Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

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

ANSA Phase III

Annotated Tour of RM-ODP

ANdrew Herbert

Abstract

A presentation of and commentary on RM-ODP for the OMG/ODP Workshop sponsored by APM and OMG in Cambridge, England, November 1997.

APM.2085.01 **Approved** 21st October 1997
Standards Contribution

Distribution:
Supersedes:
Superseded by:

An Annotated Tour of the Basic Reference Model for Open Distributed Processing

Andrew Herbert
(Editor)
(Technical Director, APM Ltd)

Reference Model for Open Distributed Processing

- Part 1: Overview
- Part 2: Foundations
- Part 3: Architecture
- Part 4: Architectural Semantics

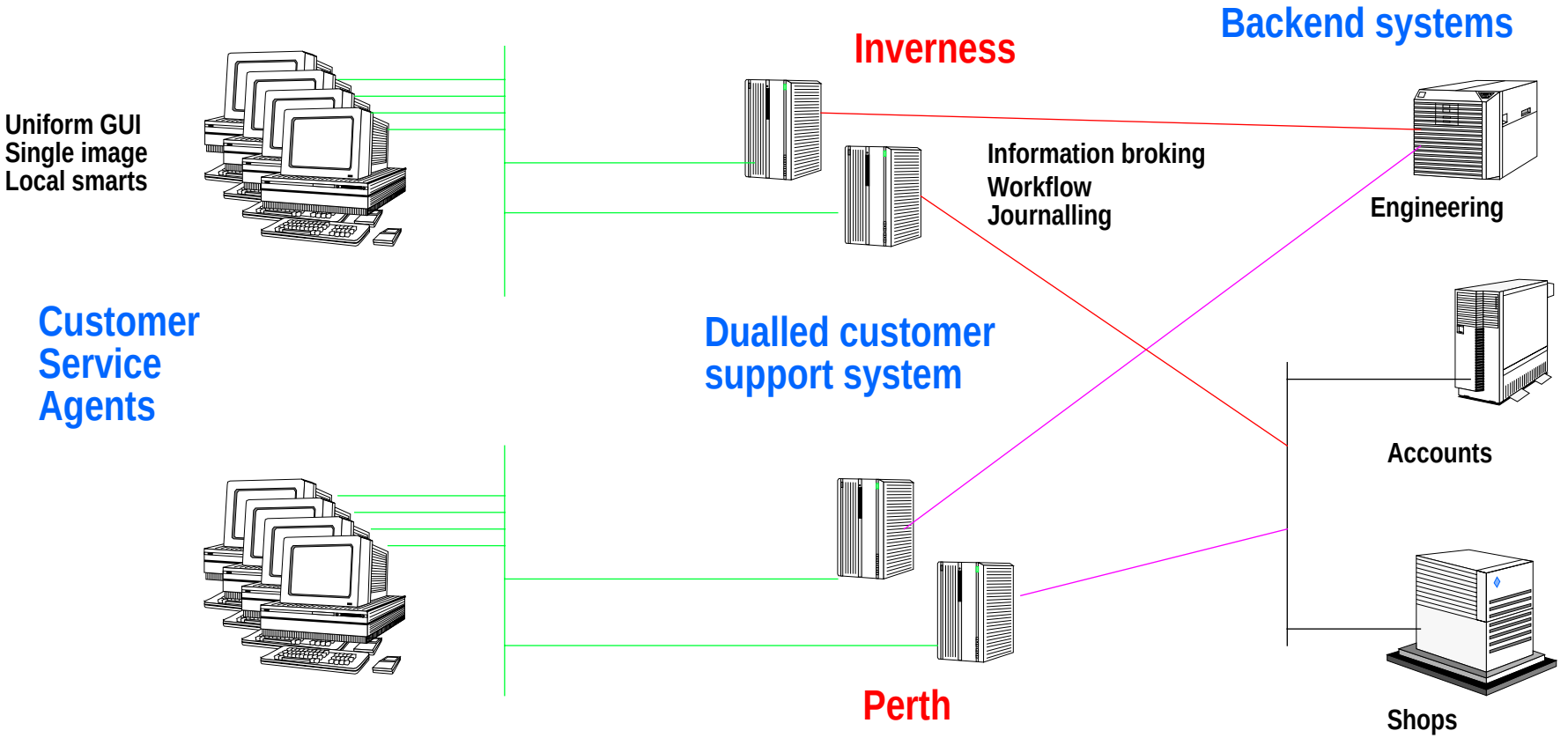
Scope

- concepts and rules for specifying ODP systems
- framework for development of ODP standards
  - standards for *specification*, *modelling* and *programming* languages
  - *language bindings* (APIs) for ODP systems
  - functional components of ODP systems

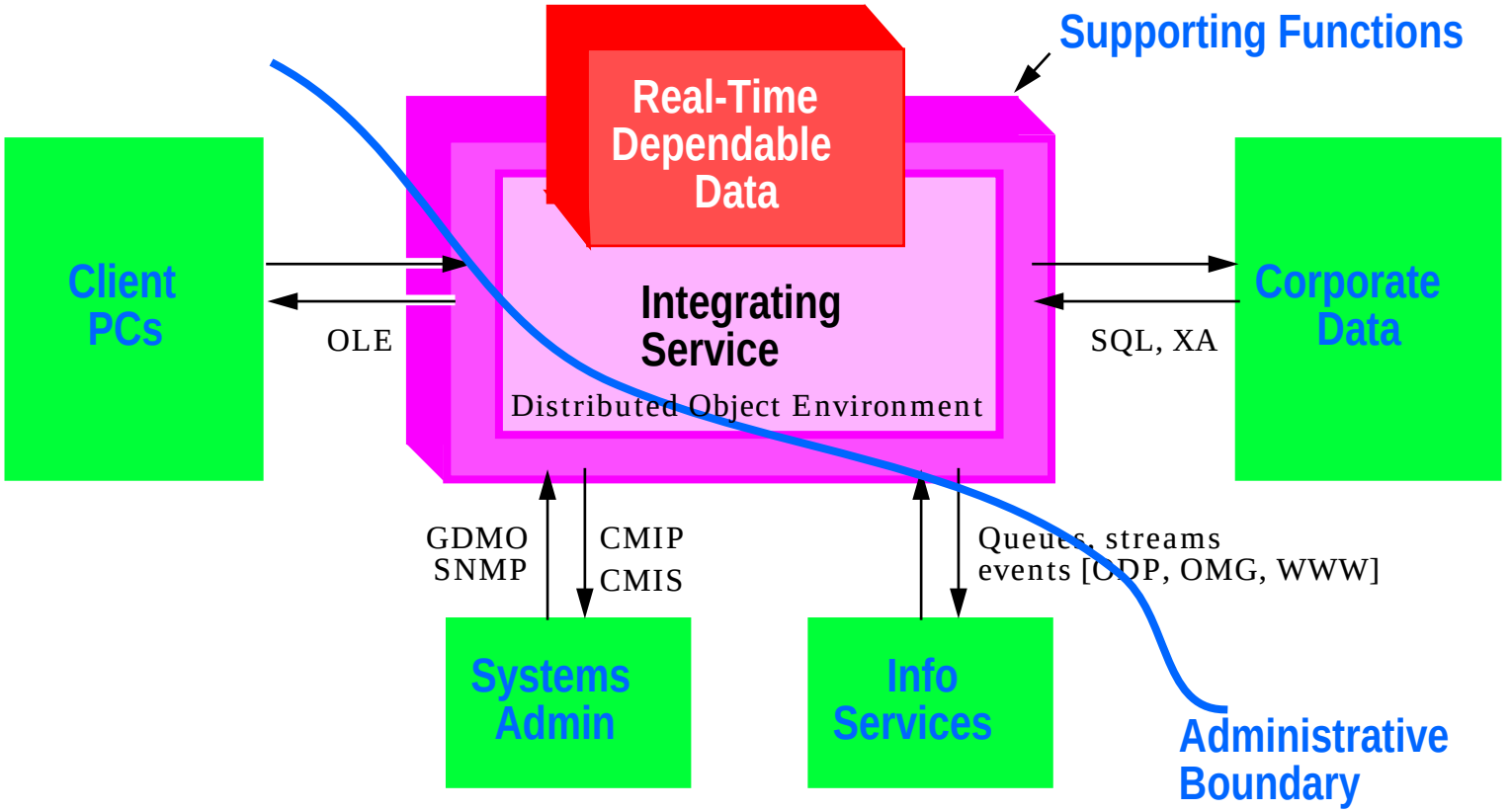
Goals for RM-ODP

- Architecture not design template
  - Tools for describing and comparing system components
  - Rules for design integrity
  - Guidelines for making choices
  - Abstract components and Interfaces
- Acknowledge rather than deny distribution issues
  - Reversed assumptions

An Example ODP System - Scottish HYDRO



Template for an ODP System



Part 2

- To define the architecture we used different styles of object modelling
- Part 2 gives us the kit of parts for
  - explaining each style rigorously
  - making correspondences between the styles
- Reflects realities of OO
  - Programming OO - composition, re-use - “inheritance”
  - Data OO - containers - “properties, relationships”
  - Systems OO - modularity, configurability - “interfaces”
  - Modelling OO - abstraction, description - “specialization, generalization, aggregation”
- Rigour shows desire to enable link to formal expression of the semantics

Objects and Interfaces

- Object: a model of an entity
  - characterized by behaviour or state
  - encapsulated - state change due either to internal action or interaction with its environment - no “action at a distance”
  - process view of objects - “data objects” are objects whose behaviour is “constant”
- Interface: subset of the interactions of an object and constraints on when they may occur
  - each interaction belongs to a unique interface
  - objects can have many interfaces which can come and go over time
  - n.b. “interface between objects” is misdescription of “binding between objects”
- Environment (of an object): The part of the model which is not part the object

Behaviour

- Action: something which occurs
  - internal action of an object
  - interaction between object and its environment
- Behaviour: collection of actions
- Activity: single headed acyclic graph of actions
  - used to describe “threads” with spawn, fork & join actions
- State: condition of an object at an instant in time
  - determines all possible future states

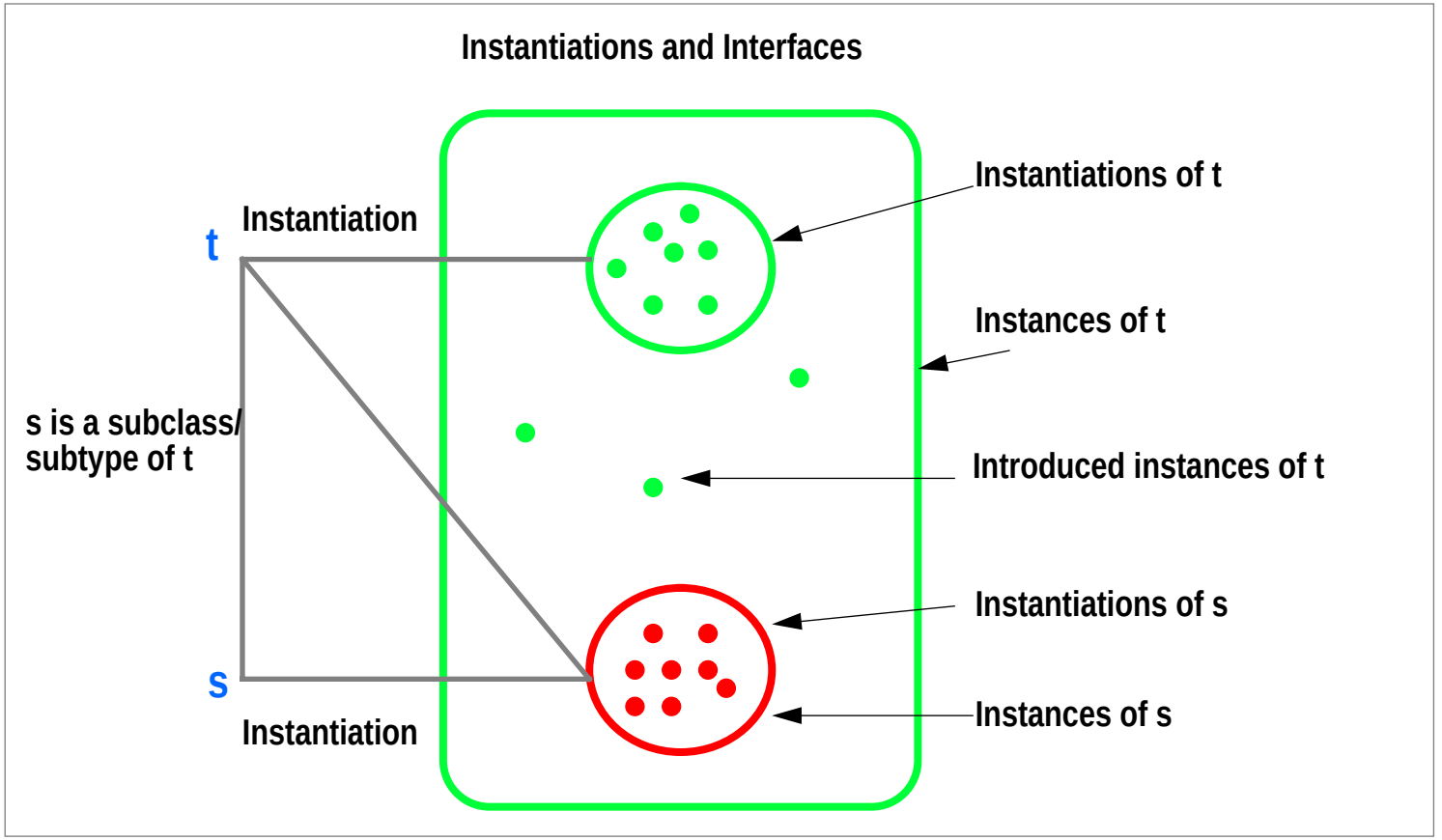
Communication

- Communication: conveyance of information between objects
  - as a consequence of one ort more interactions
  - possibly involving intermediate objects
- Location
  - in space
  - in time
  - interval reflects needs of specification - e.g. a location which is atomic in the mind of the programmer may be highly structured in its implementation
- Interaction point: A location at which there exist interfaces
  - several interaction points may exist at same location (e.g. protocol stack)
  - interaction points may be mobile

Type and Class

- OO has many religious views here because in each domain assumptions are made about instantiation and derivation (inheritance) mechanisms
  - Part 2 dissects this space in glorious detail, but hardly gets used in Part 3!
  - But removal of clutter gives Part 3 clear semantics
- Type of an <X>: A predicate characterizing a collection of <X>s
  - instance
- Class (of <X>s): the set of all <X>s characterized by a type
- <X> Template: Specification of the common features of a class such that an <X> can be instantiated using it (and possibly some instantiation parameters)
  - template type (of an <X>) - a predicate defined in a template that holds for all instantiations of the template
  - template class (of an <X>) - set of all <X>s satisfying an <X> template type
  - creation, introduction, deletion

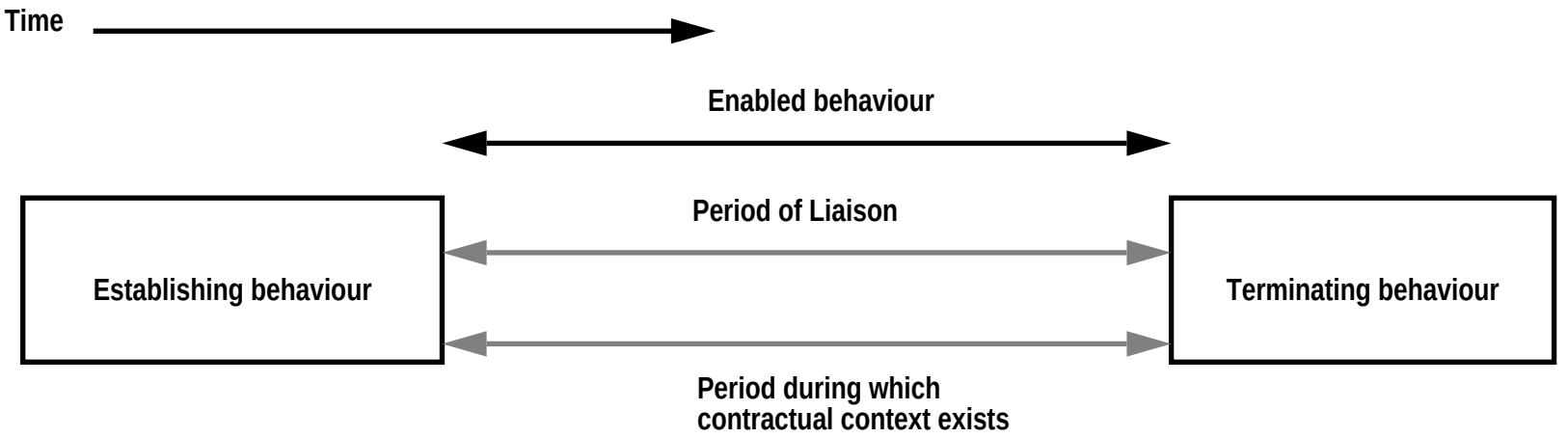
In a picture


Instantiations and Interfaces
t
Instantiation
s is a subclass/
subtype of t
s
Instantiation
Instantiations of t
Instances of t
Introduced instances of t
Instantiations of s
Instances of s

Subtypes

- A is a subtype of B if every <X> which satisfies A also satisfies B (A is “bigger”)
- P is a subclass of Q when the type of A is a subtype of the type of B
- Derived class: if template A is an incremental modification of template B, then the template class CA of instances of A is a derived class of the template class CB of instances of B; CB is a base class of CA
  - “inheritance” is set of criteria by which modification is accepted as “incremental”
  - “strict inheritance” = properties from base class cannot be suppressed
  - “multiple inheritance” = classes can have several base classes
- Inheritance (derivation hierarchy) is logically distinct from type hierarchy
  - A is a subclass of B does not imply A is derived from B
  - Part 3 Computational Model exploits this

Contractual behaviour



Binding: a contractual context arising from an establishing behaviour

Trading: interaction exchanging information about contracts

- Contract - obligations, permissions and prohibitions
  - e.g., roles and rules in a security policy, QoS attributes, liveness and safety conditions
- Much of the Part 3 engineering model is about how to use engineering bindings to hold up computational ones.

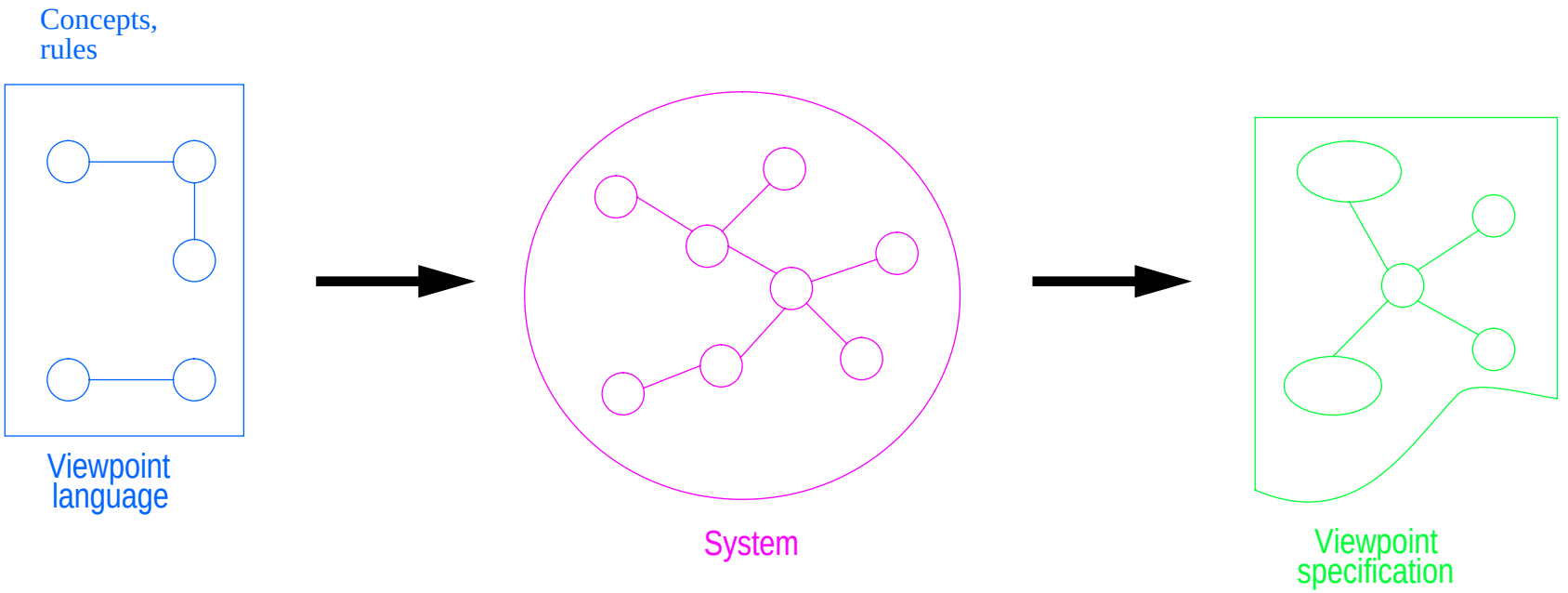
Part 3

- Defines the architectural framework
- The Body of the Reference Model
- A road map for subsequent standards

Languages and Viewpoints (1)

- *Framework* consists of five *viewpoint languages* and a set of *ODP functions*
- Each viewpoint language enables specification of an ODP system or component
- Languages are structured so that *consistency checking* between alternative viewpoint specifications is possible
- The chosen set is *necessary* and *sufficient* for needs of ODP
- Each language consists of *concepts* (vocabulary) and *rules* (grammar)

Languages and Viewpoints (2)



- **Mathematical basis in *projection* of a set of concepts and abstraction/specialisation relations over a semantic net.**

Viewpoints

- Enterprise - purpose, scope and policies for a system
- Information - kinds on information handled by the system and constraints on the use and interpretation of that information
- Computational - functional decomposition into objects suitable for distribution
- Engineering - the infrastructure required to support distribution
- Technology - the choice of technology to support distribution
- Don’t equate viewpoints to design process refinement steps!

Conformance

- Reference point - a potential conformance point
- Positioned by computational and engineering languages
- Specified in (some combination of) enterprise, information, computational, engineering and technology language languages
  - enterprise specification determines roles and policies with respect to the reference point
  - information specification determines universe of discourse for the reference point
  - computational specification determines the dialogue structure
  - engineering specification determines the infrastructure
  - technology language determines how to insert tester and interpret results
- An ODP system is one which conforms to ODP standards at all reference points asserted to be conformance points

ENTERPRISE AND INFORMATION LANGUAGES

- for writing requirements in ODP standards
  - for organizations
  - for systems
  - for components
- Enterprise language = policy, contracts
- Information language = domain of discourse, constraints

Enterprise language

- Role: e.g. service provider, service user, resource manager, agent, artefact
- Policy: security, safety, objectives, contracts, codes of practice
- Resource: accounting
- Community: configuration of objects with an objective (*contract*)
- Enterprise actions
  - incur an obligation
  - fulfil an obligation
  - waive an obligation
  - acquire permission
  - be forbidden
- OO flavour for alignment with computational viewpoint

Federation

- Federation: a community of domains
- The need to support federations that maintain domain *autonomy* underpin many of the technical aspects of the Reference Model.
- Autonomy principles
  - freedom to join
  - freedom to leave
  - subject to agreed obligations
  - retain local policies, including technology choices
  - no single administrator
- "Open Systems" is all about maximizing domain automony without compromising scope for federation

Information Language

- Concepts, relations
  - define complex concepts as relations (“is-a”, “contained in”, “owned-by”) between simpler ones
- Integrity rules
  - static schema (rules about state and structure at some point in time)
  - invariant schema (independent of behaviour)
  - dynamic schema (possible behaviour)
- OO flavour for alignment with computational viewpoint
- Compatible with methodologies such as OMT

COMPUTATIONAL LANGUAGE

- for specifying interfaces and distributed algorithms in ODP standards
- interfaces for interoperability - a lot said about this
- objects for portability - rather less said here (to my regret)
  - ANSA has a construction model - almost “abstract” bytecodes.....
- all objects are “distributable” by default
  - = Java RMI model of inherited Remote abstract class c.f. CORBA registration with an Object Adaptor
- define objects, interfaces and interactions without committing to specific engineering mechanisms
  - declarative transparency -- anticipated “reflection” in OO

Distributed Programming is Different

TRADITIONAL	REVERSED
Local	Remote
Sequential	Concurrent
Single 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

- Trying to wedge this into a traditional view won't work, hence we need a model for distributed computation

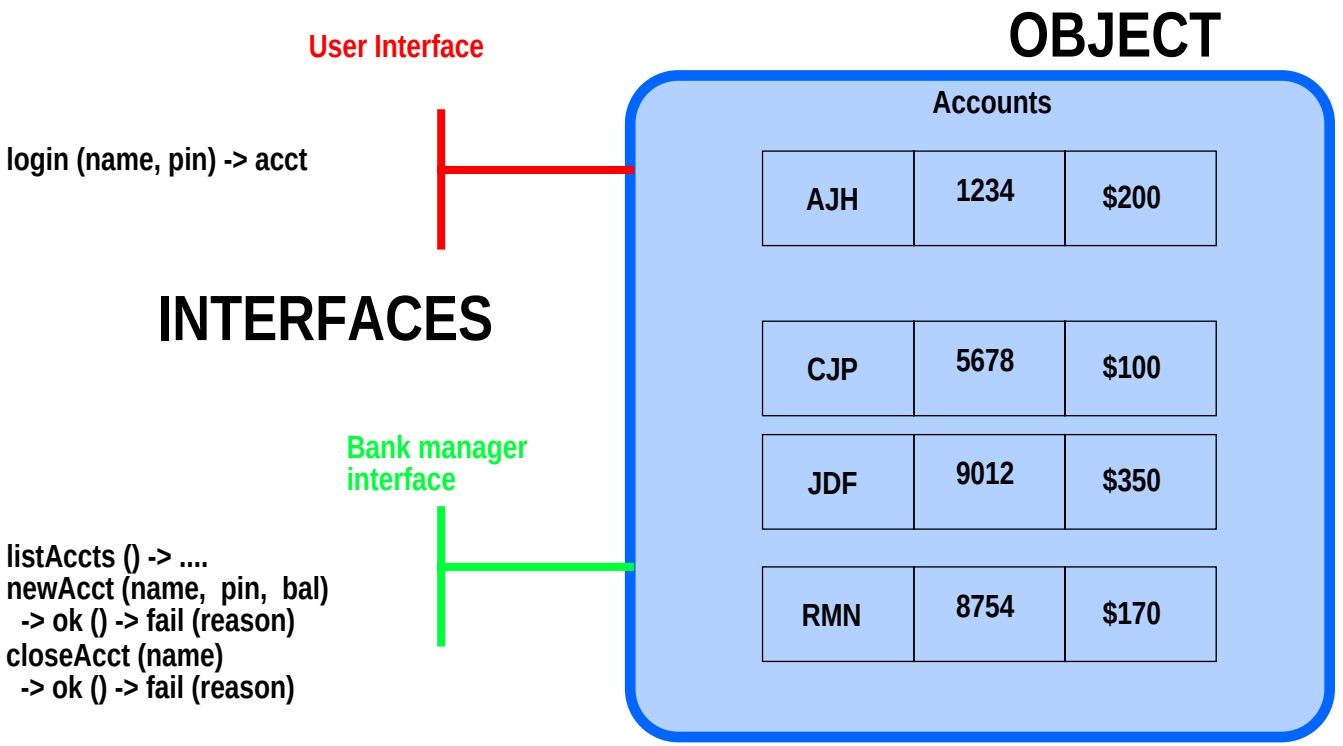
Computational language

- Significant level of prescription
  - dialogue (interaction) structures for *interoperability*
  - activity (i.e. program) structures for *portability*
- Abstracts over a wide choice of implementation mechanisms
- Object-based because encapsulation and separation of service from implementation inherent in object models is well matched to the needs of distributed systems
- Strong typing rules for maximum confidence
  - no dynamic interfaces (c.f. DII, DSI)
  - strong hint of first class types - the type “type”.
- Good alignment with OMG CORBA standard

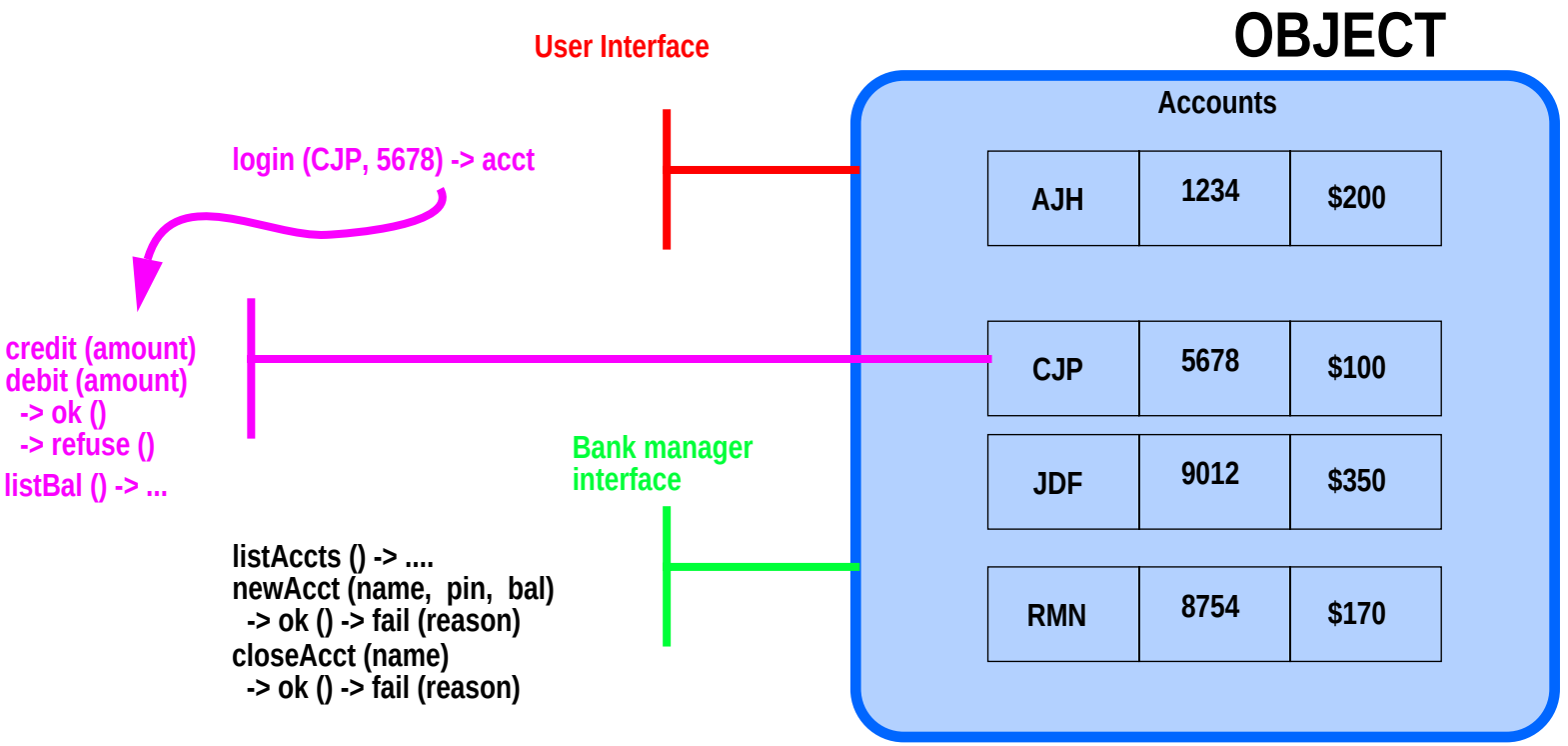
Computational Concepts - c.f. Java

- Object - unit of encapsulation and hence configuration
  - an object can have several interfaces
  - an object can create interfaces dynamically
  - an object is persistent
  - an object can migrate, fail, checkpoint, replicate, ...
- Interface - provides a service in a context
  - an operation interface has operations (methods)
  - operations have a set of terminations
  - a stream interface has flows (video, audio, Unix pipe, TCP byte stream, ..)
  - signal interfaces for event handling view of flows or operations
- call by sharing - operations, terminations and signals accept interfaces as parameters
- Activities - fork / join vs spawn, bind, call, terminate, respond

Computational Object Model (1)



Computational Object Model (2)



Types, Trading and Binding

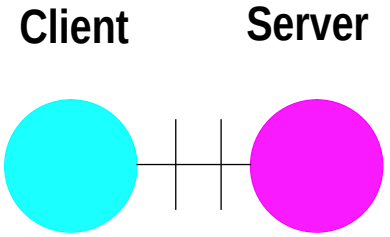
- **Trading** - discovering an interface that provides a service
  - exporters make service offers
  - importers make service requests
  - trading marries imports to exports
  - an ODP function
- **Binding** - knitting together a set of matching interfaces to enable interaction
  - typically implicit for client server operation interface pairs
  - necessarily explicit for streams and for RPCs where QoS is to be controlled
  - built into the computational model
- **Type safety** - match signatures to ensure valid interaction
  - “no surprises” rule
  - information types (called “properties”) checked when trading

Binding (1)

- SIMPLE BIND (X, Y)
  - bind interface X to interface Y (complementary types and causality)
- COMPOUND BIND <binding object template> {set of interfaces}
  - instantiates a binding object between the set of interfaces
  - binding contract determines pattern of interconnection, choice of infrastructure, quality of service, etc
- Bound interfaces have roles relative to the binding
  - client, server, producer, consumer; binding controller
- Binding controllers can
  - add/remove interfaces
  - stop/start information flows
  - change quality of service
  - monitor events

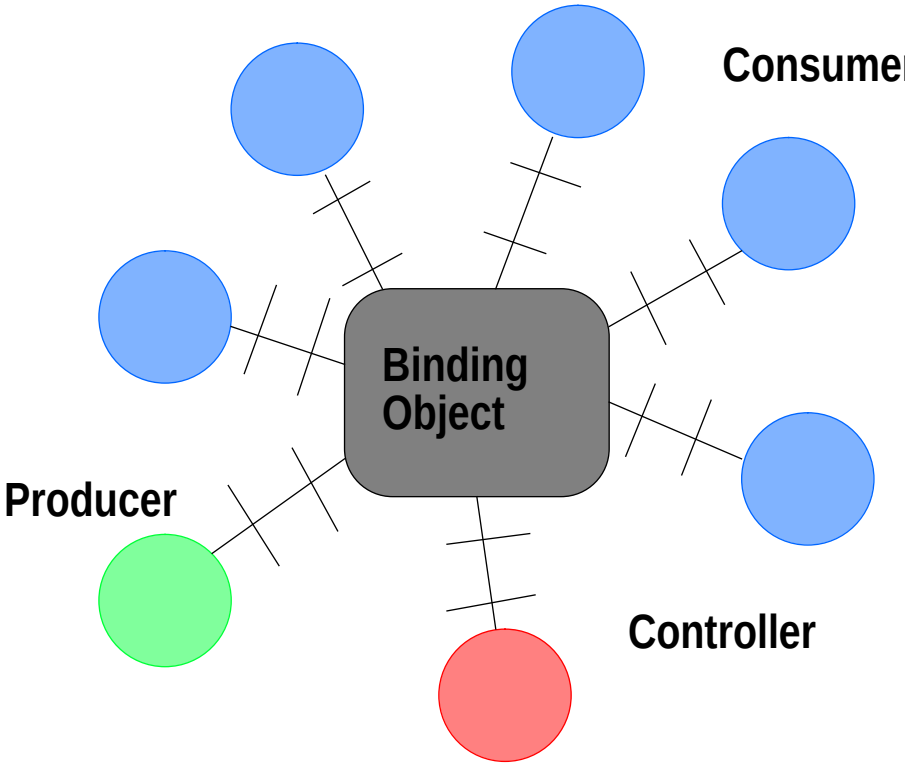
Binding (2)

Simple Bind



A binding

Use signal interfaces to
give end-to-end semantics



Compound Bind

Operational interface typing

- operation invocation is a *request-reply* model
- type checking for safety - no surprises principle
- server must provide at least operations required by client
- each operation has one or more *terminations*
- client must accept all possible terminations from server
- client arguments “smaller” than server requires
- server results “smaller” than client accepts
- arguments and results can be interfaces
- Rules for signal and stream interfaces are similar
- No linkage between type names and subtyping
  - deep religious issue -- Andrews against the world.....

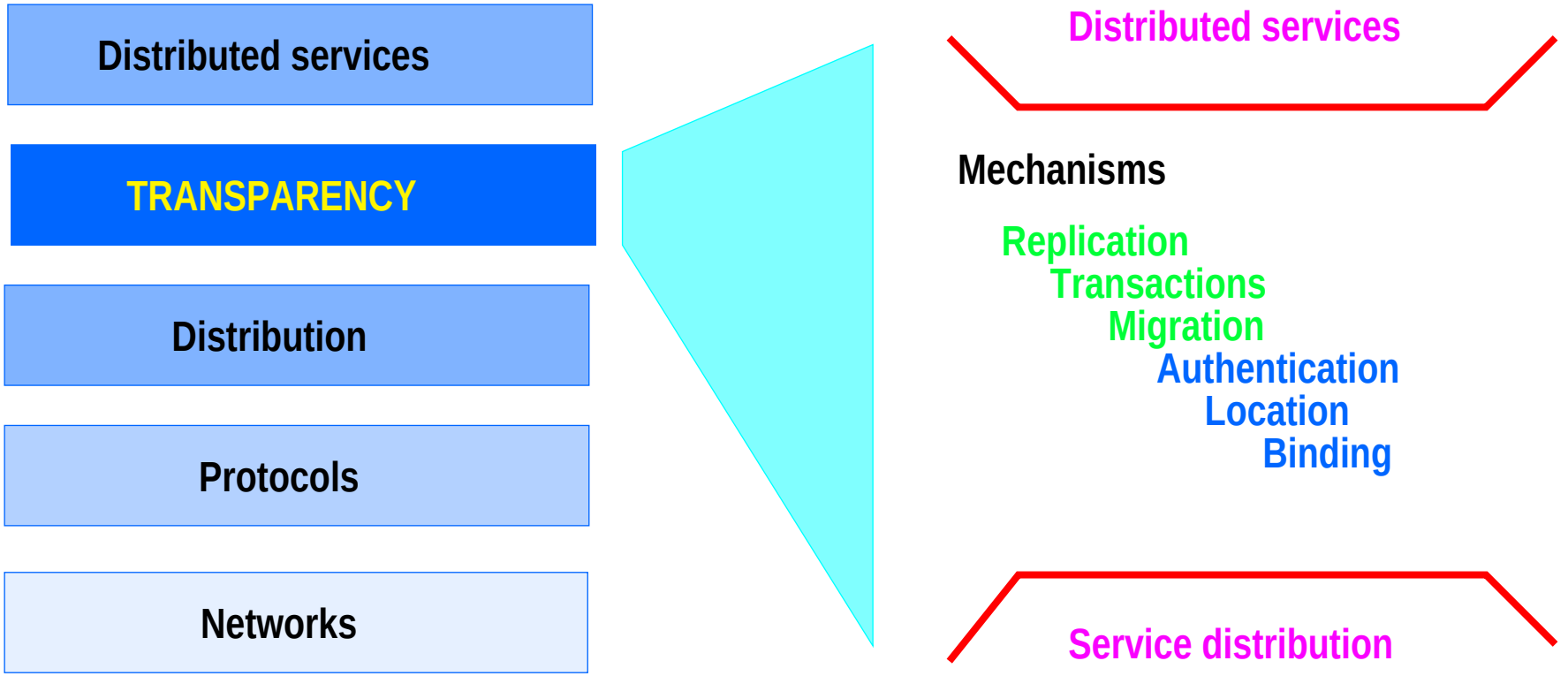
ENGINEERING LANGUAGE

- for specification of infrastructure needs in ODP systems (viz. block diagrams)
- in CORBA too much of this shows through to the applications programmer (object adaptors)
- in CORBA not enough of this shows through to the systems programmer (composing transaprent object services)

Trade-offs in Distributed Systems

- Distributed systems engineering is all about trade-offs
  - ABSTRACTION versus SPECIALIZATION -the more you hide, the less control you have
  - CONSISTENCY versus AVAILABILITY - availability implies 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
- Therefore we need a kit of parts and an open framework into which they slot
- Moreover the framework must accomodate coexistance of alternative parts for the same job

Engineering Framework



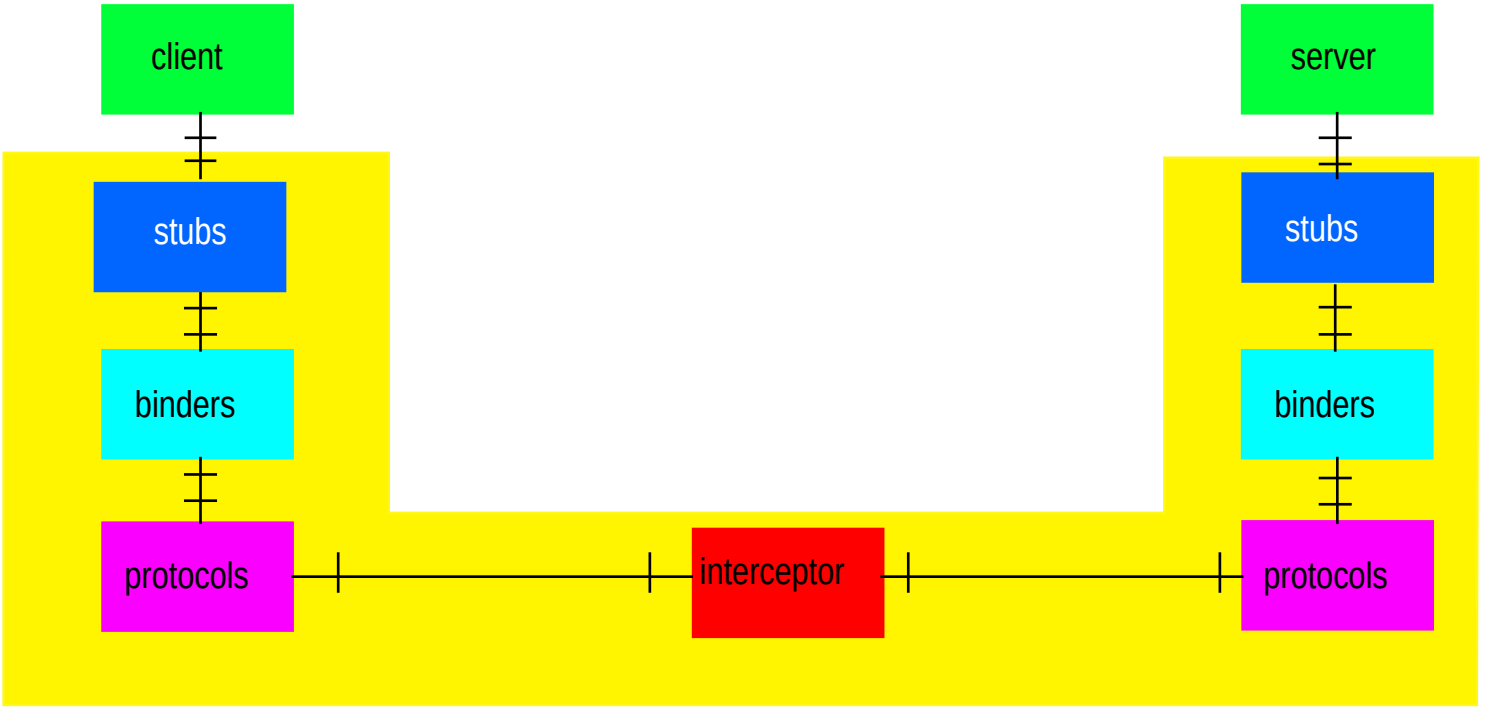
Selective Transparency

- Transparency is about hiding irrelevant complexity
  - Location — don't need to know where it is to use it
  - Access — don't need to know how it works to use it
  - Relocation — it can move while you're using it to balance loads or reduce latency
  - Migration — you can be moved.....
  - Replication — there may be copies for reliability and/or availability
  - Persistence — it only gets resources when it needs them
  - Failure — it always gets to a consistent state
- The transparency supports the functionality of the basic service distribution layer, and adds additional guarantees
- extra management functions for controlling transparency

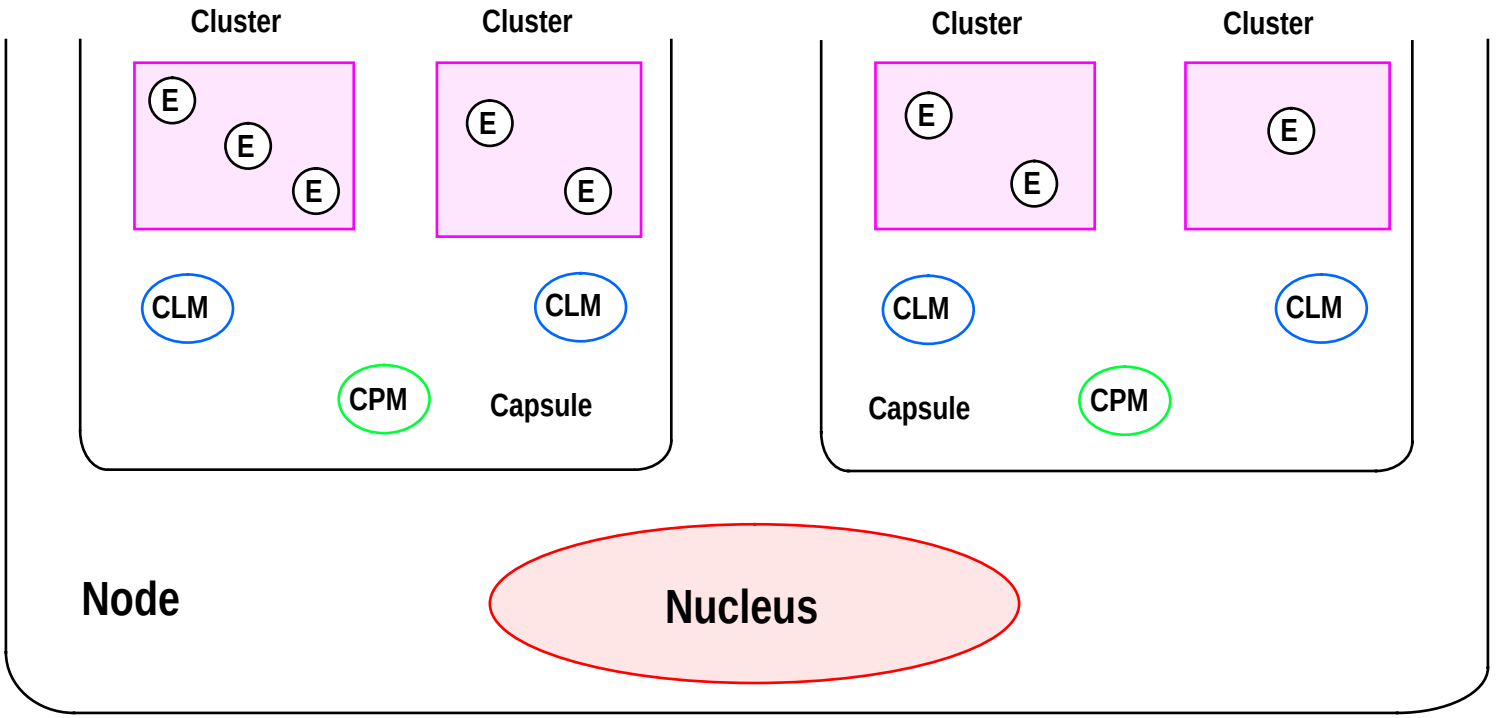
Engineering language

- structures for implementing infrastructures supporting *selective transparency*
- *channels* for communication (c.f. stubs, skeletons and object adaptors)
  - simple client server
  - multipoint channels
  - stream channels
- *clusters* for activation, deactivation, migration (= pages or CORBA objects)
- *capsules* for resource allocation, protection (= processes)
- *nodes* for network addressing (= computers)

Client-Server Channel



Engineering structures



Engineering bindings

- Stubs marshall interfaces into and from interface references
- Marshalling stub calls the nucleus stating any binding constraints
  - e.g. use of specific protocols
  - e.g. use of non-local identifiers (such as DCE UUIDs)
- Nucleus creates local supporting infrastructure and invents interface reference
  - interface reference tells other objects how to bind to the interface
  - potentially complex data structure, especially if federation is involved
- Stub send interface reference in message to stand for interface parameter
- Recipient stub asks nucleus to bind to received interface reference
  - nucleus creates supporting infrastructure and completes the binding

Federation

- Large systems are made up of autonomous islands interconnected incrementally
  - no central authority
  - legacy of old technology
  - conflicting choices of new technology
- Administrative boundaries
  - where checks and accounting are to occur at the boundary
- Technology boundaries
  - where protocol conversion and data translation are to occur at the boundary
  - Set up interceptors (gateways / bridges) on demand, when trading

Interception

- The trader instantiates an interceptor when a service is imported across the boundary
  - this makes interceptors less painful than traditional gateways
- An interceptor can
  - make a boundary transparent
  - impose an administrative boundary
- Interception can include
  - protocol conversion
  - name mapping
  - adding administration payload
- Intercept at
  - node
  - LAN
  - WAN bridge / gateway

Technology language

- Implementable standard: template for a technology object
- Implementation Extra Information for Testing
  - technology specification in a standard defines a proforma IXIT
  - IXIT defines templates and names of interfaces required for testing

Consistency constraints

- Rules for avoiding inconsistency between specifications
- Structural relationships between
  - information and computational specifications
  - computational and engineering specifications
- Pointing to image of building a system specification out of statements in multiple viewpoint languages simultaneously, cross checking as you go

Example: Modelling a directory system

- Use enterprise concepts to describe policy
  - only the directory owner can update it
- Use information concepts to describe the trading system “content”
  - e.g., only exported offers can be imported, withdrawn offers disappear
- Use computational concepts to describe the function and interfaces
  - interface in every major city
  - import, export functions
- Use engineering concepts to describe implementation
  - proxy in every major city, central database
- Use technology concepts to select standards
  - LDAP vs X.500 vs DNS

Functions

- The system components one needs to hold up the engineering structure
  - equivalent to Object Services in OMG
  - a lot of tension during development of Part 3 about how much engineering detail to prescribe
- Management (object, cluster, capsule, node)
- Coordination (checkpointing and recovery, deactivation and recovery, migration, transaction, group, replication, events, interface reference tracking)
- Repository (storage, information organization, relocation, types, trading)
- Security (access control, security audit, authentication, integrity, confidentiality, non-repudiation, key management)

Management Functions

- node management (viz., the nucleus)
  - threads
  - binding - channel setup
  - capsule manufacture
- object management
  - objects manage themselves
  - cluster managers ask objects to terminate, snapshot etc.
- cluster management
  - deactivate, checkpoint, recover, migrate, replicate, terminate

Management Functions

- capsule management
  - cluster instantiation, reactivation
  - termination
- interface reference tracking
  - tracking interface references
  - enables distributed garbage collection
- relocation
  - initiating use of coordination functions to repair broken bindings transparently (e.g. after migration)

Transactions

- Very general model - specific models defined in terms of
  - *visibility* - how much interaction possible with objects outside the transaction
  - *consistency* - invariant to be fulfilled at end of transaction
  - *recoverability* - how much of transaction is undone after failure
  - *permanence* - the degree to which failures can alter the effects of a transaction
  - *dependency* - influence of other transactions on success or fail
- Strong hint of declarative concurrency constraints in interface specifications being compiled into interface-specific transaction monitors

Groups

- general model, specific forms of cooperative computation defined by policies for
  - distribution of interactions among participations
  - collation - correlating requests (e.g. voting)
  - ordering - visibility of requests to participants
  - fault detection and recovery
- Motivated by technologies like ISIS
- Still an open subject, but research consensus on the building blocks

Repository functions

- Separate out components of repository to enable distribution
- Big debate about whether repository exists or is the cloud of distributed meta-data
- Storage
  - data storage
  - deactivated cluster repository
  - interface reference fixing done on reactivation
- Information organization
  - enables structured queries over sets of objects
- Relocation
- Types
  - abstract data types for signatures to enable dynamic type checking

Trading

- Each object has access to a trader
- A trader contains
  - services offers - type, properties, interface reference
  - links to other traders - name, properties
- A service offer can be tied to an export controller object
  - to monitor imports, to allocate resources
- A trader may be optimized for
  - speed of look up vs volume of offers stored
  - accuracy of offer
  - dependability
- No structure is forced on traders, to enable federation
  - context relative naming is the key

Security

- Every object is responsible for its own security
- Capsule concept defines the smallest protection boundary
- Functions
  - Access control
  - Audit
  - Authentication
  - Integrity
  - Confidentiality
  - Non-repudiation

Transparency Schemas: Let Compilers and Tools Take The Strain!

- Exploit abstraction, program in application oriented concepts
  - most aspects of OO really help, some hinder
- Simple (pre-processor) extensions go a long way
  - especially if leveraging an OO language
- orthogonality - e.g. “dot” and “bar” *vs*. threads and RPC API
  - languages minimize complexity without losing scope for optimization
- declarative - state requirements and policies not mechanisms
  - point already proven by IDLs and stub generators
  - decouple applications from engineering - ANSA PREPC experience
- strong type checking for safety and confidence

Abstraction and Automation in Arjuna

- Distributed transactional, persistent C++ objects
- Application objects inherit from Arjuna class for
  - transaction coordinator
  - locking
  - persistence
  - RPC
- Stubs generated from class header files
  - could also do this from class definition...
- Application programmer only has to
  - define application objects
  - defines “saveState” and “restoreState” methods
  - brackets transactions and decide ‘success” or “fail”
  - sets locks
- All the mechanisms are transparent c.f. traditional TP programing

Lessons learned (1)

- Formality helps remove noise
- People try too hard to be ascribe deep meaning to the viewpoints
  - the waterfall heresy
  - the enterprise viewpoint is business objects heresy
  - original ANSA idea was motivated by Piaget's cognitive levels and Sowa's conceptual structures
- Viewpoint modelling really works
- Telecoms orientation made us cover a broader scope (= a good thing)
- Tension between "model anything" vs "do it right" left uneven level of prescription

Lessons learned (2)

- Industry wants concrete specs to build to not abstractions
  - led to CORBA having a lot of “baggage”
- RM-ODP computational model anticipated Java object model
- RM-ODP transparency anticipated reflection and meta-object protocols
- RM-ODP should have done a computational portability model