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

Applying the ANSA failure model to active replica groups

Nigel Edwards

Abstract and Context of this document

The aim of this document is to show how to apply the ANSA failure model to the ANSA and ODP computational and engineering models. The model is applied to an implementation of active replica groups. This has gives a considerable insight into the implementation and the kinds of failures which it can tolerate.

We are still learning how to apply the failure model so suggestions for changes and improvements to our method (such as it is) are invited. This is the first attempt to apply the model. It deals with a moderately complex real world problem, as opposed to being a tutorial which analyses a simple example.

APM.1046.00.02 **Draft** 26 November 1993

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

Distribution:
Supersedes:
Superseded by:

Contents

1	**1**	**Applying the ANSA failure model to active replica groups**
1	1.1	Summary the failure model and how it is used
2	1.2	How to read the rest of this document
3	1.3	An engineering specification for active replica groups in ANSA
5	1.4	Expectations of the computational objects
5	1.5	The GEX quorum and ordering protocol
6	1.6	Failure detection and tolerance
7	1.6.1	Inter group fault tolerance
10	1.6.2	Intra group fault tolerance
11	1.6.3	What kinds of failures can be detected and what faults can be tolerated?
11	1.7	Conclusions
12	1.7.1	Insights on building dependable systems
12	1.7.2	Using expectations
13	1.8	Acknowledgements
17	**A**	**Appendix: The expectations of the engineering objects**
17	A.1	A generic client's expectation of a sever
17	A.2	A generic server's expectation of a client
18	A.3	Expectations of client-side and server-side engineering
18	A.3.1	The client object's expectations of the client stub object
19	A.3.2	The client stub object's expectations of the client object
19	A.3.3	The client stub object's expectations of the nucleus
19	A.3.4	The nucleus' expectations of the client stub object
19	A.3.5	The nucleus' expectations of the server stub object
20	A.3.6	The server stub object's expectations of the nucleus
20	A.3.7	The server object's expectations of the server stub object
20	A.3.8	The server stub object's expectations of the server object
20	A.4	Expectations of the protocol engineering
21	A.4.1	A client QOP's expectations of the GEX client stub object
21	A.4.2	GEX client stub object's expectations of the client QOP
21	A.4.3	GEX client stub object's expectations of the nucleus
22	A.4.4	Nucleus' expectations of GEX client stub object
22	A.4.5	Nucleus' expectations of GEX server stub object
22	A.4.6	GEX server stub object's expectations of nucleus
22	A.4.7	GEX server stub object's expectations of a server QOP
22	A.4.8	The server QOP's expectations of the GEX server stub object

1 Applying the ANSA failure model to active replica groups

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

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 [EDWARDS 93], 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 this document.

1.1 Summary the failure model and how it is used

The ANSA failure model [EDWARDS 93] 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

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

assumed to be in the component which engaged in the event, rather than in 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 93].

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.

1.2 How to read the rest of this document

Section 1.3 gives a brief description of the engineering objects for active replica groups.

Section 1.4 states the expectations of the computational objects.

Section 1.5 gives a brief description of the engineering objects involved in intra group interaction.

Section 1.6 applies the methodology outlined above in §1.1 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 1.7 concludes the document 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 §A.3. Appendix §A.4 gives a statement of the expectations of the objects involved in the intra group binding.

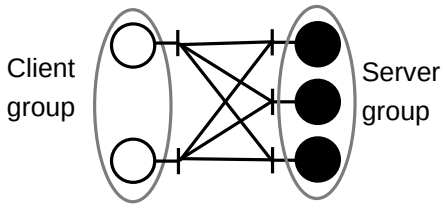
It is recommended that anybody familiar with [OSKIEWICZ 93a] or [OSKIEWICZ 93b] still reads §1.3 and §1.5, as the presentation draws attention to points which are used in the subsequent analysis. The appendices are included for completeness: the analysis in §1.6 can be read without reading these first. However, it should be remembered that the material in the appendices 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 the appendices.

1.3 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 93b]. 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 this paper. (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 1.1.

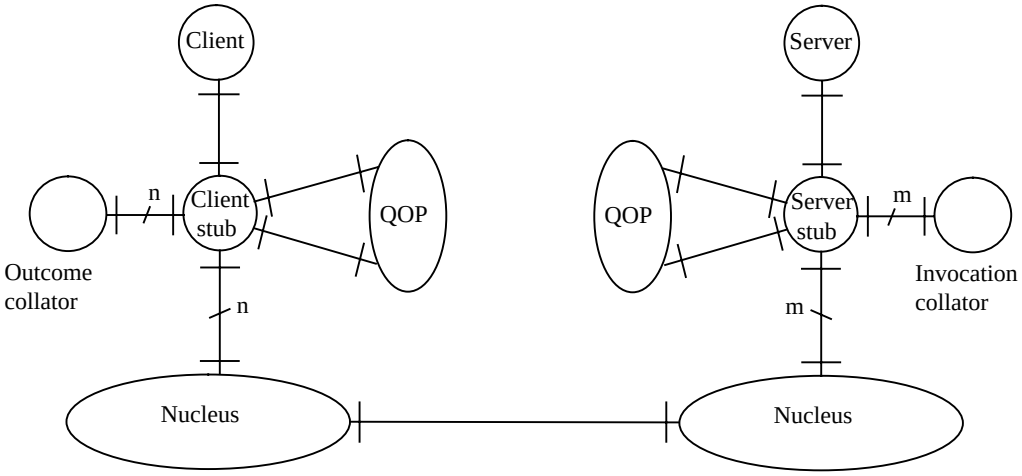
Figure 1.1: Many-many interaction



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 93b] 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 1.2 shows the engineering objects supporting one member of

Figure 1.2: The engineering objects supporting active replica group



the client group and one member of the server group in the implementation described in [OSKIEWICZ 93b]. 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.

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

1.4 Expectations of the computational objects

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

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.

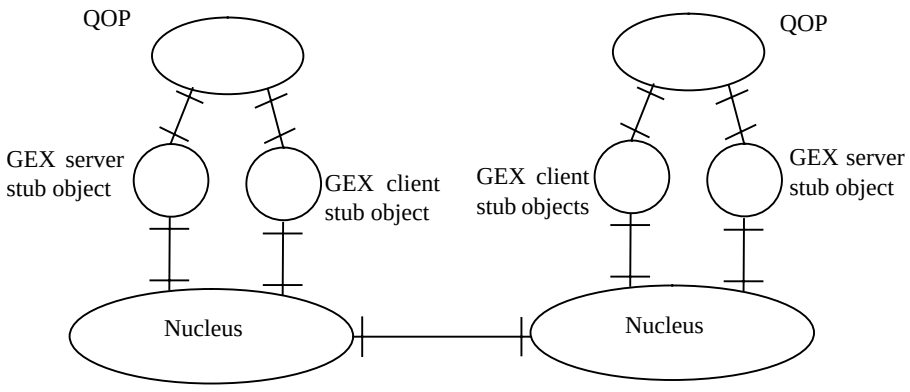
Client and servers will have expectations in the computational viewpoint which are fixed by the application semantics. For example most clients will have implicit expectations about the time and values which can be associated with outcomes it expects to receive from servers. In addition they are likely 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 93b] 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 the appendices: the client or server will fail if these expectations are not met.

1.5 The GEX quorum and ordering protocol

This section gives a brief description of the GEX quorum and ordering protocol described in [OSKIEWICZ 93b] and based on [CHANG 84]; readers needing further details are referred to these texts. Figure 1.3 shows the engineering objects

Figure 1.3: 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 1.2); this binding is not shown in figure 1.3. The remainder of this section describes the interaction between the objects in figure 1.3.

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 1.2) 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.

1.6 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 [EDWARDS 93]. These failure terminations allow clients to detect the failure.

The failure enumeration is taken from [EDWARDS 93] which lists the failures which can occur.

The nature of the tests to detect deviations from expectations are discussed in detail in the appendix. 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.

1.6.1 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 §A.3.1, [EDWARDS 93] lists 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 §A.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.

1.6.1.1 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.

1.6.1.2 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, §A.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, §A.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, §A.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.

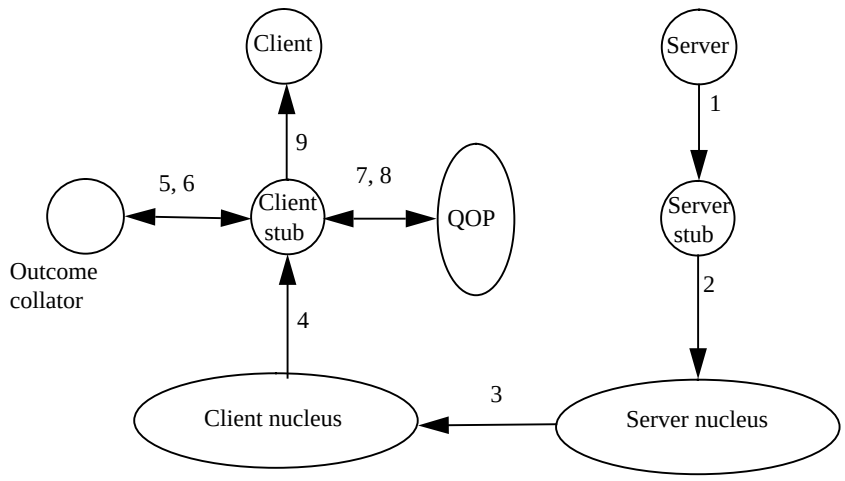
Hence three or more members are required in a client group to tolerate a single omission failures; two or more members are required in a server group to tolerate single omission failures.

1.6.1.3 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 [EDWARDS 93]. 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, §A.3.1 expresses a consistency relation between the outcomes delivered to each client object. As explained in [EDWARDS 93] 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 §1.6.1.4; this section deals with occurrences which potentially violate generic client expectation 3, §A.1 or expectation a, §A.3.1.

Figure 1.4: Incorrect occurrence in a server group member



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.

A single occurrence failure in a server (see figure 1.4) may result in unexpected outcomes being delivered to the server stub object (1) or the server's nucleus (2) (both cases violating generic client expectation 3, §A.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 §1.6.1.2 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, §A.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 (§A.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 §1.4).

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 (§A.3.3) and expectation a of the client (§A.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.

1.6.1.4 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, §A.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 §1.5).

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]

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

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.

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 §1.6.1.3).

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.

1.6.2 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.

1.6.2.1 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.

1.6.2.2 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, §A.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, §A.1). Deviations from this expectation will also deviate from the expectations of the client stub object (generic client expectation 2, §A.1). The nucleus will detect deviations from these expectations (see §A.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.

1.6.2.3 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 §A.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 §1.6.2.2 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 §A.4.8). Section §A.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.

1.6.2.4 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.

1.6.3 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).

1.7 Conclusions

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

The document 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 [OSKIEWICZ 93c].

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.

1.7.1 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].

1.7.2 Using expectations

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

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.

1.8 Acknowledgements

The author is grateful to Santosh Shrivastava of the University of Newcastle upon Tyne, and to the following members of the ANSA team for their comments on an earlier versions of this document: Ed Oskiweicz, seconded to the ANSA team by British Telecom, and Owen Rees of APM Ltd. A discussion with David Iggulden of APM Ltd. also proved very helpful.

References

[BARWISE 87]

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

[CHANG 84]

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

[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.

[EDWARDS 93]

Edwards, N.J., Rees, R.T.O, Oskiewicz, E., Warne, J.P., “A Model for Failures in Dependable Systems”, APM.1027, APM Ltd., Cambridge, U.K., 1993.

[JONES 86]

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

[LINDEN 93]

van der Linden, R.J., “An Overview of ANSA”, AR.000.00, APM Ltd., Cambridge, U.K., February 1993.

[OSKIEWICZ 93a]

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

[OSKIEWICZ 93b]

Oskiewicz, E., Edwards, N.J., “ANSAware 4.1 support for interface groups”, TR.35, APM Ltd., Cambridge U.K., February 1993.

[OSKIEWICZ 93c]

Oskiewicz, E., “An Engineering Model for Dependability”, APM.1044, APM Ltd., Cambridge U.K., 1993.

[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 93]

Rees, R.T.O, “The ANSA Computational Model”, AR.001.01, APM Ltd., Cambridge, U.K., February 1993.

[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.

A Appendix: The expectations of the engineering objects

This appendix states the expectations of the engineering objects in figures 1.2 and 1.3. 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 [EDWARDS 93]. 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.

A.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.

A.2 A generic server's expectation of a client

Any server will have the following expectation of a client.

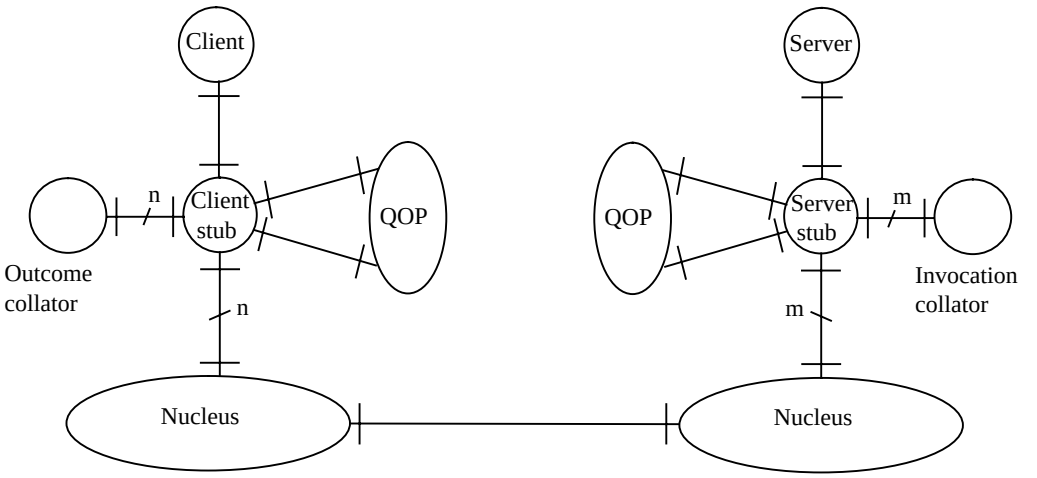
1. The invocations and their parameters will be consistent with the servers interface definition.

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.

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

This section states the expectations which the engineering objects in figure 1.2 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 1.6 considers the inter and intra group tolerance of such failures.

Figure 1.2: The engineering objects supporting active replica group



A.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 §A.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 (§1.5, §1.6.1.4 and §A.4 look at how and when this is enforced).

A.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.

A.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.

A.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.

A.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.

A.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.

A.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 1.6.1.4 (see also §1.5 and §A.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.

A.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.

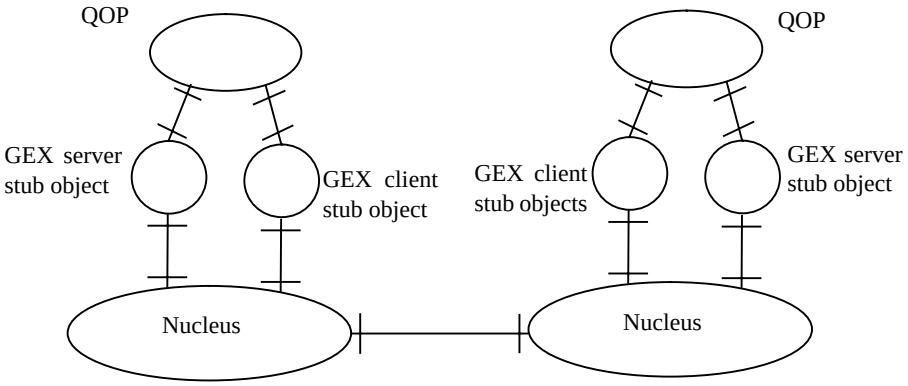
A.4 Expectations of the protocol engineering

This section states the expectations which the engineering objects in figure 1.3 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 1.2 are not aware of the token, so it does not affect their expectations.

1.3: The quorum and ordering engineering objects



A.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 §A.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.

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

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

A.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 §A.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.

A.4.4 Nucleus' expectations of GEX client stub object

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

A.4.5 Nucleus' expectations of GEX server stub object

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

A.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 §A.3.6 *The server stub object's expectations of the nucleus*. Remarks similar to those in §A.4.3 apply.

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

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

A.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! Induction? Base it on assuming the token is correct and that the holder only suffers a failure sending a different token to group members. This can 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.