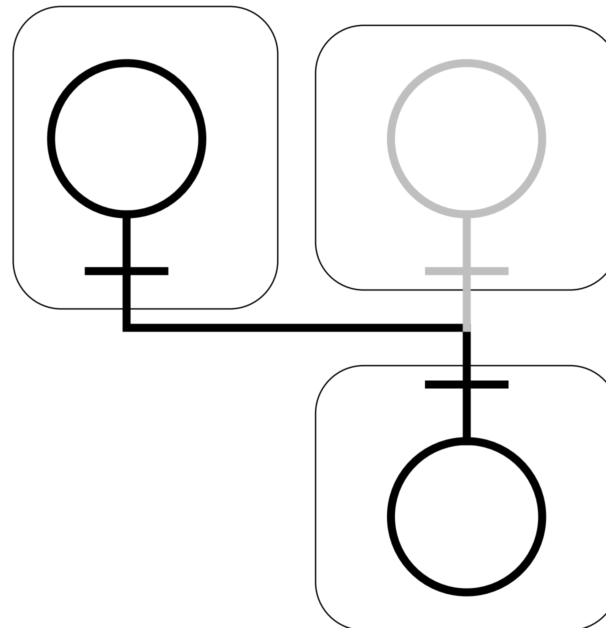


  - **objects may be in capsules on different operating systems, on different processor types (mainframe, workstation, or PC),...**

Different capsule shapes intended to suggest little-endian versus big-endian

Selective Transparency Engineering - Migration

- **Migration Transparency**
  - **application need not know where the object has moved to**



Migration Transparency

How does this differ from relocation?

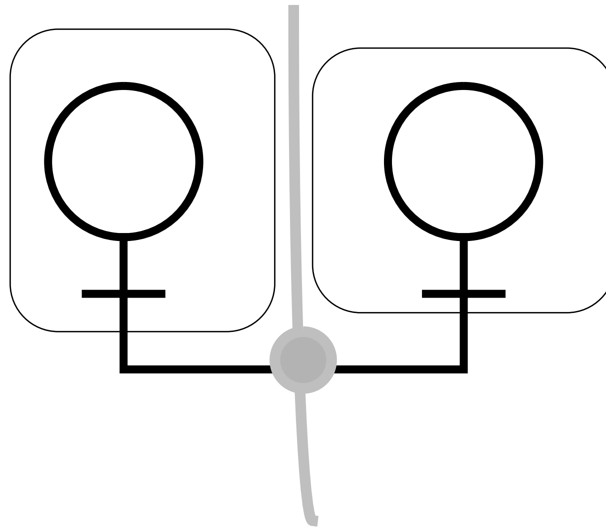
- **Object migration needed:**
  - **when a node fails, and its capsules have to be moved to another node**
  - **for load-balancing between capsules**

... and latency reduction. In practice, rather ambitious

- **Like a stronger form of location transparency**
  - **relies on location transparency mechanism**

Selective Transparency Engineering - Federation

- **Federation Transparency**
  - **application need not know where administration boundaries are**



  - **interception may happen at the boundary, but this is not visible to the application**

Federation Transparency

- **Federation is an Enterprise issue**
  - **there are many different kinds of federation boundaries: administration, organizational, contractual, and so on**
  - **constructing the transparency requires Enterprise knowledge**

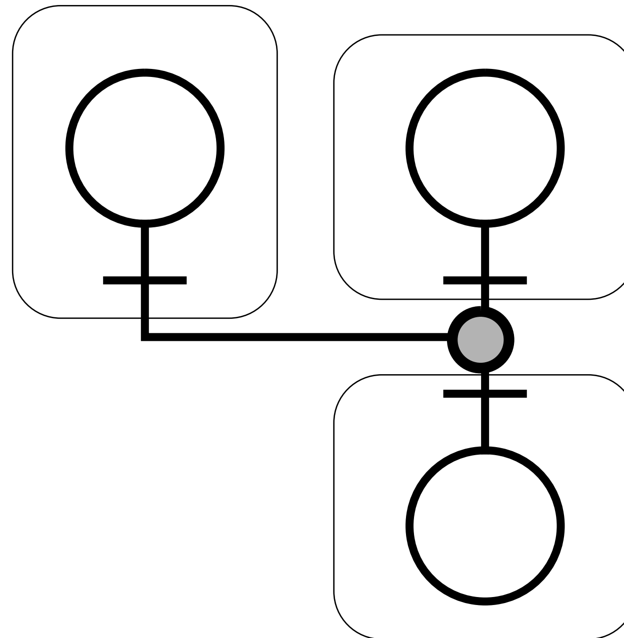
So off-the-shelf federation transparencies are unlikely; you’ll work from building blocks

- **Federation is an ANSA research area**
  - **how it relates to trading**
  - **part of ANSA Phase III**

X.500 shows just how difficult this is

Selective Transparency Engineering - Replication

- **Replication Transparency**
  - **application need not know how many copies**



  - **application only sees a single interface**

Replication Transparency

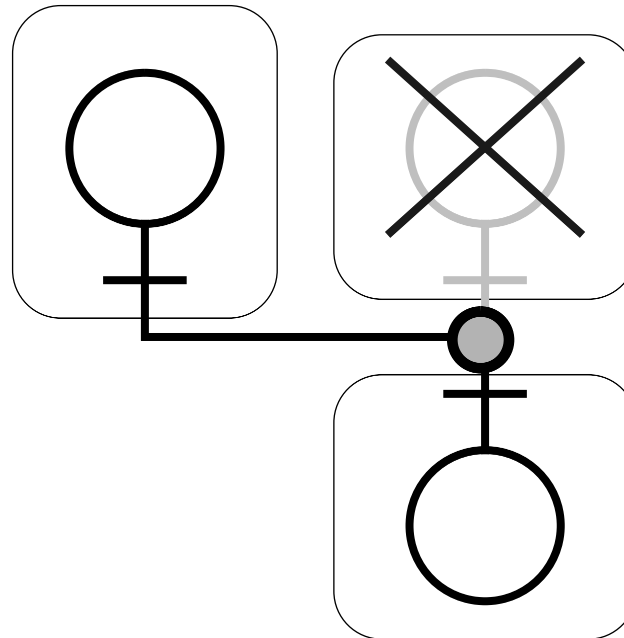
- Server objects are members of a *group*
  - do not confuse with a cluster!
- Replication transparency uses special mechanisms to make sure the group members are consistent
  - for instance, it may use multi-point channels and special protocols
- Implementing replication transparency efficiently is difficult
  - it may need information from the application
  - it is under active research in the distributed systems community

ANSAware has an experimental implementation

ANSA

Selective Transparency Engineering - Failure

- Failure Transparency
  - application need not know when an object fails



  - may use replication transparency to achieve this

Other transparencies

- **Security**
  - **application need not be aware of security policy**
- **Concurrency**
  - **application need not be aware of other concurrent operations**
- **Transaction**
  - **applications need not be aware of inconsistent states**

Implementing transparencies is a skilled task

ANSA Computational and Engineering Models - comparison

- **Computational Model concentrates on encapsulation via interfaces**
  - **interfaces consist of operations**
  - **operations have (multiple alternative) terminations**
  - **everything has a type**
- **Engineering Model is the infrastructure for Computational Model**
  - **encapsulation using capsules, clusters, and nodes**
  - **communication using channels**
  - **concurrency using threads**
  - **transparency mechanisms**

Transparency mechanisms don't appear in the Computational Model; that is why they are "transparent"

- **The Computational Model is pure...**
- **... the Engineering Model supports trade-offs**

Summary

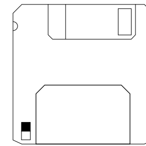
- **This has described the ANSA Engineering Model**
  - **ANSAware implements some, but not all of the Engineering Model...**

Only a few of the transparencies

  - **... and not always as described in the Engineering Model**

- **For more information:**
  - **on transparency mechanisms, see *The Challenge of ODP* (TR.033.02)**
  - **on replication transparency and groups, see *A Model for Interface Groups* (AR.002.01)**
  - **on federation, see *The ANSA Model for Trading and Federation* (AR.005.00)**
  - **on streams, see *Integrating Multimedia into the ANSA Architecture* (TR.028.00)**

Introduction to ANSAware



In this session

- **Describe the components that form ANSAware**
- **Explain the relationship between these components**
- **Mention some of the special features of ANSAware**

What is ANSAware?

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

Distinctive ANSAware features

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

ANSAware 4.1 - Infrastructure and Tools

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

ANSAware 4.1 - Services and Applications

- Services
  - Trader: provides service import/export matching
  - Factory: creates and destroys capsules
  - Node Manager: provides per-node service management
- Sample demonstration applications
- ... all supplied as ANSI C source code
  - complying to POSIX 1003.1

ANSAware 4.1 - Documentation: Manuals

- **Volume A contains...**
  - **An Overview of ANSAware 4.1: contains a general index and a summary of system requirements, also release notes**
  - **ANSAware 4.1 System Manager's Guide: contains installation and configuration instructions**
  - **System Programming in ANSAware: contains a detailed description of the ANSAware infrastructure and its internal interfaces**
- **Volume B contains...**
  - **Application Programming in ANSAware: a reference and tutorial to the whole of ANSAware:**
  - **start here!**
- **The ANSAware tape contains these manuals in PostScript form**

ANSAware 4.1 Documentation - More...

- Unix man pages
  - the PREPC language (3)
  - the service commands (1)
  - the service interfaces in IDL (3)
  - the infrastructure APIs, internal and external (3)
- ... and the complete source code

How do applications interface with ANSAware?

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

The procedural approach

- Other toolkits provide Application Programming Interfaces (APIs) between applications and systems: the *procedural* approach

Applications and Applications Programmers

Application Programming Interface

Systems and Systems Programmers

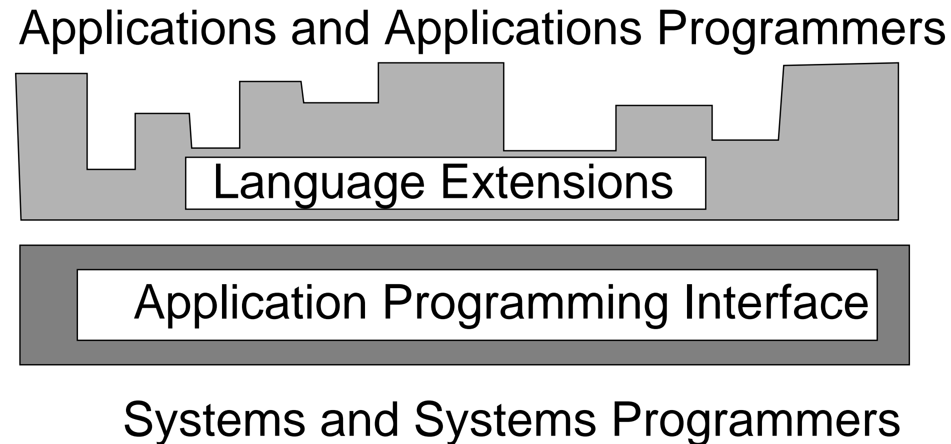
- This approach causes problems...

Problems with the API/procedural approach

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

The ANSA approach - Language Extension

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



- **The interface seen by the application programmer is expressed as a series of Language Extensions**

- **Simpler to understand and use than a procedural API**

Language Extension - Advantages

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

ANSAware components

- The ANSA Computational and Engineering Models define components that appear in every distributed system
  - ANSAware implements these components
- Explain first the relationship between
  - node
  - nucleus
  - capsule

ANSAware Node and Nucleus

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

ANSAware Capsule

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

Node support for capsules

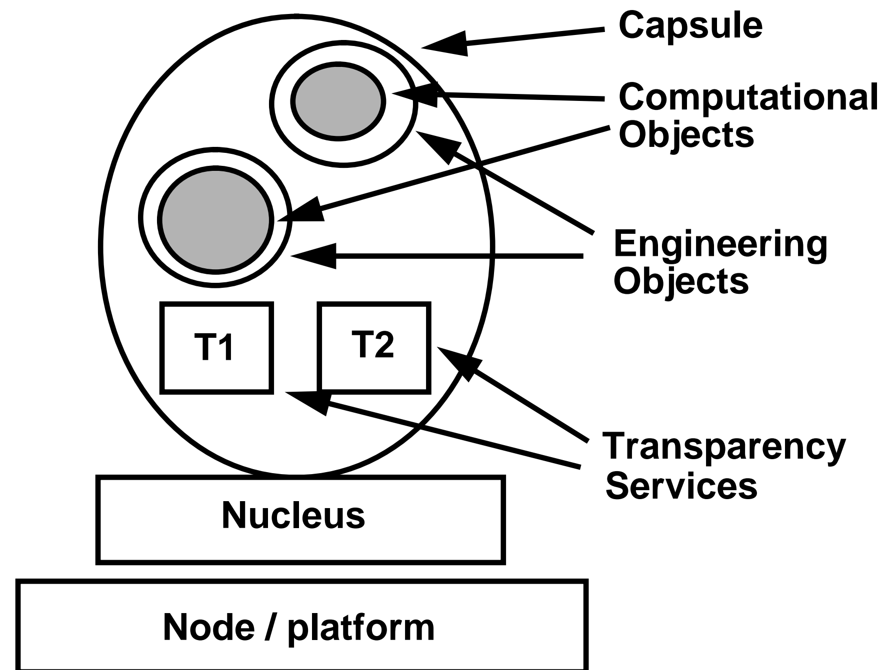
- The node support for capsules depends on the underlying operating system

  - Unix supports multiple capsules per node

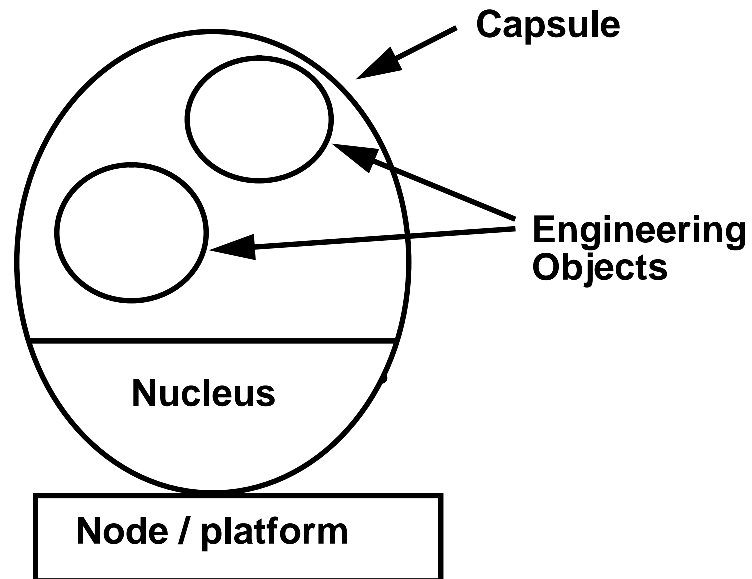
  - DOS only supports one

- Capsules can be created and destroyed dynamically

ANSA Node, Nucleus, and Capsule structure



ANSAware Node, Nucleus, and Capsule - Implementation



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

ANSAware is Lightweight

- **ANSAware applications do not require much memory**
- **ANSAware applications do not require many services**
- **ANSAware does not rely on a sophisticated operating system**
- **ANSAware has a simple and efficient RPC protocol**

ANSAware Components - IDL

- *IDL*: the language for defining service interfaces
- ... a simple example

```
Echo : INTERFACE =
-- Comment lines start with two dashes
BEGIN
   Echo : OPERATION [ Src: STRING ] RETURNS [ STRING ];

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

END.
```

ANSAware Components - STUBC

- “Stub Compiler”
- Generates stub code from IDL interface specification to
  - marshal/unmarshal IDL parameters
  - transmit the call and receive the result via the nucleus
- Stubs optimize calls within the same capsule

ANSAware Components - PREPC

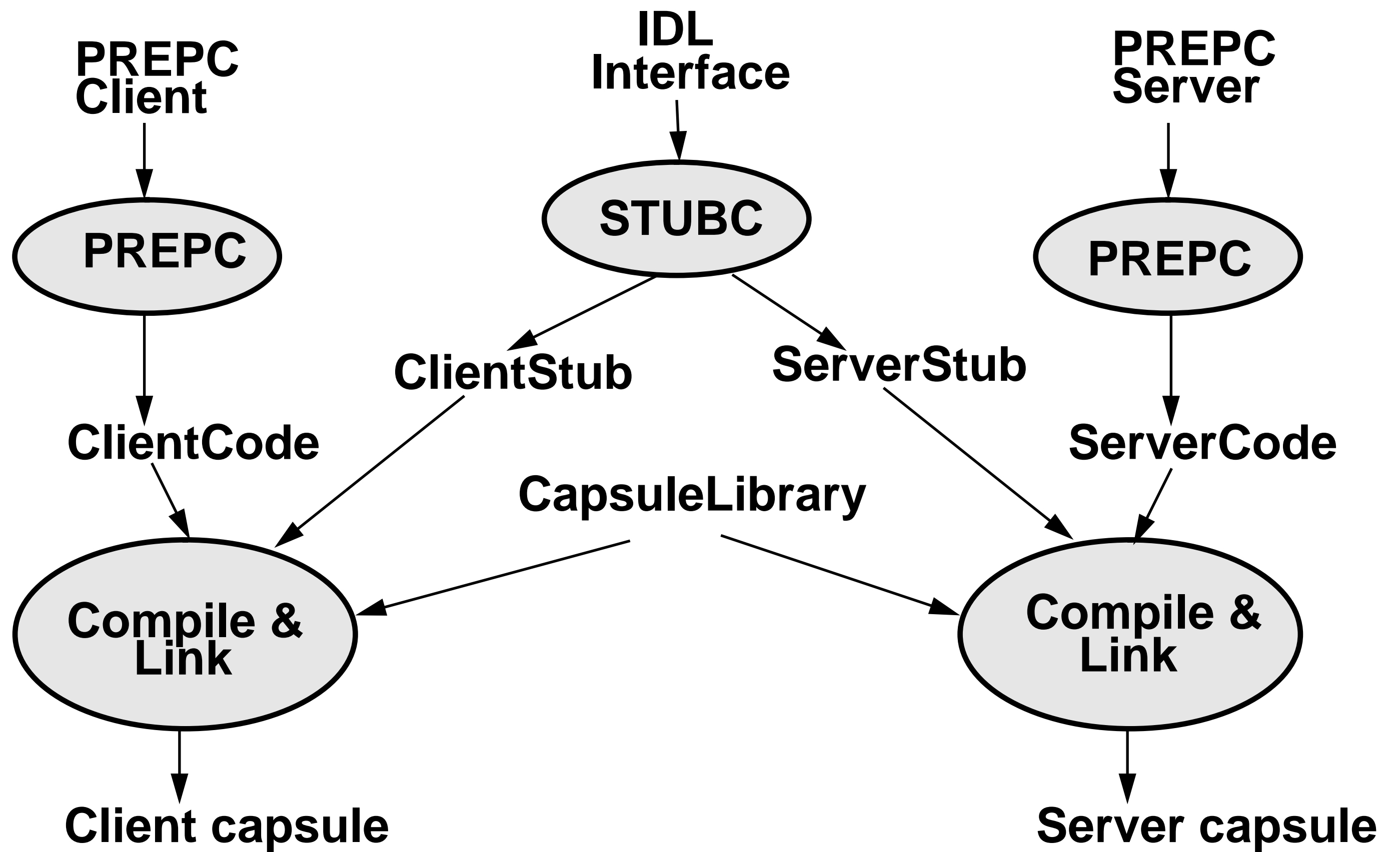
- *PREPC Language*: statements which can be embedded in C programs to access and implement services

- *PREPC Pre-processor*: translates PREPC statements into C code which interfaces to:

    - Capsule libraries

    - STUBC stubs

Building an ANSAware application



ANSAware Standard Services

- *Trader*: provides run-time matching of service requests to available services

- *Factory*: dynamic creation and destruction of services on a single node

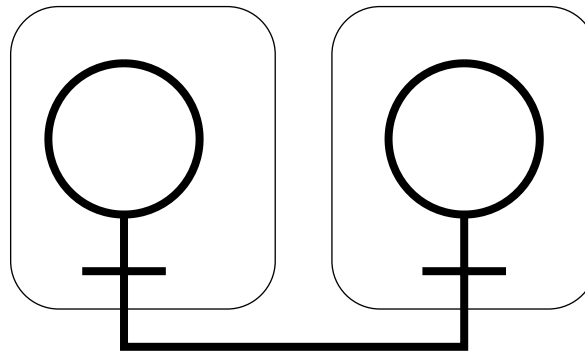
- *Node Manager*: per-node service management

ANSAware support for transparencies

- **ANSAware provides different levels of support for the various transparencies**
  - **some are automatic**
  - **some are under application control**
  - **some are provided as basic mechanisms for you to experiment with**
  - **... and some are not yet supported**
- **Here are some examples...**

Location Transparency

- **Application need not know where object is to use it**

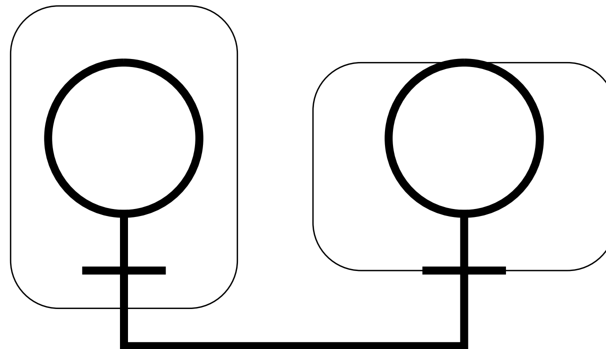


- **ANSAware provides this automatically**
  - **an application can control what happens when errors occur**

ANSA

Access Transparency

- **Application need not know the type of machine where the object is executing**

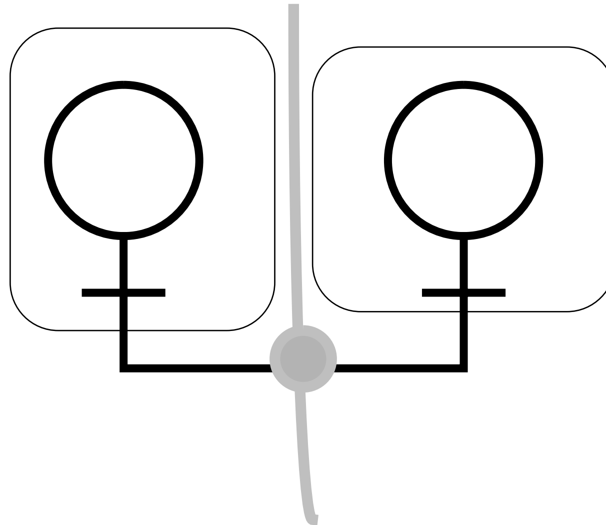


- **ANSAware provides this automatically**

  - **via the marshalling/unmarshalling in stubs**

Federation Transparency

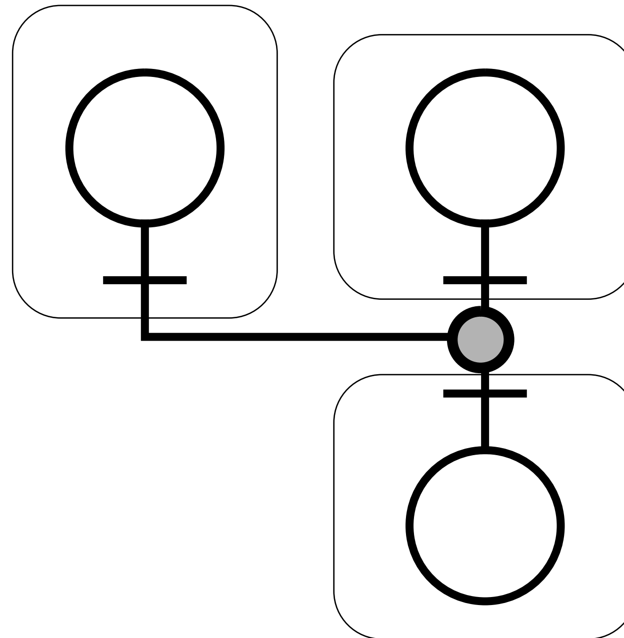
- **Application need not know where administration boundaries are**



- **ANSAware supports interconnection via the Trader**
  - **but does not provide any kind of interception**

Replication Transparency

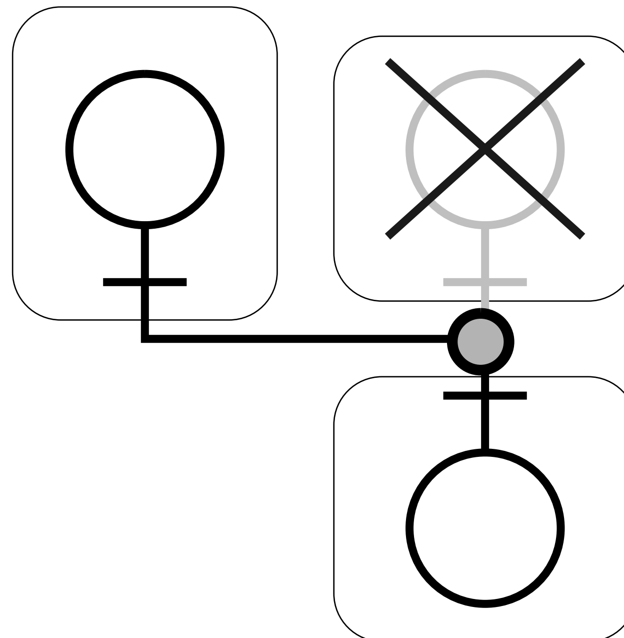
- **Application need not know how many copies**



- **ANSAware provides an experimental implementation of *groups***
  - **using a group execution protocol (GEX)**

Failure Transparency

- **Application need not know when an object fails**



- **ANSAware does not support failure transparency**

ANSAware Supported platforms

- Unix
  - HP/UX (HP/UX version 7.0/8.0/9.0 on HP 9000 series)
  - SunOS (Sun3 and Sun4)
  - OSF/1 (DEC Alpha)
- DOS/Windows
  - bare DOS
  - Windows 3.1 386 Enhanced Mode
- VMS
- ... and others to come

Other ANSAware platforms

- **ANSAware has been ported to many other platforms**
- **Some are in the ANSAware distribution**
- **... others are available on request**
- **And other products based on ANSAware are available elsewhere**

Summary

- **A light-weight, portable implementation of the ANSA Architecture**

- **Applications written in C with embedded ANSAware calls**

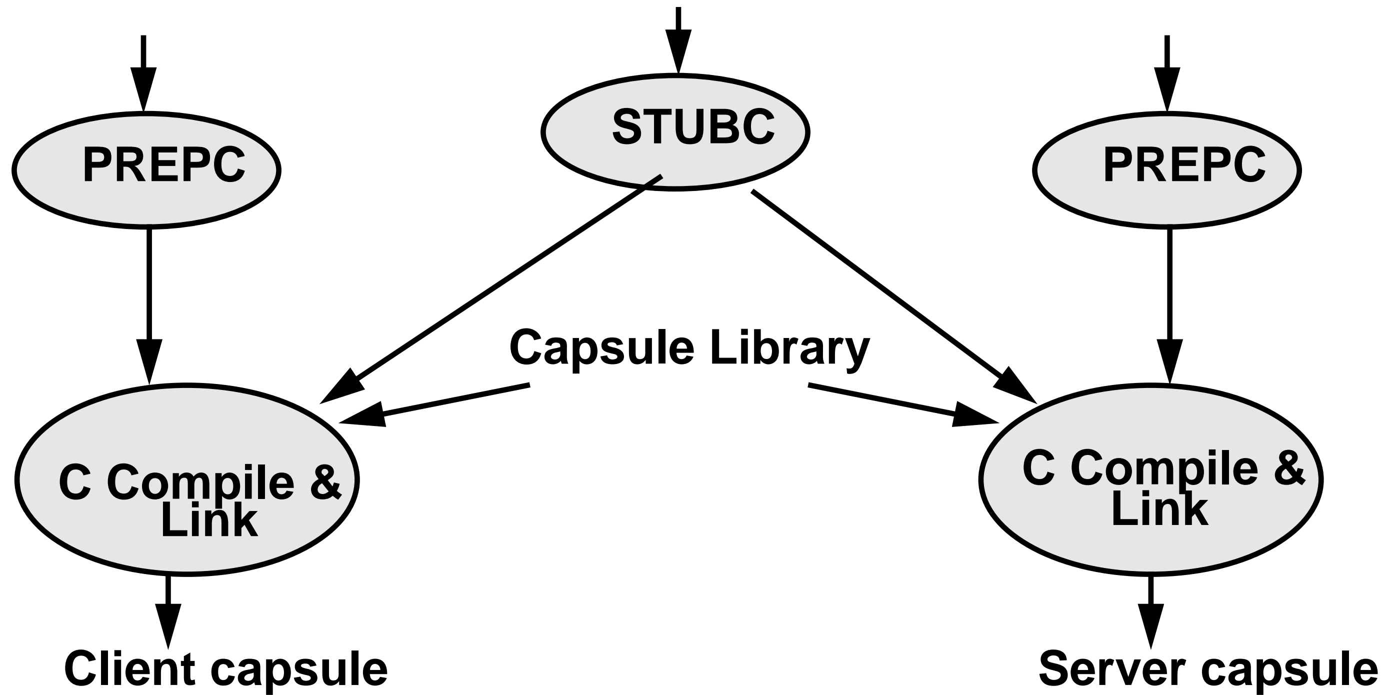
  - **These calls are a language extension**

- **Tool support for ANSA transparencies**

- **For more detail, see *Application Programming in ANSAware* (ANSAware Volume B)**

ANSA

ANSAware 4.1
Building Services with ANSAware Tools



In this session

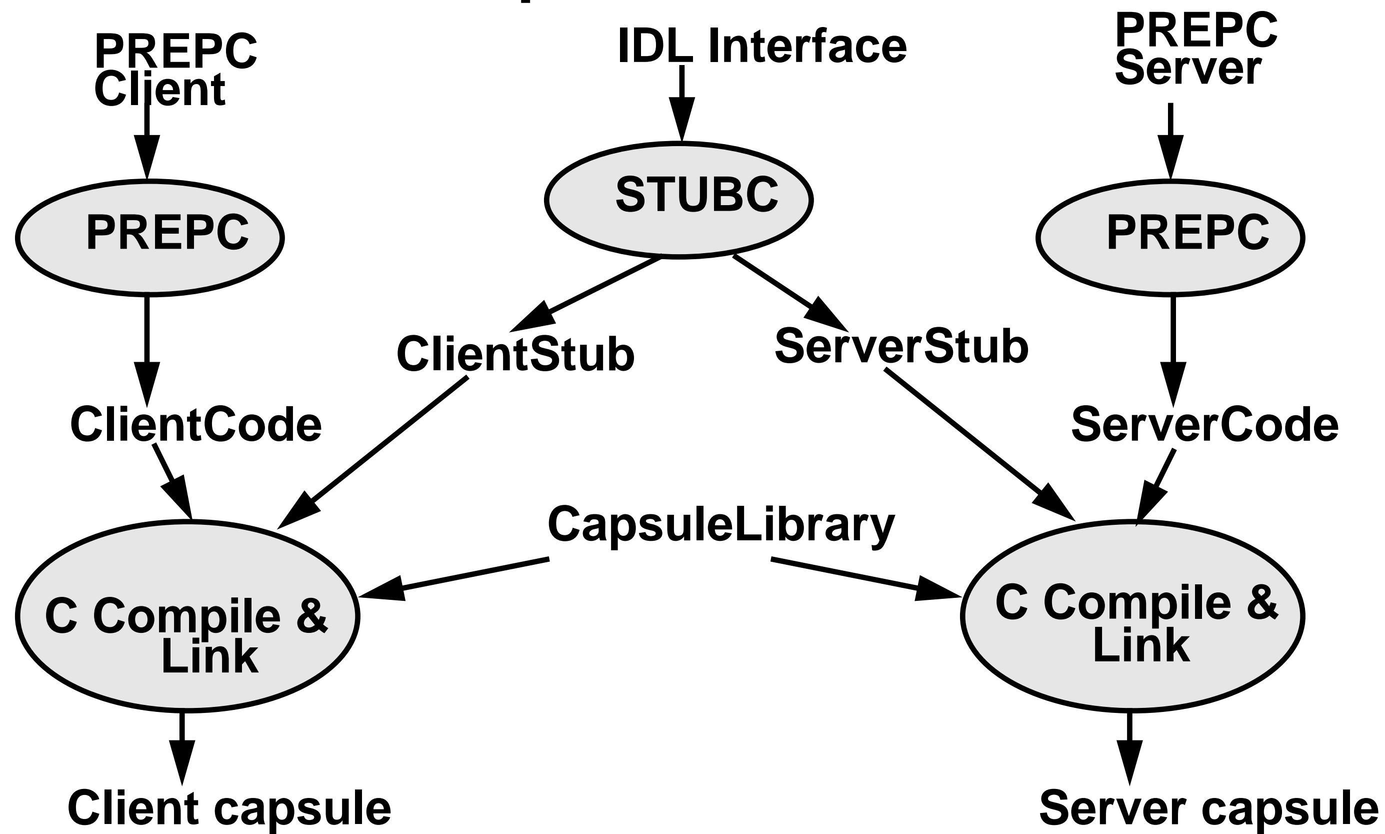
- **Show the steps needed to build a simple service (Echo) using ANSAware tools**

- **Show the IDL and PREPC languages, and how to use their preprocessors (STUBC and PREPC) to generate C**

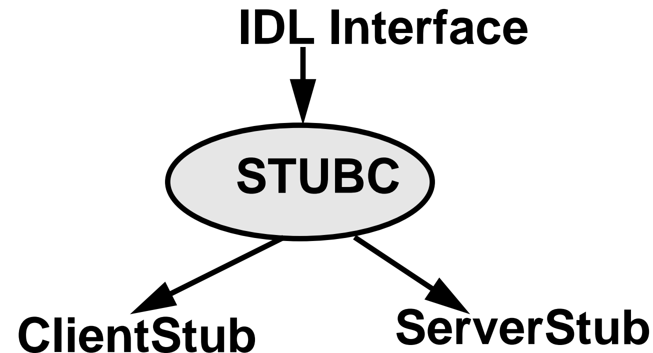
- **Explain basic use of the ANSAware Trader**

- **Build and try out a simple example service *(Echo*)**

Components - Review



IDL - the Interface Definition Language



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

IDL - Defining operations

```
Echo : INTERFACE =

-- Comment lines start with two dashes

BEGIN

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

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

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

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

END.
```

IDL Types

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

IDL Primitive Types

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

IDL Type Constructors

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

IDL - Defining new types

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

IDL - built-in types

- An abstract type definition is provided for `ansa_InterfaceRef`

- Every interface has an interface reference type defined for it automatically:

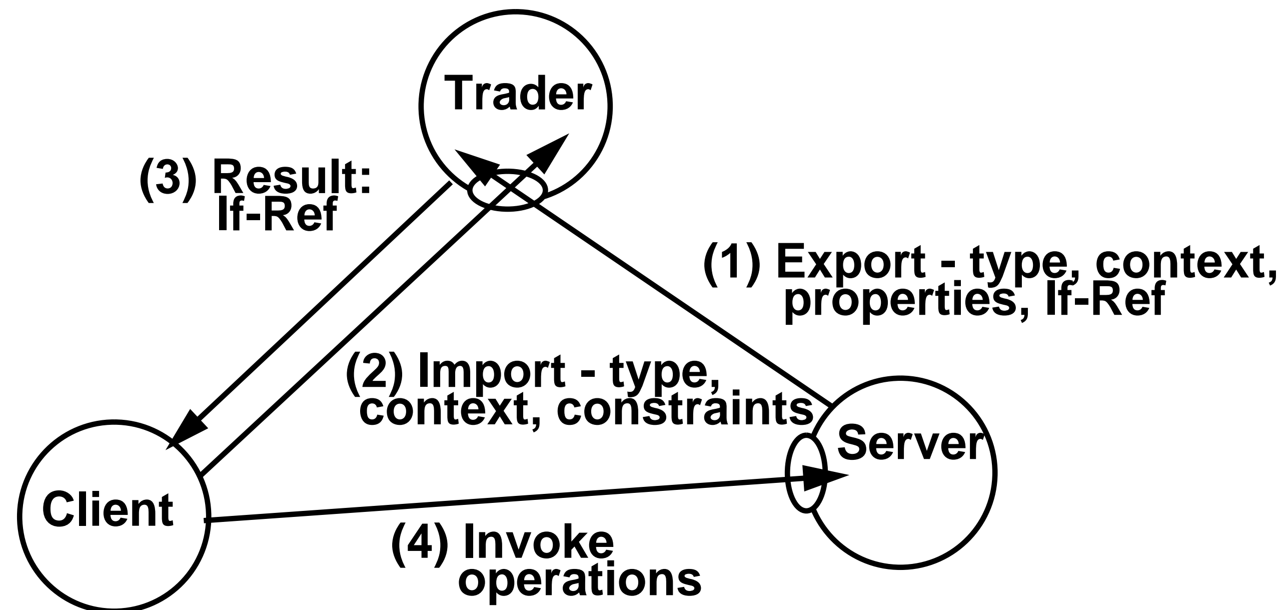
```
X: INTERFACE =

BEGIN

   XRef: TYPE = ansa_InterfaceRef;

END.
```

PREPC

- C preprocessor
- Write clients and servers in C
- Embed PREPC statements in C
- PREPC/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: the trader, factory, node manager, as well as the services you define

ANSA

Using the Trader from PREPC statements



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

PREPC statements - Echo client structure

- **Client:**

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

PREPC statements - Echo server structure

- **Server:**

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

PREPC statements - Echo server operations

- **Server Operations:**

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

{
/*  do the operation */

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

Understanding the Echo service

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

- **... but is worth examining in detail, to understand:**

  - **how the parts fit together when the example is built**

  - **how to add the operation *Reverse* to its implementation (not shown here)**

Echo service description

- **A service that, when sent a string, simply echoes back a string**

Doesn’t store the string; just responds

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

Echo service interface (IDL)

```
Echo : INTERFACE =

-- Comment lines start with two dashes

BEGIN

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

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

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

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

END.
```

Echo Client

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

Echo Server

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

Echo Server - Echo operation

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

Using Imakefiles

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

Imakefile for Echo client and server

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

# compile idl files
all:: $(SIFFILES)

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

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

Building the Echo service

$ansamkmf

to create a Makefile from the Imakefile

$make depend

to create the required dependencies

$make

to create client and server programs

Starting the Echo server

$./server &

to run the server

$trclient search Echo /ansa/testservices

to find all the available Echo servers

$trclient search Echo /ansa/testservices "Node == ’your_machine’"

to find the server on your node

Using the Echo service

- **To use the Echo service provided by any Echo server:**

$./client < Imakefile

- **echoes the contents of the Imakefile to your screen**

- **To use the Echo service provided on your node:**

$./client "Node == ’your_machine’" < Imakefile

Modifying and Extending the Echo service

- To change the implementation of the server:
  - just modify the server procedures (in C)
- To add new operations:
  1. Add the new operation to the IDL file (and compile it with STUBC)
  2. Add the new server procedure to the server implementation
- To use new operations:
  - modify the client (or write a new one)

Summary

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

Echo service - Extra information

- The Echo service is a *managed* service

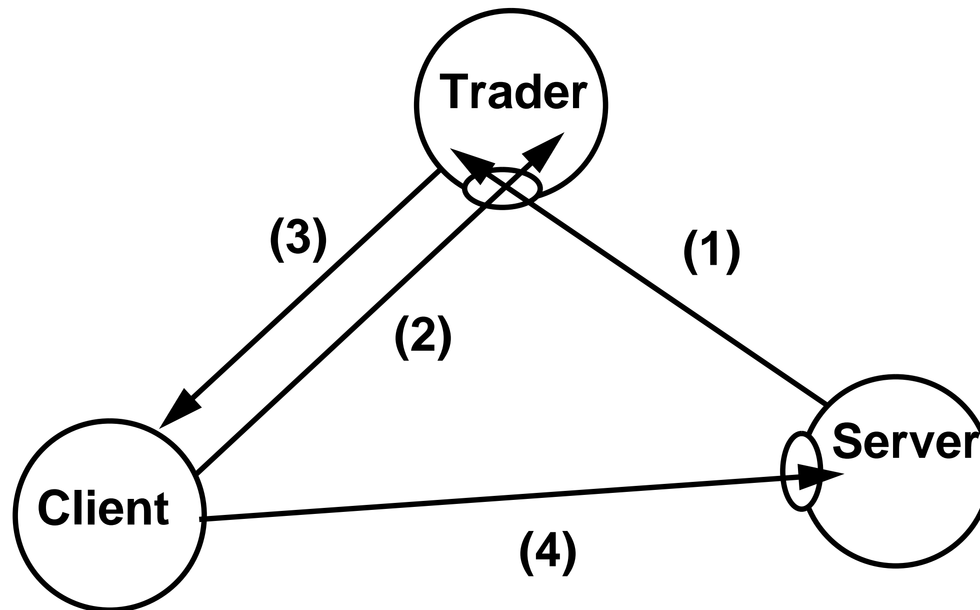
- With a managed service, client programs can start up the server automatically

  - you do not need to run the server first

- For more information, see *Application Programming in ANSAware*, Chapter 5

ANSAware 4.1

Using the Trader service



In this session

- Explain how the ANSAware Trader implements the ANSA trading concepts

- Describe the command-line tools to manipulate the Trader

- Experiment with using these tools

Review of ANSA trading concepts

- Trading allows clients to find servers that provide the services that they need

- The main criterion for matching is type conformance

- Traders can be federated together

Purpose of the ANSAware Trader service

- To implement most of the ANSA trading concepts, namely
- To allow a client to find a server which offers a service:
  - of an appropriate (matching) type
  - with other appropriate characteristics (for example, Quality of Service)
- To support federation of multiple traders
- To allow administrative management of service offers

Information in the Trader

- **The ANSAware Trader holds information about:**
  - ***service offers* and their *properties***
  - ***interface types***
  - ***trader contexts***

The ANSAware Trader interfaces

- **The Trader service has 4 separate interfaces:**
- **1 service interface (of type *Trader)***
  - **for registering, looking up, and deleting service offers**
- **3 management interfaces (of types *TrType, TrCtxt, TrFed*)**
  - **for managing interface types**
  - **for managing trader contexts**
  - **for managing trader federation**

The Trader's service interface *(Trader)*

- **It has 3 operations: Register, Lookup, and Delete**

  - ***Register*: exports a new service offer to the trader**

  - ***Lookup* - lists one, or all, offers that match**

    e.g. `traderRef$Lookup(...)`

  - ***Delete*: removes service offers**

Using the Trader service interface

- **Applications don’t generally use these Trader operations directly**
  - **it is simpler to use the $Export, $Import, and $Withdraw PREPC statements instead**
  - **these PREPC statements invoke the Trader operations**
  - **traderRef$Import invokes Lookup**
  - **traderRef$Export invokes Register**

Trader service programs

- **The Trader is supplied with some simple command-line programs**

- **The program `trtest` invokes 1000 *Lookup* operations and prints how long it took**

- **The program `trclient` invokes a *Search***

trclient search ansa /

Compatibility of interface types

- If an interface-type *StringOps* is compatible with type *Echo*, it provides at least the operations of type *Echo*
  - ... it may provide extra operations as well
- A new interface that is an extension of an existing one can be defined in IDL with the IS COMPATIBLE WITH statement:

```
StringOps: INTERFACE =
IS COMPATIBLE WITH Echo;
BEGIN
-- Define new operations here...

END.
```

- This is called *type conformance* in the ANSA Computational Model

Effect of type compatibility

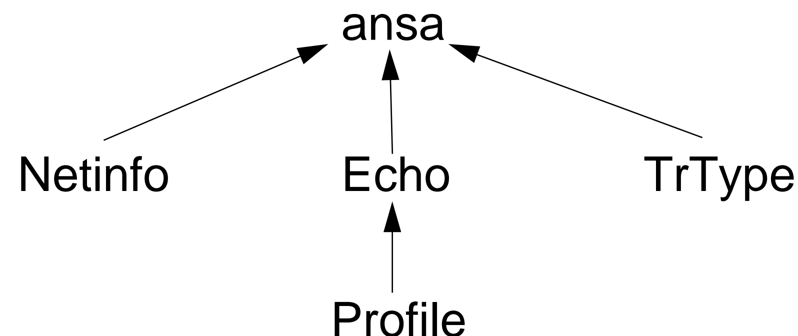
- **Because StringOps conforms to Echo, it can be used wherever Echo is needed**
  - **a client that imports type Echo may obtain an interface for either an Echo or a StringOps service**
- **code for *StringOps* must provide all the operations of Echo**
  - **and it can have others**
- ***StringOps* type must also be registered with the trader**

Trader interface types

- **Every interface type has a name**
  - **as given in the IDL**
- **Each type has one or more immediate supertypes, to which it conforms**
  - **The root type is conventionally called *ansa* (it has no supertype)**
  - **Relationship of new type to others is given when registering type with Trader**
  - **Trader maintains a list of the type relationships**

The Trader type graph

- **The relationships form a *directed acyclic graph***
  - **more general than a tree, because types can have more than one supertype**



- **conformance of offer type to requested type ensures that client and server can exchange information**
  - **Type relationships are asserted by user, and checked at invocation time**
  - **Obviously, nothing can check whether the interface is correctly implemented; this depends on the implementations of client and server**

TrType - managing the type-space

- **Trader's TrType interface provides the operations for managing Trader's type-space**

- **The command-line tool `typecl` uses this interface to manipulate trader's type-graph**

Managing types with typecl

- with the `typecl` (type client) program you can:
  - add a new type (optionally specifying types to which it conforms)
  - delete an existing type
  - mask an existing type (that is, prevent further use before deleting it)
  - unmask a masked type
  - list existing types

Adding types with typecl

- **To add a new type:**

```
typecl add type_name [supertype [supertype...]]
```

- **For example:**

```
typecl add Newtype ansa
```

- **The supertype must already exist...**

```
typecl add NewType ansa/SBank
add failed - status = TNoSuchSuperType

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

- **You can specify more than one supertype (in this case, NewType conforms to t1 and t2)**

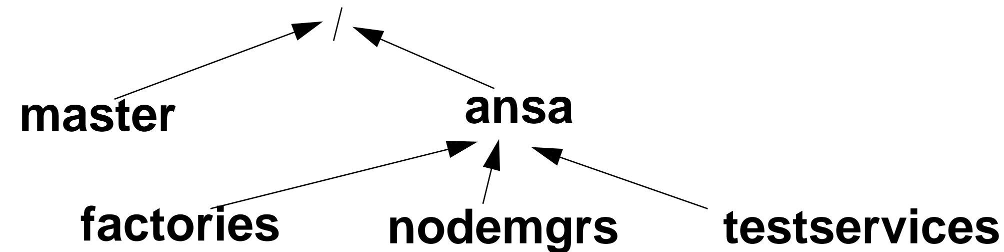
```
typecl add NewType t1 t2
```

Trader contexts

- Contexts allow administrators to group service offers together
- Every offer is registered in a context
- Trader maintains a tree of contexts
  - each context can register many service offers
  - contexts can contain other contexts within them (like Unix subdirectories)
- the Trader uses context pathnames (like Unix pathnames)
  - the root of the context tree is "/"

Administering Trader contexts

- The context tree looks like this:



- With the context client (ctxtcl) program you can add, delete, and list contexts

Adding Trader contexts with ctxtcl

- **To add a new context:**

```
ctxtcl add context
```

- **... for example:**

```
ctxtcl add /ansa/testservices/course
```

Deleting Trader contexts with ctxtcl

- **To delete a context:**

```
ctxtcl delete context
```

- **... for example:**

```
ctxtcl delete /ansa/testservices/course
```

Listing Trader contexts with ctxtcl

- **To list contexts:**

```
ctxtcl list
```

  - **for example:**

```
ctxtcl list

/master
/ansa
/ansa/factories
/ansa/nodemgrs
/ansa/testservices
```

Trader properties and constraints

- Properties hold application-specific information about a service offer
  - e.g. for a print service: LinesPerMinute, PaperSize, costPerPage, Turnaround
- Property Name/value pairs are specified when offer is registered
- The Trader automatically generates some properties
  - for example, it generates the (Type, TypeName) property for every offer registered
- Clients requesting services may specify:
  - Acceptable property constraints (e.g. PaperSize==’A4’)
  - Selection of offer with maximum (or minimum) value of a property

PREPC Export and Import syntax

- **Export and Import PREPC statements have 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

Trader properties when Exporting

- When exporting a service you specify a property list:

  - the list consists of pairs of (name, value) items

  - the pairs in the list separated by spaces

  - for example:

```
Node basilisk Price 100
```

Trader properties when Importing

- **When importing a service you specify constraints:**
  - the constraints are in the form of a logical expression, for example:

    ```
    (Node == 'basilisk') and (Price <= 200)
    ```

  - the expressions can use the usual operators (`==`, `!=`, `and`, `or`, `not`, ...)
  - for the full syntax of the constraint language, see 'Application Programming in ANSAware' (section 3.11.4)

Finding a matching offer using trclient

- To look up one matching offer:

```
trclient lookup type context [constraints]
```

- For example:

```
trclient lookup ansa /
```

- This displays one offer, selected randomly from those which match
- This is how the PREPC $Import statement works
- Selecting the offer randomly spreads the client load across servers with the same interface
- Matching is determined by the type, context, and any property constraints

Finding all matching offers using trclient

- **To search for all matching offers:**

```
trclient search type context [constraints]
```

- **For example:**

```
trclient search ansa /
```

- **Matching for type, context and constraints is the same as *Lookup***
  - **but you see all the matching offers, not just one of them**

Registering an offer using trclient

- **To register an offer:**

```
trclient register type context [constraints]
```

- **For example:**

```
trclient register test3 /ansa "Name Foo" 60
```

Deleting an offer using trclient

- To delete an offer:

```
trclient delete nonce [constraints]
```

- A *nonce* is a unique identifier for the offer generated internally by ANSAware
  - it is one of the properties displayed by trclient search
- You shouldn't normally need to delete an offer
  - Services normally withdraw their offers when terminating
  - you only need to delete the offer using trclient if the service terminates incorrectly

Local and Master Traders

- **Every ANSAware program knows the interface-references of two Traders**
  - **the Local and Master Traders**
  - **the interface-references are compiled in to the capsule library**
  - **the Local and Master Traders may in fact be the same Trader**
- **When importing, the Local Trader is tried first: if there is no match, the Master Trader is tried**
- **This is not true federation; it is an additional mechanism for grouping services**

X Graphical User Interface to the Trader

- **These are equivalent to the command-line tools:**

Interface	Command-line tool	X tool
Trader Client	trclient	offerMgr
Trader Type Client	typecl	typeMgr
Trader Context Client	ctxtcl	ctxtMgr

- **The X tools are not built and installed by default**

ANSAware Trader - Summary

- Trader holds information about:
  - service offers and their properties
  - interface types
  - trader contexts
- Traders can be federated

ANSAware Trader - More information

- **For the command-line tools, see Appendix C of *Application Programming in ANSAware***
- **For the X tools, also see Appendix C of *Application Programming in ANSAware***
- **For an overview of the ANSAware Trader, see Chapter 3 of *Application Programming in ANSAware***

ANSAware 4.1 Trader - Extra information on Types

- The ANSA computational model is not fully implemented by the Trader
- Types:
  - conformance relationships must be specified explicitly when types are added. If the relationship is incorrect, the Trader will not detect it. But there is a sanity check at invocation time
  - the Trader does not check conformance of types. It merely compares names
  - programs assert the types of their interfaces; these assertions may be invalid. The Trader cannot check this either, since it does not have the type description available

ANSAWare 4.1 Trader - Extra information

- **Contexts**
  - the name space is a tree rather than a directed acyclic graph. Context name components are separated by a "/".
- **Properties**
  - All properties are optional. There is no way of forcing an offer of type X to specify properties (Name, x), (Host, y).
- **Searching**
  - Searching is 'shallow'; federated traders are only consulted if the current trader cannot satisfy the import request.

ANSAWare 4.1 Trader - Deleting service offers

- **Service offers are deleted when:**

  - **a server terminates normally: the server can explicitly withdraw any offers it has `Export`ed, using the PREPC `Withdraw` instruction**

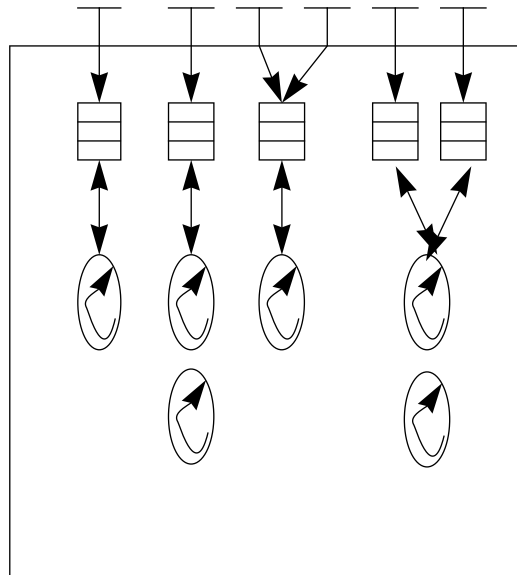
  - **a server is killed by a signal (CTRL-C): the ANSAware infrastructure withdraws all `Export`ed offers from Trader before exiting the capsule**

  - **an offer is 'stale': the trader will automatically remove an offer it considers 'stale'**

  - **a client cannot invoke operations on an interface-reference received from the trader: the client's infrastructure notifies the Trader**

ANSAware 4.1

Concurrency



In this session

- Explain the purpose of concurrency

- Explain the various ANSAware concurrency features

- Explain some of the pitfalls

What is concurrency?

- **Concurrency allows an application to overlap work**
  - **when one activity is blocked waiting for a response, other activities can execute**
- **Concurrency allows more efficient use of resources**
  - **but does not in itself guarantee a real-time response**
  - **nor does it control the use of resources**

Concurrency support in ANSAware

- **Multithreading: supporting multiple potential execution within a capsule**
  - **supported by ANSAware**
- **Multitasking: scheduling actual processor time to threads**
  - **supported by ANSAware**
- **Multiprocessing: supporting simultaneous execution on multiprocessor systems (tightly-coupled or clustered CPUs)**
  - **not currently supported by ANSAware, although experimental multiprocessor systems have been supported**

Before you start...

- **Concurrency is not transparent**
- **Most programming languages (including C and C++) provide no support for concurrency**
- **CAUTION: concurrent programs are inherently difficult to test and debug**
  - **it is easy to make mistakes, and hard to detect them**
  - **only use concurrency if you need it**
- **ANSAware simplifies some aspects of concurrency**

The need for synchronization

- Synchronization is needed to prevent concurrent invocations interfering
- Consider a simple object holding a pair of numbers (A, B)
- Each interface has one operation
  - set_AB, get_AB, and swap_AB

Implementing the Swap object

- **We might implement this using two internal variables, for the values A and B**
- **set_AB**

```
a_value = a_arg;
b_value = b_arg;
```

- **get_AB**

```
a_result = a_value;
b_result = b_value;
```

- **swap_AB**

```
temp = a_value;
a_value = b_value;
b_value = temp;
```

- **It’s trivial...**
  - **but what happens when these interfaces are invoked concurrently?**

Concurrent access to the Swap object

- Suppose set_AB and get_AB are invoked concurrently
- get_AB executes its first statement

```
-> a_result = a_value;
   b_result = b_value;
```

- set_AB executes its first statement

```
-> a_value = a_arg;
   b_value = b_arg;
```

- set_AB executes its second statement

```
   a_value = a_arg;
-> b_value = b_arg;
```

- get_AB executes the second statement

```
   a_result = a_value;
-> b_result = b_value;
```

The result?

- **It is inconsistent; it has the old value of A and the new value of B**
  - **the two invocations have interfered...**
  - **...this must not happen**
- **In this case, interference can be avoided by changing get_AB...**

```
b_result = b_value;
a_result = a_value;
```

- **... but now think about set_AB and swap_AB**
  - **or, two concurrent invocations of set_AB**
  - **interference cannot be avoided**

A solution - Mutual exclusion with locks

- **Make each of the operations an indivisible ‘critical section’ around the statements**
  - **only one invocation allowed at once**
- **Lock it at the beginning of the section, and unlock at the end**
- **If a concurrent invocation finds it locked, it will wait until the first operation leaves the critical section, and unlocks it**

Synchronization and shared state

- It is shared state that causes interference
- Synchronization is needed when accessing shared state
  - shared between interfaces, or operations in the same interface
  - ... and also concurrent invocations of the same operation
- The easiest form of synchronization is mutual exclusion using locks
  - only one invocation can access the shared state at once; other concurrent invocations must wait

Synchronization and deadlock

- If two invocations each need access to shared data that the other invocation has locked, they will deadlock

- ANSAware does not detect deadlocks

- Understand and avoiding deadlocks is a complex topic

  - to understand this, read about concurrency in databases

Tasks and Threads

- Tasks are the unit of *actual* concurrency
- Tasks have a stack and save area for their CPU state
- Threads are the unit of *potential* concurrency
- Threads must be assigned to a task in order to execute
- 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 finished
- Threads use less memory than tasks, because tasks have stack space

Task Scheduling

- **Task scheduling may be either**
  - **pre-emptive: a task may be suspended without knowing, and another task scheduled**
  - **non-pre-emptive: a task is only suspended when it explicitly yields to other tasks**
- **Applications cannot assume either scheduling policy, and must allow for both**
- **To allow for pre-emptive scheduling:**
  - **threads must use synchronization mechanisms to ensure exclusive access to shared data**
- **To allow for non-pre-emptive scheduling:**
  - **a thread that executes for a long time (in a tight loop), should yield by calling `instruct_Pause()` to allow other threads a chance to execute**

Task Creation

- The number of available tasks is controlled by the application:
  - statically:

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

  - dynamically:

```
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

Thread Creation

- **Some threads are created automatically**
  - **for example to handle incoming invocations that are queued**
  - **this is transparent to the applications**
- **Applications can create threads explicitly**
  - **using `instruct_Fork()`, `instruct_Spawn()`, and `instruct_Join()` functions**
  - **but must manage these threads themselves**

Thread Synchronization

- Threads need to synchronize with each other
  - when accessing shared data or resources
- Operating systems support a large variety of synchronization mechanisms
  - spin locks, lockouts, mutexes, mutants, eventcount/sequencers, events, event flags, flags, critical sections, semaphores, test-and-sets, zones,...
  - ... all non-portable, apparently similar, but subtly different
- ANSAware standardizes on two mechanisms
  - eventcounts and sequencers
  - mutexes

Eventcounts and Sequencers

- **These mechanisms are used together (called ecs)**
  - **eventcounts provide synchronization**
  - **sequencers provide an indivisible increment operation**
- **Eventcounts have only two operations**
  - `ecs_await ( eventcount, value)`
    blocks until the value of `eventcount` is at least `value`
  - `ecs_advance ( eventcount )`
    increments the value of `eventcount` by 1
- **Sequencers have only one operation**
  - `ecs_ticket ( sequencer )`
    an indivisible operation which returns the current value of `sequencer` and then increments `sequencer` by one.
- **These are all you need - but the mechanisms may be inconvenient to use**

Mutexes for mutual exclusion locks

- Mutexes are usually more convenient, if you only need mutual exclusion

- The functions are:

  - ansa_InitMutex(ansa_Mutex *m)
    Initializes a mutex

  - ansa_AcquireMutex(ansa_Mutex *m)
    Acquires (locks) a mutex, if necessary blocks until it has been released

  - ansa_ReleaseMutex(ansa_Mutex *m)
    Releases (unlocks) a mutex; this may unblock at most one other thread for this mutex

  - ansa_FreeMutex(ansa_Mutex *m)
    Frees a mutex

- In fact, these functions are macros that use event counters and sequencers

Timers

- Timers allow a thread to delay for a period of time, doing nothing
- Use timers rather than writing a 'busy wait' loop, because
  - other threads may be waiting to execute
  - timers are more accurate
- Two forms of timers are supported
- `timer_Sleep( unit, delay )`
  - suspends the calling thread for a `delay` `TSeconds`, `TMilliSeconds`, or `TMicroSeconds` as specified by the `unit` argument.
- `timer_setTimer ( unit, delay, action, data, owner )`
  - causes the function `action` to be called `delay` units in the future with `data` as an argument
  - `(the owner` argument is for debugging purposes only)

Invocation versus Initiate and Redeem

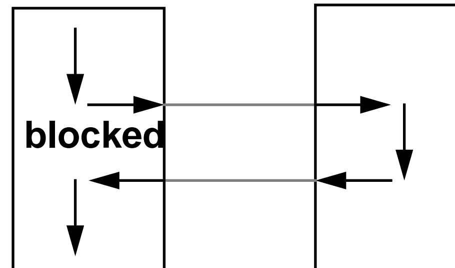
- **There are two ways of using an operation**

    - **invocation**

    - **initiate and redeem**

Invoking an operation

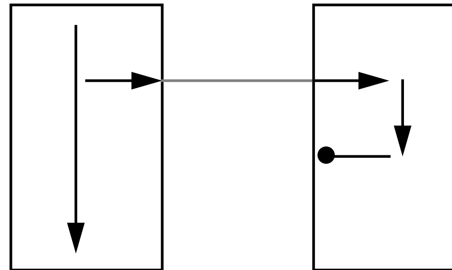
- **Invoking an operation is synchronous...**



- **the invoker blocks until the operation is complete, and the result is available**

Initiating an operation

- **Initiating an operation is asynchronous...**

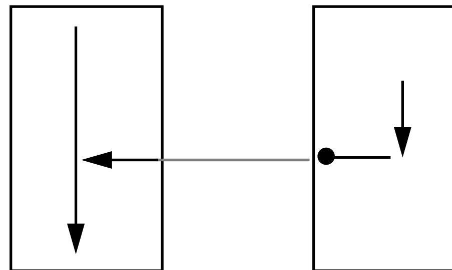


- **...the initiator continues concurrently**

ANSA

Redeeming an initiate operation

- The initiator is given a *voucher*
- When the initiator is ready for the result of the operation, it redeems it



- If the operation is not yet complete, the redeem will block
  - just as for an invocation

PREPC statements

- **Invoking...**

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

- **Initiating and redeeming...**
  - **Voucher declaration**
    `ansa_Voucher voucher;`
  - **initiating uses := rather than <-**

```
!{voucher} := IfRef$Operation( args )
...other code executes concurrently here...
```

  - **redeeming uses <-**

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

Inside Initiate and Redeem

- As you may have guessed, Initiate and Redeem simply use the `instruct_Fork` and `instruct_Join` functions...
- ... Initiate forks a new thread to execute the invocation, and Redeem joins with it
- `Redeem` is not a real operation (although it has the same syntax); it is a PREPC statement
- Vouchers do not need to be redeemed in the same order that they were initiated

Effect of Initiate and Redeem

- An initiate and redeem is equivalent to...

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

- ... but is easier to understand
  - application does not have to manipulate threads

Trying out Initiate and Redeem

- **Modify the *Echo* client**
  - **it currently invokes operations...**
  - **... change it to initiating several operations on the same (or different) Echo servers**
- **Remember that you may need to allocate extra tasks to avoid deadlock**

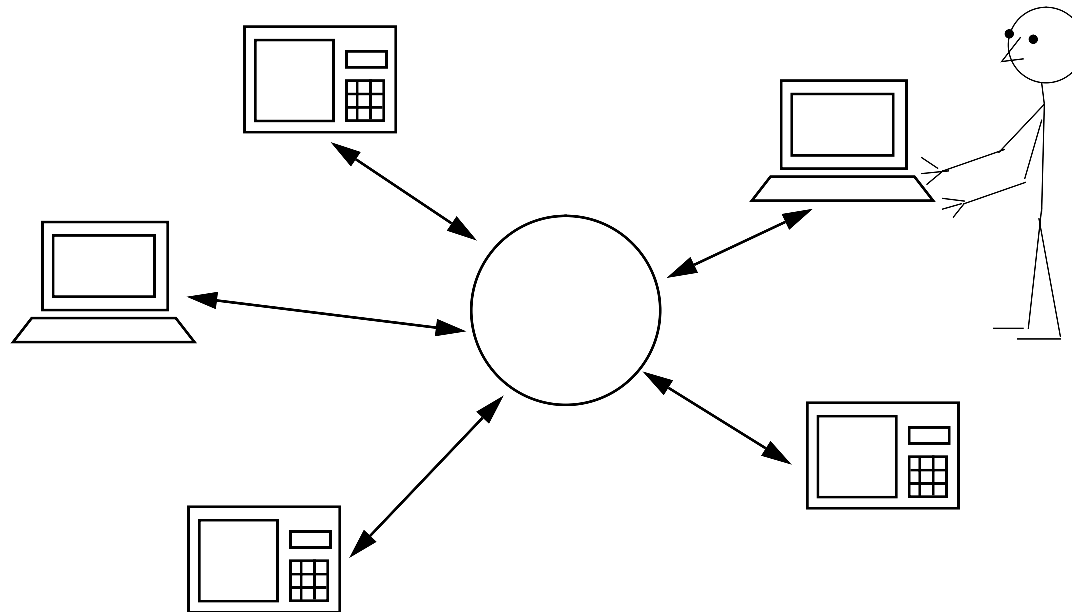
Summary

- **Concurrency can give more efficient use of resources**
- **Concurrency requires synchronization to avoid interference**
- **Use concurrency only if you need it**
- **For more information:**
  - **on ANSAware concurrency, see *Application Programming in ANSAware*, Chapter 3**
  - **on event counters and sequencers, see *Reed and Kanodia - Communications of the ACM 22(2), Feb. 1979***
  - **on database concurrency, try *Chris Date - An Introduction to Database Systems, Volume II***
  - **study the source code of the mutex macros to see how they use eventcounts and sequencers**

Concurrency - Extra information

- **ANSA has a concurrency model**
  - **see *Using Path Expressions as Concurrency Guards* (TR.022.00)...**
- **... but ANSAware does not implement it**
- **ANSA Phase III work is concentrating on dependable real-time concurrency**
  - **there will be a version of ANSAware supporting real-time guarantees**
  - **Sponsors can look at this work**

Simple Bank - An Example Application

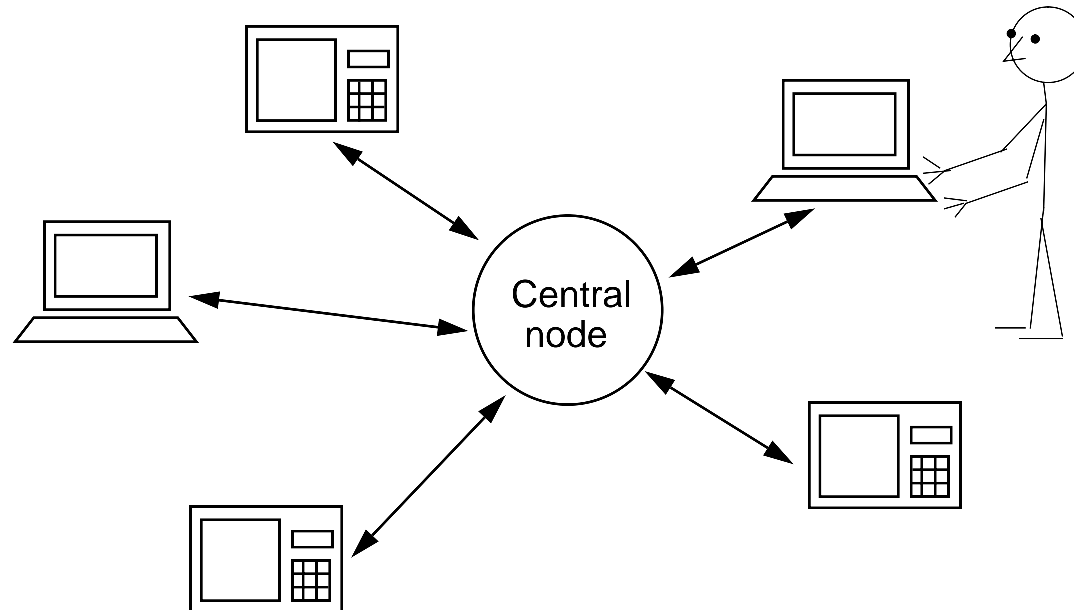


Simple Bank Example

- 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 administer bank accounts; they too are distributed around various local branches

ANSA

Simple Bank Example



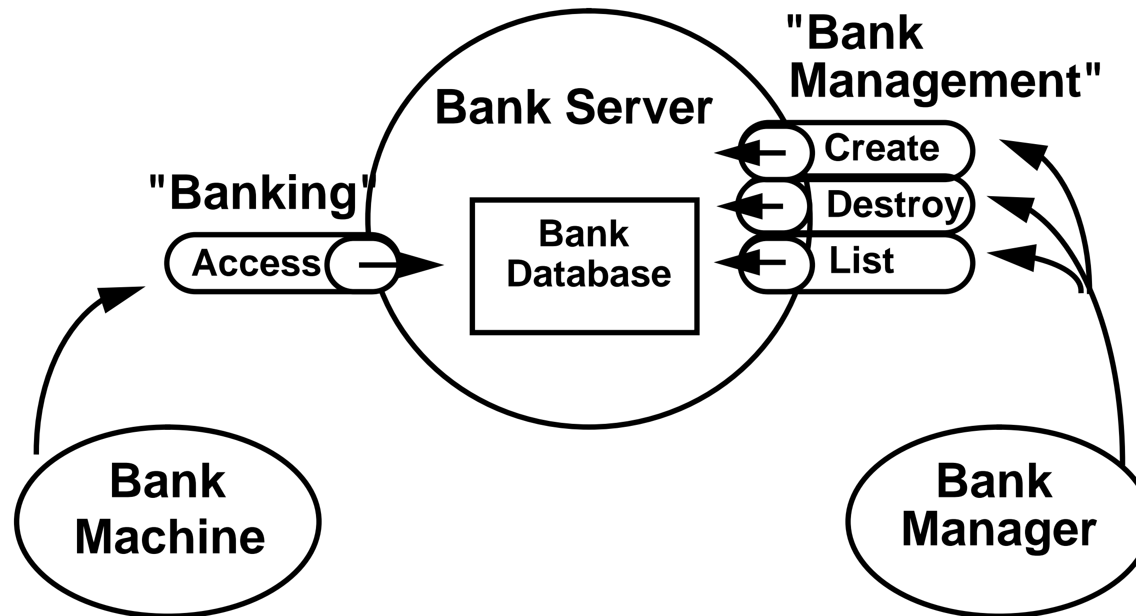
Simple Bank - Information Requirements

- **What information is required for each account?**
  - Customer name
  - Account number
  - PIN (Personal Identification Number)
  - balance
  - time of last change
- **Who accesses the information, and with which operations?**
  - Customer access via ATM (guarded by PIN): credit and debit account, list account details (balance)
  - Manager access: create, destroy, and list accounts,

Simple Bank - Dependability Requirements

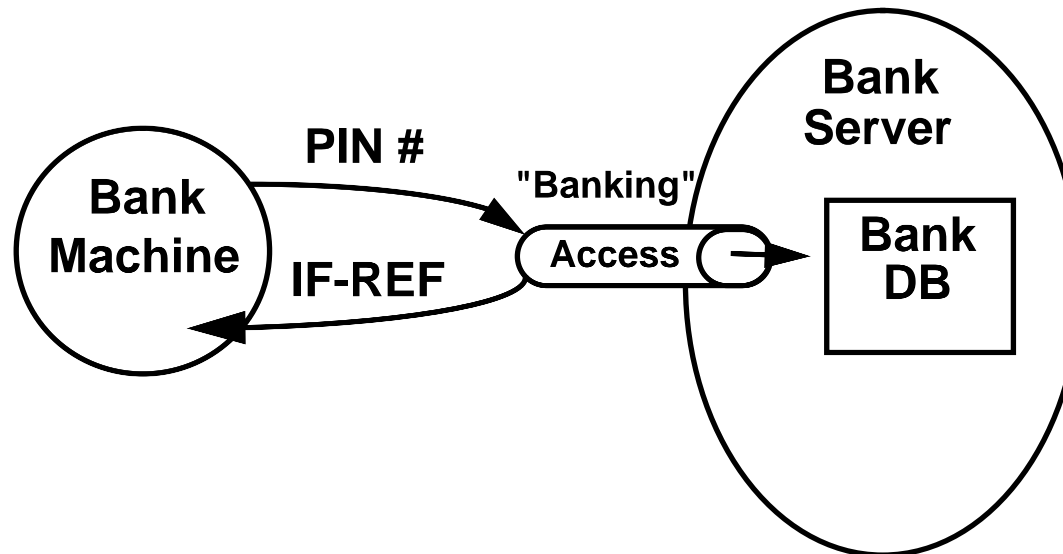
- What failure modes must be handled?
  - Failure of the bank server (central node)
  - Failure of individual ATM's (bank machines), when in the middle of some transaction
- What must be done to cope with these failures?
  - Bank server must checkpoint its state to persistent storage
  - ATM failures should be detected, and any state associated with that ATM recovered or reconstructed. (This failure mode is not handled by this example, but it is explained how to implement it)

Bank Server - Interfaces

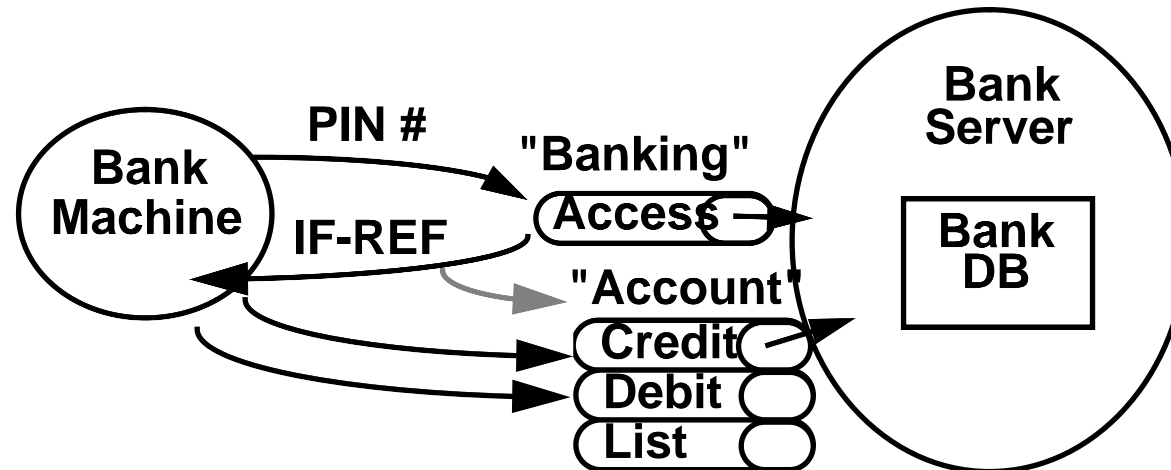


Banking Interface

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



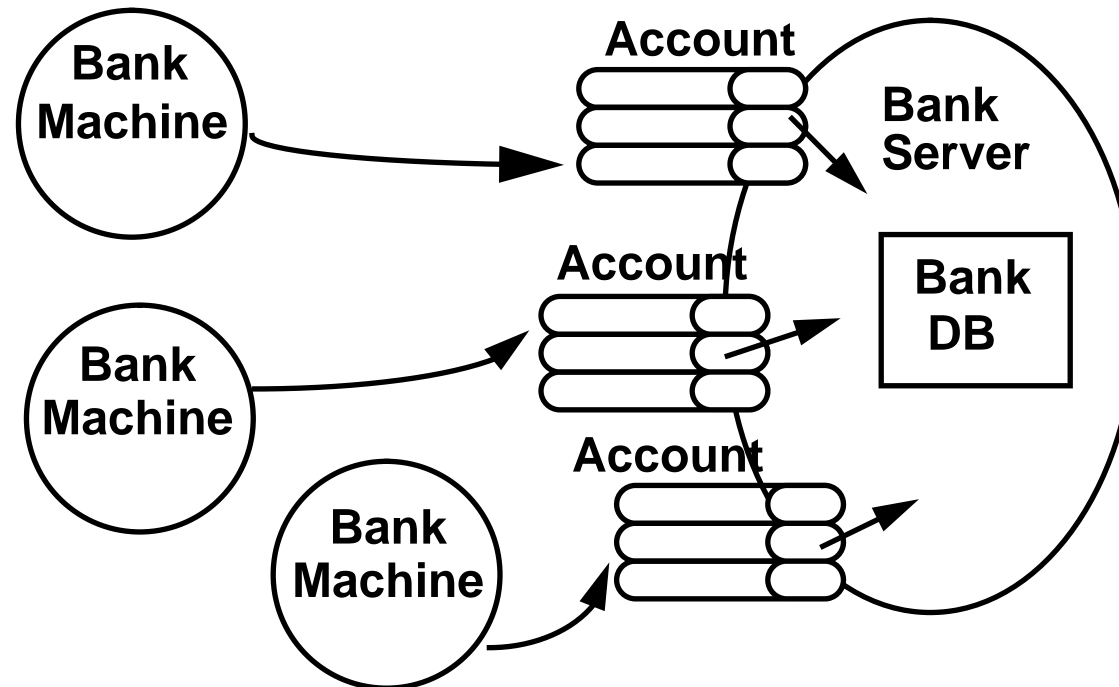
Banking and Account interfaces



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

ANSA

Multiple Bank Machines



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

Bank Server 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

Simple Bank Interfaces

- **Banking (SBank)**
  - for access with PIN by customer via Bank Machine, with PIN
- **Account (Account)**
  - for credit/debit/list by customer via Bank Machine
- **Bank Management (SBankMgmt)**
  - for create/destroy/list by manager
- **Internal (Internal)**
  - for use by the Bank server; described later
- **Common Types (SBankTypes)**
  - types used by all the other Simple Bank interfaces; described later

Banking (SBank) Interface

- **SBank - PIN based access to Account interfaces**

- **Returns an Account interface for the account identified by an account number and corresponding PIN**

- **Equivalent to the user interface offered by a real ATM to a human user**

Banking (SBank) Interface - IDL Definition

```
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.
```

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

- So there is no need to pass account numbers as arguments to the operations in the Account interface

Account Interface - IDL Definition

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

Bank Management (SBankMgmt) Interface

- **SBankMgmt - management operations for:**
    - **creating an account**
    - **destroying an account**
    - **listing an account**
    - **listing all accounts**
- **Note the use of an IDL sequence to return a variable amount of data when listing all accounts**

Bank Management (SBankMgmt) Interface - IDL Definition/1

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

Bank Management (SBankMgmt) Interface - IDL Definition/2

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

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

Bank Management (SBankMgmt) Interface - IDL Definition/3

```
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.
```

Common Types (SBankTypes) Interface

- SBankTypes - data types used by the other Simple Bank interfaces
- No operations defined in this interface
- Included in other interfaces using the `NEEDS` statement

Common Types (SBankTypes) - IDL Definition/1

```
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;
```

Common Types (SBankTypes) - IDL Definition/2

```
AccountRecord: TYPE = RECORD [

      owner: STRING,

      balance: REAL,

      lastaccess: STRING ];

  END.
```

Internal interface

- **Internal - an internal interface used by the Simple Bank server**

- **Operations of this interface are Announcements**

  - **used to spawn background activities**

Internal interface - Operations

- `StartTimer`: **starts up a thread that periodically checkpoints the server's state**

- `DestroyAccount`: **destroys an Account interface-instance**

  - **destroys the specified `Account` interface-instance once it is no longer in use**

  - **needed because a thread cannot destroy the interface-instance within which it is executing**

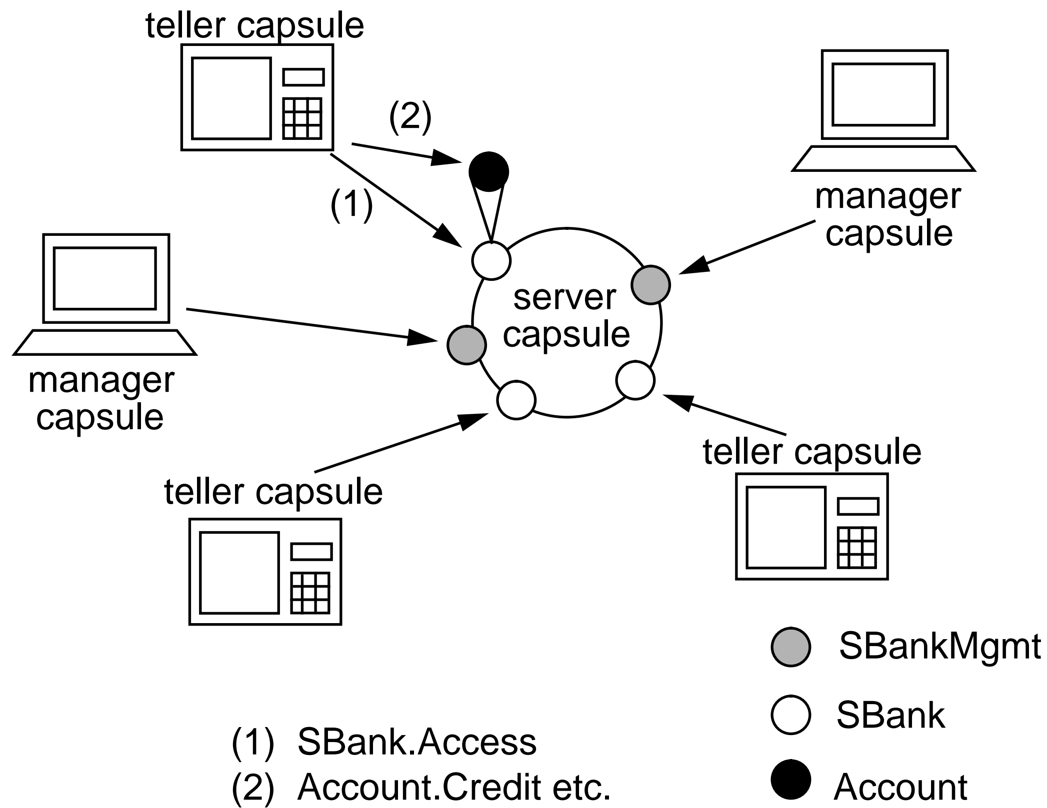
  - **`the Account` interface `Destroy` operation calls this Internal interface (via announcement)**

Internal interface - IDL Definition

```
Internal: INTERFACE =

BEGIN

   StartTimer: ANNOUNCEMENT OPERATION [] RETURNS [];


   DestroyAccount : ANNOUNCEMENT OPERATION [

               ref: AccountRef    ]

         RETURNS [];


END.
```

ANSA

Capsule Structure



Implementation - IDL Source Files

- **One IDL file for each interface previously described**
  - `SBank.idl`: **Banking**
  - `Account.idl`: **Account**
  - `SBMgmt.idl`: **SBankMgmt**
  - `Intern.idl`: **Internal**
  - `SBTypes.idl`: **SBankTypes**

Implementation - Server capsule

- **Source files**

  `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

- **Can compile server with `-DDEBUG` to enable some debugging output**

Implementation - Teller and Manager capsules

- `teller.dpl`: **teller client capsule**
- `manager.dpl`: **manager client capsule**

Data Structures - Account Record

```
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 - Interface Record

```
typedef struct ifrec {

   int index;                        /* index into accounts */

   unsigned long accno;              /* account number */

} IFRec;
```

Server - Initialization functions

- During initialization, the Server must first
  - create extra tasks for the interface-instances
  - initialise the 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 (that is, read in the checkpoint file)

Server - Interface instantiation

- ... Then instantiate the interfaces
  - instantiate the Internal interface
  - start the timer thread by invoking the `Internal$StartTimer` operation
  - 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)
- And then it waits for service requests
  - on all of its interfaces

Server - creating new accounts

- This is done with the 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
- fill in fields of the array element with client-provided arguments
  - customer name, account balance, etc.

Server - creating `Account` Interface Instances

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

- 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

- `acct_index` is the index into the array of account records for the specified account.

- `acct` is the account number for the specified account.

Server - associating Accounts and Interfaces

- **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 Simple Bank

- **This is the same procedure as for other applications:**

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 Simple Bank - Creating accounts

```
% ./server &

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

Using Simple Bank - Listing accounts

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


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

Using Simple Bank - Error handling

```
% teller 1 6263 debit 400.00

Debit(400.00) failed, reason: InsufficientFunds

% teller 1 6264 list

Access(1, 6264) failed, reason: InvalidPin

% teller 3 6263 list

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

Using Simple Bank - Destroying accounts

```
% manager destroy 2

% manager listall

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

Using Simple Bank - Shutting down the server

```
% sbshut.sh

% ./server &

% manager listall

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

Exercises

- **Identify your bank service**
  - **Export with a specific property value, for example:**

    `"BankName 'MyBank PLC'"`

- **Change the teller and manager client programs to use only your bank service,**
  - **Import with a corresponding property constraint, for example:**

    `"BankName == 'MyBank PLC' "`

- **Implement the Account interface**
  - **its Credit and Debit operations**

Checkpointing

- Checkpointing functions are in the file *state.c*
- two checkpoint files: master-file and update-file are set to:

```
<your-path>/SBMaster

<your-path>/SBUpdate
```

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

Completing the Account Interface

- You have an incomplete `Account.dpl`
- It has been generated using `stubc -t`
  - this generates a template containing function skeletons
- The `List` operation definition is already written
- You need to implement the `Credit` and `Debit` operations

Completing the Account interface - operations

- **Each operation in the Account interface has the same 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`

Summary

- **Simple Bank demonstrates several advanced ANSAware features:**
  - **multiple interfaces**
  - **storage and checkpointing**
  - **concurrency**
  - **dynamic interface instantiation**
  - **announcement operations**
- **This example is described in Chapter 6 of *Application Programming in ANSAware***

Simple Bank Example - Extra information

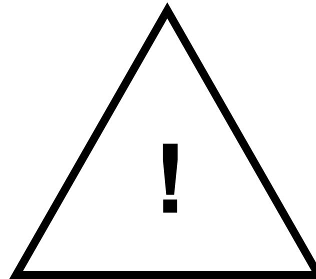
- There is an X Graphical User Interface to Simple Bank
  - provides the Bank Machine user interface only
- Like the other X tools, it is not built and installed by default
- See Appendix C of *Application Programming in ANSAware*
  - xsbteller(1)

Simple Bank - Handling Client Failure

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

ANSAware 4.1

Exception Handling



In this session

- **Explain how to handle errors that are detected by the ANSAware infrastructure**
- **Explain the difference between ANSAware exceptions and ANSA terminations**
- **Explain how to use ANSAware exceptions safely**

Operations and Terminations

- **The ANSA Computational Model allows operations with multiple terminations**

- **Remember the operations on the BankAccount:**

  Credit (amount : Integer) : (newBalance : Integer)

  Debit (amount : Integer) : (newBalance : Integer)

  - ... but the balance may be insufficient!

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

- **There is one Debit operation with two possible outcomes; two separate *terminations***

- **ANSAware does not support terminations**

Terminations and Exceptions

- **ANSAware IDL declares an operation with this syntax...**

```
... OPERATION [ arguments ] RETURNS [ results ]
```

- **... it only allows a single set of results**

- **ANSAware supports *exceptions* rather than terminations**

- **Exceptions are handled in PREPC language rather than IDL**

What is an exception?

- **An exception is an error condition detected by the ANSAware infrastructure**
  - **for example, IllegalOperation indicates that an operation was invoked on an interface which it does not support**
- **Exceptions just have a name; no arguments**
  - **they are an invocation status value**
- **Exceptions are predefined; you cannot declare new ones**
- **Exceptions are modelled on C *signals***
  - **unlike C signals, they operate predictably in the presence of concurrency**
  - **C has no other means of passing back errors from the run-time system**

PREPC - Exception syntax

- **The ordinary syntax is...**

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

- **... but you can also state how exceptions are to be handled**

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

- **The `exception-list` has the following syntax...**

```
Continue statuslist Abort statuslist Signal statuslist
```

- **... it allows you to give a list of exceptions to Continue, Abort, or Signal**

Handling with Continue, Abort, or Signal

- **Continue**
  - carries on executing
- **Abort**
  - program is aborted: an error message including the exception details is printed
- **Signal**
  - A signal function (an exception handler) is called...
  - ...with status, invocation arguments, and invocation results as parameters
  - based on these, the exception handler can take appropriate action
  - for example, Simple Bank prints out "Failed to Import Bank Service"

ANSAware signal functions and C signal handlers

- **Although modelled on C signal handlers, ANSAware signal functions are different**
  - **they integrate with the ANSAware infrastructure and distribution mechanisms**
- **C signal handlers may interfere with ANSAware**
  - **Do not use C signal handlers in ANSAware programs**

PREPC - Example exception list

- **The following exception list will**
  - **Continue on ok**
  - **Abort on everything else**

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

- **Use * to indicate "any other status values"**

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

- **The exception-list is optional**
  - **PREPC default behaviour if no exception list given**

Signal function declaration

- **The name of the signal is determined by the names of the interface and the operation**
  - **If operation-call is as follows...**

```
! DECLARE {ifref} : IfName CLIENT

...

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

  - **... then signal-function must be declared as**

```
Signal_IfName_OpName(status, arg1,...argN, res1,...resN)
```

PREPC pseudo-operations

- **PREPC $Import and $Export statements are not operations...**
  - they are not operations declared in IDL
  - they are pseudo-operations
- **... but they can have exceptions**
  - handled with a PREPC exception list
  - with a signal function if needed
- **The names of the signal function are fixed**
  - they are Signal_Prepc_Import and Signal_Prepc_Export, respectively

Returning from the Signal Function

- The signal function can return one of three return-values
  - `ExceptionContinue`
  - `ExceptionAbort`
  - `ExceptionRetry`
- Code generated by PREPC will check this return-value
  - to see how to proceed on returning from the Signal function

Communicating between invocation and signal function

- The thread that invoked the operation may need to know whether an exception occurred
  - the invocation may have decided to Continue...
  - ... and the invoking thread has to tidy up afterwards
- But signal functions cannot directly access the data of the calling thread
- Special macro-definitions are provided for this
  - thread_setExceptionCode(ansa_Cardinal code)
  - ansa_Cardinal thread_getExceptionCode()
- `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

Signal Function - Basic approach

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

}
```

Signal Function - Example

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

   /* Check that no exception occurred on Access operation. */
   if ( thread_getExceptionCode() == 1 )
   {
      bank_state == BANK_OUT_OF_SERVICE
      return( OpFailure );
   }

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

More complex communications

- **To communicate more than an exception code between signal function 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`

Initializing the exception state

- **The exception code and exception state are always initialized**
  - **initialized with `thread_setExceptionCode (0)`...**
  - **`... and threadSetExceptionState` ((`ansa_StatePtr`) 0)**
  - **this is done by PREPC-generated code before each invocation that could generate an exception**
- **So applications should not initialize them**

Exceptions - Default behaviour

- The default behaviour (if no exception-list is given) is roughly...

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

- ...but the ANSAware default signal function will be called if any of the “Signal” status-values occur
  - it will try to *relocate* the interface
  - the default signal function is called `Signal_binder_relocate()`
  - it is provided by the infrastructure

Relocation in the default signal function

- `Signal_binder_relocate()` **tries to relocate the interface**
  - **by contacting any locator services it knows about**
- **if the interface is registered with the Trader**
  - **it will have information about Trader's `Relocator` interface, and will call it**
  - **this will check whether it knows a new location for the faulty interface...**
  - **... it may have moved**

Effect of relocation attempt

- if relocation succeeds
  - the interface-reference will be updated to a new one...
  - ...`ExceptionRetry` is returned from the default signal function
  - ... the ANSAware infrastructure will retry the invocation
- if relocation fails
  - otherwise, `ExceptionAbort` is returned from the default signal handler...
  - ... which causes the program to be aborted with the message

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

- In either case, nothing happens to the faulty service itself
  - This is relocation, not migration
  - There is no support in ANSAware yet for migrating a service

Using relocation from a signal function

- If an exception list is given, the default signal function will never be called
  - so relocation will never happen
- To use relocation with your own signal function
  - just call the Signal_binder_relocate function from your signal function

Exceptions -Summary

- **ANSAware does not support terminations**
  - **use a ENUMERATION as a discriminator as one of the results, instead**
- **ANSAware supports exceptions for error conditions detected by the infrastructure**
  - **can rely on default behaviour**
  - **can use signal functions to handle specific error conditions**
  - **can use relocation in case an interface has moved**

Exceptions - more information

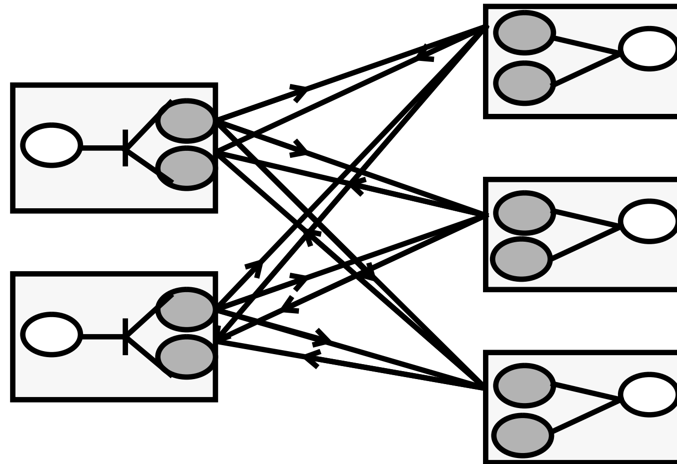
- For more information on exceptions, see Chapter 3 of *Application Programming in ANSAware*

- For a complete list of exceptions (invocation status values), also see Chapter 3 of *Application Programming in ANSAware*

- To briefly review the ANSA Computational Model, see Chapter 2 of *Application Programming in ANSAware*

ANSA

ANSAware 4.1



Using Interface Groups

In this session

- **Explain how interface groups implement replication transparency**
- **Explain the protocols used to support interface groups**
- **Explain the limitations of their implementation in ANSAware**

The purpose of groups

- **Interface groups are a construct that help you build *dependable* objects**

- **To be dependable, objects must be reliable (fault-tolerant). Reliability is achieved by replication**

  - **must keep at least two copies; this is called *redundancy***

- **Redundancy can also improve performance**

ANSA interface groups

- **Groups are defined in the ANSA Architecture**
  - **they are consistent with the computational model and its interface type system**
- **Groups are groups of interfaces, not of implementations**
  - **they make the *service* reliable, not the implementation**
  - **all members of a group must conform to the same interface type**
- **Groups need to be configured flexibly**
  - **there are mechanisms and policies to tailor them**
- **Groups manage themselves**
  - **external managers enforce the client policy (for instance, the minimum number of members of a group)**

ANSA group configurations

- **Groups can use active or passive replication**
  - **active replication to reduce recovery time**
  - **passive replication to reduce overheads**
- **Groups may need to satisfy various requirements**
  - **distribution**
  - **collation**
  - **consistency**

Group membership

- **The membership of a group is dynamic**
  - **members may join**
  - **members may leave**
  - **members may fail**

Groups and Singletons

- **An ordinary interface is called a *singleton* interface**

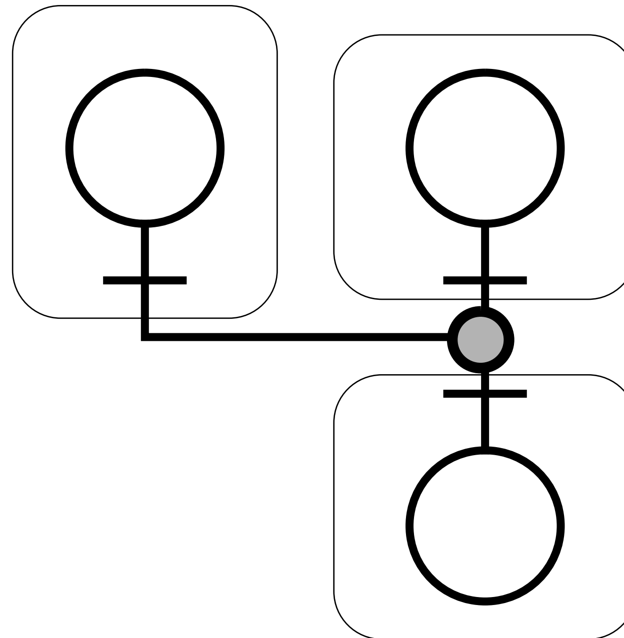
- **But, in theory, groups are the general kind of interface...**

  - **singletons are the special case of a group with 1 member**

ANSA

Replication transparency

- **Replication Transparency**
  - **application need not know how many copies**



  - **application only sees a single interface**

Transparency of groups

- The use of an interface group is almost transparent to the client
  - they appear in the Trader just like a singleton interface
  - they are invoked just like a singleton interface
  - the difference is largely handled by the infrastructure
- Where possible, membership needs should be defined in the IDL for the interface
- The use of an interface group is not quite so transparent to the members
  - but it is transparent at the application level

Invocation Ordering

- **All members must process invocations from clients in the same order**

  - **interrogations from same activity are (inherently) sequenced**

  - **interrogations from different activities and announcements are independent**

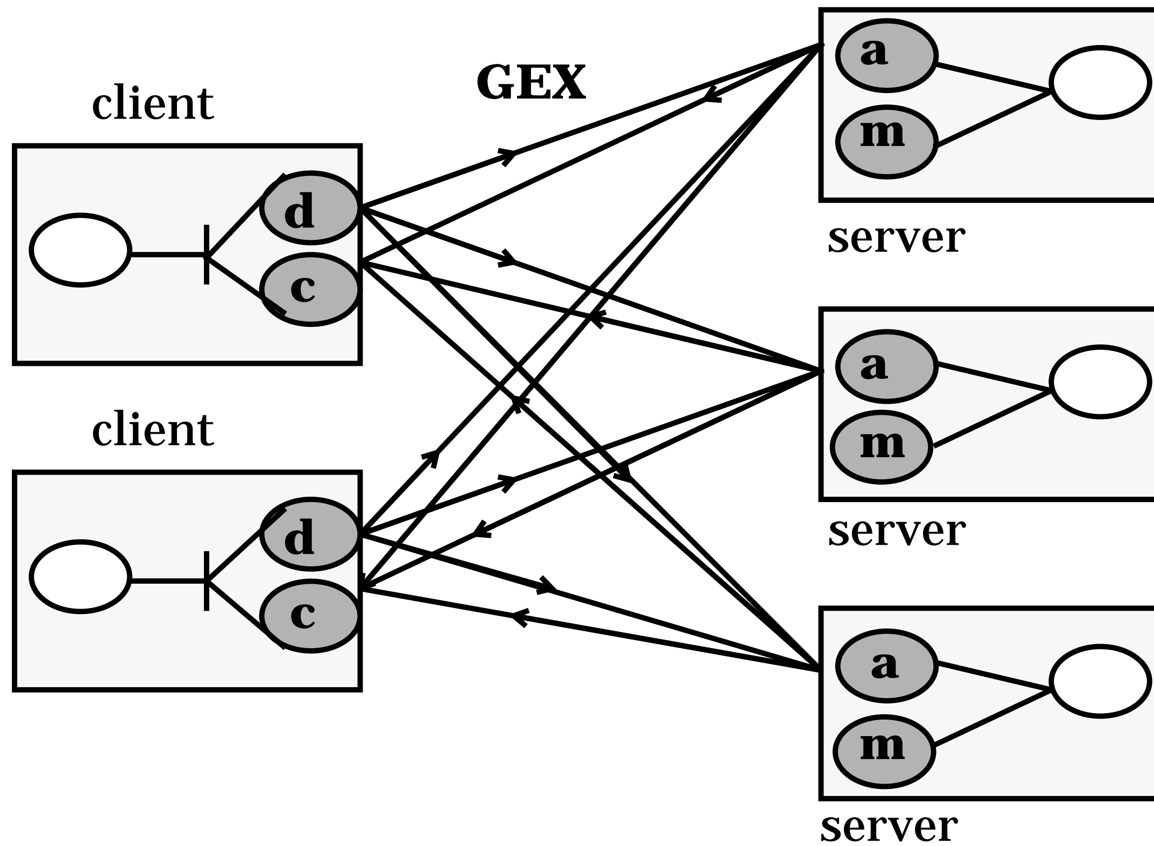
GEX and REX

- **A special group execution protocol (GEX) is incorporated within the ANSAware nucleus**

  - **it uses the standard ANSAware remote execution protocol (REX) to communicate with the members**

- **The standard group interaction mechanisms are used for population management and recovery**

Groups in ANSAware - Interfaces



Groups in ANSAware

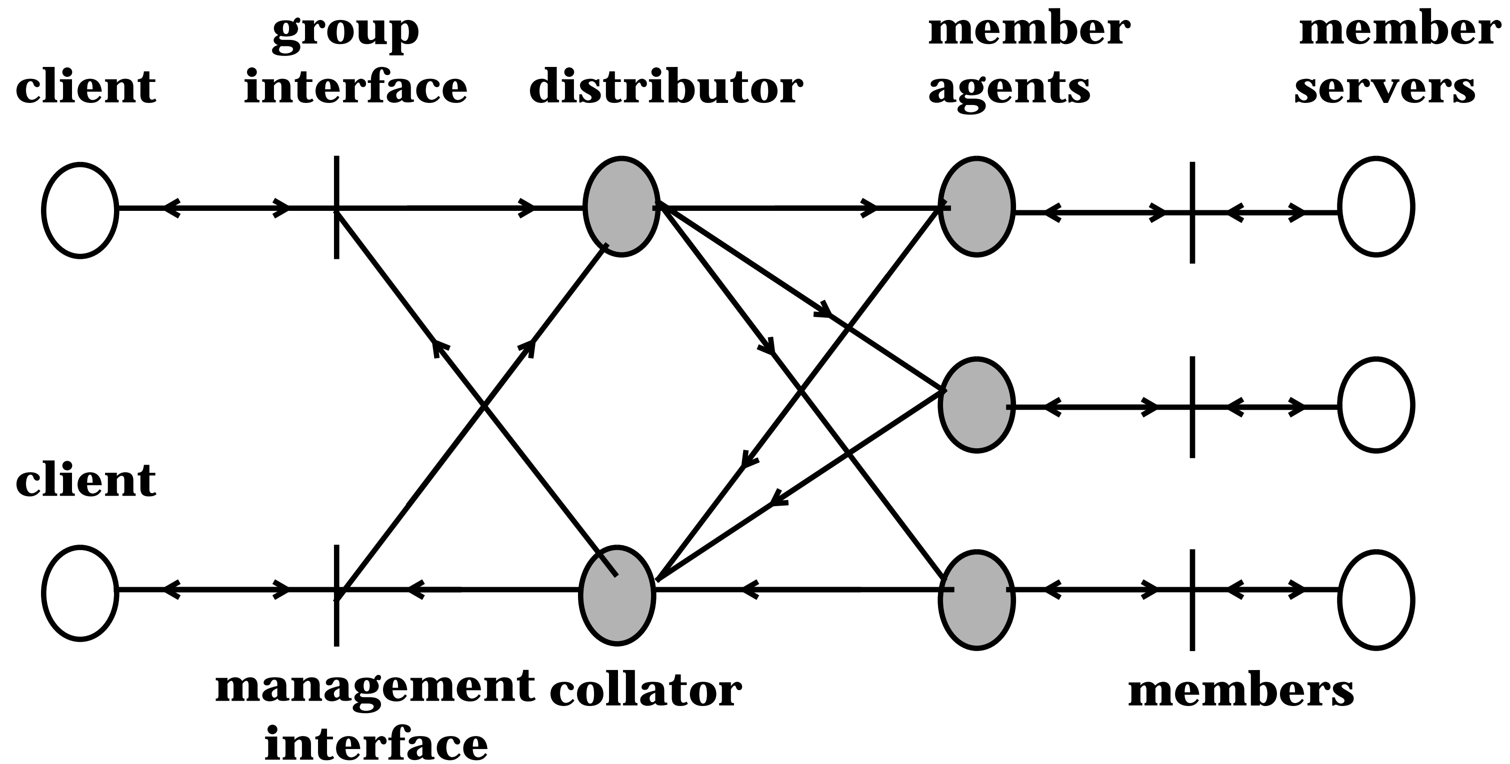
Internal Group interfaces

- Each member of a group is created via a Factory
- The group itself has two interfaces
  - the group interface: communicates between members, synchronizes them, checks whether they have failed
  - management interface: initializes the group, changes its population
- These two interfaces separate...
  - ...the service of a group
  - ...the management of a group’s membership

Components of a group

- **distributor**
  - **broadcasts invocations on group interface to all members**
- **member agent**
  - **co-operates with other members to ensure...**
  - **... no invocation have been missed (co-operation)**
  - **... all invocations have been processed in the same order (ordering)**
  - **... failed members are detected (failure detection)**
- **collator**
  - **collects each members result to produce single result for client**

Components of a group



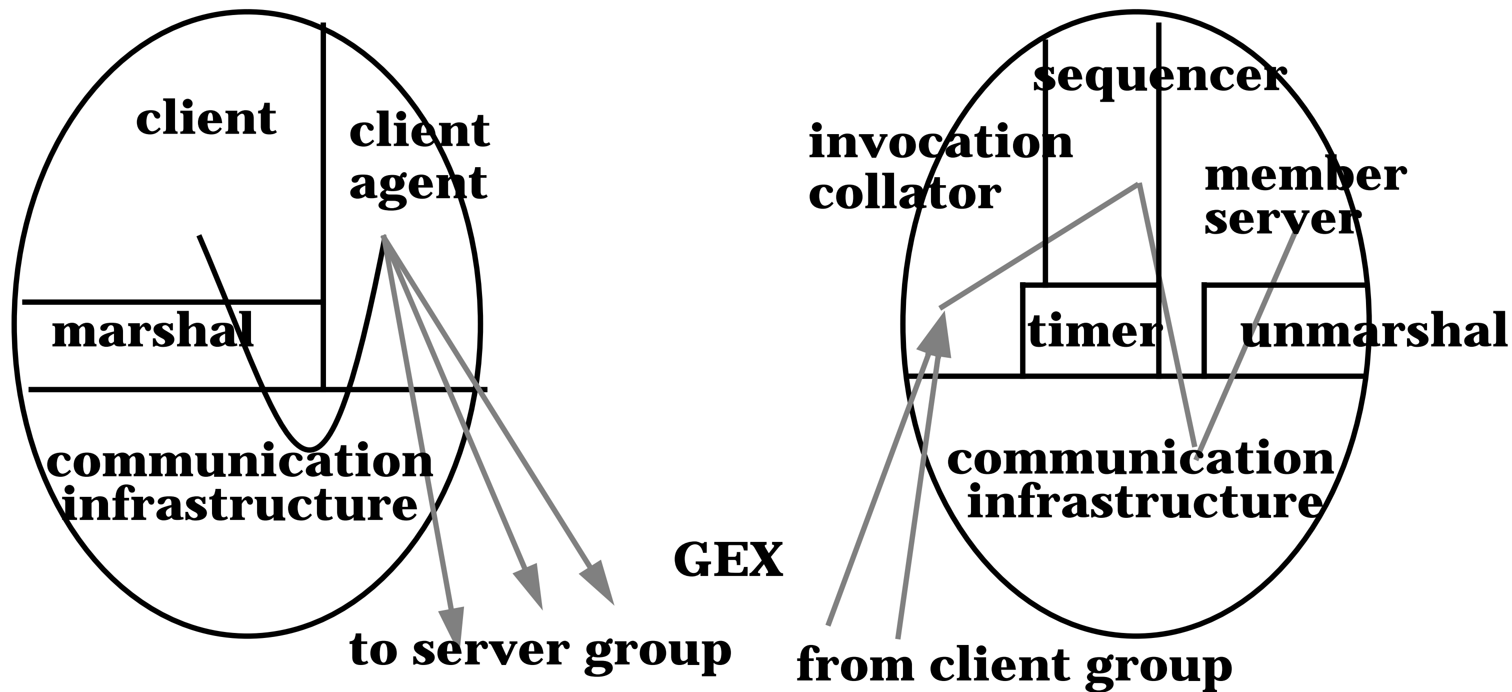
Groups - Implementation of clients

- **For clients:**
  - **arguments are marshalled by stubs in the usual way**
  - **arguments are passed to dispatcher function**
  - **dispatcher passes to distributor**
  - **distributor broadcasts invocation to each member (non-blocking RPC)**
  - **control passes to collator which awaits response**
  - **when all results received, one result is returned to client invocation**

Groups - Implementation of members

- **For members:**
  - **invocation collator: ensures all invocations arrive**
  - **invocation sequencer: for agreeing order of processing**
  - **timer: expiry indicates member failure**

Groups - client and server Infrastructure



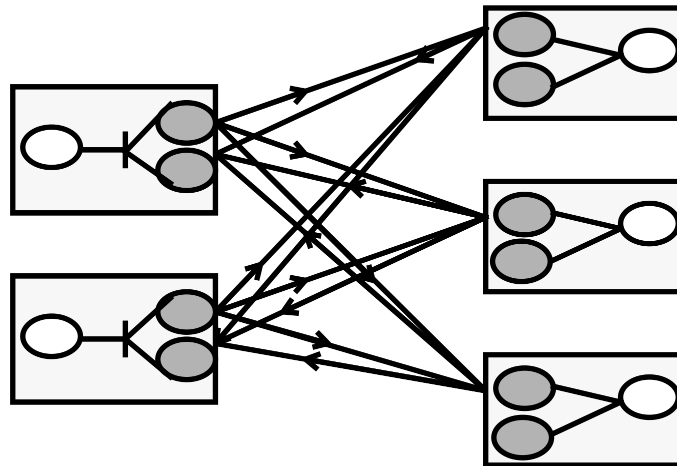
Group Management in ANSAware

- **A group management tool (gc) manages group membership**
  - **for population control**
  - **to initialise first group member**

Experimental results

- Replication transparency really works
- Full non-selective replication transparency is possible
  - we made it work...
  - ...but it was difficult to achieve
- Group management and configuration are central problems
- Insertion of user mechanisms and policies requires much work
- Performance is disappointing
  - multicast protocols would help...
  - .... fragmentation remains a problem

The N:M problem



- **Need to reduce the number of messages**

  - **use a multicast protocol**

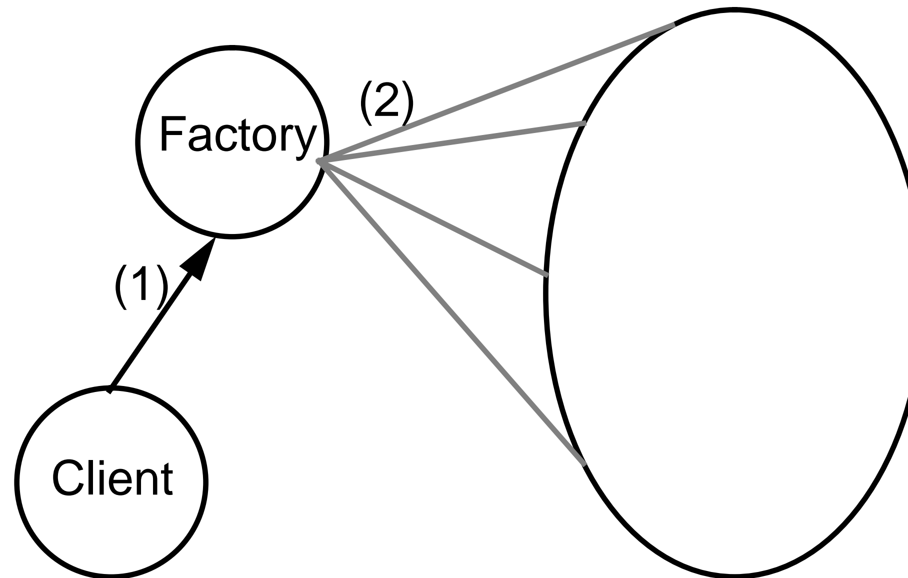
Implications for dependability in general

- **Universal solutions will often be unacceptable**
  - unacceptable performance
  - unacceptable cost
- **The application must participate in dependability provision**
  - to exploit semantic knowledge
  - to relax synchronization requirements
  - to set the configuration, protocol parameters, synchronization mechanisms and policies, and state synchronization
- **The infrastructure should provide a framework for constructing dependable applications**
  - use standard components...
  - ... or provide your own

Groups - summary

- **Replication transparency uses special mechanisms to make sure the group members are consistent**
  - **for instance, it may use multi-point channels and special protocols**
- **Implementing replication transparency efficiently is difficult**
  - **it may need information from the application**
  - **it is under active research in the distributed systems community**
- **For more information, see *A Model for Interface Groups* (AR.002.01)**

ANSAware 4.1
Using the Factory service



Speaker Notes
Exercise at end, so lots of details included for referring to when doing exercise. [Is there?]
This really needs diagrams showing how all the *IDL* interfaces are related

In this session

- **Explain the use of the Factory service**

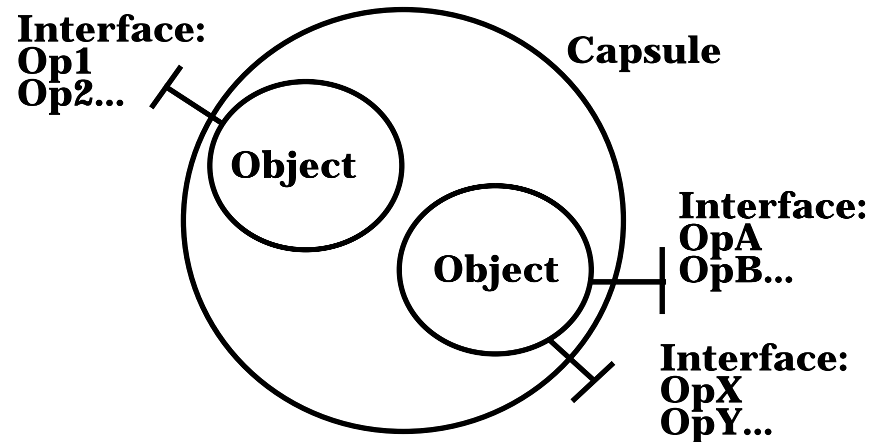
- **Show how the Factory service supports the dynamic creation of capsules, and objects and interfaces within them**

- **Explain the use of the Factory command-line tools**

ANSA

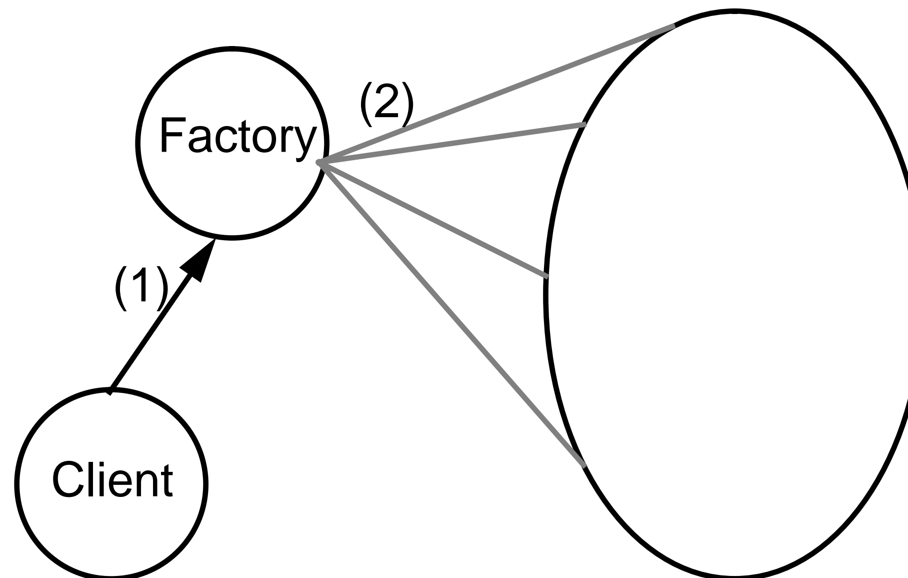
Factory services

- **Factory services (called *Factories*) provide a service for:**
  - Creation/destruction of capsules
  - Simple monitoring of the capsules it creates
- **Once created, capsules then provide a service for:**
  - Creation/destruction of objects within a capsule
  - Creation/destruction of interfaces within an object



ANSA

Capsule Creation

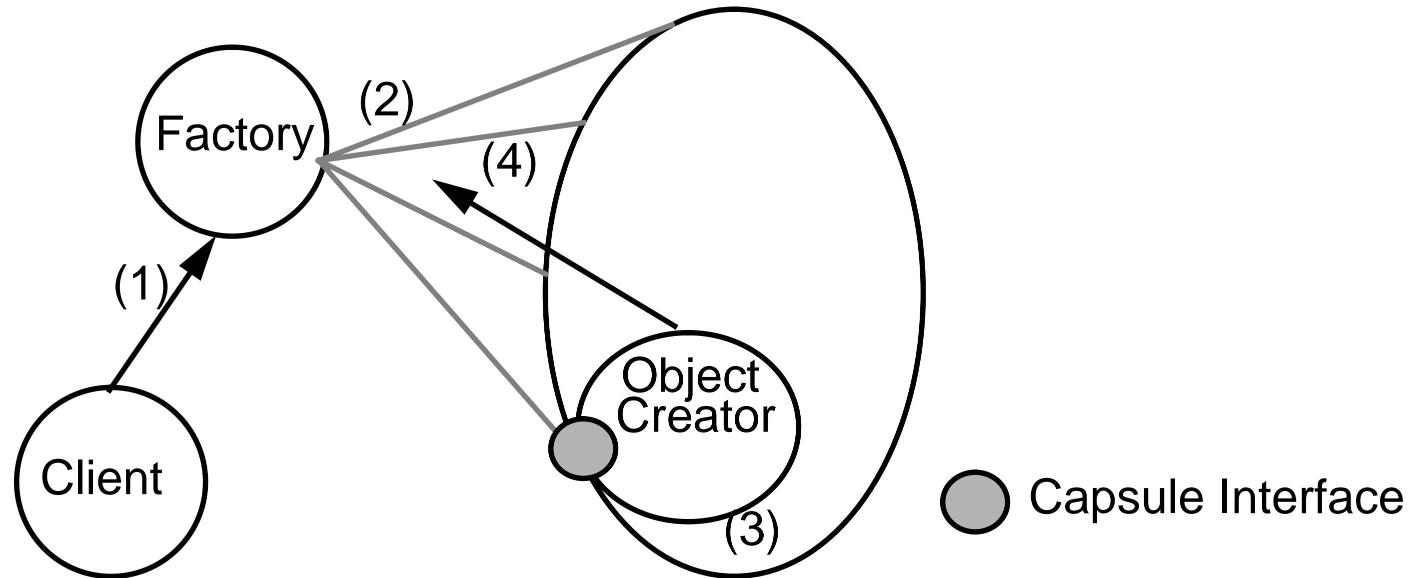


1. **Client asks factory to Instantiate a capsule**

2. **Factory instantiates a new capsule**

ANSA

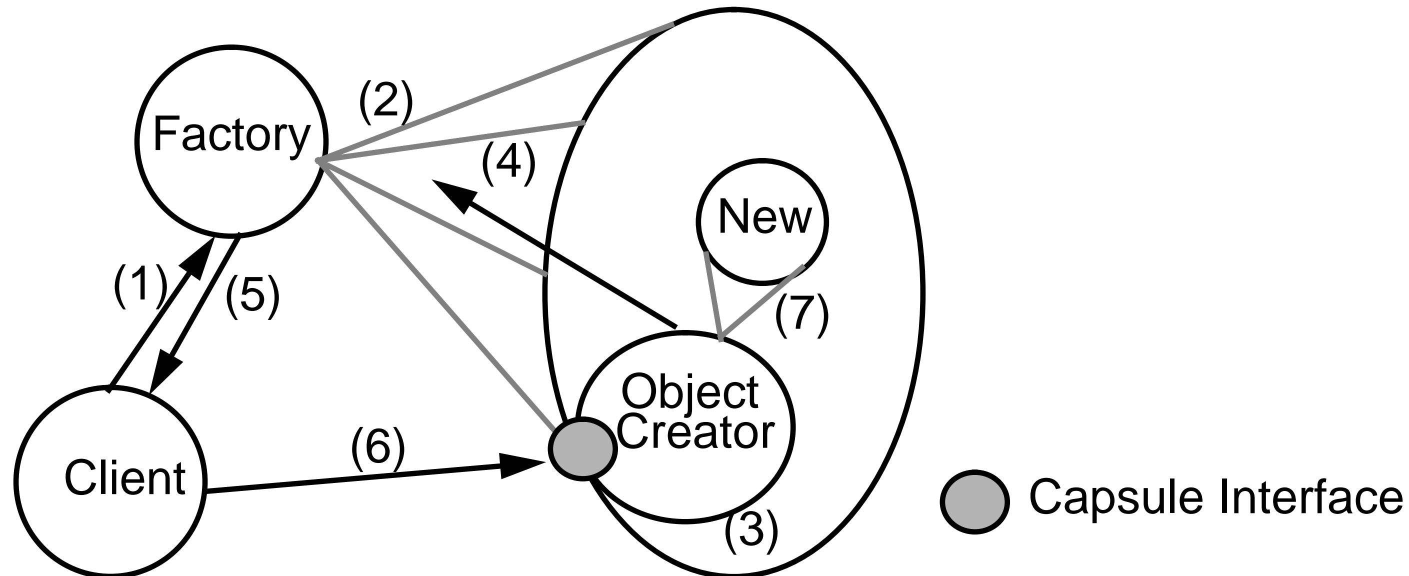
Initial Object Creation



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**

ANSA

Interface Creation



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 /1

```
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 the Capsule interface of the created capsule - more shortly
        Cid : ansa_CapsuleId ]; process id of capsule
```

Factory Interface/2

```
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 ];
```

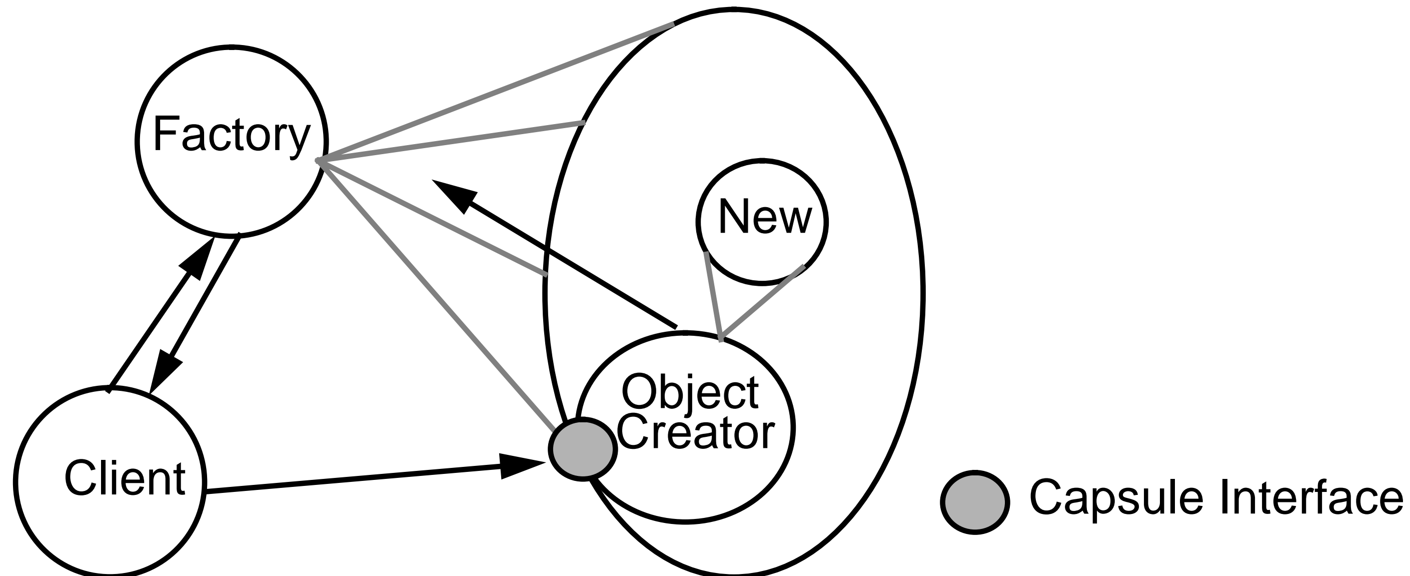
ditto this one

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

used to kill capsules created by Instantiate above

```
END.
```

Capsule Interface Usage



- **Every capsule contains a Capsule interface, provided by the ANSAware infrastructure**
- **This is used to instantiate objects within that capsule**

Capsule Interface

```
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 - a simple client program/1

```
! 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
```

Using the Factory - a simple client program/2

```
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 );
! {factRef} <- traderRef$Import( "Factory", "/", "")
! {capRef, cid} <- factRef$Instantiate("", "fooserver",\
                                       "", "", nullRef, 0)
! {objRef, res} <- capRef$Instantiate( nullRef, nullRef,\
                                       "Foo", "", "")
...
```

InstantiateResult res; — this data structure is returned by Capsule$Instantiate (prev slide)

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

Using the Factory - a simple client program/3

```
/* 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

```
}
```

Dynamic Services

- **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")**
- **This is the *object constructor* of the ANSA Engineering Model**

Capsule structure for Object Creation

- A `! MANAGED` PREPC statement is required

  `! MANAGED ObjName`

  - the *ObjName* is a name you choose for this template
  - this name determines the name of Create and Destroy functions for each such object

- `Create_`*`ObjName`*`_Object` is invoked by the Capsule interface
  - when its `Instantiate` operation is invoked with an object name of *`ObjName`*.

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

- The Capsule interface is supported by all capsules

The Create and Destroy functions

- **Create_... function**

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

- **Destroy_... function**

```
ansa_Boolean Destroy_ObjName_Object ( state )
   ansa_StatePtr state;
```

Steps in creating an object

- **The Create_*ObjName*_Object function must:**
  - **1. use arguments and environment (if any) to decide what action is required**
  - **2. instantiate any interfaces required**
  - **3. allocate and initialise any object state required (explained later)**
  - **4. return any state and set up results (of type `InstantiateResult`) - a sequence of interface references to instantiated interfaces**

InstantiateResult type defined in Capsule interface

Multiple objects in a capsule

- **Capsules can support multiple objects**
  - **effectively, they are templates for creating interfaces**
  - **the *ObjName* specifies the object to be used**
- **The 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**

The Object Interface

```
Object : INTERFACE =

BEGIN

    Terminate : OPERATION [] RETURNS [];

END.
```

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

Terminating Objects

- **When this Terminate operation is invoked...**

```
!   {} <- objRef$Terminate()
```

as shown in example client, earlier (page 410)

- **...the user-provided Destroy_*ObjName*_Object function is called**
- **Destroy_*ObjName*_Object should**
  - **destroy any interfaces**
  - **free any object state set up by Create_*ObjName*_Object**
  - **(Object state is discussed below)**

Simple factory-startable service/1

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
```

Simple factory-startable service/2

```
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)
```

inst of Object i/f - will describe this shortly

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

Simple factory-startable service/3

```
ansa_Boolean Destroy_FooObj_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**
  - usually based on arguments and/or environment
  - it can be initialised by Create_*ObjName*_Object function
  - it is easily accessible from Destroy_*ObjName*_Object function
- **Object state can be anything; it is application-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;
}
```

Using object state

- Object state is often used for keeping track of interface instances

  - To record whether interface-instance has been exported to Trader

  - If so, Destroy_...() function can Withdraw the offer from Trader before destroying interface

  - the earlier example could have used this technique

  - the next example does

ANSA_MANAGED_EXPORT

- **ANSA_MANAGED_EXPORT is an environment variable**

- **By convention, it is used to indicate whether the interface should be exported to the Trader**

- **The frun command sets this environment variable**

    - **this is shown later**

Factory client program - using object state/1

```
! 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
```

Factory client program - using object state/2

```
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")
```

Factory client program - using object state/3

```
/* 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 /1

```
#include “ansa.h”
#include “tFoo.h”

#define PROPSIZE 1024
char pbuf[PROPSIZE];

! MANAGED Foo

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

Factory-startable server program/2

```
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

Factory-startable server program/3

```
ansa_StatePtr Create_String_Object(ac, av, envp, results)
int ac;
char *av[], *envp[];
```
these params corresp to the last 2 args of the cap$Inst call

```
InstantiateResult *results;
{
ObjState *p;
```
need pointer to our state-type

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

Earlier example used system_allocate rather than malloc - why?

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

Factory-startable server program/4

```
  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;
}
```

ANSAware Factory

Factory-startable server program/5

```
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 would have own state which it would point to

Capsule termination

- **Here is the Factory interface 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 argument to provide an interface-instance which is to be invoked when the capsule terminates**

The Terminated Interface

```
Terminated : INTERFACE =
BEGIN

   CallerData: TYPE = CARDINAL;

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

END.
```

- **The 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

Factory client example - using Terminated/1

```
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;
```

Factory client example - using Terminated/2

```
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

```
...
```

Factory client example - using Terminated/3

```
/* 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

```
}
```

ANSAware Factory

Factory client example - using Terminated/4

```
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 tools

- **Two command-line tools are provided to create and destroy capsules**

  - **frun**

  - **fkill**

Creating a capsule using frun

- The command is:
  - frun node template object arguments environment [client args ...]
  - the arguments and environment are passed to the object constructor/template
  - ... for example

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

- The capsule id of the created capsule is displayed
- frun sets `ANSA_MANAGED_EXPORT` before instantiating capsule to make sure any interface-instances will be exported
- if the `client` argument is given, it will be started as a sub-process,
  - ...`args` will be passed to it
  - ...when `client` terminates, `frun` will terminate the object, then terminate the capsule

Destroying a capsule using fkill

- **The command is**
  - **fkill host cid**
  - **... for example**

```
fkill "" 15307
```

This means run on current node (the default)

  - **cid is the capsule id as displayed by frun**

- **You can only destroy capsules that were created dynamically**

Factory availability

- Factories can only create capsules on nodes that support capsule creation
  - DOS does not support capsule creation
- But the frun and fkill commands can be run from DOS
  - to access a factory on another node

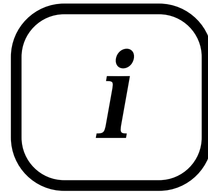
Summary

- **Factories are used to create capsules**

- **Capsules create objects and interfaces**

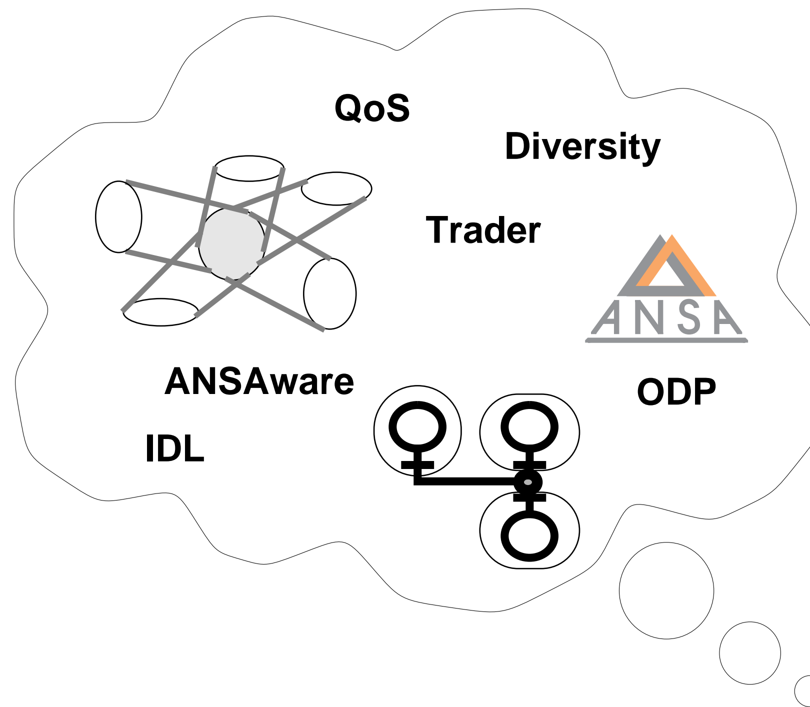
- **The frun command-line tool can be used to create a capsule**

Distributed Processing and Distributed Systems



...Learning more...

New Ideas



More about ANSA?

- **If you want a brief introduction to the Architecture:**
  - ***An Overview of ANSA* (AR.000.00)**
- **For more detail, see the individual Architecture Reports and Technical Reports**
  - **see leaflet in your information pack**
- **ANSA is online on the World Wide Web!**
  - **our URL is http://www.ansa.co.uk/**
  - **... note the final /**
- **E-mail address is apm@ansa.co.uk**

More about ANSAware?

- **The Manual Set contains a lot of detail**

- **E-mail to ansaware@ansa.co.uk**

More about distributed systems?

- **If you want one introductory book on both technical issues and products**
  - ***Distributed Computing: A Practical Synthesis*, by Amjad Umar (Prentice Hall)**
- **If you want one introductory book on the technical issues only:**
  - ***Distributed Systems Concepts and Design*, by Coulouris, Dollimore, and Kindberg (Addison-Wesley)**
- **If you prefer a more theoretical approach:**
  - ***Distributed Systems*, edited by Sape Mullender (Addison-Wesley)**

Latest state of play?

- **Magazines**
  - ***First Class*** **from the Object Management Group (OMG)**
  - ***Network Monitor*** **from Patricia Seybold**
  - **...and the general computer press**
- **Internet newsgroups**
  - **comp.client-server**
  - **comp.object**
  - **comp.unix.osf.misc**

Latest distributed systems research?

- **Journals**
  - **Distributed Systems Engineering**
  - **Internetworking Research and Experience**
  - **IEEE Network**
  - **IEEE Computer**
  - **IEEE Communications**
  - **ACM Communications**
  - **Distributed Computing**
  - **IEEE Parallel and Distributed Systems**

How we can help

- **ANSA*ware***
  - technical support
- **ANSA*works***
  - the annual ANSA conference
- **ANSA*wise***
  - training in distributed systems
- **ANSA*web***
  - consultancy and advice

Stay in touch

with

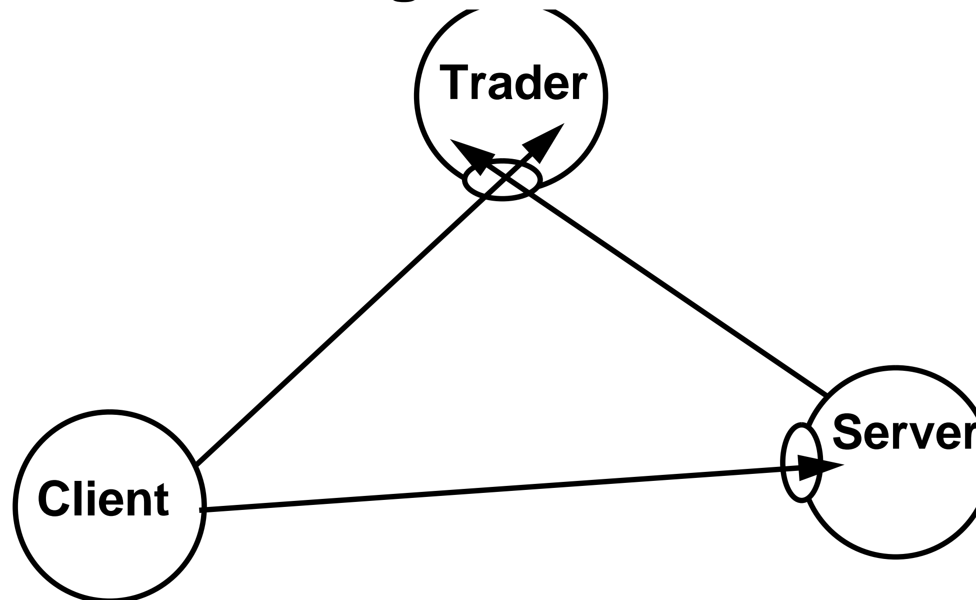
apm@ansa.co.uk

http://www.ansa.co.uk/

ANSA

ANSAware 4.1

Using Federation facilities



Speaker Notes

This module overlaps with the Using the Trader module. All the work with federation facilities should be transferred into this separate module and expanded.

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**

Is there a fedcl? If not, why not?

- **Requirements for federation:**
  - **mutual agreement on common interface types**

Do the administrators have to add the types to all federated traders?

  - **appropriate mapping functions to maintain InterfaceRef guarantees (InterfaceRef's are guaranteed to be unique within a trading domain)**

Is uniqueness the only kind of guarantee? What's a trading domain?

Trader federation techniques

- **Federation is achieved via *naming context* or via *proxy offers***

Are naming contexts the contexts that we covered earlier?

- **Federation via naming context binds a whole *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**

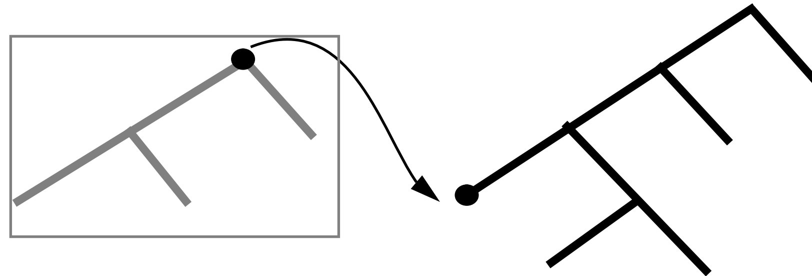
Can you do this within the same trader?

- **Federation via proxy offer binds one *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 called "proxy offers"**

Only lookup requests?

Federation via a naming context

- **Federation via naming context:**



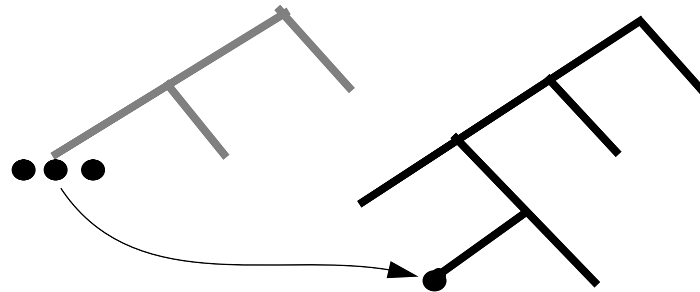
- **Currently you can only federate the root of a context**

  - **like a Unix mount**

In the ANSA computational model you are able to federate any part of the context tree to anywhere, but this is an ANSAware implementation limitation

Federation via proxy offer

- **Federation via proxy offer:**



Is this still federation within the context tree?

- **Can federate any offer to anywhere**

ANSAware Federation - Summary

- **Traders can be federated**

- **Searching**

  - **Searching is ‘shallow’; federated traders are only consulted if the current [local?] trader cannot satisfy the import request.**

What about LOCAL and MASTER traders?

ANSAware 4.1

Using the Node Manager

Speaker Notes

This is just the original Jane Dunlop section on the Node Manager. It needs to be completely revised and matched up with the session on the Factory service.

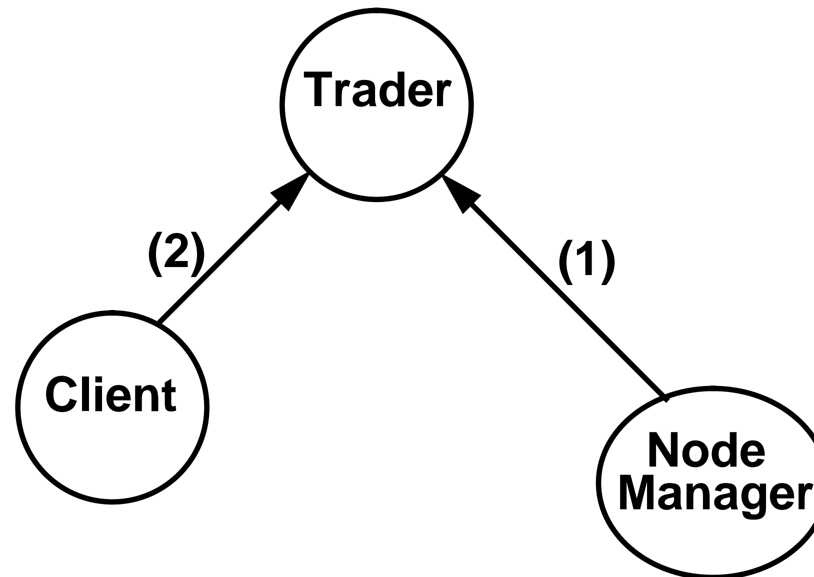
In this session

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)

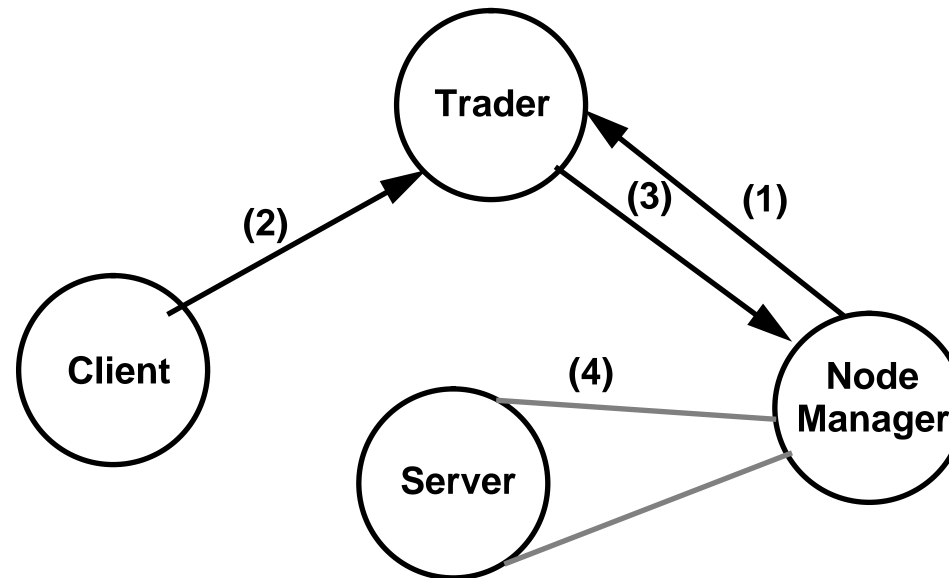
ANSA

Dynamic Service Creation

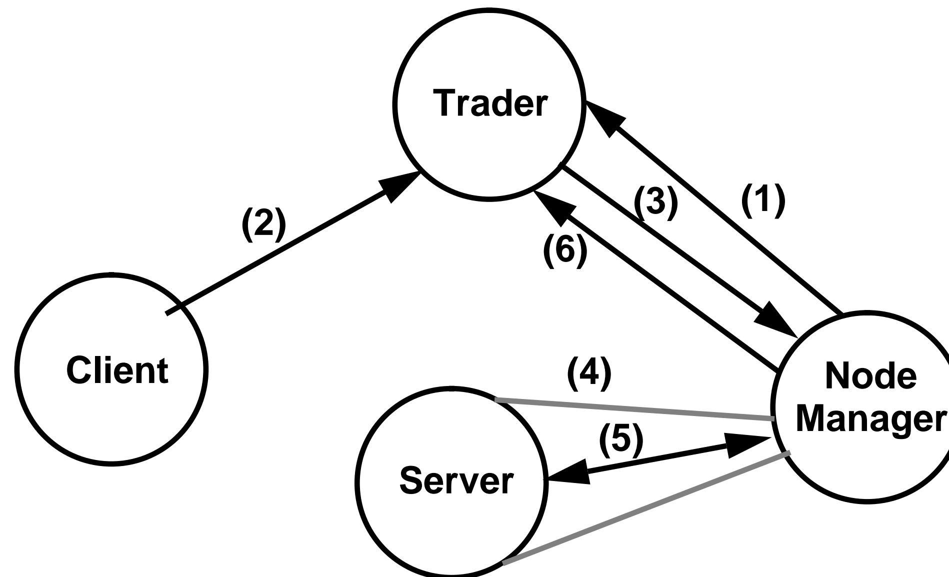


1. Node manager registers a proxy offer with the trader.

2. Client performs an import.



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).

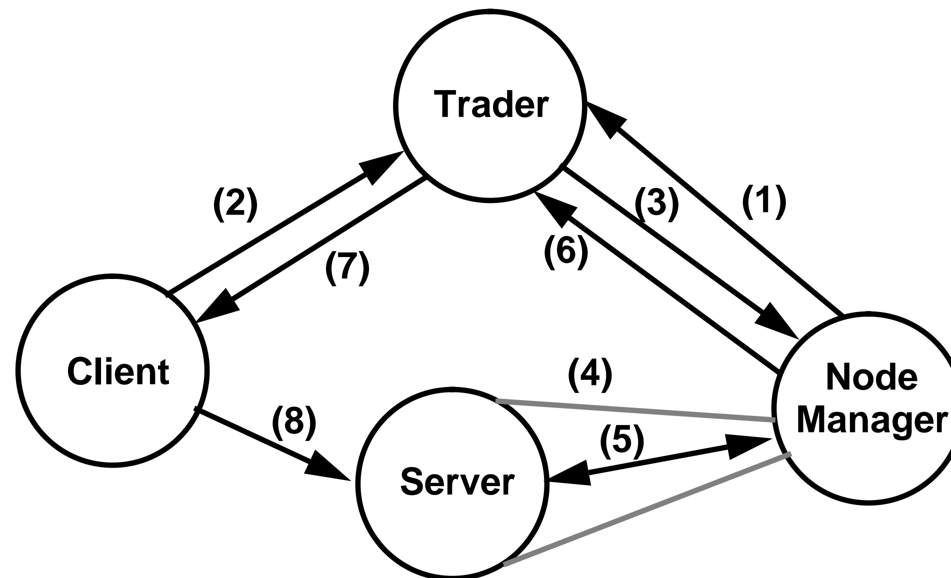


5. **Node manager invokes the Instantiate operation on the newly created capsule's Capsule interface.**

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

ANSA

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**

Summary

ANSAware 4.1

Storage and Migration

Speaker Notes
This is only a roughly formatted version of the original draft Jane Dunlop material. It is technically inaccurate and needs a complete rewrite.

In this session

- **Explain the concepts behind Storage and Migration**
- **Illustrate how these can be used**

Storage and Migration

- **Storage** **of inactive objects/services**
  - **Activation / Passivation**
- **Migration**
- **Location** **of services that have moved / become passive**

The ANSA computational model shown earlier was just basic - work going on in architecture on all of these topics. These new additions to ANSAware should be regarded as experimental.

Object persistence

- **Capsules and objects that are inactive (not in use) could be stored away until needed**
  - **this is called *passivation***

Or deactivation?

  - **passivation saves resources**
  - **but services can still be re-activated when needed**

- **Only activation/passivation/migration/storage of *objects* is available**

No, capsules - see ANSAware manual?

  - **Capsules containing objects need to be dealt with as well**

You need to worry about finding the right executables for capsules

  - **... future work will address this**

[Has this in fact been done?]

Snapshot Service

- a *Snapshot* is a representation of object and 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 (snbase) 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

Snapshot Components

- **Snapshot consists of three components**
  - **object type**
  - **current state of object: bindings, references to interface-instances**
  - **representation of interface: types and instances currently supported by object**

[I don't think this is actually the way ANSAware works?]

- **Restrict snapshot to be of idle object only before 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**

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**

Steps in Passivation

- **An object goes through the following steps:**
  - **It receives a passivation request (via an interface), or decides to passivate itself (for example, because it is idle)**
  - **It waits until current activities have completed, and prevent new activities from starting**
  - **It produce a snapshot**
  - **It stores the snapshot in a SnapshotBase**
  - **in appropriate Locators, it register each interface instance as being mapped to the stored snapshot**

We'll come to Locators in a minute

  - **It terminates**

Steps in Reactivation

- **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 retrieve reference to the Snapshot in a SnapshotBase**
- **Activation steps are then:**
  - **Instantiate a new object via a factory**

Of course something has to be done abt finding/creating capsule to support object - big hand-wave [How is this done in ANSAware?]

  - **pass new object the snapshot reference from the SnapshotBase as an argument to the 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**

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**

? The locator is not a service; it is part of the infrastructure?

Migration

- **Migration is the process of moving an active object from one location in a distributed system to another**

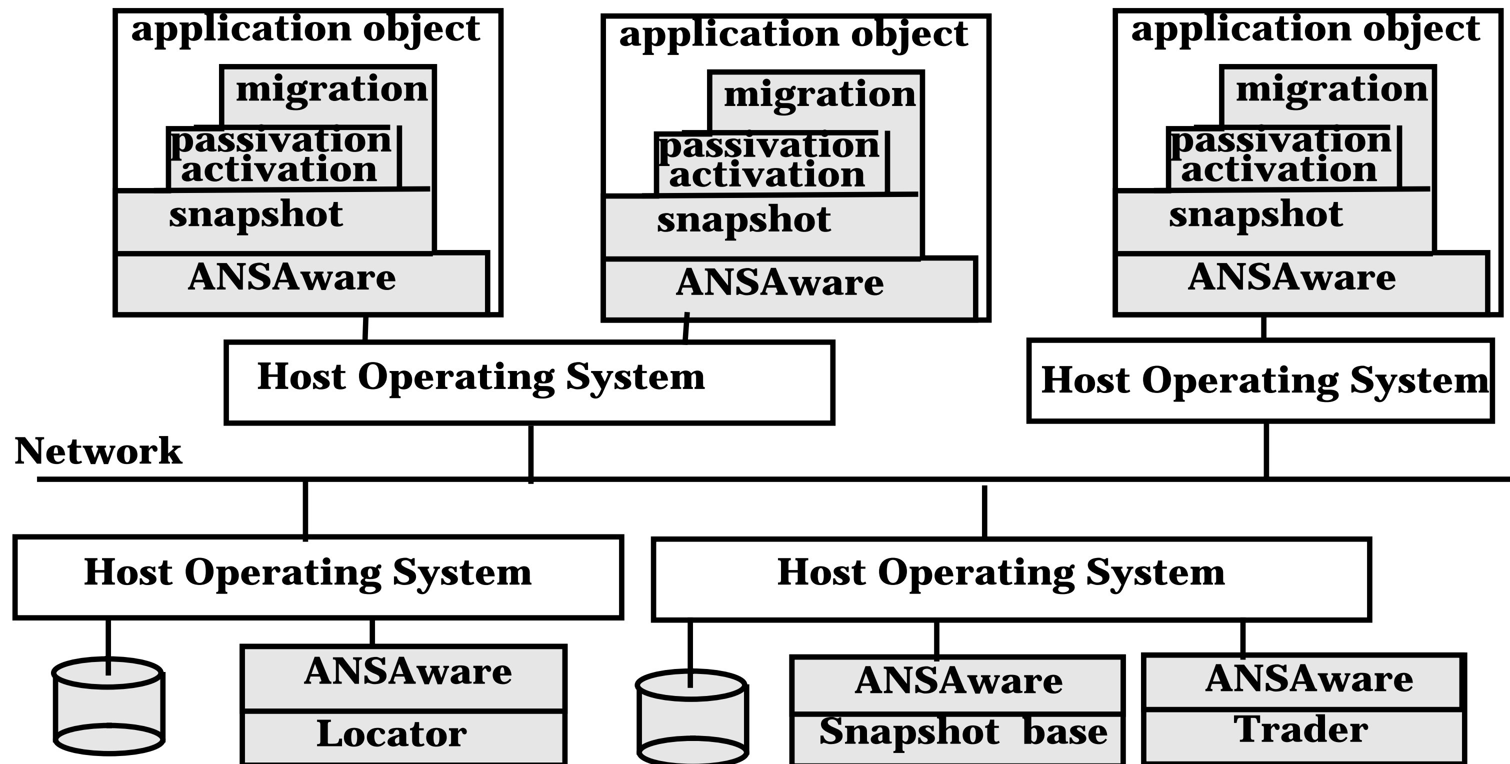
Miigration is managed and requested; it does not spontaneously happen

  - **Migration is needed when a node fails, or has to be switched off for maintenance, and so on**

- **Services of object are temporarily unavailable during migration**
- **Migration is also accomplished via snapshots**
- **Can migrate passive object by activating it at new location**
- **Active object migrates by:**
  - **activating clone of object at new location**
  - **taking snapshot of existing object, and installing into new capsule**
  - **update locators**

- **terminate**

Persistent Object Infrastructure



yes - it's mostly all done in infrastructure

Summary

- **For more information:**

  - **see Chapter 3 of Application Programming in ANSAware**

Only a few pages, but it seems useful

  - **examine the Simple Bank example source code**

Note: AR.006 is The ANSA Storage Model; it appears on the *ANSA Manuals and Reports* flyer as ‘in preparation’, but has never been issued.

ANSAware 4.1

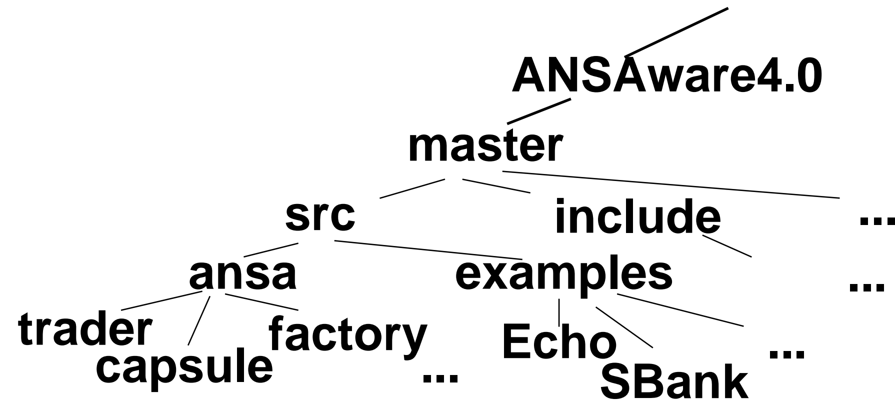
Installing and Configuring ANSAware

This needs considerable expansion and rewriting into several modules (perhaps platform-specific). Also add more technical support information. Should really be a 1-day course in its own right..

Installing

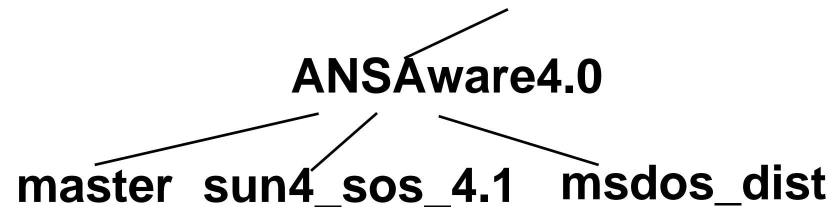
- ANSAware4.0 versions available for Unix, VAX/VMS, MS-DOS
- Unix release contains "master" copy - can build any version from it
- Others contain platform-specific code only
- Once you have read the files onto your system, you will have a master-tree

- **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**

ANSAware 4.1

(Bits and pieces)

All this stuff needs to be reorganized, rewritten, and moved to other (possibly new) modules.

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 initialized 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`

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.