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**

ANSA Phase III

The Dependability Engineering Model

List of author names goes here

Abstract

Need some instructions here.

APM.1044.00.01 **Draft** 23 February 1994

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

The Dependability Engineering Model

The Dependability Engineering Model

List of author names goes here

APM.1044.00.01

23 February 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

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

Contents

1 The Dependability Engineering Model

Note: This draft is mostly a collection of requirements statements.

1.1 Introduction

The scenario paints a picture of a variety of ANSA and non-ANSA systems (aka legacy systems) inter-connected by an ANSA infrastructure. This leads to a system view in which the following interaction possibilities are of interest:

- ANSA client and ANSA server
- ANSA client and legacy server
- legacy client and ANSA server
- legacy client and legacy server (via ANSA infrastructure)[1]

In the context of ANSA, a legacy system is any system which was explicitly designed and built to use a non-ANSA infrastructure. Access to the internals may not be available, but even if it is such systems may be uneconomic to re-engineer to exploit ANSA directly and so must be accomodated indirectly.

Note: There is of course a spectrum ranging from “written as an ANSA application” through to “written for a specific hardware configuration, no access to source code”. If you could re-engineer it would it still be a legacy system?, what’s the least re-engineering you would need to do?

From a dependability viewpoint, we take as a starting point the end-to-end argument [ref], that is the model must concern itself with ensuring that service consumers are provided with a reliable/dependable service as and when they require it.

There is a spectrum of approaches to providing dependability,the endpoints of this spectrum are: *fault avoidance* which attempts to pre-empt errors by designing them out of the system and *fault tolerance* which attempts to survive errors by detecting and recovering from them. Practical systems will exploit a mixture of both techniques.

Fault tolerant systems require additional runtime support to be provided to enhance both client and server. For ANSA clients and service providers, this support can be selectively transparent (or even non-transparent). For legacy objects, the support can only be transparent and there may be application imposed limitations as to how much transparency can be deployed.

1.2 Fault avoidance

Fault avoidance is especially useful where it is possible to predict the nature of failures which can occur. It is commonly used for non stop hardware

1. The case of legacy client and legacy server interacting entirely via a non-ANSA infrastructure is of no interest here.

configurations [tandem ref]. Software fault avoidance techniques include type checking, testing and formal verification to remove logic and coding errors.

Note: This model is primarily about software dependability techniques. To guarantee the overall dependability of a system requires knowledge of hardware dependability, this drags in lots of management and configuration issues.

Another use of fault avoidance is the use of self test techniques. An object can use self test to detect it's own failure before any third parties percieve a fault in the service.

The present state of the art is that fault avoidance is best suited to those situations which are simple (to describe) but the techniques are still valuable and all feasible fault avoidance techniques should be used during the design and implementation of the system to minimise the failures that can occur. Note however that many legacy systems will not be amenable to any (extra) fault avoidance enhancement.

While software fault avoidance techniques may someday be able to guarantee fault-free software, failures will still occur, in particular systems will always be susceptible to hardware failure or environmental errors (network partition, earthquakes etc). Clearly fault avoidance is not viable as the sole means of defence andcoping with these other faults requires fault tolerance techniques.

1.3 Fault tolerance

Note: need a link to expectations in [failure model].

Note: FIX: align terminology with [LAPRIE]

The engineering model for dependability initially concentrates on fault tolerance as this gives us a greater ability to withstand arbitrary component and environmental failures. Fault tolerance combines two independant strategies, a means of *fault detection* together with some form of *fault recovery*; that is, in order for detecting a fault to have been a useful act, the designer must know what action to take as a consequence.

1.3.1 Fault detection

The essence of fault detection is *monitoring*, that is objects of interest must somehow be observed in order to detect timing and/or value failures.

Note: The point here is that the onus is on me to detect faults in services I consume, e.g if the output is in error then the interface has failed and hence I have detected a fault. As fault notifications are generated by the detecting party, this will in general be an external object.

For monitoring to be useful we need some way to decide whether the output (or lack of it) is incorrect. To allow this, fault tolerance is based on two fundamental techniques which enable expectations to be tested: measuring the passage of *time* and exploiting *redundancy*.

1.3.1.1 Measuring time

Being able to measure the passage of time enables application designers to infer the existence of a failure by defining the time interval during which an action can legitimately take place, e.g. the time when a reply to a request is regarded as valid. Actions occuring outside this interval (before and after) are regarded as evidence of a failure. Having decided that an action is untimely it

is necessary to be able to cope with actions occuring after they have been declared late, e.g late messages.

Note: More needed, this requires accurate clocks, the ability to synchronise remote clocks and the ability to describe time in common terms.

1.3.1.2 Redundancy

Redundancy enables faults to be detected by having a means to decide whether an action is valid or not. The redundancy may be *implicit* and involve exploiting prior knowledge of the semantics of an action (e.g. knowing that counters and clocks should never go backwards) or it may be *explicit* and involve generating additional information (c.f. sequence numbers and CRCs in data communications protocols).

As the fundamental attributes of a computation are storage, processing and communications [ref], then redundancy can only involve *redundancy of storage* (e.g. replication), *redundancy of processing* (e.g. checksums) or *redundancy of communications* (e.g. retransmission of messages). In practice, fault tolerant systems will use a combination of techniques.

The distinction between explicit and implicit redundancy is that of understanding semantics: with implicit redundancy the client must have some knowledge of the semantics of the server, with explicit redundancy, only the designer of the server need have this knowledge.

There are two ways in which this classification is useful, firstly it gives us a way to classify reliable building blocks (that is reliable storage, processing and communications) out of which dependable systems may be constructed. Secondly it gives us a set of simple abstractions so that we may categorise transparencies, mechanisms and policies acording to their underlying properties. Both ways enable high level abstractions to be built up by selecting from standard choices with known characteristics.

1.3.2 Fault recovery strategies

Interaction with management, reconfiguration, renegotiation between interacting parties, rebinding, resynchronisation etc.

Note: Talk about recovery of storage, processing, communications perhaps?

1.4 How clients and servers interact

Note: Somewhere, we have a service/interface oriented view of dependability.

Lack of dependability within a client-server interaction arises because client and server may fail independantly or because there may be a communication failure. Hence (from the client's viewpoint) dependability within an interaction is ensured by first selecting a server with the most appropriate dependability characteristics and then enhancing the dependability of the interaction path.

For interaction to be possible the client must know how to communicate with the server, i.e. it must determine its name and address and a suitable protocol. To maximise flexibility, this ought to be done as late as possible. Within ANSA, this is made possible by the trader which provides a match making service, servers publish descriptions of services which they can offer, clients then try to meet their requirements by constraining the available offers which they are prepared to use.

After selecting a likely candidate the client then needs to establish a suitable communications path to the service. N.B. the service offer can only be regarded as a hint as the server may have failed after publishing it's offer.

It is also necessary for servers and clients to describe services in precisely the same terms. This may cause problems when trying to match objects created at different times as the vocabulary may have altered, i.e. old clients may be unable to recognise that services are suitable for use because of the terms in which they are described.

The client's handle on the service is called an interface reference.

Clients consume service features by invoking operations on the service provider, this causes a result to be returned to the client [ref interaction model].

The failure model [ref], discusses the client's view of a service in terms of expectations. To ensure expectations are met requires that the service has the desired attributes and that the behaviour of the service be continually monitored (otherwise faults cannot be detected). If operations are invoked using announcements then there is no result to monitor, hence the client cannot even know if the invocation took place let alone if it's expectations of the service are being met. This effectively precludes the use of announcements when interacting with services which are required to be reliable.

Servers must also take whatever measures are available to protect themselves against errant or crashed clients, many measures can be taken at bind time. For instance to guard against client crashes where the client has assumed some responsibility for server correctness (e.g. by acquiring a lock which it is expected to relinquish) the server may require that it's clients themselves be groups with a membership greater than n.

1.5 Transparencies

Note: Vocabulary leaves a lot to be desired here.

The concealment of aspects of an interaction from users is called *transparency*. A transparency masks, or compensates for, some unwanted property of the infrastructure which separates client and server. Transparencies may be thought of as comprising three sets of actions (*name needed here*) , client side actions, server side actions and those actions which occur within the end to end communications path. Not all of these need be present in all cases, however if more than one is present then compatibility problems may arise.

We need to know how to make choices so that the end-to-end behaviour meets the expectations of the objects at each end, i.e. we need a vocabulary for describing their properties and rules for determining conformity of behaviour.

A transparency may be composed of smaller units, either other transparencies or some more fundamental units of behaviour.

The smallest units are called *mechanisms* and these correspond to sufficient engineering functionality so that a particular characteristic of the infrastructure can be hidden, implemented or compensated for (*example needed*). Of course there may be many different mechanisms which can have the same end result.

To allow precise control over the operation of a mechanism requires that detailed guidance be provided as to to how low level issues are handled. This is

called supplying a *policy*. Typically, application control over transparencies will consist of supplying policies, only in rare circumstances will applications supply their own mechanisms and even more rarely will they provide their own transparencies.

Note: What are the interfaces to mechanisms and to policies?

Note: Transparencies and mechanisms may be combined, the composition structures are based on interfaces, the composition rules are in the computational model.

Note: It might be useful to attempt classify transparencies and mechanisms in terms of the fault tolerant redundancy classes above, i.e. it porvides so much storage redundancy, so much processing redundancy etc. This might give us the basis for a taxonomy by which we could compare and contrast different mechanisms and/or transparencies

Note: How can you determine the equivalence of different mechanism/policy options? How can you determine the compatibility of given client/server choices?

Note: Can a given mechanism constrain policy choices (e.g. user supplied policies), what is the language?

Note: Need a section on management, what it is, how do these actions interact with external management etc.

1.6 Options for implementing dependability

The provision of dependability for a service may be *transparent* or *non-transparent*, clients and servers need not have the same view, i.e. it may be transparent to the server but non-transparent to the client.

It is also possible for the support to be provided on only one side of the interaction, e.g. the client may do 'whatever it takes' (rebind, retry) to make the best of otherwise unreliable services.

Note: More needed.

If the service is provided transparently then the object which benefits from the transparency is unable to detect this by any difference in interaction characteristics (except that it may take longer or fail less often). Applications are also unable to direct or otherwise influence the actions of transparent dependability, but this may be done on their behalf by the user/system manager.

If the service is provided non-transparently then the object is able to influence the dependability attributes indirectly or directly. *Indirect influence* may mean that the object is aware that dependability is being provided and is able to observe or change some management parameters. *Direct influence* means that the application actively participates in the dependability process, perhaps even to the extent that some of the dependability support is implemented within the application.

There are three options for locating dependability functions: within the application, in the local infrastructure or in the communications path. These correspond to placement within the application, within the operating system or along the communications path.

Note: How are these named?, would they have the same interfaces?, how would you determine their presence or absence?

Note: The following tries to show what the three options mean in terms of transparent and both forms of non-transparent dependability, it's not altogether successful.

1.6.1 Within the application

This is the case when knowledge of dependability is being exploited, or it's behaviour guided, by the application, that is the dependability is non-transparent (see below for the case when support code is linked in).

Indirect influence may consist of selecting management and QoS parameters, either via the trader or at run time. Direct influence may comprise providing mechanisms and/or policies and participating in failure recovery (e.g. deciding if the recovered configuration is adequate).

1.6.2 Within the infrastructure

This is the most likely placement for common (or default) dependability choices. This placement is transparent to applications by default though applications which require to exert an influence are still able to do so. Placement within the infrastructure means that the placement is logically external to the application, e.g. the location may be in underlying layers of the system or in a wrapper placed around the application or even within the application in some situations, e.g. linked from a library.

Indirect influence may consist of supplying management parameters or making a selection from locally available options. Direct influence may consist of participating in failure recovery.

1.6.3 In the communications path

Transparent to remote (non-colocated/all?) applications, the point here is that because it is not in the local scope and hence not under local control it is not amenable to direct influence. Also attempting to influence the transparency may have wide ranging consequences.

In the transparent case, these would be chosen at bind time. The application may be able to exert indirect influence by specifying QoS constraints when trading.

Note: How would you detect the existence of these transparencies?, how would you name them?

1.6.4 Rough notes

General approach application code + declarative QoS statements => dependable application.

Stages:

select Qos requirements => need language/notation to express needs, allow for standard capabilities and ad-hoc extensions

augment application => choose whether to do this in application or in infrastructure

provide own capabilities => done before trading/binding - and so might limit servers which can be used

name own policies

trading and binding => how to decide conformance of mechanism and policy choices; defaults related to what is commonly available

dynamic modification of policies and recovery from errors => retrade, rebind, relocate

1.7 How dependability is structured

Note: These points probably need to be merged elsewhere.

The end-to-end dependability of an interaction is based on three factors: what the client does to enhance dependability, what happens in the communications path and what the server does to enhance dependability.

This can be broken down into five distinct activities:

- client coordination (including unilateral client actions to enhance dependability)
- client interaction with service
- transformations implemented along the communications path
- server interaction with clients
- server coordination (including unilateral server actions to enhance dependability)

Note: Naming

1.7.1 Client view of dependability

Three types of mechanism: those needed to prop up client itself (e.g. management), those actions which the client may take unilaterally regardless of the particular service and those which are needed to interact with a reliable service (e.g. collation of replies from an interface group).

Clients are able to select servers with the most suitable attributes.

Clients may be the first to detect server failures, they need to take corrective action and may choose to notify external management to initiate recovery.

1.7.2 Server view of dependability

Two types of mechanism: those needed to prop up the service itself (e.g. synchronisation for groups) and those which interact with clients directly.

Servers have no real control over who attempts to invoke them.

As above, servers may detect client failures.

1.7.3 Communications enhancements

1.7.4 Dependability and trading

Matching client requirements with server capabilities - need language for QoS

1.8 Making legacy applications dependable

Note: This is a set of questions at the moment, one way to make progress is to see how a couple of example legacy applications (UNIX client and server and DOS client and server) could be ANSAfied.

Within ANSA, a legacy application is one which was designed without an ANSA infrastructure in mind. If it is not possible to modify the internals to take advantage of an ANSA infrastructure any enhancements to increase dependability must be fitted outside of the application, such dependability is necessarily transparent.

Legacy applications which are network aware may be enhanced 'externally' by intercepting network traffic thus giving access to remote objects with modified characteristics. Others may need to be embedded in a wrapper to position them within a modified infrastructure.

Note: What about legacy applications that interact explicitly with their devices?

Here we can only do so much because there may be limitations in the way in which legacy applications can behave.

Clients and servers pose separate problems.

Note: Naming

1.8.1 Legacy clients

Initially the legacy client must provide a name and type of the service with which it wishes to interact. Often this is implicit (or equivalently) built into the application. This will determine the IDL representing the client's expectations and will require interaction with a trader so that an appropriate service may be imported.

The service is then 'opened' so that the client may access it, this may take the form of opening a file (or equivalent view, e.g. a UNIX pipe), accessing a device driver or by some other method.

Thereafter the client consumes the service, this must be mapped into invocations of the ANSA service. This requires argument mapping/marshalling, result unmarshalling/mapping, dealing with error terminations etc.

Finally the user will close the service.

What if the client uses multiple services at once?

How do we detect/handle conflicts between multiple clients onto the same server?

What is the client's name as far as the server is concerned, e.g. for charging purposes.

How do we provide the illusion of a dependable service (comes for free?)

How do you make the client appear to be dependable?

1.8.2 Legacy servers

How are these created, e.g. is it possible to manufacture them on demand?

How do they do they name themselves and interact with the trader in order to advertise their existence?, how is their IDL determined?

How they may be invoked by ANSA clients, how do you do validation to guard against erroneous clients? and detect/manage conflict between multiple clients?

Legacy servers may use client names of a given form for accounting purposes, how can ANSA clients manufacture such names

How to make the service appear to be dependable by e.g. replicating it or providing access to stable storage

Servers may also be clients.

1.8.3 Legacy clients invoking legacy servers

Essentially a free side effect of the above two, big issues here are IDL mapping, naming and trading.

References

[ANSA 91]

ANSA: A Systems Designer's Introduction to the Architecture, APM Ltd., Cambridge U.K., April 1991.