Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

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

Training

ANSAwise - Dependability in Open Distributed Systems

Chris Mayers

Abstract

Organizations are seeking competitive advantage by offering high service quality for information services.

One characteristic of service quality is dependability. Achieving high dependability is a difficult challenge in distributed systems.

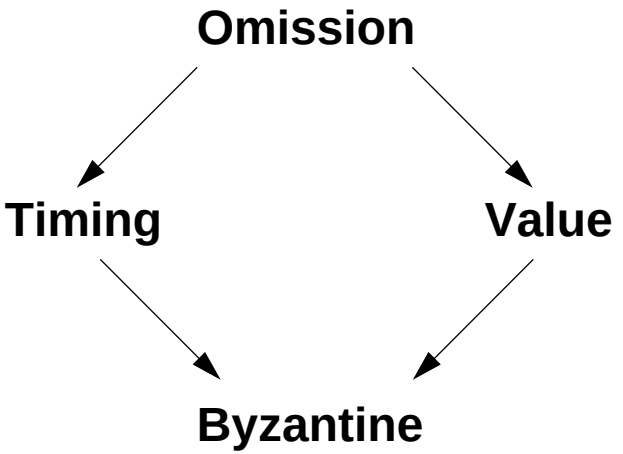
This module of the ANSAware training programme reviews dependability concepts and shows how they can be applied to distributed systems, focusing on reliability and availability aspects of dependability.

APM.1341.01 **Approved** 25th November 1994

Briefing Note

Distribution:
Supersedes:
Superseded by:

Dependability in Open Distributed Systems



In this session

- *Review concepts in dependability*
  - failure models
- *Review the importance of interfaces*
  - importance of specifying the error conditions that occur
- *See how objects help build dependable systems*

Dependability

- *"The property of a system that allows reliance to be justifiably placed on the services it delivers"*
- *Defined in terms of five non-functional properties*
  - availability: readiness of usage
  - reliability: continuity of service
  - integrity: maintaining the consistency of data
  - safety: avoidance of catastrophic consequences
  - security: prevention of unauthorized disclosure (or handling) of information

Prerequisites for reliability

- ***Error detection***
  - **in a distributed system, this must be done at interfaces**
- ***Damage assessment***
  - **this must be done by the objects themselves, assisted by infrastructure**
- ***Error recovery***
  - **backward to a previous good state**
  - **forward to a new good state**
- ***Fault treatment***
  - **fault location, repair or avoidance**

Faults are a fact of life

- *Recall the centralized assumptions we must now reverse...*
- *... So*
  - faults are frequent and must be handled
  - partial failure must be handled
  - malicious intent is likely and must be handled

Availability grades

Availability	Unavailability (mins/year)	Grade	Comments
90%	52,560	Unmanaged	
99%	5,256	Managed	Good (in 1980)
99.9%	526	Well managed	
99.99%	53	Fault-tolerant	
99.999%	5	High-availability	Best (in 1990)
99.9999%	0.5	Very-high-availability	
99.9999%	0.05	Ultra-high-availability	

Threats to Commercial Network Security

Threat	Frequency	Cost
Natural disaster	Unpredictable	Unpredictable
Accidental failure	25%	30%
Human error	70%	25%
Malicious attacks	5%	45%

Transport safety and reliability

- *These specifications are considered 'ultra-reliable'*

Probability of failure (per hour)	Service
10^{-12}	Urban trains
10^{-11}	
10^{-10}	Commercial aircraft
10^{-9}	
10^{-8}	
10^{-7}	Military aircraft
10^{-6}	Unmanned vehicles

Hard facts

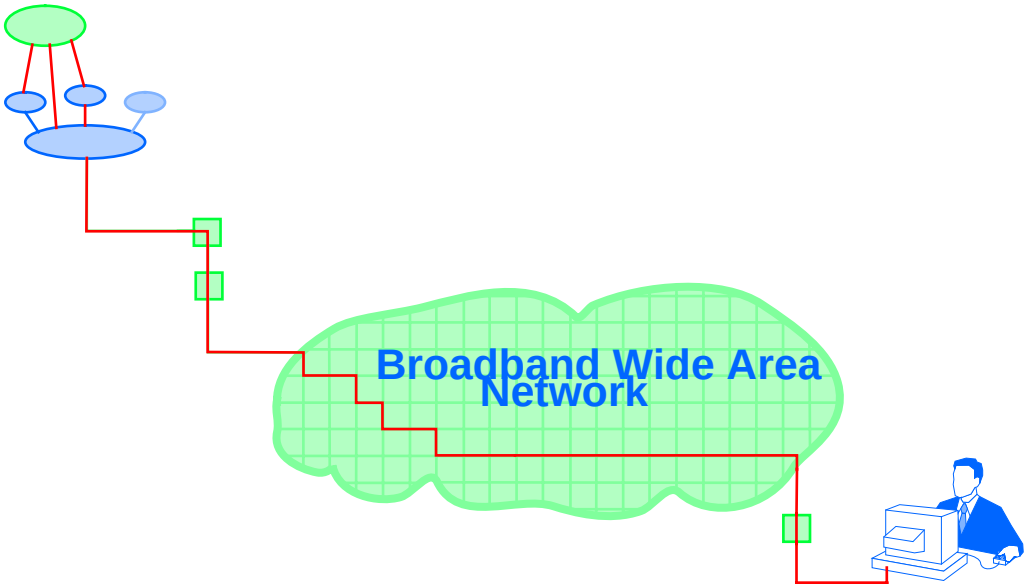
- *With a failure probability of 10^{-9} per hour*
  - ...fleet of 1000 aircraft
  - ...over 30 years
  - ... about 1 in 10 chance of at least one failure
- *As a comparison*
  - your chance of winning the jackpot in the UK national lottery is about $7x10^{-8}$

Hardware Isn't The Real Problem

- *Outages are rare*
  - and few of them are caused by hardware faults...
  - ... hardware fault tolerance masks them
- *From a study by Tandem*
  - out of 30,000 hardware faults, only 29 caused system outage
- *By comparison*
  - UK nuclear safety standards mandate that software can only be assumed to be 10^{-4} (probability of failure on demand)

End-to-end reliability

- *In a distributed system, reliability must be end-to-end*



  - including the end-systems...
  - ... not just the communications

Cost-effective reliability

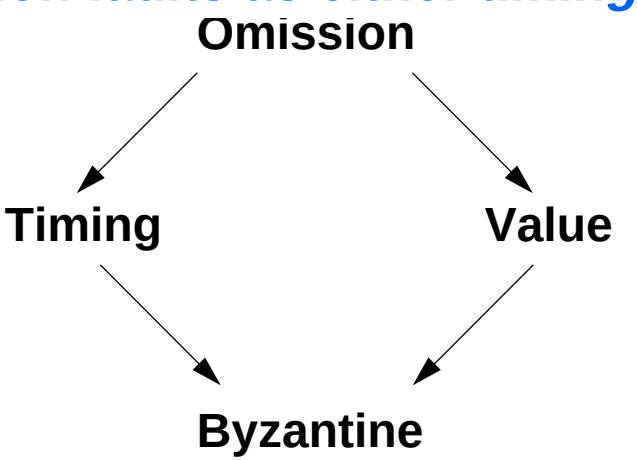
- ***Having quantified the requirement, we must now achieve it***
  - **cost-effectively**
- ***This requires engineering trade-offs***
  - **based on understanding the kinds of system failure**

Kinds of fault

- ***Omission fault***
  - the expected response never arrives
- ***Timing fault***
  - the expected response arrives too early or too late
- ***Value fault***
  - the expected response arrives with the wrong value
- ***Byzantine fault***
  - the expected response violates specified behaviour
  - ... including a response when none was requested
  - ... including arbitrary ‘malicious’ behaviour; literally anything

The ‘fault tolerance lattice’

- *You can regard omission faults as either timing or value faults...*



- *... and Byzantine faults encompass everything*

Coping with faults

- ***If systems just stopped when they were faulty, fault tolerance would be simpler***
  - **all failures would be omission failures**
- ***We can achieve this for hardware faults***
  - **by hardware replication**
- ***With duplexed hardware and comparison***
  - **system stops when outputs disagree...**
  - **... *fail-fast* or *fail-stop* operation**
- ***The software equivalent needs N-version programming***
  - **controversial, because all software faults are design faults**

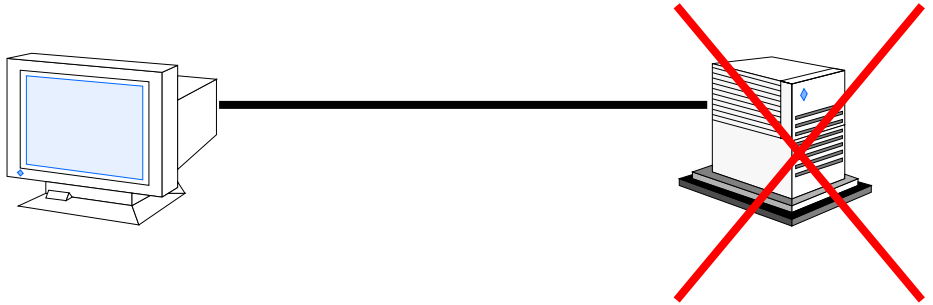
Timeouts in RPCs

- ***A particular difficulty is to determine the cause of an RPC timeout***
  - **the server may never have received the request**
  - **the server may have crashed during the request**
  - **the server's response may have been lost**
  - **the server may still be processing the request**
- ***Without bounded-time communications, it is impossible to distinguish 'slow' and 'failed' service***

The partial failure problem in distributed file systems

- *In a distributed system, workstation client and file server can fail independently (a partial failure of the system)*

- *How should file server failures be handled?*



Failure handling in NFS

- *In stand-alone file systems, errors are rarely handled by applications*
  - errors are usually catastrophic...
  - ...for example, disk full, file corrupt,...
  - human involvement is needed to recover from these situations anyway
- *When network file systems were introduced, NFS offered two options*
  - soft-mounting (not transparent)
  - hard-mount (transparent)

Soft mounts and hard mounts in NFS

- *Soft mounts expose network and server failures to the application*
  - Existing applications did not cope well with these failures...
  - ... mishandling new, unexpected, error codes
  - ... they often corrupted files
- *Hard mounts hide network and server failures from the application*
  - Workstations freeze when a server fails...
  - ... the effect cascades between servers
  - ... users must sit and wait
- *Hard mounts are now universal*
  - the lesser of the two evils

Implications for NFS

- *NFS has good reliability, but poor availability*
  - the effects of failure tend to propagate
- *NFS requires centralized administration to recover from failure*
  - one person who can diagnose and sort out failures
- *NFS does not scale*
  - it is best suited to small or medium-sized workgroups

The flaw with NFS?

- *The communications technique?*
  - No; NFS is built on RPC, but the same problems would arise with the other two techniques
- *The NFS server design?*
  - No; it is pure - the client, server, and network are the only points of failure, and are separated
- *The NFS recovery mechanism?*
  - No; NFS is stateless
- *The applications file system interface*
  -

Failure handling in applications

- *In distributed systems, failures are normally handled using exceptions*
  - in CORBA
  - in ANSAware
  - in DCE (partially)
- *Up-to-date programming languages have this as a standard construct*
  - special support may be provided for older languages...
  - ... ANSAware provides this for C

Operations Must Specify The Exceptions They Raise

- *An example from the CORBA Naming service*

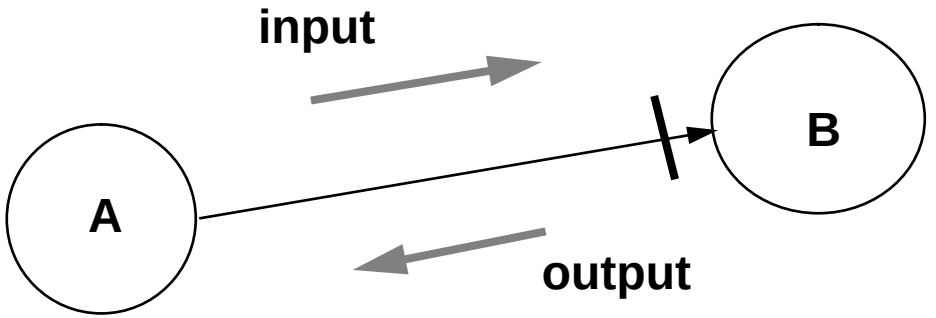
```
interface NamingContext {
   enum NotFoundReason (missing_node,
                        not_context,
                        not_object);
   exception NotFound {
      NotFoundReason why,
      Name rest_of_name
   }
   exception InvalidName ();
   exception AlreadyBound ();
   exception NotEmpty ();
   void bind (in Name n, in Object obj)
    raises(NotFound,CannotProceed,InvalidName,AlreadyBound);
   ...
};
```

Exception specifications

- *Additional information can be associated with exceptions*
  - as in the NotFound exception above
- *This information may be necessary for error recovery*
  - it must be returned as the response to an operation...
  - ... in a distributed system the client cannot fetch it in a separate call
- *Operations must only raise the exceptions they specify*
  - it is part of their ‘contract’ with the client
  - ...there is a standard set of ‘system exceptions’ that can occur on any operation

Type Conformance Has Two Sides

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



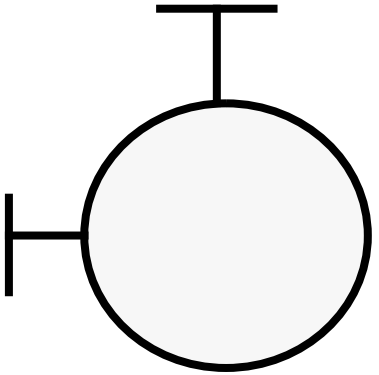
- *...but also, A must handle at least the exceptions given by B*
  - same idea, but the ‘at least’ rule is the other way round for exceptions
- To conform, both of these must be checked... (a two-sided *contract*)
  - B must not receive unknown operations
  - A must not receive unknown exceptions

Two basic techniques for dependability

- ***Fault avoidance***
  - **good engineering practice, to make faults less likely**
  - **... dependable implementations**
- ***Fault tolerance***
  - **to negate the effects of faults**
  - **... dependable interfaces**
- ***... these two techniques are complementary, not exclusive***

How objects help

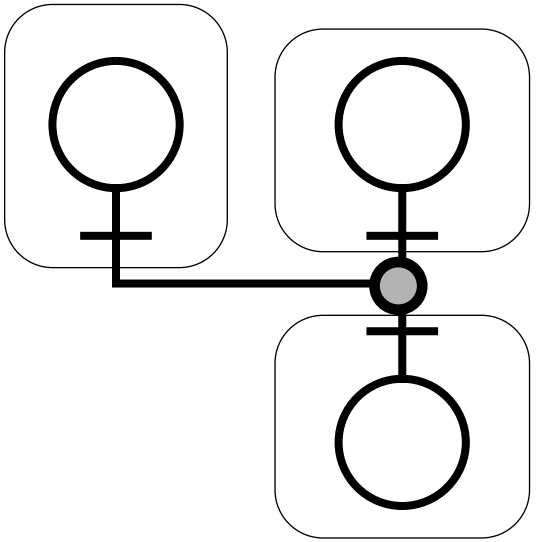
- *The encapsulation boundary confines faults...*



- *...failure of one object does not imply failure of another*
- *... failure modularity*

Tolerating failure with replication transparency

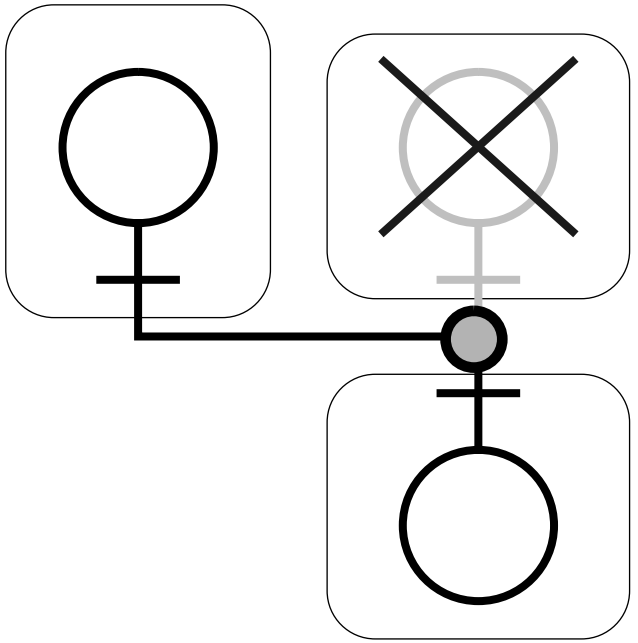
- *The application need not know how many copies*



  - the application only sees a single interface

Transparency Engineering - Failure

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



  - **may use replication transparency to achieve this**

General guidance for reliability

- *Use simple and proven reliability engineering mechanisms*
  - particularly in error recovery
  - ... (selectively) transparent to applications
- *Be clear what the reliability requirements are*
  - over-engineering has high performance penalties...
  - ... the application must decide
- *Be clear what each mechanism is giving you*
  - the more mechanism there is, the more there is to go wrong
- *Use both fault tolerance and fault avoidance techniques*
  - for dependable interfaces and implementations

Summary

- ***Objects are responsible for their own dependability***
  - in a distributed system, there is no ‘global overseer’ to look after their welfare
  - applications are responsible for end-to-end reliability
- ***Faults must be detected as early as possible***
  - ideally, before installation
- ***Interfaces must specify their error conditions***
  - typically, as exceptions

More information?

- ***For a comprehensive treatment of dependability***
  - **see *Reliable Computer Systems*, by Daniel Siewiorek and Robert Swarz (Digital Press)**
  - **... covers software and hardware**
- ***For more on fault tolerance***
  - **see *Transaction Processing: Concepts and Techniques*, by Jim Gray and Andreas Reuter (Morgan Kaufmann)...**
  - **... a general approach, not just for TP systems**
- ***For more on dependability in distributed systems***
  - **see *An ANSA Analysis of Open Dependable Distributed Computing*, by N.J. Edwards (APM.1149)**

A wider view

- ***And, for a wider perspective...***

  - ***Computer Related Risks*, by Peter G. Neumann (Addison-Wesley)**

  - **... and the inside back page of Communications of the ACM**