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

A Model for Failures in Dependable Systems

Nigel Edwards, Owen Rees

Abstract

This document describes a model for failures in dependable systems. A general failure model is described in terms of a system consisting of interacting components. This model is then applied to an object-based interaction model.

The model is based on events which occur with some value at some time. Components in the system observe events and have expectations which define regions in a value, time space. A failure is detected when what is observed does not match what is expected.

The concepts in the model can be used to analyse a given configuration of engineering mechanisms, application components and infrastructure to determine what failures can and cannot be tolerated by this configuration. This can then be mapped onto an application-level statement: what failures the applications can and cannot tolerate.

The intention is that the model should provide the underlying framework for further work on dependable distributed computing. Some familiarity with basic principles of object-based distributed computing is assumed.

.

APM.1143.01 **Approved** 14 March 1994
Technical Report

Distribution:
Supersedes: APM.1027 and APM.1046
Superseded by:

A Model for Failures in Dependable Systems

A Model for Failures in Dependable Systems

Nigel Edwards, Owen Rees

APM.1143.01

14 March 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

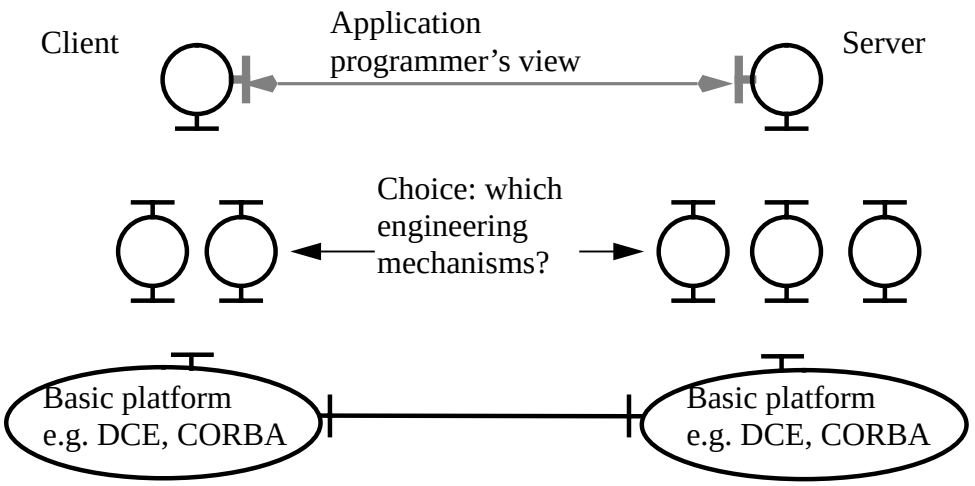
1 A Model for Failures in Dependable Systems

This paper describes a model for failures in dependable systems. The motivation for the model comes from the need for dependability in Open Distributed Systems [EDWARDS 94a].

From an application programmer's point of view such a system consists of a set of interacting application components (clients and servers). Sometimes these components will fail. The model describes such failures in terms of values and time. These are concepts which can be measured directly by components to detect failure.

From the infrastructure designer's point of view an open distributed system will consist of a set of interacting platforms (e.g. DCE, CORBA). On top of these platforms will be a set of engineering mechanisms which support a set of interacting application components. The purpose of the engineering mechanisms is to provide the application components with more guarantees (e.g. dependability) than are offered by the basic platform. This arrangement is shown in figure 1.1. Any mechanism, platform or application component may fail. The same modelling concepts can be used to describe all such failures.

Figure 1.1: An infrastructure designer's view of open distributed systems



The concepts in the model can be used to analyse a given configuration of engineering mechanisms, application components and infrastructure to determine what failures can and cannot be tolerated by this configuration. This can then be mapped onto an application-level statement: what failures the applications can and cannot tolerate. A methodology for this is outlined in §1.7, a detailed example is given in the appendices. One of the aims of the ANSA work on dependability is to develop a set of tools and abstractions which use this failure model to help application programmers match their

dependability requirements onto a specific configuration of engineering mechanisms and platforms (further details are given in [EDWARDS 94b]). In general each application will have a different set of requirements which will lead to different configurations.

The remainder of this paper is structured as follows. §1.1 describes the target audience and prerequisites for understanding the rest of this paper. §1.2 describes the relationship to other work, including other ANSA work. §1.3 states the assumed system model. §1.4 describes a model for failure in terms of abstract components. §1.5 uses the model to analyse the ODP and ANSA object-based interaction models. §1.6 looks at some of the failure modes often discussed in the literature and considers how they are related to the model. §1.7 outlines how the model can be applied to analyse a configuration of engineering mechanisms, infrastructure and application components. §1.8 states some conclusions and summarises the main points of the paper. The appendices demonstrate a use of the failure model by analysing an implementation of active replica groups.

1.1 Audience

Some familiarity with the basic principles of object-based distributed computing is assumed (e.g. ODP [ODP 93], CORBA [OMG 91] or ANSA [LINDEN 93a]). The audience is assumed to be those who are interested in how to characterise the failure of a system component and relate this to system dependability. This includes designers of dependability infrastructures and reference models for distributed dependability, as well as system implementors.

1.2 Relationship to other work

Modern computer systems are enormously complex; distribution introduces further complexity [LINDEN 93a]. Managing and understanding these complexities within a single computer requires the introduction of a hierarchy of levels (e.g. circuit level, logic level, program level). A hierarchy of failure models are needed so that dependability at lower levels can be related to dependability at higher levels (e.g. relating circuit level faults to logic functions) [SIEWIOREK 92].

ANSA uses a set of projections for managing the complexities introduced by distribution: the enterprise, information, computational, engineering and technology projections [LINDEN 93a] (ODP calls these viewpoints). Each of these projections describes a different aspect of a distributed system. The failure model presented in this paper can be used to relate failures in the engineering and computational projections.

The paper does not present a hierarchical failure mode classification (i.e. a partial ordering on failure modes); there are many in the literature (e.g. [BARBORAK 93], [SHRIVASTAVA 90], [CRISTIAN 90]). Such orderings are useful, because they say when one engineering mechanism can replace another. For example suppose we have an ordering $\subseteq$ on failure modes, and suppose we have 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. 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

describes the engineering objects and the mechanisms which they contain) which will determine the ordering on failure modes. Different engineering models will give rise to different orderings. The failure model presented in this paper can be used to show that a particular engineering model does or does not implement a particular ordering.

Taken together with the basic concepts of dependability, such as availability and reliability (discussed in [EDWARDS 94b]) this failure model provides a means of describing dependability requirements. This is deliverable D1 in [LINDEN 93b].

Previous ANSA work on dependability has concentrated on groups [OSKIEWICZ 93] and transactions [WARNE 92]. The model provides a framework within which to discuss these technologies, enabling us to understand what aspects of dependability they address and how they can be used to build dependable systems.

The intention is that the model should provide the underlying framework for further work on dependable distributed computing. This work is summarised in [EDWARDS 94a], which also investigates why dependability is important in open distributed computing. [EDWARDS 94b] gives a more detailed overview as well as describing the basic concepts for dependability in open distributed systems.

1.3 System structure and activity

It is assumed that a system is composed of components. These components can engage in events which can be observed by other components. For example a client makes an invocation which is later observed by a server when it receives this invocation (see §1.5.1). The structure of the system limits which components may observe which other components.

Each component makes its own observations and determines which events it considers to have occurred. An event is considered to occur with some value at some time, by the observer. The model does not require any notion of a global observer, a global ordering on events, nor does it require any notion of global time.

The model looks at only single occurrences of events. A similar assumption is made in the failures/divergence model of CSP [HOARE 85] which considers the ability of a process to refuse to engage in a single event, as opposed to sequences of events. Sequences of events can be built up from individual occurrences; §1.4.5 considers this issue.

1.4 A model for failure

Prompted by [SHRIVASTAVA 90] this section takes a set theoretical approach to modelling failure. Here the aim is to understand what constitutes a failure rather than to build up a hierarchical class of failures as in [SHRIVASTAVA 90].

The formal definition of a failure is given in §1.4.4. Informally, a failure occurs when the event which is observed does not match what is expected or when some event is expected and none is observed. An example of the former is a server receiving an invocation of an operation which it does not support. An example of the latter is a client failing to observe a reply after it has made an

invocation. Not all events are failures, for example a server may observe that a client has invoked an operation which it supports.

This definition of failure does not assign fault to either the component which engages (or does not engage) in the event or the component which observes (or does not observe) the event. To resolve this issue, the parameters used to determine correctness must be made explicit [BARWISE 87]. This can be done by using a specification.

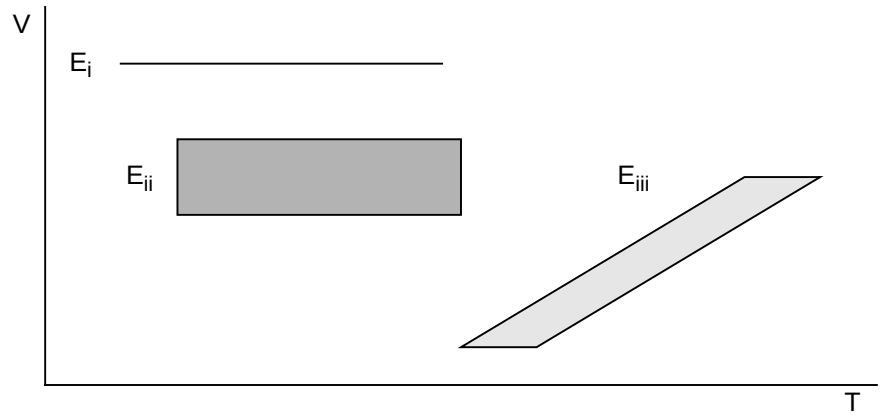
A specification sets expectations: it forms a contract between the component engaging in the event and the component which observes it. However, a specification may not be complete. In such a case a component may engage in an event which is allowed by the specification, but unexpected by the observer. Arbitration may be necessary to determine what to do about this and who is responsible for the failure. The notions of contracts between clients and servers and arbitration is being studied within the ANSA project.

§1.5 looks at other ways of setting expectations which do not involve specifications. This section considers expectations, occurrences, and the kind of mismatches that can arise. The various kinds of mismatch are the failure modes of the system.

1.4.1 Expectations

The event is usually expected to occur within some time interval, and with one of a restricted set of values. In the case of a server, the time interval within which it expects to receive an event may be very long (see §1.5.3). Figure 1.2 illustrates three expectation sets that will be used to illustrate the various failure modes.

Figure 1.2: Expectations



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
- 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.

All of these cases can be expressed as a set of (value, time) pairs at which the occurrence is expected. The expectation set is a subset of the set of all (value, time) pairs.

$$E \subseteq V \times T \tag{1}$$

This formulation can be used to express not only the cases illustrated above, but also many others, for example E could consist of several disjoint regions.

1.4.2 Occurrences

Occurrences of the event can also be considered as a set of value, time pairs:

$$O \subseteq V \times T \tag{2}$$

In the ANSA model, an individual event can occur at most once. Issues such as retransmission and replay can be dealt with by sets of distinct but related events, as discussed in §1.4.5.

This assumption means that the set O contains either zero or one element.

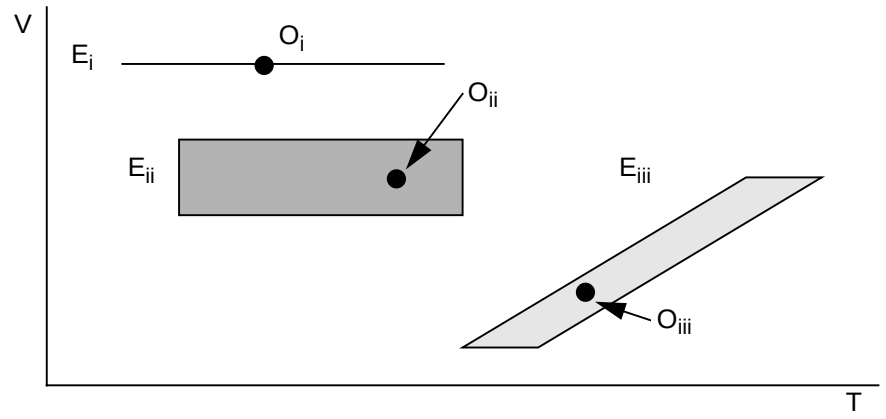
1.4.3 Correctness

The occurrence of an event is considered correct if what happens matches what is expected. Similarly if an event is not expected to occur then it is correct that it does not occur.

1.4.3.1 Correct occurrence of an event

An occurrence is correct if its value and time match one of the expected value, time pairs. Figure 1.3 adds an example of a correct occurrence for each of the three expectation sets of figure 1.2.

Figure 1.3: Correct Events



In terms of the expectation and occurrence sets, the correctness of the occurrence is given by the proposition

$$O \cap E \neq \varnothing \tag{3}$$

The informal definition at the beginning of this section is captured more directly by the equivalent proposition, which says that some event occurs and its value and time are expected

$$\exists v, t \bullet (v, t) \in O \wedge (v, t) \in E \tag{4}$$

1.4.3.2 Correct non-occurrence of an event

If an event is not expected to occur then it is correct that it does not occur.

It is important to consider this case since we wish to consider unexpected occurrences as failures. This means that we must consider the cases where an event is not expected to occur.

A correct non-occurrence satisfies the proposition

$$O \cup E = \varnothing \tag{5}$$

The equivalent proposition that captures the informal statement more directly is

$$\neg(\exists v, t \bullet (v, t) \in O) \wedge \neg(\exists v, t \bullet (v, t) \in E) \tag{6}$$

This says that the event does not occur and it is not expected.

1.4.3.3 *Formal definition of correctness*

Since an event occurs either zero times or once, the event is correct if it has either a correct occurrence or a correct non-occurrence.

Combining propositions (3) and (5) gives a proposition for a correct event

$$(O \cap E \neq \varnothing) \vee (O \cup E = \varnothing) \tag{7}$$

1.4.4 Failures

A failure occurs when what happens does not match what was expected. This could be because the observer is faulty (its expectations are wrong) or the event itself is not correct. It is usual to assume that the observer is correct and that a failure occurs when an event is not correct, but this is not fundamental to the model.

The proposition for failure is therefore the negation of (7), the proposition for correctness

$$\neg((O \cap E \neq \varnothing) \vee (O \cup E = \varnothing)) \tag{8}$$

This can be simplified to

$$(O \cup E \neq \varnothing) \wedge (O \cap E = \varnothing) \tag{9}$$

The failure modes which correspond to different mismatches of expectation and occurrence are described in the rest of this section.

1.4.4.1 *Unexpected occurrence*

If the event occurs and is not expected to occur, then this is a failure.

The proposition which captures this case is

$$O \neq \varnothing \wedge E = \varnothing \tag{10}$$

1.4.4.2 *Omission failure*

The event is expected to occur and does not occur.

$$E \neq \varnothing \wedge O = \varnothing \tag{11}$$

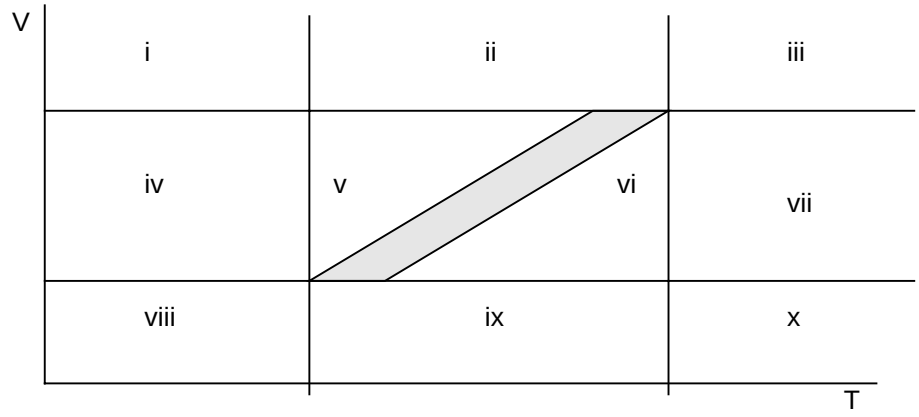
1.4.4.3 *Incorrect occurrence*

This leaves those cases when an event occurs, an event is expected, and the occurrence does not match the expectation.

$$O \neq \varnothing \wedge E \neq \varnothing \wedge (O \cap E = \varnothing) \tag{12}$$

Since both the expectation and occurrence sets are non-empty, the various cases can be illustrated graphically. Figure 1.5 shows a shaded region for the expectation and divides the space outside the expected region into ten areas.

Figure 1.4: Failure occurrence regions



The following regions are value failures: i, ii, iii, viii, ix and x.

The following regions are timing failures (early or late): i, iv, viii, iii, vii and x.

Regions v and vi are not covered in many failure mode classifications: they represent an occurrence of an event with a value and a time which are not expected together. In [SHRIVASTAVA 90], failures of this kind are considered in the discussion of clock faults, and they are classified as emission failures.

1.4.5 Consistency between multiple events

Consistency relations between events describe how the expectations of components vary over time.The simplest consistency relations are between events which are local to a single component. These are relations which constrain the region of expectation using the events which have so far been observed by the component and events in which the component has engaged. For example the relation may constrain the value of the next event expected using the sequence of events which have been observed so far. Hence an event which breaks the consistency relation will be an incorrect occurrence failure (or possibly an unexpected occurrence failure, if the expectation region collapses to zero). Such incorrect occurrences can be detected by mechanisms which are local to the component observing the event.

More complicated consistency relations may constrain the expectation regions using events which are observed by different components. For example the relation may require that the next events observed by a group of components each have the same value. Again events which break the consistency relation will be incorrect occurrence failures. However, these failures cannot be detected by mechanisms which are local to individual components. Detecting such failures is a distributed consensus problem, requiring collaboration between mechanisms at each of the observing components.

Often, rather than employing mechanisms which detect deviations from the consistency relation, mechanisms are used which enforce the consistency relation. For example [OSKIEWICZ 92] describes a mechanism which runs a distributed consensus algorithm to ensure that all members of a group receive the same invocations and terminations.

Sometimes it is possible to express a global property as a set of local ones. This makes the job of enforcing the consistency relation or detecting deviations

easier, since the mechanisms which do this can be made local to individual components.

1.4.6 Understanding the value domain

This presentation of the model has only associated a single value with an event. This allows an event to be represented in two dimensional space. In a real system it may be convenient to regard an event as having many values associated with it: for example each parameter of an invocation could be regarded as being a separate value.

The collection of values can be treated as a single value for the purposes of the model described above. Alternatively, the components of the value can be considered separately, making it possible to have an event that is correct with respect to some components of the value, but a failure with respect to others.

Some components of the value may be more significant than others in terms of value failures. In the safety community, the severity of the potential mishap is sometimes referred to as “hazard criticality” [LEVESON 86], and failures in different components of the value may correspond to hazards with different criticalities. The idea that some parts of a message are more sensitive than others is well established in the security community. The OSI Security Architecture [ISO 89] defines “selective field protection” to support this requirement.

If an event has n values then the expectation set occupies a region in n+1 dimensional space, and figures 1.2, 1.3 and 1.5 can be used to represent a two dimensional cut through this space. An occurrence occupies a point in this space. The same idea is used to model the program input space in [BISHOP 93].

1.5 Failures in object based interaction models

This section applies the failure model to an object-based interaction model, specifically the ANSA and ODP interaction model. This analysis is also applicable to other object-based interaction models such as CORBA [OMG 91], and RPC interaction models such as that supported by DCE.

The failure model itself does not make any assumptions about local or remote interaction. Hence it can be applied to objects which are interacting remotely and to objects which are interacting within the same address space.

The components of an ANSA and an ODP system are objects which interact through interfaces. The interaction allowed is described in [REES 93a] and [ODP 93] and is shown in figure 1.5.

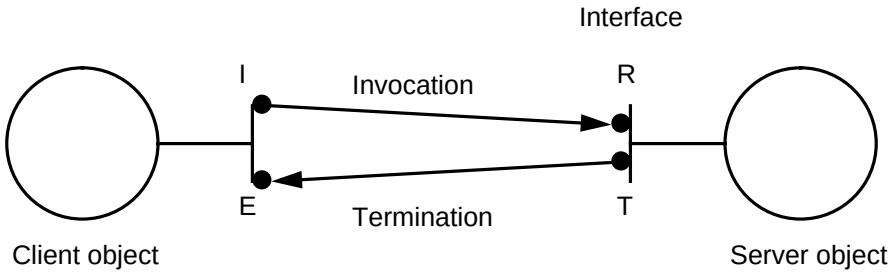
A client object interacts with a server object by invoking operations on interfaces provided by the server.

1.5.1 Events in an interaction

Four events that occur in an interaction are illustrated in figure 1.5. Of these, two are events in which components engage, and the other two are observations of events made by components.

The event labelled I is the event in which the client engages to initiate the invocation. The value for this event consists of an operation name and zero or more parameters. The client’s expectation will change when it engages in this

Figure 1.5: The interaction model



event; it will be expecting a response that corresponds to the requested invocation.

The event labelled R is the server's observation of the invocation request. This is the "request event" described in [REES 93b] that informs the server that it has been invoked. The value of this event is expected to be the same as the value of the event I.

The event T is the "terminate event" described in [REES 93b] that occurs when the server has completed the evaluation of the operation. Its value consists of a termination name followed by zero or more parameters. The particular name and parameters are expected to be the ones defined by the behaviour of the server given the value of the event R, the state of the server at the time of the event R, and any other events that may have been observed by the server.[1]

The event E is the client's observation of the termination. It is expected to have the same value as the event T.

The expectations above are that the engineering mechanisms (or objects) and platforms involved in the binding between client and server do not corrupt values. However, failures can occur in these objects. Identifying four separate events in the interaction model allows a link to be made between expectations and failures at the application-level (or in the computational model) and expectations and failures in the engineering. It would be harder to introduce this link if invocation and termination were regarded as atomic events, as these atomic events would need to be subdivided when studying the role of engineering objects in the interaction.

1.5.2 Constraints on expectations

An interface type is defined by a set of signatures. Each signature specifies:

- the name of an operation
- the number and types of argument parameters
- the termination names and result parameters

1. The "activate event" described in [REES 93b] has not been discussed. An elaboration on the behaviour of the server would be the right place to introduce it. There is scope for a discussion of how concurrency control can cause timing faults in its efforts to preserve consistency.

Both ODP and ANSA allow for types to specify more properties, e.g.: concurrency constraints and environmental constraints. DCE and CORBA have no such facilities.

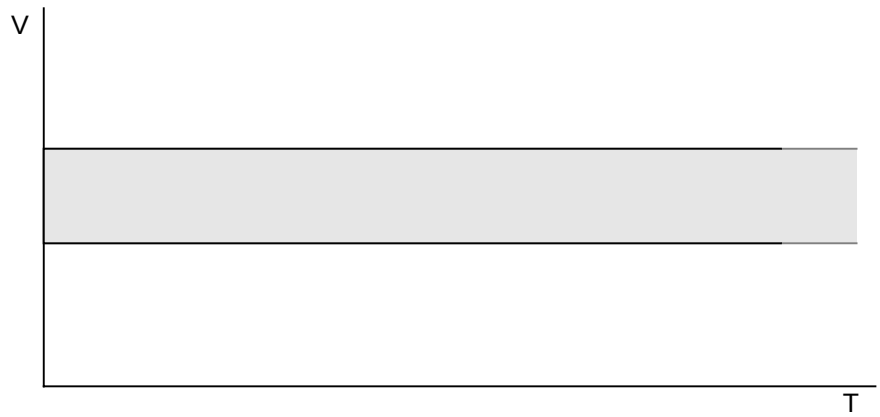
The interface type limits the expectations of both the client and the server. It may be thought of as part of a contract between the client and the server which sets bounds on the expectation region. The stronger the contract between the client and server, the smaller the expectation region.

The technology for stating timing constraints in ANSA and ODP like systems is not well understood. ANSA does not specify an means of constraining time within the interface type, although a proposal has been made in [STEFANI 93] for using Quality of Service to express timing constraints. ODP states that timing constraints are part of the environment contract in an interface type, but no technology is specified for stating these constraints. In the model presented here, expressing frequency or throughput failures can only be done by looking at sequences of occurrences and defining consistency relations over those occurrences.

1.5.3 A server's expectations of clients

A server offering a service through an interface must expect operation invocations to select any of the operations defined by the interface type. The expected value space is constrained by the operation name and the parameter types associated with that operation; figure 1.5 shows the two dimensional representation for this simple case.

Figure 1.6: A servers expectations



A server's expectations may be further constrained if the invocation is received from an active replicated client group [OSKIEWICZ 93]: it expects the values and times of invocations from members of the replica group to agree.

A server may expect a client to invoke operations in a certain sequence and within certain time constraints.

Security constraints may restrict from whom the server is expecting to receive invocations. The observed event must include information that allows the server to check that the invocation is authorised. This may be an explicit part of the value (e.g. a Privilege Attribute Certificate as defined in [ECMA 89]). Alternatively, the server may use a previously negotiated key either to

decipher the value, or to check a cryptographic checkvalue. These techniques allow unauthorised invocations to be detected as value failures.

1.5.4 A client's expectations of servers

A client's expectation of a server is also constrained by the interface type: this constrains the termination a client expects when it invokes one of the set of operations supported by the interface[1]. Typically a client will expect some response within a given time of making the invocation. This simple situation corresponds to the sets shown in figure 1.2. A client does not expect a termination from a server until it has made the invocation, this is an example of a consistency relation between events which constrain expectations (§1.4.5).

A client's expectations will be constrained further if it invokes an active replica group (i.e. it expects the values and times of the terminations to agree) or if it has some semantic knowledge of the service (e.g. time should not run backwards).

The process of trading [ANSA 93], [ODP 93] will also constrain the client's expectation region. A client's expectation of the trading service is that it will get back an interface which matches its request. If it is unable to obtain something which satisfies its request, negotiation may take place which will affect the client's expectations of the service it subsequently decides to use.

1.5.5 The engineering components' expectations of clients and servers

In the engineering model, a platform provides services to computational objects (i.e. client and server objects) through typed interfaces [ODP 93], [OSKIEWICZ 92]. Hence within the engineering model we can use exactly the same ideas of expectations as within the computational model.

For example most platforms have an implicit expectation that computational objects will transmit below a certain frequency, f. If they exceed this frequency of transmission they are regarded as flooding the network. Detecting this kind of failure requires the ability to look a sequences of occurrences. Although it could in principle be detected anywhere, it is probably be easier to take remedial action if it is detected as close as possible to the client or server object responsible.

Trading sets the expectations of clients and servers in the computational model: similarly binding sets expectations in the engineering model. When binding occurs it establishes the expectations of the engineering objects which support interaction between the client and server object. Binding may also be visible in the computational model. If it is, it will also set client and server expectations.

At federation boundaries interceptors [ANSA 93] may set and enforce expectations.

1.6 Crash, Byzantine and arbitrary failures

This section considers how some of the failure modes discussed in the literature relate to the model described in §1.4.

1. ODP takes the view that a binding occurs between client and server interfaces to capture the polarity and duality of the interface concept.

The model described in §1.4 says nothing about crash failures (or fail-silence). This is because a crash failure usually involves multiple non-occurrences. For example [CRISTIAN 90] states that a server is suffering a crash failure "if, after a first omission to respond, ... [it]... omits to respond to all subsequent inputs, until its restart". Hence a crash failure involves observing multiple non-occurrences, deciding that the component has suffered a crash failure and repairing it. It is something which cannot be demonstrated by testing.

The model does not describe Byzantine (arbitrary) failures. Such failures cannot be detected; they can only be masked through redundancy. The classic Byzantine failure is sending different values to two or more receivers (e.g. in a multicast), when consistency constraints demand that these should be the same value [DOLEV 87].

It is known that using authentication techniques substantially simplifies the handling of Byzantine failures [BARBORAK 93], [DOLEV 87]. The authentication techniques can range from simple checksums to complicated cryptographic techniques. These kinds of techniques operate in the value domain, making the value space as sparse as possible. Time domain techniques which look at sequences of occurrences have to be used to detect some other kinds of malicious behaviour, such as attempts to flood the network (§1.5.5).

1.7 Applying the failure model

Failure detection is the detection of deviations from what is expected. The failure model provides a way of describing expectations. Hence we can also describe deviations from expected behaviour. This allows us to state the behaviour of failure detection mechanisms and also the behaviour of mechanisms which enforce expectations.

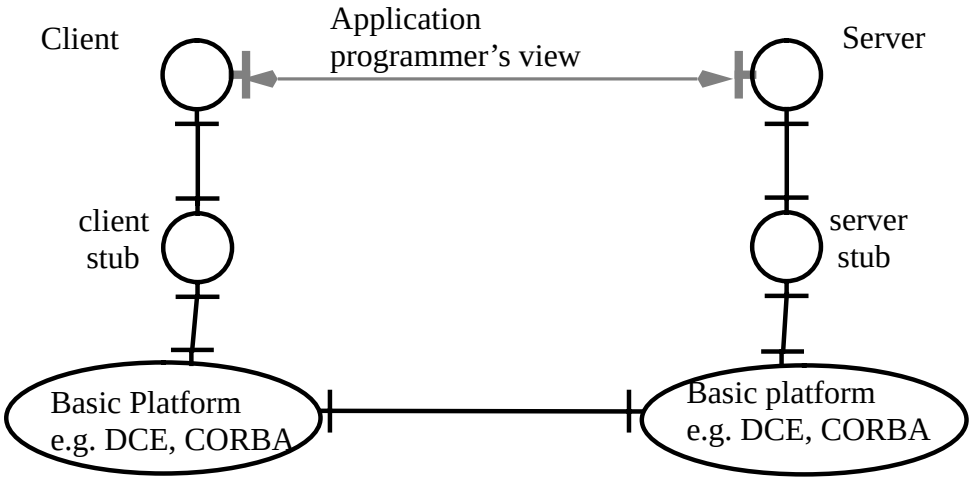
As a very simple illustration of how the model can be used consider figure 1.1. This shows the objects supporting a client and server interacting with each other. Suppose the client invokes the server. After receiving the invocation the server's node crashes. This means that the server, its stub and platform will not send any more messages. Now the client's platform is expecting to receive a message from the server's platform within a certain time in response to its invocation. The client's stub is expecting to receive a response from its platform and the client is expecting to receive a response from its stub.

After some fixed time, the client's platform times-out on the server's platform and returns a response indicating this to the client stub which in turn returns it to the client. Hence the failure is detected (although not tolerated unless the client has a mechanism for dealing with it).

The appendices show how to use the failure model presented here to analyse the implementation of active replica groups described in [OSKIEWICZ 93]. This involves analysing a set of engineering objects which implemented active replication over a basic platform supporting RPC interaction. The following is a summary of the method used, readers requiring further details are referred to the appendices.

1. The expectations of engineering objects and platform involved in the binding between a client group and a server group are stated.
2. The expectations of the application objects are derived from 1 (i.e. the expectations which the clients and servers involved in the interaction have of each other).

Figure 1.7: Infrastructure designers view of client/server interaction



3. Each failure mode of a server object (listed in §1.4) is considered in turn to see if the engineering objects or platform can detect and tolerate this failure. If the failure cannot be tolerated, the expectations of the client object will not be met.
4. A similar analysis is conducted for each client object

This analysis results in a statement within the model of the failures which can be tolerated and those which cannot be tolerated: something which could not be done in [OSKIEWICZ 93] because the technology to perform this analysis was not available. The analysis also gives considerable insight into how the implementation might be improved to tolerate more kinds of failures.

The notion of a contract between components is crucial to setting expectations. Present technology allows only very weak contracts (e.g. interface definition languages only define signatures). Stronger contracts will allow better failure detection mechanisms to be built. Currently, active replication is one of the best forms of failure detection: the members of active replica groups have an implicit contract to agree (of course, they may all be wrong). However, active replication is not appropriate for all applications.

The model described does not explicitly cover events which report detected failures. In one sense these are correct since they are anticipated. In another sense they are failures since usually it is expected that the system will operate correctly, so events reporting failures are unexpected. As discussed in the appendices, experience of applying the model has shown that these events should be treated as unexpected events (and therefore failures).

1.8 Summary and Conclusions

This document describes an abstract failure model in terms of a system consisting of interacting components. This model is then applied to an object based interaction model.

The model can be used to analyse a given configuration of engineering mechanisms, application components and infrastructure to determine what failures can and cannot be tolerated by this configuration. From this a

statement is generated, within the model, of the failures which can and cannot be tolerated by the configuration (see appendices).

More work is needed to investigate the nature of contracts between clients and servers which fix expectations. The stronger these contracts the more powerful the failure detection mechanisms which can be used.

The model has been successfully applied to previous work on active replica group mechanisms (see appendices). More experience of applying the model is needed to identify improvements either in the model itself, or in its use. The use of the failure model as part of a design methodology for building dependable systems is being studied.

1.9 Acknowledgements

The authors acknowledge the valuable contributions made by Ed Oskiewicz, seconded to the ANSA Team by BT, and John Warne, seconded to the ANSA Team by BNR-Europe.

The authors are grateful to Santosh Shrivastava of the University of Newcastle upon Tyne: his comments on an earlier versions of this document enabled substantial improvements to be made. The authors are also grateful to Brian Coan of Bellcore, Paul Harry of Hewlett-Packard and Andrew Herbert of APM Ltd. for their comments. A discussion with David Iggulden of APM Ltd. also proved very helpful.

References

[ANSA 93]

"The ANSA model for Trading and Federation", AR.005.00, February 1993, APM Ltd., Cambridge

[BARBORAK 93]

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

[BARWISE 87]

Barwise, J., Etchemendy, J., "The Liar — an Essay on Truth and Circularity", Oxford University Press, 1987.

[BISHOP 93]

Bishop, P.G., The Variation of Software Survival Time for Different Operational Input Profiles, in FTCS 23, the 23rd International Symposium on Fault-Tolerant Computing, Toulouse, France, June 1993, pp98-107.

[CHANG 84]

Chang, J., Maxemchuk, N.F., "Reliable Broadcast Protocols", ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp251-273.

[CRISTIAN 90]

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.

[DOLEV 87]

Dolev, D., Lamport, L., Pease, M., Shostak, R., "The Byzantine Generals", in Concurrency Control and Reliability in Distributed Systems, van Nostrand Reinhold, 1987, Bhargava, B.K., (Ed.), pp348-369.

[ECMA 89]

"Standard ECMA-138, Security in Open Systems, Data Elements and Service Definitions", December 1989, ECMA, Geneva.

[EDWARDS 94a]

Edwards, N.J. " Open Dependable Dsitributed Systems", APM.1045, February, 1994, APM Ltd., Cambridge, U.K.

[EDWARDS 94b]

Edwards, N.J., "Building Dependable Distributed Systems", APM.1044, February, 1994, APM Ltd., Cambridge, U.K.

[HOARE 85]

Hoare, C.A.R , "Communicating Sequential Processes", Prentice Hall International, 1985

[IGGULDEN 93]

Iggulden, D., Architecture and Frameworks, TR.038.00, February 1993, APM Ltd., Cambridge U.K.

[ISO 89]

Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 2: Security Architecture", ISO 7498-2 : 1989 (E)

[JONES 86]

Systematic Software Development using VDM, Prentice-Hall International 1986.

[LEVESON 86]

Leveson, N. G., "Software Safety: Why, What, and How", ACM Computing Surveys, June 1986.

[LINDEN 93a]

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

[LINDEN 93b]

van der Linden, R., "Operational Plan: 1/3/93 - 28/2/95", APM.1031, Cambridge, U.K., June 1993.

[OMG 91]

"The Common Object Request Broker: Architecture and Specification", OMG Document number 91.12.1, Revision 1.1, 1991.

[OSKIEWICZ 93]

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

[OSKIEWICZ 92]

Oskiewicz, E., Edwards, N.J., "GEX Design Notes — Revised", RC328.01, APM Ltd., Cambridge U.K., September, 1992.

[ODP 93]

Basic Reference model of Open Distributed Processing - Part 3: Prescriptive Model, Secretariat ISO/IEC JTC1/SC21, American National Standards Institute, June 1993.

[POWELL 91]

Powell, D. (ed.), "Delta-4: A Genric Architecture for Distributed Computing", Springer-Verlag, 1991.

[REES 93a]

Rees, R.T.O. "The ANSA Computational Model", AR.001.01, January 1993, APM Ltd., Cambridge, U.K.

[REES 93b]

Rees, R.T.O., "Using path expressions as concurrency guards", TR.022.00, January 1993, APM Ltd., Cambridge, U.K.

[STEFANI 93]

Stefani, J.B., "Some Computational Aspects of QoS in ANSA", CNET/ RC.W03.JBS.001, January 1993, (Available from APM Ltd., Cambridge, U.K.).

[SHRIVASTAVA 92]

Shrivastava, S.K., Ezhilchelvan, P.D., Speirs, N.A., Seaton, D.T., “Fail-Controlled Computer Architectures for Distributed Systems”, Technical Report No. 333, University of Newcastle upon Tyne, Department of Computer Science, revised August 1992.

[SHRIVASTAVA 90]

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.

[SIEWIOREK 92]

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

[WARNE 92]

Warne, J.P., Rees, R.T.O, “ANSA Atomic Activity Model and Infrastructure”, AR.004, February 1992, APM Ltd., Cambridge U.K.

A Applying the ANSA failure model to active replica groups

The aim of the appendices is to show how to apply the ANSA failure model to the ANSA and ODP computational and engineering models. It is applied to the model for active replica groups described in [OSKIEWICZ 93] and the implementation of that model described in [OSKIEWICZ 92]. The reader is assumed to be familiar with ANSA [LINDEN 93a] or ODP [ODP 93] and also the ANSA failure model.

The behaviour of the mechanisms in the infrastructure supporting active replica groups are described and analysed. This analysis takes place within the engineering model and results in a statement, within the failure model, of those failures which can be detected and tolerated by the engineering objects. This allows a statement to be made about which failures can and cannot be tolerated by applications (i.e. within the computational model). The analysis also yields some insight into the implementation and how it might be improved to tolerate even more failures.

The style of analysis is one of "rigorous argument" [JONES 86] rather than a completely formal presentation. The arguments are founded on the concepts described in the ANSA failure model, however, no formalism is available to allow a completely formal check.

The next section summarises the failure model and explains how it is used, and the following section explains how to read the remainder of the appendices.

A.1 How the failure model is used

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** which define a time interval and a restricted set of values within which a component expects to observe an event.

A **failure** occurs when the event which occurs does not match what is expected. This leads to a situation which is ambiguous and somewhat paradoxical[1], since 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 [BARWISE 87]. This document assumes that a component's expectations are correct, so when a failure occurs, the fault is assumed to be in the component which engaged in the event, rather than in

1. The work of Barwise and Etchemendy show that the concepts of paradox and ambiguity are closely related [BARWISE 87].

the component which observed the event. It will be seen that many of the expectations will be set by interface definitions and will therefore be independent of event observer or generator. Interface definitions are a form of contract between a client and server, the stronger these contracts the easier it is to avoid the ambiguous situation where fault cannot be assigned.

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, assuming those objects are "*correct*". The notion of what is correct ideally should be captured by a formal contract between two objects. Unfortunately, usually it is only partially captured in an interface definition, and written text.

A computational object will have expectations in both the computational and engineering viewpoints. The computational expectations will be associated with the semantics of the application, whilst the engineering expectations will be fixed by the engineering model. These correspond to the interworking and portability conformance points identified in [REES 93a].

The failure model is used to analyse separately the interaction between groups (inter group) and the interaction between group members (intra group). The inter group analysis comprises of the following.

1. The expectations of engineering objects involved in the binding between a client group and a server group are stated.
2. The expectations of the computational objects are derived from 1 (i.e. the expectations which the clients and servers involved in the interaction have of each other).
3. Each failure mode of a server object (listed in the failure model) is considered in turn to see if the engineering objects can detect and tolerate this fault or if it will lead to a failure to meet the expectations of a client. Only single failures are considered.
4. A similar analysis is conducted for each client object

The aim of the implementation is transparent tolerance of failures, so it is assumed that the application code in clients and servers makes no attempts to detect and tolerate failures. Hence failures are detected only if the engineering objects involved in the binding are able to detect them. This will only happen if the failure also does not match the expectations of the engineering objects. The failure will be tolerated if the engineering objects are able to meet the expectations of clients and servers in spite of it. (In this case the failure will be transparent to the computational objects.)

The analysis of the intra group interaction is similar. The main difference is that quorum and order processors which act as the clients and servers in this interaction have their own failure detection and handling mechanisms, so they do not have to rely on the engineering objects involved in the binding to detect and tolerate failures.

A.2 How to read the appendices

Section A.3 gives a brief description of the engineering objects for active replica groups.

Section A.4 states the expectations of the computational objects.

Section A.5 gives a brief description of the engineering objects involved in intra group interaction.

Section A.6 applies the methodology outlined above in §A.1 to the inter group and intra group interactions. This leads to a statement of the failures within the model which can and cannot be tolerated by the implementation of active replica groups.

Section A.7 concludes by using the insight given by the analysis to make recommendations for improving the implementation and commenting on how engineering can exploit the concept of expectations.

A statement of the expectations of the engineering objects involved in an inter group binding is given in appendix §B.3. Appendix §B.4 gives a statement of the expectations of the objects involved in the intra group binding.

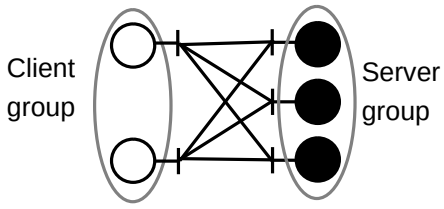
It is recommended that anybody familiar with [OSKIEWICZ 93] or [OSKIEWICZ 92] still reads §A.3 and §A.5, as the presentation draws attention to points which are used in the subsequent analysis. Appendix B is included for completeness: the analysis in §A.6 can be read without reading these first. However, it should be remembered that the material in appendix B was needed to do the analysis: generating the list of expectations of the engineering objects is the first step. A reader contemplating applying the ANSA failure model is urged to study both appendices A and B.

A.3 An engineering specification for active replica groups in ANSA

This section gives a brief description of the implementation of active replica groups described in [OSKIEWICZ 92]. This is a description of a particular implementation and design, rather than a general description of the principles of active replication. The description identifies the mechanisms in the infrastructure which are needed to support active replica groups. These will be analysed in the rest of these appendices. (In ODP these mechanisms are called engineering objects.)

The basic group abstraction is to treat a number of object's interfaces as though they were one. This enables many-many interaction structures to be built like the one shown in figure A.1.

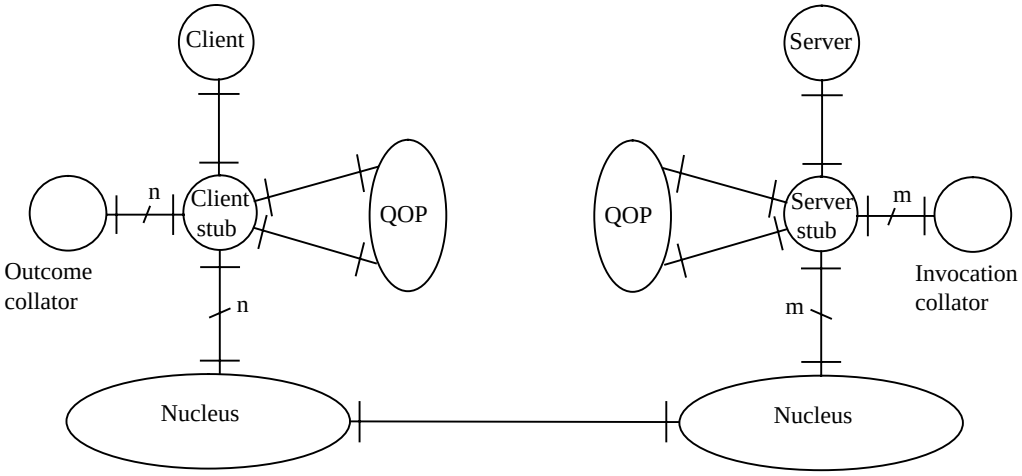
Figure A.1: Many-many interaction



Active replica groups can be used to deliver a service even if some of the replicas fails. This requires that the state of each replica is synchronised so that any individual replica can deliver the service. [OSKIEWICZ 92] describes an implementation in which groups are transparent so it appears to each client and server object, as though there were a single client invoking a single server.

Suppose a client group with m members invokes a server group with n members; figure A.2 shows the engineering objects supporting one member of

Figure A.2: The engineering objects supporting active replica group



the client group and one member of the server group in the implementation described in [OSKIEWICZ 92]. Each member of the client group will send the invocation to each member of the server group. The client stub takes a single invocation and hands it to the nucleus which multicasts it to the server group. Assuming there are no failures, each nucleus supporting a server will receive m invocations (one from each member of the client group) and pass these to the server stub.

The server object must only receive a single invocation, as though it comes from a single client. So the server stub waits for all m invocations and uses the invocation collator to ensure that the invocations agree. Maintaining state synchronisation requires that multiple invocations (possibly from different client groups) are evaluated in the same order at each server object. Furthermore if one of the non-faulty server objects receives an invocation then all of them should. The server stub passes the successfully collated invocation to the QOP (quorum and order processor) which is responsible for this.

The QOP agrees with the other QOP's in the server group which invocations have been received by the group and in which order they will be invoked. It then invokes the server stub which finally invokes the server object.

A similar situation occurs when the server group sends the outcome[1] back to the client group. The server stub receives a single outcome and invokes the nucleus m times to deliver the outcome to each of the m client objects. Each client stub will receive n replies (one from each server object) which it needs to collate using the outcome collator and then pass to the QOP.

A.4 Expectations of the computational objects

Appendix B.3 discusses expectations within the engineering model. Within the computational model interactions occur between clients and servers — the engineering objects are transparent.

1. In ANSA an outcome is what is returned in response to an invocation; a termination is the type of the outcome [REES 93a]. ODP uses the term termination to describe what is returned in response to an invocation.

The expectations a client has of a server group, and those a server has of a client group in the engineering viewpoint, can be derived by noting that the stub objects are acting as proxies. The client stub object is acting as a local proxy of the server at the client; the server stub object is acting as a local proxy for the client at the server. Hence, in the engineering viewpoint, the expectation a client has when interacting with a server group are those it has of the client stub object. Similarly the expectations a server has when interacting with a client group, in the engineering viewpoint are those it has of the server stub object.

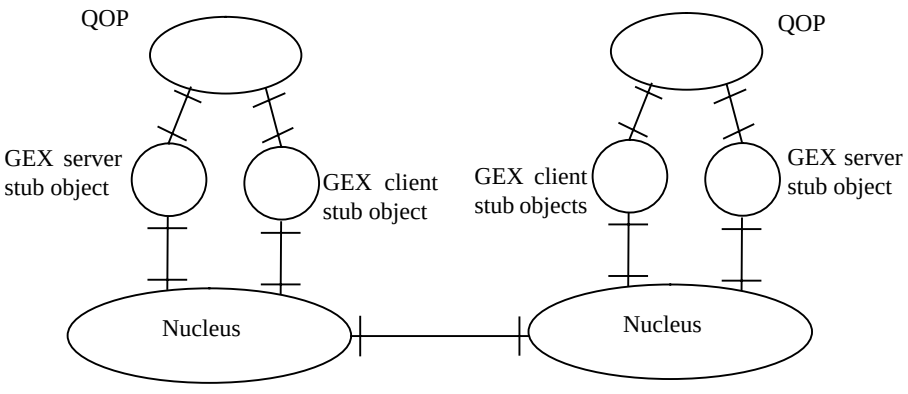
Client and servers will have expectations in the computational viewpoint which are fixed by the application semantics. For example most clients will have implicit expectations about the time and values which can be associated with outcomes it expects to receive from servers. In addition they are likely to have implicit expectations about the consistency of successive outcomes. If these expectations are not met, the client may fail because of a faulty input, or because it has received an exception which it cannot deal with. These expectations are usually associated with some notional idea about what a correct server will do, but are rarely stated explicitly. (It is not clear what technology could be used to state these expectations in the contract between client and server.)

[OSKIEWICZ 92] describes an implementation in which replication is transparent to clients and servers. This means that enforcing and detecting deviations from their expectations is transparent. However, they do have the expectations described above and in appendix B: the client or server will fail if these expectations are not met.

A.5 The GEX quorum and ordering protocol

This section gives a brief description of the GEX quorum and ordering protocol described in [OSKIEWICZ 92] and based on [CHANG 84]; readers needing further details are referred to these texts. Figure A.3 shows the engineering objects

Figure A.3: The quorum and ordering engineering objects



needed to run the protocol in a two member group. The QOP's are also bound to a client stub object or a server stub object (see figure A.2); this binding is not shown in figure A.3. The remainder of this section describes the interaction between the objects in figure A.3.

The basic idea in the protocol is that a token circulates around the group in a logical ring, so that each QOP in the group takes turns to hold the token. The token holder decides which invocation (or outcome) the group will receive next. The identity of this invocation is attached to the token before it is passed on to the next QOP. The token pass is multicast to all QOP's in the group, but only one QOP will become the next token holder.

If a QOP sees an invocation attached to the token which it has not received, it requests the QOP who multicast the token for a copy of that invocation. Before the token holder is able to pass on the token, it must ensure that it has received a copy of each invocation in the old token. The next time a QOP holds the token, it knows that every other QOP in the group has held the token since it last held it. Hence every QOP must have the old invocation it placed in the token the last time: if they had not received it they would have requested a copy before passing the token. The token holder can therefore invoke the server stub object (shown in figure A.2) which in turn invokes the server object with the old invocation. It then removes the old invocation from the token, attaches a new one and multicasts the new token. When other QOP's see the new token pass they too can invoke their server stub objects with the invocation which was removed from the token.

For the sake of brevity the reformation protocol, which is invoked when a group member fails, is not described here.

A.6 Failure detection and tolerance

This section enumerates the failures which can and cannot be detected and also those which can and cannot be tolerated by the replica protocol. First the section considers inter-group fault-tolerance: the failures which a client replica group can tolerate in a server replica group (and vice-versa) and still deliver a correct service. Finally it considers intra-group fault tolerance: the failures which the group members can tolerate in each other and still continue to deliver a correct service.

Expected outcomes are those which would occur if the interaction between all components was successful. Some IDL's include failure terminations in the interface definition, however, by definition a failure is something which is unexpected. These failure terminations allow clients to detect the failure.

The failure enumeration is taken from §1.4.4 and §1.4.5 which lists the failures which can occur.

The nature of the tests to detect deviations from expectations are discussed in detail in appendix B. In ANSAware, by default, these are limited to link-time and run-time type checking (value checking), and the used of timers. The value checking is very simple, for example it is unlikely to detect a rogue pointer passed as an argument. The latter will probably cause the receiver of the pointer to crash (and hence suffer an omission failure to be tolerated by other components). Comparing replicated invocations and outcomes allows better value checking, in the absence of common mode failures.

A.6.1 Inter group fault tolerance

This section considers the failure modes of a member of a server replica group after it has been invoked by a client (replica group). The expectations of each member of a client group are listed in §B.3.1, §1.4.4 and §1.4.5 list the possible

failure modes. The analysis proceeds by considering each of the failure modes of the server group and how this may fail to match the expectations of the client (group members). Only single failures are considered.

It is assumed that the clients do not contain code for detecting and tolerating failures. Hence the failure will be detected, only if the engineering objects are able to detect it. This will only happen if the failure also does not match the expectations of the engineering objects (which are listed in §B.3). The failure will be tolerated if the engineering objects are able to meet the expectations of each member of the client group in spite of the failure.

The exercise is repeated for a server (replica group) which is expecting to receive invocations from a client replica group.

A.6.1.1 Unexpected occurrence failures

A client group expects an occurrence after having made an invocation. The nucleus will discard all other messages intended for the client group from the server group. So the client group will tolerate this kind of fault in a server group.

A server group always expects an occurrence: so a client group cannot suffer this kind of failure so far as a server group is concerned.

A.6.1.2 Omission failures

Consider a server replica group suffering a single omission failure. The nucleus of each member of the client group will time-out and return a outcome which indicates this to the client stub object. This is not an expected outcome (expectation d, §B.3.3).

Suppose the server group has only one member. In this case the collator will have no other outcomes to mask the unexpected outcome, so it will be delivered to the client, violating generic client expectation 3, §B.1.

If the server group has two members, the client stub will receive one unexpected outcome and one expected outcome. This cannot be masked by a simple majority vote in the collator. However, the outcome collator can merely discard the unexpected outcome, so a two member server group can be used to tolerate a single omission failure.

If the server group has three or more members the client will receive two or more expected outcomes and one unexpected outcome. This can be masked by the majority vote in the collator. So a server group with three or more members can tolerate a single omission failure.

Similar arguments apply for a server group receiving invocations from a client group. If the client group has only one member, the server will never receive the invocation, so the failure cannot be tolerated. This may cause damage at the server if the client has obligations to it: for example, the client is holding locks which it fails to release.

If the client group has two members, the server stub will receive one invocation, but it expects two (expectation a, §B.3.6). There are two possible cases which could give rise to this: the received invocation is correct and the other client has suffered an omission failure; or the received invocation is an incorrect occurrence[1] and the other client has correctly done nothing. In the absence of further information (such as assumptions limiting the possible failure modes) it is impossible to resolve this ambiguity.

The above ambiguity can be resolved by the addition of a third replica (see below) or a statement (contract) which says what the behaviour of the client should be. In the analysis of a server group suffering a single omission failure the ambiguity is avoided by using a fact about the client/server relationship: the client knows the server should respond, because it has invoked it.

If the client group has three or more members the server stub object will receive two or more invocations, the invocation collator will detect and mask the failure. So the server group can tolerate a single omission failure in a client group with three or more members.

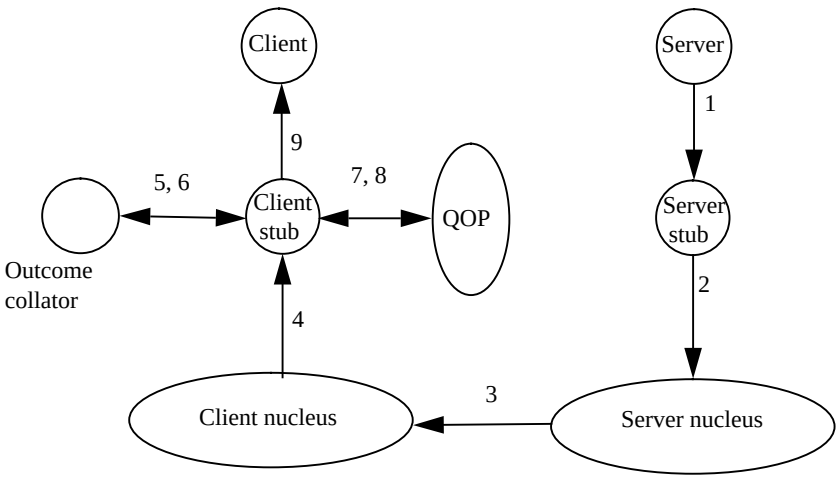
Hence three or more members are required in a client group to tolerate a single omission failures; two or more members are required in a server group to tolerate single omission failures.

A.6.1.3 Incorrect occurrence failures

Consider a server replica group suffering a single incorrect occurrence failure upon being invoked by a client group. Hence one member of the server group delivers an outcome which is unexpected in value or time or both. The termination name of the outcome is represented as part of the value, so “unknown termination errors” can be detected as value failures in the engineering, although they are “type errors” in terms of the computational model.

Expectation b, §B.3.1 expresses a consistency relation between the outcomes delivered to each client object. As explained in §1.4.5 this constrains the expectation region, so deviations from this expectation region will appear as incorrect occurrence failures. For the sake of simplicity occurrences which violate this expectation are treated separately in §A.6.1.4; this section deals with occurrences which potentially violate generic client expectation 3, §B.1 or expectation a, §B.3.1.

Figure A.4: Incorrect occurrence in a server group member



A single occurrence failure in a server (see figure A.4) may result in unexpected outcomes being delivered to the server stub object (1) or the

1. This is an incorrect occurrence as observed by the server which always expects something. From the point of view of an observer which knows that the clients should do nothing, it is an unexpected occurrence failure.

server's nucleus (2) (both cases violating generic client expectation 3, §B.1). Some link-time checking is done to try to prevent this. In addition there is limited value checking made by the server stub object and nucleus (the programming model supported by C limits what can be done). If either the stub or nucleus detect the failure, they would suppress the outcome, so it would appear to the client as though the server had suffered an omission failure. In both of these cases the discussion of §A.6.1.2 applies.

Such a failure may also result in an unexpected outcome being delivered to the client's stub object (4), which may detect it in a value check (i.e. a run-time type check) and discard the outcome (expectation d, §B.3.3). An outcome indicating this failure is delivered to the outcome collator; the collator will treat this in exactly the same way as it would incorrect occurrence failures which are not detected by the client stub — these are discussed below.

The unexpected occurrence may not be detected by any of the value checks in the server stub, nucleus or client stub. However, the client stub object will pass the incorrect occurrence to the outcome collator (5) which will compare it to the correct outcomes. The collator enforces expectation c of the client stub object (§B.3.3), which requires the outcomes to agree, passing the agreed outcome back to the client stub (6). This is the passed to the QOP (7, 8) before eventually reaching the client object itself (9).

Suppose the server group has only one member. The collator has no other correct occurrences to compare with the outcome which is an incorrect occurrence. So the latter will be passed to the client, potentially violating expectations it has associated with some notionally correct server (see §A.4).

If the server group has only two members, the collator will not be able to enforce an agreement between the two outcomes. This violates expectation c of the client stub object (§B.3.3) and expectation a of the client (§B.3.1).

If the server group has three or more members, the collator will be able to enforce an agreement using a majority vote.

Similar (but not identical) arguments apply to a client group suffering a single incorrect occurrence failure invoking a server group. Hence three or more group members are needed to tolerate incorrect occurrences.

A.6.1.4 Consistency failures

Consider a server replica group suffering a single consistency failure. Each member of a client group expects to receive the same outcomes as the other members in the same order (expectation b, §B.3.1). Note as individual occurrences these outcomes might be considered "correct"; it is only when considered in relation to each other that they are incorrect. The GEX quorum and ordering protocol ensures that all members receive outcomes in the same order, but not necessarily that the outcomes are the same (see §A.5).

Suppose a server sends different messages to each member of the client replica group (this is often called Byzantine failure in the literature [DOLEV 87]).

If the server group has only one member each client object in the client group will receive different outcomes.[1]

If the server group has two members, invocation collation will detect that the two server members disagree, but nothing can be done to tolerate the failure.

1. This could be detected if checksums of the invocation were placed in the token, instead of invocation identifiers.

If the server group has three members or more members, the invocation collator at each client will detect that one of the servers has suffered an incorrect occurrence failure. This failure can be tolerated (see §A.6.1.3).

Similar arguments apply when a member of a client replica group sends different invocations to different members of a server replica group: the failure can be tolerated if the client group has three or more members.

A.6.2 Intra group fault tolerance

Once a failure is detected within the group, it attempts to reform. If the group successfully reforms and continues to deliver a correct service, the failure is tolerated. Hence to understand the failures which can be tolerated, the reformation protocol needs to be examined. This section first looks at the failures which group members (QOP's) can detect in other group members (QOP's), and then considers the reformation protocol.

A.6.2.1 Unexpected occurrence failures

Each group member always expects an invocation from other group members (either a token pass or a request for a missed invocation), so an unexpected occurrence failure cannot occur.

A.6.2.2 Omission failures

There are two cases in which an omission failure can occur: a group member may omit to return an outcome, or the token holder may omit to pass the token to one or more members.

Each QOP expects to see a token pass from the token holder within a given time (expectation a, §B.4.8). If the token holder suffers an omission failure it will be detected by the timers in the other QOP's and a reformation would be initiated. (Note that once a group member has timed-out and decided that the token holder has suffered an omission failure, it must be prepared to reject the token pass, should it arrive eventually).

Each QOP expects to see one and only one outcome for each invocation it makes on other QOP's within some time-limit (generic client expectation 2, §B.1). Deviations from this expectation will also deviate from the expectations of the client stub object (generic client expectation 2, §B.1). The nucleus will detect deviations from these expectations (see §B.3.3), returning a time-out termination. Should the outcome be unexpected (e.g. it indicates a time-out), a reformation will be initiated by the receiving QOP.

Provided reformation is successful, a single omission failure will be tolerated.

A.6.2.3 Incorrect occurrence failures

An incorrect occurrence failure occurs if the QOP receives an invocation or outcome which is not consistent with the interface definition (generic server expectation 1, and generic client expectation 3), or if it receives a token which violates the consistency relation expressed in expectations b and c of §B.4.8. For simplicity the two situations are considered separately.

Consider a client QOP attempting to make an invocation which is not defined in the QOP's interface definition (assuming it is not detected by link-time checking). This will deviate from the expectations of the server stub object which will detect it in a run-time type check and suppress the invocation. (It may also be detected by run-time type checks in the nucleus). Hence the

failure will be manifested as an omission failure and the discussion of §A.6.2.2 applies.

Consider a server QOP attempting to return an unexpected outcome (either it indicates a failure or the termination is not defined in the interface definition). (Assume the failure is not detected by link-time checking.) If the outcome would result in an undefined termination being returned to the client, it will also deviate from the expectations of the client stub object. The latter will detect it in a run-time type check and ensure an outcome indicating failure is returned. Once the outcome indicating failure is returned, reformation will occur. Hence such a single incorrect occurrence failure can be tolerated, provided reformation is successful.

An incorrect occurrence failure also occurs if the token holder sends different tokens to different QOP's, or a QOP receives a token in which the body of the list of outcomes or invocations has been changed. This deviates from expectations b and c of a QOP (see §B.4.8). Section §B.4.8 explains how these failures are detected. Once such a failure has been detected a reformation is initiated. Hence a single consistency failure can be tolerated, provided reformation is successful.

A.6.2.4 The reformation protocol

The reformation protocol was the most complicated part of the active replica group protocol to build. The reason for this is that it has to cope with failures while it is trying to reform the group, and the code to handle this must be built explicitly into the QOP. (Compare with clients and servers which are active replica groups, which rely on the infrastructure to detect and tolerate failures.) For simplicity the reformation protocol assumes that group members will only suffer omission failures, if they suffer any failures at all.

A.6.3 What kinds of failures can be detected and what faults can be tolerated?

The above shows that a three member replica group can detect any kind of single failure (inter and intra group) within the failure model. However, the reformation protocol assumes that only omission failures can occur. This means that the active replica group protocol will only tolerate omission failures: i.e. a group with two or more members will continue to deliver a correct service if a single omission failure occurs; if another kind of failure occurs, the group may fail. (Note that many incorrect occurrence failures are mapped by the nucleus or stubs into omission failures which can be tolerated).

A.7 Benefits of and comments on the analysis

This appendix has shown how to apply the ANSA failure model to the implementation of active replica groups described in [OSKIEWICZ 92]. This has given a considerable insight into the implementation and the kinds of failures which it can tolerate: this is absent from [OSKIEWICZ 93] and [OSKIEWICZ 92].

Appendix B analyses a set of engineering mechanisms to show under what failure modes the expectations of the computational objects will be met. Another way of using the failure model would be to start with the expectations of the computational objects and a set of assumed failure modes (a subset of those listed in the model). Then make an attempt to synthesize the engineering objects to ensure the expectations of the computational objects

will be met under the assumed failure modes. This is the purpose of the engineering model for dependability.

The analysis has not resulted in a statement about the dependability attributes of a service using the active replica protocol (e.g. mean time to failure or the percentage availability). Rather it has concentrated on the failures within the model which can and cannot be tolerated. Measuring dependability involves managing and monitoring issues which are for further study.

A formalism is needed for expressing and checking expectations, this would reduce the chance of faulty reasoning.

A.7.1 Insights on building dependable systems

By running the nucleus and engineering objects within some protection domain, for example somewhere where the client and server object cannot write, the nucleus and engineering objects can substantially restrict the failure modes of the application objects. They can restrict the frequency with which they can send messages, and can restrict the messages they can send. This does not mitigate against hardware failures or failure in the engineering objects or nucleus.

Applying the failure model has also given an insight into how the implementation might be improved. Although the implementation can only tolerate omission failures, a group with two or more members can detect any kind of single failure within the model. Thus a two member group could be used to form a fail-silent service from components which can exhibit any kind of failure.

A fail-silent component delivers a correct service until it suffers an omission failure. At this point the component is deemed to have crashed and will make no further invocations or outcomes until it is repaired. (Hence this is sometimes called a crash failure.) It is much easier to build dependable systems with components which only suffer such simple failure modes. For example, the reformation protocol of an active replica group would be very simple if it could be assumed that the faulty component had suffered a crash-failure. This is the approach which is being investigated in the Voltan project [SHRIVASTAVA 92] and has also been used in the Delta-4 project [POWELL 91].

A.7.2 Using expectations

There are two ways in which expectations are used in the active replica protocol studied in this appendix.

1. An object may have some mechanism to detect deviation from its expectations and perhaps enforce adherence to these expectations (e.g only allowing the invocation of operations whose names are known).
2. An object may rely on a third party to detect deviations from its expectations and enforce them. For example a client object relies on the quorum and ordering processor to enforce its expectation that all client objects will receive outcomes in the same order.

There is a high degree of duality between the client-side and server-side expectations: for many expectations in the server-side infrastructure there is a corresponding expectation in the client-side infrastructure and vice-versa. The differences in expectations reflect the polarity in the client/server relationship.

The duality is a reflection of the duality in the client/server relationship and can be used to cross-check client and server expectations.

B The expectations of the engineering objects

This appendix states the expectations of the engineering objects in figures A.2 and A.3. It also looks at the mechanisms which are used to enforce expectations and the mechanisms which are used to detect deviations from expectations. If no such mechanism is discussed for a particular expectation it should be assumed that deviations cannot be detected and the expectation is not enforced.

The expectations listed enable boundaries to be drawn around the expectation region. The problem of whether or not this is a complete set of expectations or a “good” set of expectations is not discussed; this is analogous to whether a specification is “good” or complete. Adding more expectations to the list will have the effect of further constraining the expectation region.

Expected outcomes are those which would occur if the interaction between all components was successful. Some IDL's include failure terminations in the interface definition (i.e. outcomes whose types indicate a failure has occurred). However, by definition a failure is something which is unexpected. In the following expected outcomes are referred to as being consistent with the interface definition, unexpected outcomes are not consistent.

ANSAware 4.1 is written in C. There is a notion of an interface between computational objects (i.e. clients and servers) and preprocessing support is provided to support these interfaces. The following assumes that the same abstractions are used for interaction between engineering objects. However, strictly there are no such interfaces between the engineering objects. Invocation corresponds to a procedure call; the value checks referred to are checks on the values of the arguments and results of these procedures.

To avoid repetition, the expectations of a generic client and a generic server are stated.

B.1 A generic client’s expectation of a sever

Any client will have the following expectations of a server.

1. The server will do nothing until it is invoked.
2. The server will return a single outcome within some time limit in response to a single invocation.
3. The outcome and its parameters will be consistent with the server's interface definition.

B.2 A generic server’s expectation of a client

Any server will have the following expectation of a client.

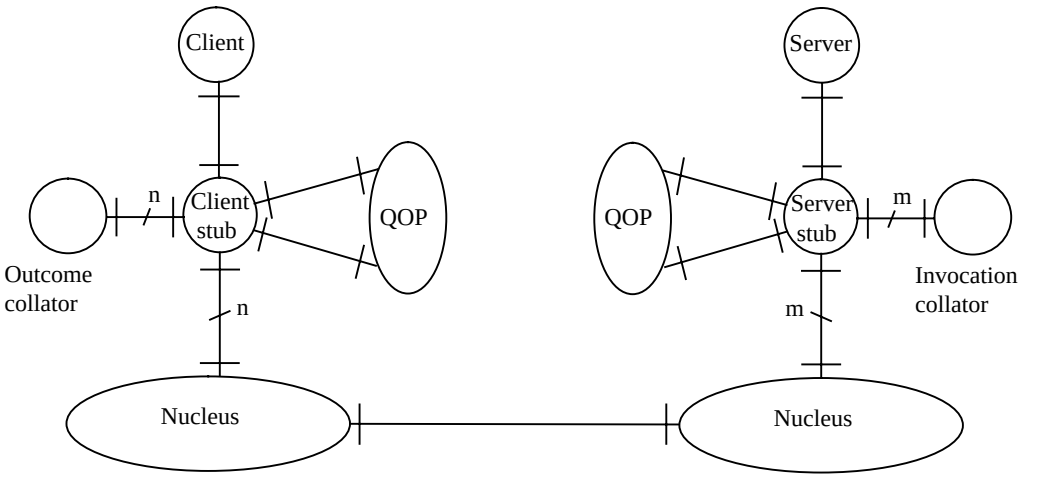
1. The invocations and their parameters will be consistent with the servers interface definition.

In the following the interface definitions will be those of the engineering objects (and not the computational objects). For example: the interface provided by the client stub to clients, the interface provided by the nucleus to client stubs, the interface provided by server stubs to the nucleus, etc. The interfaces between clients and servers, and their stubs are determined by the computational interfaces.

B.3 Expectations of client-side and server-side engineering

This section states the expectations which the engineering objects in figure A.2 have of each other. The expectations of the engineering objects reflect the expectations of the client and server objects. The function of the engineering objects is to ensure that the expectations of the client and server object are met. If for any reason, the nucleus or the engineering objects supporting a client or server object fail to meet its expectations, the client or server object will fail. This may cause other objects interacting with the client or server to fail. Section A.6 considers the inter and intra group tolerance of such failures.

Figure B.1: The engineering objects supporting active replica group



B.3.1 The client object's expectations of the client stub object

The client object has a generic client's expectation of the client stub object. In addition it expects the following.

a. The replicated servers should agree upon the outcome (both in value and in time).

b. The outcome should be consistent with the outcome delivered to the other clients.

The nucleus (see §B.3.3) and the outcome collator enforce generic client expectation 2. The nucleus delivers n and only n outcomes to the client stub object which uses the collator to convert these to a single outcome.

Deviations from generic client expectation 3 are detected by value checking in the client object. In ANSAware 4.1 a preprocessor inserts value checks in the client.

The client stub object uses the collator to enforce a.

The consistency relation in b requires that each client receives the same outcome, that the ordering of outcome delivery is the same at all clients, and that the property of uniformity is preserved. The QOP attempts to enforce this expectation (§A.5, §A.6.1.4 and §B.4 look at how and when this is enforced).

B.3.2 The client stub object's expectations of the client object

The client stub has the generic server's expectation of the client object. In ANSAware 4.1 link-time checks ensure that 1 is enforced.

B.3.3 The client stub object's expectations of the nucleus

A client stub object does not have a generic client's expectation of a nucleus, because the interaction is slightly different: it makes a single invocation of the nucleus and receives multiple outcomes.

a. The client stub expects the nucleus to do nothing until after it has made an invocation.

b. The nucleus should return one and only one outcome per server group member within some time limit.

c. The outcomes must agree with each other (both in value and in time).

d. The outcome and the arguments must be consistent with the nucleus interface definition.

The nucleus enforces b: it has a time out which ensures it will always deliver a outcome to the client stub, even if the outcome indicates a time out and is therefore unexpected. The nucleus also suppresses late outcomes and duplicates ensuring that only one outcome is returned per invocation and that a outcome is not delivered after a time-out.

The outcome collator defines and enforces the agreement in c. It will attempt to mask unexpected outcomes such as time outs by a majority vote. This will occur before the client stub passes an outcome to the QOP.

Ansaware 4.1 inserts some value checks in the client stub to detect deviations from d.

B.3.4 The nucleus' expectations of the client stub object

The nucleus has a generic server's expectations of the client stub object. Link-time checking and run-time value checks in the nucleus detect deviations from 1.

B.3.5 The nucleus' expectations of the server stub object

The nucleus has a generic client's expectation of the server stub object.

A time-out could be used to detect deviation from timing expectations in 2, but none is used in ANSAware 4.1 (it is not clear how it would interact with the time-out within the client's nucleus). The return of one and only one outcome is implicit in the thread model.

Link-time checking and run-time value checks in the nucleus detect deviations from 3.

B.3.6 The server stub object's expectations of the nucleus

a. Once it has received an invocation from the nucleus, the server stub object expects the nucleus to make m and only m invocations — this corresponds to each member of the client group making the invocation.

b. The invocations are expected to agree with each other (both in value and in time).

c. The server stub expects the nucleus' invocations and their parameters to be consistent with the server stub's interface definition.

The collator and nucleus enforce a: the collator has a time-out and suppression of late invocations to ensure m invocations are "received" once the first is received. The nucleus suppresses duplicate invocations to ensure no more than m invocations are received.

The collator defines and detects deviations from the agreement in b.

Link-time checking enforces enforce c.

B.3.7 The server object's expectations of the server stub object

The server object always has a generic server's expectation of the server stub object. In addition it has the following expectations.

a. The server object expects the replicated clients to agree upon the invocation (both in value and in time).

b. The server object expects the invocation to be consistent with the invocation delivered to the other servers.

Value checks (run-time type checking) by the server stub immediately before it invokes the server object enforce generic expectation 1.

The server stub object uses the invocation collator to enforce the agreement in a.

The consistency relation in b requires that each server receives the same invocation, that the ordering of invocation delivery is the same at all servers, and that the property of uniformity is preserved. Section A.6.1.4 (see also §A.5 and §B.4) looks at how and when this expectation is enforced by the QOP.

Server objects expect replicated clients to behave as a singleton. Expectations a and b express this in terms of conditions which are testable.

B.3.8 The server stub object's expectations of the server object

The server stub object has a generic client's expectation of the server object. The thread model means that enforcement of one and only one outcome being returned is implicit. There is no detection of deviations from timing expectations. Value checks could be used to detect deviations from expectation 3, but in ANSAware the programming model supported by C limits what can be done.

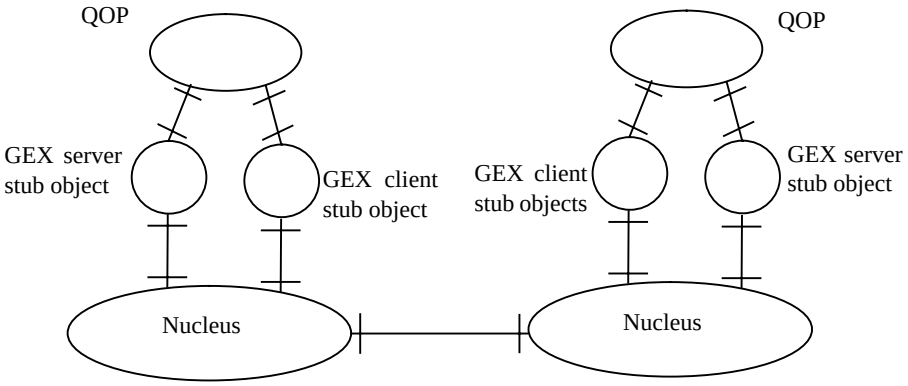
B.4 Expectations of the protocol engineering

This section states the expectations which the engineering objects in figure A.3 have of each other. In addition it describes the mechanisms which are used to enforce and detect deviations from these expectations.

QOP's are in a peer-peer relationship with each other. A QOP can be a client of the other QOP's invoking them to pass the token, requesting a missing invocation/outcome, or requesting a reformation. A QOP will also act as a server when it receives these requests from other QOP's.

The notion of holding the token is relevant only to the QOP: its expectations change when it holds the token. The other objects in figure A.2 are not aware of the token, so it does not affect their expectations.

Figure B.2: The quorum and ordering engineering objects



B.4.1 A client QOP's expectations of the GEX client stub object

A client QOP has a generic clients expectations of the client stub object. The nucleus (see §B.3.3) enforces expectation 2. Deviations from expectation 3 can be detected by value checking in the QOP — in ANSAware 4.1 a preprocessor inserts value checks in the client.

B.4.2 GEX client stub object's expectations of the client QOP

This is the same as stated in §B.3.2 *The client stub object's expectations of the client object*. This is because replication is transparent to the client object in §B.3.2, so the client stub object cannot expect any more from a replicated object than a non-replicated object.

B.4.3 GEX client stub object's expectations of the nucleus

The client stub object has a generic client's expectations of the nucleus.

The nucleus enforces expectation 2. The nucleus enforces the time limit by using time-outs, so a outcome will be delivered within the time limit (although it may be an exception indicating a failure and therefore unexpected). This ensure that one outcome is returned per invocation. The nucleus also suppresses late outcomes and duplicates ensuring that only one outcome is returned per invocation.

ANSAware 4.1 inserts some value checks in the client stub to detect deviations from expectation 3.

These expectations are those which most client server stub object's will have of their nucleus. In §B.3.3 the client stub knows it is participating in a many-many interaction and this affects its expectations. Some of its expectations are about how the remote servers will behave. They are projected onto the nucleus which effectively acts as a local proxy for the remote servers.

B.4.4 Nucleus' expectations of GEX client stub object

This is the same as stated in §B.3.4 *The nucleus' expectations of the client stub object*.

B.4.5 Nucleus' expectations of GEX server stub object

This is the same as stated in §B.3.5 *The nucleus' expectations of the server stub object*. (The nucleus expects all server stubs to offer the same service.)

B.4.6 GEX server stub object's expectations of nucleus

The server stub object has a generic server's expectation of the nucleus. Link-time checking enforces expectation 1.

These are different to the expectations stated in §B.3.6 *The server stub object's expectations of the nucleus*. Remarks similar to those in §B.4.3 apply.

B.4.7 GEX server stub object's expectations of a server QOP

These are the same as stated in §B.3.8 *The server stub object's expectations of the server object*. Remarks similar to those in §B.4.2 apply.

B.4.8 The server QOP's expectations of the GEX server stub object

A server QOP's has a generic server's expectations of its server stub which is acting as a remote proxy for the other QOP's. In addition if the QOP is a non-token holder it also expects the following.

a. It expects to see a token pass invocation from the current token holder within a certain time.

b. It expects the token to be consistent with the last one it saw.

c. It expects the same token to be sent to all members of the group.

Value checks in the GEX server stub enforce generic expectation 1.

A timer detects deviations from a. If a deviation is detected then a reformation of the group will be initiated. Security techniques could be employed to check that the invoker really is the current token holder.

The token contains a list of invocation identifiers. The job of the token holder is to remove the head of the list and add a new invocation to the tail of the list. The body of the list should remain unaltered — this is the consistency constraint referred to in b. Value checks can be used to detect deviation from this expectation (checksums can be used to help make this more efficient).

Deviations from c will be detected by the consistency check for b.

Note: A formal proof is needed to confirm this. Possibly induction based on assuming the token is correct and that the holder only suffers a failure sending a different token to group members. This can only be by inserting different invocations at the head of the list (otherwise it will violate expectation b). When the next token pass occurs from a correct group member, some members will see that the body of the list of invocations is different to the body of the list they have. This will violate b, and will be detected in a value check.