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 ANSA Approach to Open Dependable Distributed Computing

Nigel Edwards

Abstract

The basic technology for open distributed processing is now understood. The challenge now is to be able to deliver appropriate non-functional guarantees (e.g. reliability, availability and performance), and to be able to integrate existing services and systems into this world of open dependable distributed computing. It is important to realize that there will not be one set of non-functional guarantees which are appropriate to all applications; any solution must allow the selection of guarantees to match different application requirements. Until this challenge is met, open distributed computing will not be used in business critical applications.

The ANSA work on dependability aims to develop the technology for building open dependable distributed systems on industry standards platforms such as DCE and CORBA. A failure model has been developed and its use in the design of dependable systems is being investigated. An engineering model is being developed which will provide a choice of mechanisms to enhance the functionality of the basic platform to meet the requirements of applications for dependability. A programming model is being developed to help programmers meet the requirements of the chosen engineering. A core component of both the engineering and programming models is an extended transaction framework.

APM.1127.00.03 **Draft** 7 February 1994

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

Distribution:
Supersedes:
Superseded by:

Contents

The ANSA Approach to Open Dependable Distributed Computing

Dependability is increasing in importance in the market place. A recent Gartner report predicts the market for fault-tolerant systems will double in the next three years (from 'mid 1993) [Gartner]. In an increasingly fierce market, reliability and availability can have significant effects in reducing the cost of ownership [Siewiorek and Swarz, 1992], thus giving a vendor a competitive advantage. Within the context of large distributed systems, dependability will be particularly important: the more components a system has the greater the probability that one of those components will be faulty. In addition, openness further reduces the cost of ownership by allowing easy integration and incremental evolution of the information system [Herbert, 1993], [Harris and Fraser, 1993].

This paper argues that the basic technology for open distributed processing is now understood: there are now several de-jure and de-facto standards which are emerging. The challenge now is to be able to deliver appropriate non-functional guarantees (e.g. reliability, availability and performance), and to be able to integrate existing services and systems into this world of open dependable distributed computing. It is important to realize that there will not be one set of non-functional guarantees which are appropriate to all applications; any solution must allow the selection of guarantees to match different application requirements. Until this challenge is met, open distributed computing will not be used in business critical applications.

The purpose of this paper is as follows:

- to explain why dependability is important in open distributed processing §1;
- to examine what consequences the ANSA principles have for dependability §2;
- to explain why it is important to take an end-to-end view of dependability §3;
- to describe the technology being developed for dependability by the ANSA project §4 - §9.

1 The need for open dependable distributed systems (ODDS)

This section looks at the need for dependability in open distributed systems, the advantages to be gained by delivering the right solution quickly, and some of the constraints on the technology used to deliver ODDS.

1.1 Business critical applications need dependability

Deploying a business critical application or information service without any guarantees about the dependability of that application is analogous to

participating in a business transaction without any formal contractual arrangements. In the absence of any contract to set expectations, there is more chance of something unexpected happening — something which may be viewed by one of the parties involved as a failure. The consequences of such failure could be severe.

Similarly it could be disastrous for a business to rely on an application without well defined expectations — without clearly defined dependability guarantees. Hence exploiting open distributed computing to deliver business-critical information services will require the ability to offer both functional and non-functional (dependability) guarantees which are appropriate to the information service being provided.

1.2 Current technology does not address dependability

There are a number of de-facto and de-jure standards emerging which incorporate technology for distributed computing: ODP [ODP 93], CORBA [OMG 91], Atlas [UI], DCE [OSF 91] (including DME and ENCINA [Sherman, 1993]) and the various OSI standards (e.g. GDMO [GDMO], OSI RPC [OSI RPC], OSI TP [OSI TP] etc.). All of these provide applications (objects) with a means of communication. Perhaps the highest level of functionality is delivered by CORBA and related products such as DAIS [ICL 93] which supports object based distributed computing: objects can invoke each other regardless of whether or not they are co-located. In addition all these standards identify some basic services which are needed by applications, such as naming. Hence the technology for delivering basic “open distributed computing” is becoming well understood and standardised.

With the exception of ODP (which has been heavily influenced by previous ANSA work), very little work has been done on providing appropriate dependability guarantees [Herbert, 1993]. ODP with its notions of transaction, group and replication transparencies lays some of the foundations [ODP 93].

1.3 Gain a competitive advantage: match customer requirements

One of the basic principles of ANSA is that different customers and different applications will have different dependability requirements. Even within one application the different components will have different availability, reliability and consistency requirements. Understanding the engineering and cost trade-offs in building dependable distributed systems will enable the vendors to match the dependability delivered to the requirements of the customer and the application, giving them a competitive advantage over those who cannot do this.

1.4 Gain a competitive advantage: deliver the solution quickly

Competitive advantages are also gained by being able to deliver the right solution more quickly than the competition. One way of doing this is to minimise the amount of bespoke engineering in a solution. The approach should be to use tools, configuring basic standard engineering components to deliver the guarantees which are needed by the application.

1.5 The need to incorporate existing systems into ODDS

The need to preserve investments in existing information technology infrastructure means that new information services will have to interwork with so-called legacy systems and yet still provide a some guarantees about

dependability. This means that there will be few opportunities to build systems from scratch; rather, it will be important to understand how to configure mechanisms to get appropriate non-functional guarantees from what already exists. Openness implies the ability to be able to cope with heterogeneity at all levels: different machines using different operating systems interworking between different administrations.

1.6 Hardware versus software techniques

The ANSA work on dependability is about developing concepts which can be used for open dependable distributed computing. It aims to put in place the technology which enables the construction of information services with various dependability guarantees. Since openness implies minimising the assumptions about the underlying hardware and operating system, this work concentrates on software rather than hardware techniques for dependability, and on techniques which do not require one particular underlying platform for distribution.

2 The ANSA principles and dependability

[van der Linden, 1993] describes the principles of ANSA. This section looks at those principles which are particularly relevant to dependability; the principles are divided into seven categories — each category is considered in turn.

2.1 Separation

Systems should be designed so that separation amongst their parts can be achieved; this means that they can be more flexibly configured. However, this can have the effect of introducing more components reducing the dependability of the system.

Separation means that services may be remote. This introduces the possibility of partial failure: a failure may occur in a remote service request even though the requester's local system has not failed.

The ANSA work on dependability aims to ensure that the required dependability can be achieved in spite of the effects of separation.

2.2 Diversity

Large distributed systems will include many significantly different individual systems. This means that data will be widely distributed with multiple representations and different consistency requirements. It is inevitable that different standards and different dependability mechanisms will be adopted in different parts of the system; designers need to be prepared for this and the dependability mechanisms need to allow it.

2.3 Scaling

The dependability mechanisms used in a system must not impose constraints on the extent to which it can be interconnected and its applications made to interwork. Scaling is about scaling up and down: mechanisms which are efficient in large systems should be designed so they are efficient in small systems or else should be replaceable by mechanisms which are efficient in small systems.

In large distributed systems it is very difficult to implement a notion of universal time or an observer which can observe every event. This means that technologies which assume a global clock or a global ordering on events are not appropriate.

Larger systems will contain more components which increases the probability that there are one or more faulty components in the system.

2.4 Federation

Federation deals with heterogeneous authority and how to retain local control in a large distributed system spanning boundaries of authority. The consequences of federation mean that objects are responsible for their own dependability. In addition objects will need to negotiate contracts with objects subject to other authorities: there may be no common higher authority which lays down what the contract should be. The contract will state what each object is entitled to expect of the other (i.e. what the “correct behaviour” should be).

2.5 Transparency

A property of a system is transparent if application programmers need not be concerned with it. The aim of the ANSA work on dependability is to hide the details of the dependability mechanisms from the application programmer.

Previous experience suggests that it is possible to make dependability mechanisms such as replication completely transparent to the programmer [Oskiewicz and Edwards, 1993]. However, there are limitations on making dependability fully transparent. These are explored in §3.

There is no universal set of requirements for dependability hence, there is no universal configuration of mechanisms. This means that transparency must be selective: programmers can select and configure the mechanisms which are most appropriate to the job at hand.

Selecting and configuring the appropriate mechanisms is likely to be a complex and error prone task. Programmers may well be tempted to implement their own mechanisms, ignoring the ones provided, because they are too difficult to understand and use. This means that programmers must at least be given guidance on how to select and configure mechanisms to match the requirements of their programs. Where possible, tools should be provided to configure and select the mechanisms (this is automated transparency).

Ideally the dependability requirements should be declared as attributes of the object; tools would then configure the most appropriate mechanisms. The difficulty of capturing requirements and the lack of tools which work directly from them, means that programmers will probably have to specify specific mechanisms. Automated transparency techniques will configure the specific mechanisms selected. The programmer is protected from the details of the mechanisms (e.g. see [Warne and Rees, 1993]).

2.6 Concurrency

Concurrency is inevitable in distributed systems. This means that there is potential for conflicting inconsistent changes to be made to data. Mechanisms are needed to prevent this.

2.7 Configuration

Systems evolve over time: new parts are added and old parts are removed. ANSA advocates detection and correction of faults as early as possible, ideally before a new component is configured into the system. This limits the potential for a fault in one component to cause damage to the rest of the system. To achieve this in a dynamic system, the description (of the correct behaviour) of a component must be on-line. Such descriptions will form the basis of the contracts described in §2.4 and are important in fault diagnosis (see §5.1).

3 An end-to-end view of dependability

Dependability is an end-to-end concept. What is dependable and what constitutes a failure to an application can only be understood by understanding the application semantics: it is not sufficient to consider dependability purely in terms of protocols provided by the underlying engineering and platforms. For example, a file transfer is completed successfully when all the file data has been safely and correctly stored in the file system of the recipient machine, not just when the data has been delivered by the network to the machine (it may crash before storing the data) [Saltzer et al., 1981]. Consequently dependability needs to be considered at the system design stage and throughout the development of the system.

Some techniques have a minimal effect on the structure of the system. For example within ANSA technology for transparent replication has been developed [Oskiewicz and Edwards, 1993]. Provided the client or server satisfied certain (well understood) assumptions it is possible to hide replication from the application programmer, so that the decision to replicate or not can be made after the code has been written. This dramatically increases the complexity of the underlying engineering required to support the application components with consequent loss of performance (compared with non-replicated code). Hence, the scope and potential of technology for transparent dependability is limited.

Other techniques for dependability require more participation from the designer and programmer, but may result in applications which need less complicated engineering to support them and have better performance. For example, suppose a service needs to replicated. If a service can be designed so that it is immutable (does not change its state when invoked) the underlying support for replication can be made much simpler, since no mechanisms is required to coordinate state changes between the replicas (it never changes). To make a service immutable may require the designer to take special steps, for example, to ensure that any state concerning the interaction between client and server is held by the client. In addition the programmer of the service needs to avoid code which makes changes to state.

This paper describes technology which is being developed which allows designers and programmers to consider dependability issues, as well as technology which tries to make dependability transparent. In the former case the concept of selective transparency is important: hiding irrelevant detail from the programmer.

4 Dependability in ANSA

There are two basic techniques used to build dependable systems: fault tolerance and fault avoidance (sometimes called fault intolerance). Fault avoidance involves using good engineering practice to minimise the occurrence of faults. Fault tolerance exploits redundancy to negate the effects of faults.

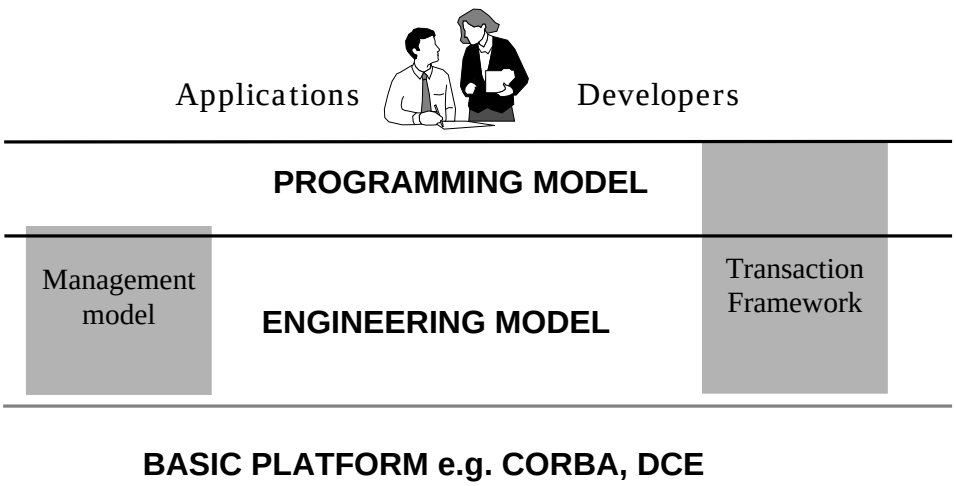
It is important to realise that these two techniques are complementary and not alternatives. Good engineering practice reduces the occurrence of faults, unless the rate at which faults occur is reduced to an acceptable level any redundancy (fault tolerance) will be quickly overwhelmed.

The aim of the ANSA work on dependability is to develop technology which allows application programmers and system designers to use a set of simple concepts to declare their dependability requirements. These requirements will be mapped quickly and efficiently onto a rich set of engineering mechanisms which exploit various redundancy and consistency techniques to deliver the required dependability. The component technologies are:

- A **failure model** which provides the underlying concepts;
- A **programming model** which provides programmers with abstractions for building dependable applications;
- An **engineering model** which provides a set of engineering mechanisms and sets of standard configurations of these mechanisms;
- An **extended transaction framework** providing a programming model and set of engineering mechanisms based on transactions;
- A **management model** which provides the mechanisms and concepts for fault diagnosis and reconfiguration to maintain dependability.

Figure 1 shows the relationship between the programming model, the engineering model, the management model and the transaction framework.

Figure 1: How the ANSA dependability work fits together



The engineering model provides a set of services to enhance the functionality provided by basic platforms for distribution such as DCE and CORBA. Application programmers build dependable applications using concepts provided by the programming model and services provided in the engineering model.

The system designer needs to elucidate the requirements of the application components for dependability (from the supporting engineering) and also the requirements the engineering components impose on the behaviour of the application components. Designers need to be familiar with both the engineering and programming models to make the trade-offs between what (dependability) can be provided by the engineering and what can be provided by the application components.

The role of the engineering model is to provide a choice of services. It helps the system designer to choose the service which satisfy the application components requirements (for dependability). The role of the programming model is to ensure that the application components meet the requirements of the engineering components. The role of the failure model is to provide the concepts for stating requirements.

The management model is concerned with such issues as the maintenance of dependability in the presence of faults (fault diagnosis and reconfiguration), how to install new applications and how to upgrade existing ones. The services required for management form part of the engineering model. In addition some concepts in the programming model may deal specifically with management issues.

The use of different kinds of transactions to build dependable systems is being studied. The services provided by the advance transaction framework will form part of the engineering model. The framework will also provide abstractions to help programmers use these services. These abstractions are part of the programming model.

The remainder of this paper describes the above work in more detail. At the time of writing, the failure model has been developed, development of the extended transaction model is ongoing and work on the remaining areas is just beginning.

5 Failures and the ANSA failure model

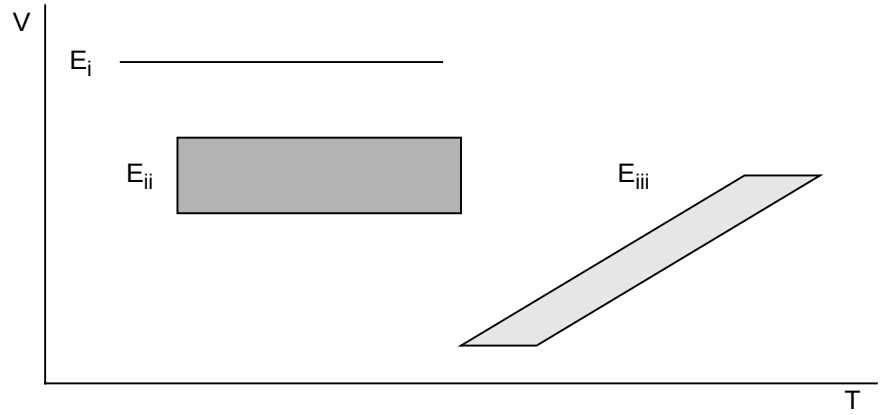
Understanding the concept of failure is crucial to building dependable systems. This section looks at failures: how to understand who is responsible for a failures and the role of failure models. We begin with a summary of the ANSA failure model [Edwards and Rees, 1993].

The ANSA failure model assumes that a system is composed of components which can engage in events which are observed by other components in the system. An **event** is considered to occur with some value at some time, by the observer; there is no notion of a global observer, a global ordering on events or a global time. An event which occurs is called an **occurrence**. The model defines **expectation regions** (a region in the $value \times time$ space) which define a time interval and a restricted set of values within which a component expects to observe an event. Boundaries can be drawn around an object's expectation region by determining an objects **expectations**: what it expects from the objects with which it interacts. Three example are shown in figure 2.

The three examples are:

- E_i: a particular value is expected in some time interval
- E_{ii}: one of a range of values is expected in some time interval

Figure 2: Expectations



- E_{iii}: one of a range of values is expected in some time interval but the value is related to the time at which the event occurs.

A **failure** is a mismatch between an occurrence and an expectation. For example, an occurrence which does not match what is expected: any occurrence not in regions E_i, E_{ii}, or E_{iii} would be a failure. The absence of an occurrence when there is an expectation of one is also a failure; this kind of failure is conventionally called an omission failure [Laprie, 1992].

Currently we are investigating how to use this failure model in the design of dependable systems. The methodology we are testing is as follows. Having identified the major application components (clients and servers) using appropriate modeling and design techniques (e.g. [Rumbaugh et al., 1991]), we analyse the expectations which each component has of the components with which it is interacting.

At this stage the design may be refined so that the expectations of some components are minimised. The less that is expected of a component the less its potential to cause damage to the rest of the system. For example, suppose it is known that a client must be located on a host which has inherent low availability — perhaps because it is mobile. It may be appropriate to redesign the interaction between the client and the services it uses so that any interaction with the client is stateless. This means that the services are never in a state in which they are expecting the client to do something (e.g. commit or abort a transaction).

The next step is to identify the facilities which the engineering and underlying platform needs to provide to meet the expectations of the application components. For example, suppose a client has an expectation it will be charged for a service only if it successfully uses it (i.e. both events occur or neither occurs). One option to satisfy this expectation could be to use atomic actions [Warne and Rees, 1993].

The engineering model helps the system designer or programmer to choose the appropriate configuration of engineering mechanisms (see §7). The programming model helps the programmer to write application components which have the properties required by these engineering mechanisms (see §6).

5.1 Fault diagnosis

Fault diagnosis is the process of identifying the faulty component which is responsible for a failure. This can be difficult because the fault may have propagated from the original faulty component by causing other components to fail. If a fault is wrongly attributed to a particular component, erroneous reconfigurations are sure to follow [Schnieder, F.B., 1993]. In particular good components may be decommissioned while the faulty component is left in the system.

The relevant components in an ANSA system are clients and interfaces. Interfaces are the place where contracts take effect (between client and server), and where reconfiguration is possible. They are also the place where federation boundaries may exist. Fault diagnosis tries to isolate the fault to the particular client or interface from which the fault originated. Sometimes fault diagnosis may have to stop at a federation boundary: beyond that boundary diagnosis is the responsibility of another organisation.

The traditional concept of a failure focuses on service: a failure is said to occur when a service deviates from its specification [Laprie, 1992], [Siewiorek and Swarz, 1992]. In ANSA the consequences of federation and separation mean that the consequences of mutual suspicion are extremely important. One should not take a client's word for it that a service has failed — it may be that the client itself has failed. The ANSA failure model [Edwards and Rees, 1993] captures this: it does not prejudge whether the faulty component is the one which engages in the event or the one which observes or expects to observe the event.

This leads to a situation which is potentially ambiguous: either the observer or the component which engaged in the event may have failed. To avoid this, the parameters used to determine correctness must be made explicit.

The notion of what is correct ideally should be captured by a formal contract between two objects. Interface definitions are a form of contract between a client and server, the stronger these contracts the easier it is to avoid ambiguity. Unfortunately, usually correctness is captured only partially in an interface definition and written text.

The ANSA work on federation is investigating contracts between clients and servers [Beasley et al, 1994]. The programming model for dependability will involve investigating enhanced interface definitions (see §6). This will allow stronger statements to be made about expected behaviour.

5.2 Failure models and hierarchies of failure modes

A failure mode describes the characteristics of a class of failures (e.g. omission failures, value failures, crash failures, fail-stop [Laprie, 1992]). There are many hierarchies of failure modes in the literature (e.g. [Barborak et al, 1993], [Shrivastava et al., 1990], [Cristian, 1990]). Such hierarchies are sometimes referred to as "failure models". In contrast the ANSA failure model is not a failure hierarchy; it provides a set of concepts for understanding the semantics of failure.

Failure hierarchies arise from partial orders on failure modes; they are useful, because they say when one engineering mechanism can replace another. For example suppose there is an ordering $\subseteq$ on failure modes, and suppose there are two failure modes x and y such that $x \subseteq y$. Then any mechanism which can detect and tolerate y will also detect and tolerate x. Suppose x is omission failures and y is value failures, whether or not a mechanism actually detects

and tolerates both *x* and *y* will depend on the implementation of that mechanism. Hence it is the engineering model (which prescribes the configuration and implementation of engineering mechanisms) which will determine the ordering on failure modes. Different engineering models will give rise to different orderings; within an engineering model different arrangements of components may produce different orderings. The ANSA engineering model for dependability is discussed further in §7.

Failure modes are useful in the engineering of dependable systems. For example, if the failure behaviour of a server is known to be restricted to a well understood mode (e.g. fail-stop), the engineering mechanisms used by its clients only need to be able to deal with this behaviour.

[Edwards and Rees, 1993] shows how the ANSA failure model can be used to describe the failure modes which are discussed in the literature. During the development of the engineering model for dependability it is intended to use the ANSA failure model to analyse configurations of engineering mechanisms to determine what failure modes they can detect and tolerate.

6 The programming model

This section discusses a number of requirements have been identified for the programming model. The role of the programming model is to provide the concepts to assist programmers in making sure that application components meet the expectations (requirements) of their supporting engineering components. Together with the engineering model, it enables system designers to make trade-offs between what is provided by the engineering components and what is provided by the application components. As an example suppose there is a choice of replication mechanisms including one that consumes fewer resources, but can only be used if the state of a service is immutable. There are a variety of possible tools and techniques that can be used to exploit this opportunity.

1. A tool to analyse the application code automatically and report whether or not it has the immutability property.
2. Guidelines and rules for the programmer which explain how to write the code so that it has the appropriate characteristics.
3. A tool that transforms the code so that the most appropriate mechanism is used. This approach has already been investigated for an atomic activity infrastructure: code transformation was used to insert calls to appropriate lock mechanisms whenever mutable state was accessed [Warne and Rees, 1993].

The scope of this technology will be set in part by the programming language which is used. Some languages are less amenable to transformation and automatic analysis than others.

It is usual to perform type checking based on information provided in an interface definition. The interface definition is a very limited specification of the expected or allowed behaviour. Part of the intended work on the programming model will be to extend this technology by adding information about the expected behaviour of the interface. For example:

- Whether an operation updates or observes mutable state.

- How an operation affects the outside world: does it read information on the outside world (sensor) or does it make changes to the outside world (actuator).

If type checking tools can be enhanced to check these attributes, then at least some expectations can be checked automatically.

Programmers may choose to exploit application level redundancy — redundancy which is associated with the application semantics. For example:

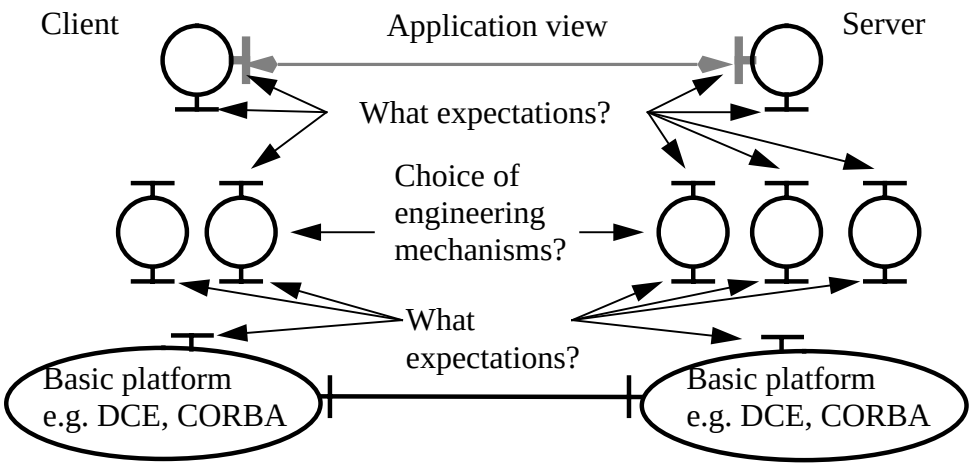
- Knowledge which restricts the time or value of a result, e.g. time should not run backwards; this kind of redundancy could be captured by behavioural descriptions of objects
- Alternative algorithms for achieving a particular aim (this is sometimes called resourcefulness [Abbott, 1990], recovery blocks is one example which exploits this idea [Randell, 1975])
- The use of stability and self stabilisation [Schneider, M., 1993]

At present there are no plans to investigate explicitly such redundancy; redundancy is provided by the engineering model.

7 The engineering model

This section discusses some of the requirements which have been identified for the engineering model. The engineering mechanisms are positioned between the underlying platform and the application components as shown in figure 3. Failures can occur in the application components, the underlying basic platform or the engineering mechanisms themselves.

Figure 3: The relationship between the programming and engineering models



As shown in figure 3 the application components and engineering will have requirements (expectations of each other). The role of the engineering model is to help the designer select a configuration of engineering mechanisms which ensure that the expectations of the application components are satisfied even when failures occur. The designer also uses the programming model to makes trade-offs between what is provided by the engineering and what is provided by the application components.The engineering mechanisms enhance the functionality of the basic platform so that it meets the requirements for

dependability. The engineering model also need to ensure that the expectations of the engineering mechanisms and underlying basic platform are matched and vice-versa.

The engineering model will consist of a set of basic services which are useful for building dependable systems, for example: reliable multicast, state transfer mechanisms, audit services, persistent storage services.

Each service will have a typed interface (just like any service which is visible to the application). This will allow type checking to be applied to engineering objects (as well as application objects). Currently checking tends to be limited to application level objects, because the configuration of supporting engineering objects tends to be rather static and uniform. Well understood, standard configurations do not require constant checking and validation. In the future the aim is to allow dynamic configuration of engineering objects specifically to match client and server requirements, each new configuration will require checking.

The engineering model will prescribe standard configurations of mechanisms which will have well understood behaviours (and expectations). These configurations must themselves be dependable. It is intended to use the concepts in the failure model and the methodology outlined in §5 in their development. Examples of such configurations could be:

- a configuration of mechanisms to ensure fail-stop behaviour;
- replication for a non-mutable service;
- replication for a mutable service.

Initially the engineering model is likely to consist of a set of mechanisms and a set of standard configurations for those mechanisms. However, structuring the mechanisms as services will allow recombination of these mechanisms in application specific ways. Application specific configuring of the engineering mechanisms is likely to be a complex an error prone task. Unless proper support is provided to help programmers select the right configuration of mechanisms they are likely to be tempted to ignore what is provided and build their own. This suggests that eventually it will be necessary to provide tool support for configuration.

The opportunities to build completely new systems are becoming fewer and fewer. In general new applications and systems will have to interwork with what already exists. If a new application is to be dependable, the dependability of the existing services with which it interworks needs to be evaluated. If they do not provide sufficient dependability they need to be enhanced in some way, or at the very least the new application needs to be protected from them. The engineering model needs to provide mechanisms and configurations of mechanisms to do this.

Much of the engineering model is concerned with the provision of redundancy to give fault tolerance. Redundancy can be provided in the form of extra storage, processing or communications, further it can be provided in space or time (e.g. doing the same thing twice simultaneously or sequentially). Examples include:

- Replication [Oskiewicz and Edwards, 1993]
- Checkpointing [Birrell et al., 1987]

Comprehensive lists of redundancy technology are given in [Siewiorek and Swarz, 1992] and [Smethurst and Wharton, 1993].

The engineering model will also provides some mechanisms which enforce requirements; these can be regarded as fault avoidance mechanisms. An example of such a mechanism is a protocol which enforces ordering between messages to ensure a group of servers see messages in the same order. This avoids the state of the servers becoming inconsistent. (Note that the above mechanism may be itself part of a replication protocol which is intended for fault tolerance).

The engineering model provides a set of mechanisms to supplement and support application redundancy. The engineering model is not only concerned with the provision of redundancy, it is also concerned with the management of redundancy, the latter is considered separately in §8.

8 The management model

This section discusses some of the requirements which have been identified for the management model.

The part of the engineering model discussed in §7 is concerned mostly with the provision of mechanisms which provide redundancy. The management model includes the part of the engineering model which is concerned with how to manage this redundancy to tolerate faults, how to maintain dependability, how to install new applications and how to upgrade existing ones. For example consider an active replica group; managing the group involves providing mechanisms which can detect failures, reconfigure the group to remove the faulty member, and adding new members to maintain the level of dependability when existing ones fail. A redundant system may go through as many as eight stages when a failure occurs [Siewiorek and Swarz, 1992].

1. **Fault confinement** is concerned with limiting propagation of the fault. This involves liberal use of detection mechanisms to try and detect a fault as soon as possible.
2. **Fault detection** is measuring value and time and comparing what is observed to what is expected.
3. **Fault diagnosis** is used if fault detection does not identify the faulty component. Fault diagnosis is discussed in §5.1.
4. **Reconfiguration** takes place once the faulty component has been identified. The aim is either to isolate the system from the faulty component or to replace it with a spare.
5. **Recovery** attempts to remove the effect of the fault. Redundant information can be used to correct the erroneous state (space redundancy). Alternatively the system can roll (backwards or forwards) and either retry or try an alternative strategy (time redundancy).
6. **Restart** takes place once all the damaged state has been removed. In extreme cases large parts of the system may need to be restarted from its initial state.
7. **Repair** restores the faulty component to an undamaged state. Redundancy might be used to correct erroneous state.
8. **Reintegration** involves reconfiguring the system to introduce the repaired component.

If the system has requirements for high availability all these stages may have to take place “on-line”.

Management will usually be embedded into the redundancy mechanisms which they manage. However, often it is necessary and convenient to have separate management and service interfaces for the redundancy mechanisms e.g. [Oskiewicz and Edwards, 1993], even if the interfaces are onto the same object.

Just like all the engineering mechanisms, the management mechanisms themselves need to be dependable — system recovery mechanisms can be responsible for 35% of system failures [Toy, 1993].

9 Extended transactions framework (ETF)

Development of the extended transaction framework (ETF) is ongoing. This section summarises the work to date and possible future directions.

Transactions exploit both fault avoidance and fault tolerance. For example computations enclosed within traditional transactions have well defined relationships with each other: they are constrained by the fundamental properties of atomicity, consistency, isolation, and durability (collectively known as the ACID properties [Bernstein et al., 1987]). This helps to avoid faults which can be introduced by concurrent interfering computations.

Transactions use redundancy to undo their effects should they need to abort (fault tolerance). For example, the Tandem transaction processing monitor (Pathway) distributes work to available processors. Should any of this work be lost or compromised by failure it is automatically restarted after being rolled back to its initial state [Bartlett et al. 1992].

The traditional transaction model, with its strict ACID properties, is highly effective in some application areas such as conventional databases. However, it frequently found lacking in functionality, flexibility, and performance when used in other applications areas, especially those involving collaborative or long-lived activities. Such applications typically require some, but not all of the ACID properties. This has led to the development of many different kinds of transaction models, for example: Split Transactions [Pu et al., 1988], Coloured Transactions [Shrivastava and Wheater, 1990] and Transaction Groups [Skarra, 1989].

As observed in [Chrysanthis and Ramamritham, 1990], irrespective of how successful these extended transaction models are in supporting their intended application domains, they merely represent points within the spectrum of interactions possible within competitive and cooperative environments. Therefore, they each capture only a subset of the interactions to be found in any complex information system.

ETF is intended to address these concerns; it is intended to support a wide range of telecommunications and other business applications. Inevitably, different classes of applications will require different transaction models. These model will have: differing concurrency control methods; differing recovery procedures; differing resource placement, migration and replication strategies; and differing timeliness (execution responsiveness) guarantees. ETF must enable such application diversity to interoperate effectively.

ETF identifies a number of new primitives for controlling the behaviour of different transaction models. The inspiration for these primitives stems from the abstract concepts of the ACTA meta-model [Chrysanthis and Ramamritham, 1992]. The transaction model is characterised by controlling four basic attributes of a transaction. ETF provides primitives to control these attributes (VPCR).

- **Visibility**: the degree with which members of the extended transaction are able to observe each others effects before the transaction as a whole terminates its execution and commits or aborts.
- **Correctness**: the acceptable effects on system state that members are permitted to produce.
- **Permanence**: the rules by which members are allowed to record their results in the stable state of the system.
- **Recoverability**: the capability of an extended transaction as a whole, or its members in part, in the event of failure, to recover and take the system to some state that is considered correct.

Hence ETF allows programmers and designers to construct transaction models which match the requirements of the application object (providing a service). It is intended to use the concept of transaction-based work flows (e.g. [Dayal et al., 1993], [Sheth, 1993]) to describe how different application objects supporting different transactions models fit together. A workflow will consists of several related transactions which interwork to achieve a specific goal. Each transaction in a work flow may be different when characterised in terms of VPCR.

From figure 1, it can be seen that EFT spans both the programming and engineering models. It is intended to build engineering mechanisms for EFT which support the control of the VPCR attributes. Additional mechanisms will be needed to support the concept of work flows. The programming model will contain the concepts needed to drive the engineering mechanisms supporting EFT.

At the time of writing ANSA is experimenting with primitives for controlling the VPCR attributes. The next step is to develop the concept of workflow in the manner described above. Finally the engineering mechanisms and programming model concepts can be developed.

10 Evaluating dependability

It is hard to measure all the parameters which affect the dependability of a design. For example, it depends on the effects of other applications and users on the network. Thus in general it is difficult to make absolute measurements of dependability; it is easier to perform evaluations which compare one design to another. For example a statement such as: “Design A has an MTTF (mean time to failure) 1.6 times that of Design B” is a comparative statement. Given a comparative measurement it may be possible to turn it into an absolute measurement. It may be known that design B has an MTTF of 8000 hours, so A’s MTTF is predicted to be 12,800 hours. Thus comparative measurements allow the effects of changes in the engineering to be measured.

One possible piece of future work is to analyse and compare the configurations of engineering mechanisms in the engineering model using techniques such as Markov Modeling [Siewiorek and Swarz, 1992]. This will give relative measures of the dependability of different engineering options

11 Summary and conclusions

There are several de-jure and de-facto standards which are emerging which will provide the technology to make open distributed computing possible. However, for open distributed computing to be exploited in business-critical applications more technology is required. In particular there is a lack of technology which will enable services to be delivered with appropriate and defined dependability guarantees.

It is important to realise that there is not going to be one set of dependability requirements. Rather the dependability requirements will be determined by the application semantics. The concept of selective transparency can be used to select an appropriate set of engineering mechanisms and configure those mechanisms to satisfy a particular requirement. This needs to be automated.

The ANSA principles reveal a number of issues for dependability in open distributed systems.

- Objects are responsible for their own dependability; contracts must be used in a federated environment to state what each object is entitled to expect of others.
- Technologies based on a notion of a global observer or global clocks are not appropriate: they cannot be realised in large distributed systems.
- Faults should be detected as early as possible, ideally before a component is installed into the system.

Designers of dependable systems should take an end-to-end view: careful design and partitioning of functionality can reduce the need for complicated and sophisticated engineering mechanisms to support an application.

The ANSA work on dependability aims to develop the technology for building open dependable distributed systems. It is anticipated that such systems will be built using industry standards such as DCE and CORBA platforms.

A failure model has been developed and its use in the design of dependable systems is being investigated. One of the roles of the designer is to identify the requirements of the application components in terms of what each component expects from the underlying engineering. The notion of expectations in the failure model can be used for this. The engineering model then helps to identify a suitable configuration of mechanisms to meet these expectations. The engineering components enhance the functionality of the basic platform so that it can meet the application's requirements for dependability.

The designer also need to state what constraints the application components must meet, i.e. what the engineering expects of the application components. Again the notion of expectations can be used for this. The programming model helps the programmer to build application components which satisfy these constraints.

Taken together the programming and engineering model enable system designers to make trade-offs between what dependability is provided by the engineering mechanisms and what is provided by the application components.

As part of the work on programming and engineering models, an extended transaction framework is being developed: we believe transactions are a fundamental technology in the building of dependable systems.

12 Acknowledgements

The author would like to acknowledge the contribution of his colleagues in the ANSA team to this work: Ed Oskiewicz, seconded to the ANSA team by BT; Owen Rees of APM Ltd.; John Warne, seconded to the ANSA team by BNR. In addition comments and discussions with the following on various aspects of the work reported here were most helpful: Brian Coan of Bellcore, Andrew Herbert of APM Ltd., Santosh Shrivastava of The University of Newcastle and Paul Vickers of Hewlett-Packard.

References

[Abbott, 1990]

Abbott, R.J., “Resourceful Systems for Fault-Tolerance, Reliability and Safety”, ACM Computing Surveys, Vol. 22, No. 1, March 1990, p35-68.

[Barborak et al, 1993]

Barborak, M., Malek, M., Dahbura, A., “The Consensus Problem in Fault-Tolerant Computing”, ACM Computing Surveys, Vol, 25, No. 2, June 1993.

[Bartlett et al. 1992]

Bartlett, J., Bartlett, W., Carr, R., Garcia, D., Gray, J., Horst, R., Jardine, R., Jewett, D., Lenoski, D., McGuire, D., “Fault Tolerance in Tandem Computer Systems”, in [Siewiorek and Swarz, 1992], p586- 648.

[Bernstein et al., 1987]

Bernstein, P.A., Hadzilacos, V., Goodman, N., “Concurrency Control and Recovery in Database Systems”, Addison-Wesley Publishing Company Inc., 1989.

[Birrell et al., 1987]

Birrell, A.D., Jones, M.B., Wobber, E.P., “A Simple and Efficient Implementation for Small Databases”, in Proc 11th ACM Symp on OS Principles, 1987, ACM OS Review, Vol. 21, No. 5 p149-154.

[Beasley et al, 1994]

Beasley, M., Thomas, G., Cameron, J., Hoffner, Y., van der Linden, R., “Establishing Co-operation in Federated Systems”, to be published, ICL Technical Journal.

[Chrysanthis and Ramamritham, 1992]

Chrysanthis, P.K., Ramamritham, K., “ACTA: The Saga Continues”, Database Transaction Models for Advanced Applications, Edited by Ahmed K. Elmargarmid, Morgan Kaufmann Publishers, 1992.

[Chrysanthis and Ramamritham, 1990]

Chrysanthis, P. K., Ramamritham, K., “ACTA: A Franework for Specifying and Reasoning about Transaction Structure and Behavior”, Proceeding of the ACM SIGMOD International Conference on the Management of Data, 1990.

[Cristian, 1990]

Cristian, F., “Understanding Fault-Tolerant Distributed Systems”, IBM Research Report, RJ 6980 (66517) 8/24/89 (revised 4/6/90), Almaden Research Center, California, USA.

[Dayal et al., 1993]

Dayal, U., Hsu, M., Ladin R., “Organizing Long-Running Activities and Triggers and Transactions”, ACM SIGMOD Proceedings, 1990.

[Edwards and Rees, 1993]

Edwards, N.J., Rees, R.T.O, "A Model for Failures in Dependable Systems", APM.1027, APM Ltd., Cambridge, U.K., 1993. (Submitted for publication.)

[Gartner]

Figures quoted and attributed to the Gartner group in "Supply and demand", J. Kaye, Informatics September 1993.

[GDMO]

"Guidelines for the Definition of Managed Objects", ISO/IEC 10165 Part 4.

[Harris and Fraser, 1993]

Harris, R.J., Fraser, R.J.C., "Command and Control Infrastructures: The need for Open System Solutions", Keynote Address, IEE International Workshop on Systems Engineering for Real Time Applications, 13 -14 September 1993.

[Herbert, 1993]

Herbert, A.J., "Open Distributed Processing — the Solution to a Business Need", APM.1055, APM Ltd., Cambridge U.K., 1993.

[ICL 93]

"DAIS: System Overview", ICL manual R30428/03, December 1993.

[Laprie, 1992]

Laprie, J.C. (ed.), "Dependability: Basic Concepts and Terminology", Springer-Verlag, 1992

[van der Linden, 1993]

van der Linden, R., "An Overview of ANSA", AR.000.00, APM Ltd., Cambridge U.K., May 1993.

[ODP 93]

"Basic Reference Model of Open Distributed Processing", ISO/ IEC JTC1/SC21, American National Standards Institute, New York, USA, 1993.

[OMG 91]

The Common Object Request Broker: Architecture and Specification, OMG Document Number 91.8.1, August 1991.

[OSI TP]

"OSI Distributed Transaction Processing (OSI TP), IOS/IEC 10026.

[OSI RPC]

"Open Systems Interconnection — Remote Procedure Call", ISO/IEC 11578 (draft).

[OSF 91]

Introduction to OSF DCE, Open Software Foundation, December 1991.

[Oskiewicz and Edwards, 1993]

Oskiewicz, E.O., Edwards, N.J., "A Model for Interface Groups", AR.002.01, APM Ltd., Cambridge U.K., February 1993.

[Pu et al., 1988]

Pu, C., Kaiser, G., Hutchinson, N., "Split Transactions for Open-Ended Activities", IEEE Proceedings of the 14th Conference on VLDB, 1988.

[Randell, 1975]

Randell, B., “System Structure for Software Fault Tolerance”, IEEE Trans. on Software Engineering, SE-1, No. 2, June 1975, p220-232.

[Rumbaugh et al., 1991]

James Rumbaugh, Michael Blaha, William Premerlani, Fredrick Eddy, William Lorensen, “Object-Oriented Modeling and Design”, Prentice-Hall International, 1991.

[Saltzer et al., 1981]

Saltzer, J.H., Reed, D.P., Clark, D.D., “End-To-End Arguments in System Design”, in Proc. 2nd International Conference on Distributed Systems, Paris, France, 8-10th April, 1981, p509-512.

[Schnieder, F.B., 1993]

Schnieder, F.B., “What Good are Models and What Models are Good?” in Distributed Systems, Second Edition, Mullender, S., (ed), Addison-Wesley, 1993.

[Schneider, M., 1993]

Schnieder, M., “Self-Stabiliszation”, ACM Compting Surveys, Vol. 25, No. 1, March 1993, p45-67.

[Sheth, 1993]

Amit Sheth, Marek Rusinkiewicz, “On Transaction Workflows”, Data Engineering Bullitin, June 1993.

[Shrivastava and Wheater, 1990]

Shrivastava, S.K. , Wheater, S.M., “Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions”, n Proc. 10th International Conference on Distributed Systems, Paris, France, 28th May - June 1st, 1990.

[Shrivastava et al., 1990]

Shrivastava, S.K., Ezhilchelvan, P., Little, M., “Understanding Component Failures and Replication in Distributed Systems”, ISA Project Report: UNT/ TR1, University of Newcastle May 1990.

[Sherman, 1993]

Sherman, M., “Distributed Transaction Processing in a DCE Environment with Encina”, Tutorial presented at 13th International Conference on Distributed Computing Systems, Pittsburgh, USA, May 1993.

[Siewiorek and Swarz, 1992]

Siewiorek, D.P., Swarz, R.S., “Reliable Computer Systems — design and evaluation”, Second Edition, Digital Press, 1992.

[Skarra, 1989]

Andrea H Skarra, “Concurrency control for cooperating transactions in an object-oriented database”, SIGPLAN Notices, 24(4), April 89.

[Smethurst and Wharton, 1993]

Roy Smethurst, Peter Wharton, “ OPENFramework Availability”, Prentice-Hall 1993.

[Toy, 1993]

Toy, W.N., “Fault-Tolerant Deisgn of AT&T Telephone Switching System Processors”, in [Siewiorek and Swarz, 1992], pp533-574.

[UI]

“UI ATLAS Distributed Computing Architecture: A Technical Overview”, Unix International.

[Warne and Rees, 1993]

Warne, J.P, Rees, R.T.O, “ANSA Atomic Activity Model and Infrastructure”, AR.004.01, January 1993, APM, Ltd., Cambridge, U.K.