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 Interface Groups

Ed Oskiewicz, Nigel Edwards

Abstract

Distributed systems differ from centralised systems because application designers and system builders have to learn to deal with properties like concurrency and partial failure. Although these properties are a source of complexity they can be exploited to increase performance and robustness, and to build applications supporting interactions between sets of entities.

To exploit these properties easily, abstractions are needed for the wide variety of communication, synchronization and error recovery mechanisms which are possible in a distributed system. This document presents one such abstraction, the *interface group*. In addition to resolving distribution issues, interface groups provide an easy-to-understand programming structure for replicated services.

The basic group abstraction is to treat a number of objects as though they were one. This enables clients to invoke operations on collections of objects without needing to know the exact membership of the collection or the location of the members. This capability provides the basis for distributing the implementation of a service over a set of objects.

APM.1002.01 **Approved** 19 May 1994
Architecture Report

Distribution:
Supersedes:
Superseded by:

A Model for Interface Groups

A Model for Interface Groups

Ed Oskiewicz, Nigel Edwards

APM.1002.01

19 May 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 Introduction

The purpose of this report is to explain how interface groups fit into the ANSA architecture. In particular it describes the engineering consequences of making interface groups computationally transparent and identifies a set of requirements for an engineering model supporting interface groups. A companion document, [OSKIEWICZ 93] describes an example implementation. The target audience is system and application designers interested in the consequences of exploiting multicast protocols and replication techniques within distributed systems. The reader is assumed to be familiar with the basic concepts of ANSA [ANSA 91], ODP [ODP] or OMG-CORBA [CORBA 92].

1.1 What are interface groups?

Distributed systems differ from centralised systems because application designers and system builders have to learn to deal with properties like concurrency and partial failure. Although these properties are a source of complexity they can also be exploited to increase performance and robustness, and to build applications supporting interactions between sets of entities.

To exploit these properties easily, abstractions are needed for the wide variety of communication, synchronization and error recovery mechanisms which are possible in a distributed system. This document presents one such abstraction, the **interface group**. In addition to resolving distribution issues, interface groups provide an easy-to-understand programming structure for replicated services.

The generality of interface groups is such that in ANSA all interfaces may be group interfaces by default: non-group (singleton[1]) interfaces can be treated as a special case optimization.

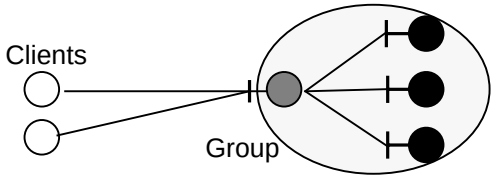
1.1.1 The group abstraction

The basic group abstraction is to treat a number of objects' interfaces as though they were one. This enables clients to invoke operations on collections of objects without needing to know the exact membership of the collection or the location of the members and thus provides the basis for implementing replicated services.

An interface group is a collection of one or more interfaces which are accessible externally through a single service interface (see figure 1.1). This service interface can be used just like any singleton interface. All the interfaces in a group must conform to a common interface type — this will be used as the type of the group service interface.

1. **Terminology**: unless qualified, terms are used with the following meanings: **group** is short for interface group; **member** is short for interface group member; **singleton** is short for singleton interface (i.e. a non-group interface.

Figure 1.1: An interface group



If a group is **transparent**, interaction with the group is (computationally) indistinguishable from interaction with a singleton providing the same service, i.e. the client makes a single **invocation** and receives a single **termination**. If a group is not transparent the client may expect to receive a response from every member and must construct a single termination for itself by inspecting and collating all the responses.

There is no simple one-to-one relationship between objects and members. An object may provide many interface instances as multiple independent group members, or it may provide only a single interface. Higher level criteria, such as dependability requirements and fault model assumptions, may determine whether these attempts result in membership. For example, particular groups may have policies in force restricting the number of members allowed.

1.1.2 Active and passive replication

There are two basic kinds of replication. In an **active replication** scheme each input message is processed concurrently by all replicas, so their internal state is synchronised closely [POWELL 91]. In a **passive replication** scheme only one (primary) replica processes the input messages and provides the output messages: the others regularly update their state by processing a log (see 5.4.3).

If they are transparent there is no computational distinction between active and passive replica groups although the engineering will contain differences in detail and approach. So far in ANSA, experience of transparent groups has been confined to active replica groups, much of the discussion of the engineering required for transparent groups will reflect this. The computational model does not preclude other kinds of transparent groups.

1.2 Why do we need groups?

Groups are a consequence of multi-endpoint communications and have two important uses: they are a way of providing transparent fault tolerance and reliability using replication; and they provide a programming abstraction to control parallelism and build distributed applications (e.g conferencing).

Group transparency allows clients to interact with server groups, and vice-versa, regardless of the details. It is, however, possible to relax the transparency to allow applications to use some aspects of group functionality directly, e.g. message distribution.

A group is said to be **closed** if invocations on the group are only made by other members of the group, otherwise it is **open**. In this document the emphasis is on open groups[1].

A variety of applications can be built using groups:

Highly available applications: many applications require continuous availability of function even during hardware maintenance or software upgrade. Implementing the service as a group allows enough redundancy to mask change-overs or failures.

Conferencing applications: some applications are based solely on the ability to perform multicasting: there may be no shared state *per se*, simply the exploitation of a membership list. The conference persists regardless of who is in the current membership. More structured conferences require members to share common state can be built using higher level group abstractions.

Management applications: a system manager may wish to treat a pool of otherwise independent objects as a group to exert control policies by issuing *generic* invocations, e.g. gathering accounting details. This is grouping of instances to form classes. The internal state of each instance may differ, so different responses are expected and collation is done by the application. This example shows how the physical distribution of a logical entity may be masked from the manager of that entity.

1.3 Overview of the rest of the document

An interface group is a collection of one or more interfaces which are accessible externally through a single service interface. If the interaction with a (server or client) group is indistinguishable from an interaction with a singleton, the group is said to be transparent. Non-transparent interaction is also possible, non-transparent interaction may involve obtaining explicit knowledge of who is in the group and dealing with multiple responses when invoking a group.

A transparency is implemented by an orchestrated set of mechanisms. These mechanisms provide services through interfaces. Policy objects partially define the behaviour of these mechanisms; different policy objects defining different behaviours. A number of mechanisms are necessary to support transparent group interaction: distribution, collation, sequencing and quorum processing.

Groups need to be managed so that they deliver a stated quality of service. The most important aspect of quality of service for active replica groups is the class of failures which they can tolerate. A rigorous failure model is needed to compare different kinds of groups delivering different quality of service guarantees.

Implementing groups raises a number of issues for the communication infrastructure. Multicast communication is useful, but error handling is not straightforward. As well as inter-group communications it is necessary to provide a high degree of intra-group interaction so that the group may keep itself in synchronisation. Handling population changes transparently requires a relocation service so that a distributor can obtain an up to date view of the membership. It is necessary to synchronise new members when they join a group.

1. Closed groups are relevant to application areas such as conferencing and for distributing management information within a group. However, open groups have greater generality and an open group can always interact with itself via the group's service interface, if such a group has no external clients it is effectively a closed group.

1.4 Acknowledgement

The authors acknowledge the valuable contributions made by all the members of the ANSA Team, in particular Michael Olsen, seconded to the ANSA Team by Hewlett-Packard, and John Warne, seconded to the ANSA Team by BNR-Europe.

2 Computational issues for interface groups

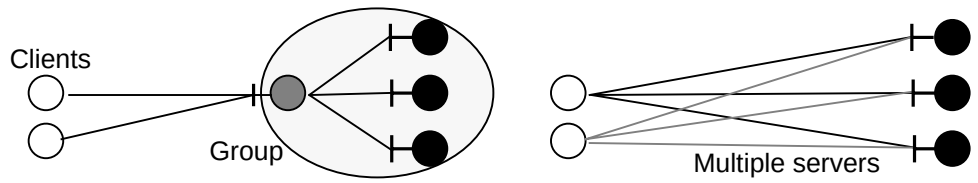
2.1 Introduction

When designing group-based applications, there is a tension between the wish to make unwanted behaviour transparent (which tends to mask group structures) and the need to exploit decentralization (which tends to expose group structures). The trade-off is simplicity versus flexibility.

The ANSA computational model is described in [AR.001]. While groups introduce no new notions to the computational model, maintaining computational transparency requires a great deal of engineering support. Hence this chapter supplements [AR.001] by describing the engineering consequences arising from the need to make interface groups invocationally (computationally) transparent, i.e. allowing a set of interfaces to be invoked by a client as though they were a single interface. Transparency means it is also necessary to hide the consequences of group membership from members of a group, e.g. masking synchronisation of members. The emphasis is on active replica groups (see 1.1.2).

Server groups can mask replication and distribution by hiding details of addressing, parallel execution, consistency and error recovery from clients of the group. This is in contrast to the effort that would be necessary if a client were to orchestrate a set of servers directly; for example, creating or locating the servers, communicating with them, coordinating their multiple replies, reconfiguring in the event of failure and so on. Furthermore, without support for groups, clients using the same service would have to liaise with each other to avoid conflicts (see figure 2.1).

Figure 2.1: Transparent and explicit groups



A client group arises when members of an active replica group invoke a third party. Active replication requires that all the members must have the same state, and so they must all see the same results of interactions. Arranging that they invoke operations at the same instant is infeasible; some other means of masking replication, such as collation of requests or caching of results is needed.

Computationally, two factors which members of a group have in common are that they all conform to a common interface type and that all receive each invocation of operations in the group interface. Internal mechanisms within the group will ensure that the required degree of synchronisation between members is maintained, e.g. that invocations are evaluated in the same order.

The group interaction model hides population changes. Any population changes which do occur must not affect the interaction i.e. they should be masked by engineering mechanisms. This may involve deferring the interaction until a safe moment or causing it to be restarted transparently. Uncontrolled population decrements during an invocation may, given sufficient replication, be ignored.

2.2 Interaction model

The ANSA interaction model defines the allowed forms of interaction between objects and the type system used to classify those interactions. The interaction model for groups is the same as that for singletons, however there are many extra engineering mechanisms necessary to make this transparent, e.g. collation and distribution. This section introduces these concepts and shows what happens when transparency is relaxed. Table 2.1 summarises the difference between transparent and non-transparent behaviour when interacting with a group or being a group member.

Table 2.1: Summary of transparency vs non-transparency

	Transparent	Non-transparent
Interacting clients and servers	cannot tell they are interacting with a group, i.e. the interaction style is identical for groups and non-groups.	are group-aware and may choose to exploit this knowledge, e.g. application specific termination or invocation collation
Group members	cannot tell they are members of a group, i.e. there is nothing about receiving or making invocations which differs for group members or singletons.	are aware of their membership and may choose to exploit this, e.g. by choosing when to join/leave the group or to interact with other members in a non-transparent fashion.

2.2.1 Invocations

The invocation scheme is identical to that for non-groups (but there are extra terminations for group run-time errors). An invocation may be either an announcement or an interrogation.

When activities within members invoke operations on some other interface, they are capable of doing so either as a singleton, when they behave as any other singleton, or as a group member, when the invocation is made by one or more group members acting as a ‘client group’.

2.2.2 Inter-group mechanisms to support active replication

Figure 2.2 shows a client group interacting with a server group — a many to many interaction, note that the groups need not be the same size. To make the interaction appear to be between singletons, the infrastructure must provide mechanisms to coordinate the behaviour of group members so that they appear to behave as a singleton.

This behaviour is shown in more detail in figure 2.3 and consists of the distribution and collation of messages to support transparent replica groups. For simplicity, intra-group interactions (e.g. to synchronise distributors and collators) are not shown.

Figure 2.2: Many-many interaction

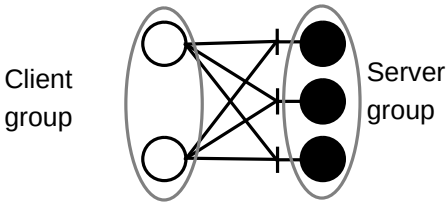
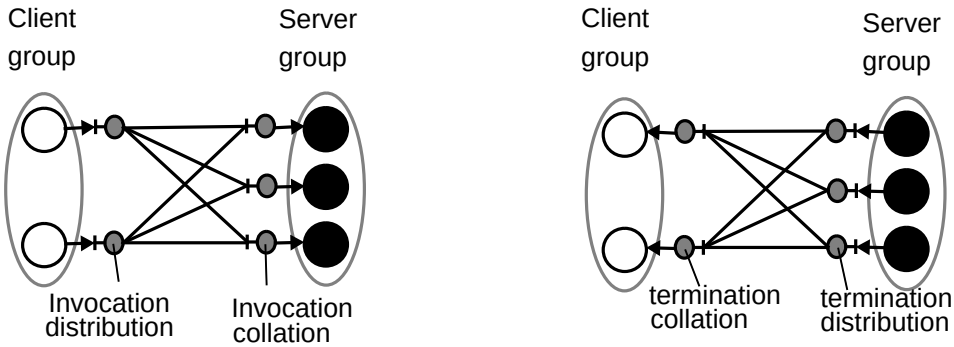


Figure 2.3: Invoking a server group



Distribution is the process of sending an invocation or termination to all members of a group.

Collation is the process of reducing multiple messages from a group into a single message. When a client group makes an invocation each member of the client group may try to invoke the server (see figure 2.3). These invocations need to be collated into a single invocation which is delivered to the server application. Conversely each member of the server group may return terminations which require collation.

The collation process may be prepared to produce a result from a subset of messages, this can provide fault tolerance. In the presence of faulty communications it is possible that messages will be lost and the invocation or termination may be incomplete. If the incomplete invocation or termination fails to pass through the collation process it will be ignored and may cause some form of error termination to be returned to the sender by the collator.

2.2.3 Intra-group mechanisms to support active replication

It is also necessary for an active replica group to compensate for the failure of individual group members. If the state of group members was allowed to diverge, it would not be possible to handle failures transparently. Hence group members need to be coordinated by the infrastructure to prevent state divergence; by default, this coordination should be transparent to the programmer. This coordination fulfils two requirements: it ensures that group members process invocations and terminations in the same order; and it ensures that they do not begin this processing until they are immune to some number of failures.

Sequencing involves group members arriving at an agreement about the order in which invocations and terminations will be delivered to the application.This is necessary because different orders at different members could cause state divergence.

When an invocation or termination message is sent to a group it is possible that some group members may not receive the message. To survive k failures the infrastructure needs to ensure that k+1 members have received the message before any of them act upon it, this is called **quorum determination** [OSKIEWICZ 93] [5.3.1].

Concurrency within a group member can introduce nondeterminism which, in turn, could cause state divergence. The infrastructure needs to provide **concurrency control** which ensures that concurrent execution is deterministic (see [AR.004]). Note also that whatever ordering is established by the inter-group protocol must also be made available to the concurrency control logic so that the intended determinism can be maintained or restored. The difference between sequencing and concurrency control is that the former is concerned with establishing an order between members while the latter is concerned with establishing an order within a member.

Additional mechanisms to support active replication such as the handling of population changes and failure handling are discussed in chapter 5.

2.2.4 Activities

An **activity** is the means by which a computation makes progress.

The computational model [AR.001] describes only singleton interactions and thus describes activities being transferred by interrogations or being created by announcements. With groups, the situation is a little more complicated because a client group with n active replicas may invoke a server group with m active replicas and thus sub-activities may appear to be created or made dormant to give the illusion of there being a single activity on each side of the invocation.

If a singleton invokes a group then extra activities are created (at the recipients) for the duration of the evaluation of the operation. If a group invokes a singleton then all but one of the recipient activities will appear to be dormant until the reply is sent back. When a group invokes another group then activities may appear to be created or made dormant depending on the protocols used and the relative cardinalities of the groups.

2.3 Collation

Collation is central to the ability to provide transparent interface groups. It is the means by which interactions between groups can be hidden from the application program. The purpose of collation is to take a set of requests or replies and derive from them a single request or reply which is seen by the application.

This section raises two of the principal questions, what does it mean to say that two requests or replies are the same (or different)? and how may references to interfaces be collated so that they may then be invoked? These need to be reconciled with the computational model view that everything may be modelled as a service so that sameness may only be determined via invocation. This area contains many unresolved issues, some of which are outlined in 2.6.

2.3.1 How is sameness determined?

In the computational model, arguments and results are references to interfaces. For transparency (i.e. to preserve the single client/server illusion), arguments or results from a group are subject to collation. Typically collation is used to determine the (in)equality of things; however, what is meant by collation of references to interfaces is still an unresolved question. There are at least four answers depending on what sameness criteria you are applying.

The simplest is to require that their concrete representation be identical, this is *simple equivalence*. However two interface references whose concrete representations are different (e.g. contain non-identical network addresses) may still refer to the same interface (i.e. they are *invocationally identical*). Another possibility is the interface references may be different but deliver the 'same' result if invoked, this is similar to CCS *observational equivalence*[1] [HENNESSY 85]. Finally, there is sameness in the information sense where we are interested in whether two things have the *same meaning*, i.e. does applying interpretation_1 to data_1 have the same meaning as applying interpretation_2 to data_2?

This area is critical for groups and to make progress we need to make and document some simplifying assumptions about our desired sameness criteria — there is no generic sameness test which will give a reliable answer under all circumstances. Indeed in a purely computational model there is no way to determine sameness. However things aren't as bleak as this might indicate because many things are immutable or have invariant concrete representations so there are usually engineering solutions to many of these problems.

2.3.2 The group factory problem- what is the result of collation?

This problem is stated as follows: if a group is invoked and each member returns an interface reference as a result, how is the single server illusion preserved when the client decides to invoke the result it is passed?. This is still an open question (see 2.6) which requires answering before interface groups can be fully reconciled with the computational model [AR.001].

2.4 Type scheme

In [ODP], environmental constraints are part of the computational type system. ANSA needs a similar notion adding to it (possibly as a type system for the engineering model) because client service requirements must be matched with server properties before interaction can occur.

Client and server must be bound together before interaction can occur. This may be done at run time by passing interface references as arguments or results of invocations; it may also be done at compile or link time by declarative statements in a programming language. To enable the binding to occur, the client will give a type specification for the service it requires and the server will provide a type specification for the service it is offering.

1. Note that even observational equivalence only identifies sameness or difference at the instant(s) of interaction, they may diverge or coincide at a later time. In particular, if an interface gives different results on successive invocations, then even if you have the 'same' interface reference invoking it twice to determine sameness will indicate otherwise.

The process of binding a client to a server interface must ensure that the two specifications are conformant. To enable members to satisfy the type specification of the service offered, each must conform to some common supertype so that clients are able to invoke each member (member interfaces may contain extra operations which clients are unable to invoke because they do not appear in the supertype).

Because groups can return group specific error terminations these must also be dealt with if a client is able to invoke a group service transparently and so the description of the supertype must contain all possible terminations. So that the client is not surprised by unexpected terminations the interface which the client invokes must conform to the supertype.

Because it is not usually known in advance whether a group will be invoked or not this implies that group failures like other engineering failures are ubiquitous, i.e. they are terminations associated with all interfaces but are only manifested when invoking a group. It is still a local infrastructure decision whether the extra terminations are masked or made to appear as an apparent service failure.

2.5 Abstract Representation

A type specification consists of a set of operation signatures together with the attributes of those operations. In a programming language the attributes name a set of transformations which modify the source code to produce the desired effect. For example an attribute processor may be applied to the client and server source code; this may result in altered source code (a macro expansion process) or the generation of appropriate stubs or other wrapper code.

Groups provide a good opportunity for the use of attributes to simplify the application programmer's view of the system. Modified source code can take account of any new behaviour introduced by knowledge that a group is involved in an interaction, by selecting a suitable protocol, and choosing collation and management policies.

In this fashion a basic non-transparent server application may be augmented by the provision of increasingly powerful transparency managers which build higher level abstractions. Given a knowledge of the group service type and attributes these group agents may be constructed automatically.

2.6 Outstanding issues

The computational description of groups other than active replica groups.

The description of activities needs to aligned with the work on activity trees in [AR.004].

What does collation of interface references really mean?.

The group factory problem is a general architectural problem and needs a clear architectural solution - there are many engineering solutions. Avoiding complex collation problems by some action at the server group seems like a good strategy.

Complete transparency for group members is not possible. For transparency, non-members invoke operations as singletons and members of a group invoke

operations as a member of a client group. However, there are times when the underlying engineering needs to be aware of whether operations are being made by a group member or not as some operations, e.g. relocation need to be made as a singleton. Operations explicitly invoked as though by a singleton are necessarily done in a non-transparent fashion.

We have still to investigate and document the simplifying assumptions which will make sameness tests for references to interface references practical.

Programming language attributes have not yet received sufficient investigation.

3 Group transparencies

This chapter looks at the transparencies needed for groups. It defines the transparency requirements for active replica groups and discusses how the behaviour of these transparencies can be changed. Finally it discusses how the transparencies might be relaxed to build different kinds of groups and how these groups can be defined. Ultimately the aim is to understand how to configure a structured kit of parts to build different kinds of groups.

3.1 What is a transparency?

A property of a system is said to be transparent if application programmers need not be concerned with it. For example, replication is transparent if it is not apparent to either clients or servers, i.e. the interaction model is exactly the same for replicas as it is for singletons.

3.1.1 Mechanisms and policies

Transparency is achieved by employing a set of mechanisms in the infrastructure to compensate for an unwanted property of the system. One cannot generally identify a particular transparency with a particular mechanism, rather a set of mechanisms are orchestrated to implement a particular transparency. These compensating mechanisms can be sited in the client capsule, in the server capsule, or in both.

A mechanism provides a service through a typed interface which can be defined in an interface definition language (IDL). This enables existing mechanisms to be substituted by new mechanisms conforming to the same type as the original.

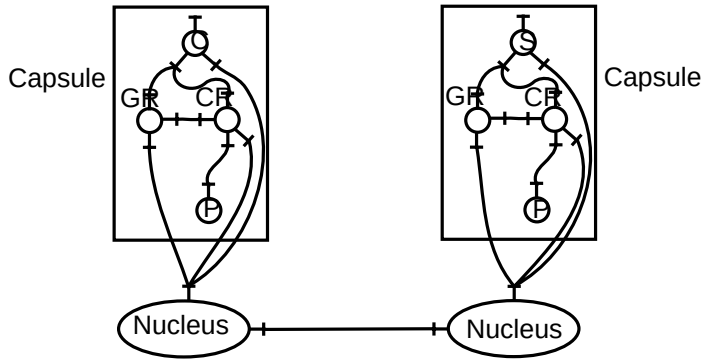
The behaviour[1] of a mechanism may be altered by policy objects which the mechanism invokes. These policy objects have typed interfaces which can be defined in an IDL. The behaviour of a mechanism can be partially changed by passing it an interface reference to different policy objects.

Policy objects are an engineering convenience for changing the behaviour of a generic mechanism without having to build a completely new one. For example, an interface to a collator mechanism will support the operations necessary to reduce multiple messages to a single message. Precisely how this is done will be determined by policy objects (see 3.3.1 and [OSKIEWICZ 93]). Some mechanisms may not make use of policy objects; a change of behaviour may involve using a new mechanism.

1. **Terminology note**: the distinction between mechanism and policy is somewhat fuzzy, a mechanism can be thought of as a rather coarse grained piece of the system, e.g. the means to create new members to replace failed members. A policy is something that parameterises that mechanism, e.g. to ensure there are always at least three members. Behaviour is used in it's usual English sense.

As transparencies may be deployed in many parts of the system, a mechanism can be invoked by applications, other mechanisms or the nucleus. Figure 3.1 is adapted from [ODP]; it shows the set of mechanism and policy objects acting as proxies and agents for client and server objects to provide replication transparency in the implementation described in [OSKIEWICZ 93]. The client application has the illusion of invoking a server's interface directly, in reality it is invoking a chain of mechanisms which are transparent to it.

Figure 3.1: Mechanisms supporting active replication



C: client group member S: Server group member
GR: Group runtime mechanism object
CR: Collator mechanism object P: Policy object

Transparencies are selectable or graduated (see sections 3.3 and 3.4). The transparency requirements of an application constrains the set of mechanisms and policies which can be used in the infrastructure.

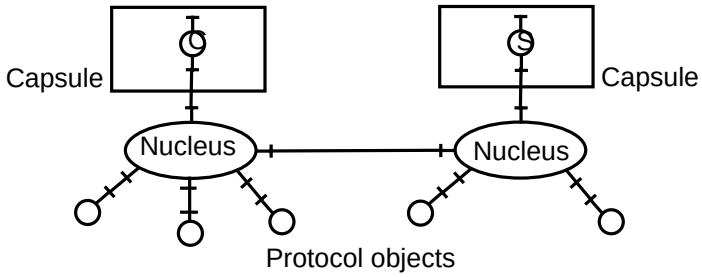
Clients and servers may make independent choices of mechanisms and policies. For example replication may be transparent to members of a server group, but visible to a client of the group. Different clients can also make independent choices of mechanism and policy. This enables different clients to have different views of a server group.

As elsewhere in ANSA, transparency cannot mask time delays and may break down in the presence of unrecoverable errors. Both mechanisms and policies can fail: every member of an active replica group may fail. If the mechanisms and policies cannot deal with the failures themselves, then a termination must be issued so that an application can attempt recovery such as rebinding to another group. To preserve transparency the termination may need to be mapped to a termination which is not specific to groups (see section 2.4).

3.1.2 Implementation options

The advantage of implementing the support for groups above the nucleus (as shown in figure 3.1) is that it is likely to be easy to port the application to other platforms such as DCE and OMG-CORBA. A disadvantage is that some of the functionality of the nucleus is likely to be duplicated. Figure 3.2 illustrates an alternative strategy in which the nucleus acts as switch between protocols, some of which will implement active replication. This is likely to be a more efficient implementation which minimises the duplication of functionality. However, it requires the nucleus to provide support for modular composition of protocols as in designs like the x-Kernel [HUTCHINSON 91].

Figure 3.2: Active replication below the nucleus



C: client group member S: Server group member

3.2 Trading on policies and mechanisms

Trading may take place on mechanisms and policies. Precisely what mechanisms and policies are used in an interaction between a client and a server will not be resolved until bind-time.

Mechanisms and policies can be configured to build different protocols and different kinds of groups (see §3.3). Since interface references contain information about protocols, there is an obligation for a server to support the protocols included in the interface reference. A trader may project this protocol information as the properties of an offer.

When a trader returns an interface reference there is no guarantee that the client will be able to interact with that service [DESCHREVEL 93]. This cannot be resolved until the client attempts to binds to the service because many failures are possible: e.g. the client may get a termination indicating the server does not support (or will not use) the protocol required by the client; it may get a termination saying the server has failed. If the client is not successful in binding to the server it may consult the trader for other service offers.

If a server makes a change to a policy or mechanism which may invalidate a binding (e.g. by changing a property), it may need to replace the offer it has exported to the trader atomically. Any clients of the server may need to rebind. Whether or not a change to a policy or mechanism invalidates a binding is subjective, so it is only possible to determine guidelines, not absolute rules.

A trader may have several offers with the same property. However, a client may be able to interact with some of these services. To ensure that it gets a service to which it can bind, the client will have to ask the trader for all matching offers and decide for itself to which server it should bind.

3.3 Client transparencies for active replication

A client group interacting with a server group may select different policies for certain mechanisms. To maximise programmer abstraction, policies should be specified by name. This requires agreement on the names of a set of standard policies. The common transparent cases will use standard policies.

More sophisticated (and non-transparent) clients may choose to provide their own policies. Certain policies will expose some aspects of the behaviour of individual group members — *relaxing the transparency.* The following

subsections described the mechanisms required by single threaded client groups for active replication and lists different policies which can be set.

3.3.1 Termination collation

When a termination is delivered to a client group, it is important that each member of the client group uses an equivalent policy for each mechanism otherwise state divergence of the client group may occur.

A client group may set different policies for different server groups for collating terminations. There are many possible policies e.g.:

- Insist all terminations are identical, wait until all have arrived and return one, otherwise return an error
- Insist that the majority are identical, return one
- Impose a time limit and insist that the majority received within this limit are identical, return one
- Return all the terminations to the client application
- Return the first termination, ignore the others

Invocations of groups may terminate due to excessive communication failures just as they can with singletons. There are also other causes for the number of terminations to vary, such as member failures. Therefore clients must specify collation policies which determine when sufficient terminations have been delivered to form a valid result. To avoid waiting indefinitely it must also be possible for client groups to determine when all responses which are likely to occur have occurred.

3.3.2 Termination quorum processing

A client group may also have different policies for different server groups for termination quorum processing, although typically the group will have the same policy for all server groups. This policy will state the number of members, *n*, which must have a termination before it can be delivered to the client application. This fixes the number of simultaneous failures a client group can tolerate and is called the **resiliency** of the group (i.e. the group can tolerate *n-1* member failures).

3.3.3 Termination distribution

Although the mechanism for distributing terminations may reside in the capsules of the server group, a client and a server may trade over which policy to use for distributing terminations to the client group. This allows client groups to interact with server groups using different protocols. Different policies can be set for distributing terminations from different server groups, some possible policies are:

- Each member of the server group sends a termination to each member of the client group; this is the policy in GEX [OSKIEWICZ 93]
- One member of the server group sends a termination to each member of the client group; this is the policy in lightweight GEX [WARNE 92]
- One member of the server groups sends a termination to one member of the client group; the receiving client sends the remaining members of the client group the termination piggybacked onto the sequencing message; this is a variant of Amoeba protocol [KAASHOEK 91]

3.3.4 Invocation distribution

A client and server may trade over which policy to use for distributing invocations to the server group, although the mechanism for invocation distribution may well reside in the capsules of the client group. The possible policies are similar to those for termination distribution.

3.4 Server transparencies for active replication

The following mechanisms are required by server groups, different policies may be set for each: invocation collation, invocation quorum processing, invocation distribution and termination distribution. Some policies will relax transparency.

A server's view of these transparencies is completely isomorphic to the client's view discussed in §3.3. Client groups send invocations to and receive terminations from server groups; server groups send terminations to and receive invocations from client groups. Additionally, server groups need an additional mechanism called invocation sequencing to ensure that members evaluate invocations in the same order (see 3.4.1). This is not an issue for single threaded client groups because the determinism inherent in a client group ensures that invocation patterns which might cause a sequencing problem for terminations cannot arise, this needs to be revisited when we know more about multi-threaded clients.

All members of a server group must use the same policies when handling an invocation, otherwise state divergence may occur. For transparent active replication all members of the group must use the same policy for a given client, otherwise clients would see different behaviours and quality of service from different members of the server group.

3.4.1 Invocation sequencing

A server group will need to ensure that invocations are evaluated in the same order at all members, this is called invocation sequencing, all members of the group must choose the same policy. Some possible policies are:

- Sequenced invocations: a total delivery order imposed at all members
- Unsequenced invocations: delivery order is arbitrary

Whether or not sequencing should be enabled will depend on whether or not invocations can change the state of the server group. If they can, sequencing should be enabled to prevent state divergence.[1]

3.5 Generic transparencies for active replication

Some mechanisms are generic to both client and server groups; any group requires the following transparencies.

1. For example, suppose invocations can be partitioned into those which change the state of the receiver (write) and those which have no side effects (read only). Write invocations need to be sequenced with all other invocations while read-only invocations need sequencing with write invocations not other read-only invocations.

3.5.1 Handling membership change

A group can have only one policy at a time for handling intentional membership changes in the group (i.e. join and leave operations). For some protocols (e.g. GEX [OSKIEWICZ 93]) membership changes need to synchronise the state of the group and so should not be concurrent with any other invocation. A policy could specify the following: a maximum or minimum number of members for the group; and how to transfer application state to a new member.

There are several possible policies for transferring application state to new members of a group, including not transferring it at all. The application state may be transferred by obtaining a snapshot of the state or by replaying a log of all the invocations and terminations delivered to the group. Both the snapshot and log may be obtained from a number of group members, so collation may be required (see also 5.4.3).

It is much harder to synchronised the log or snapshot with the state of client groups which generate activity internally. Some notion of quiescence might be helpful here: when the group is quiescent it can be reconfigured [MAGEE 87].

3.5.2 Member failure handling

A group can only have one policy at a time for handling failures. These policies may specify actions to be taken by the failure handling mechanism in addition to reforming the group. Some suitable policies are: create another member and add it to the group to replace the failed member; take no additional action. Care needs to be taken to ensure that any new members created are independent of the failed member: it may be inappropriate to instantiate a new member on the node on which a member has just failed.

3.5.3 Concurrency control

A client group may specify the degree of concurrency allowed in each member: no concurrency (single threaded); allow some fixed number of threads, scheduled deterministically.

3.6 Relaxing transparencies to build other kinds of groups

By selecting appropriate policies and mechanisms which provide more direct access to the underlying communications, applications other than transparent replica groups can be supported. For example, by disabling the transfer of application state and allowing members to discover who else is in the group “groupware” or conferencing applications can be built.

Where it is not possible to get the required behaviour by changing a policy it may be necessary to replace the mechanism with a new mechanism which conforms to the type of the old one.

3.6.1 Group templates

Group templates can be used to define the behaviour of a group. A group template consists of a service interface and a management interface, followed by a (possibly empty) sequence of mechanism and policy interface tuples. This is an extension of the concept of a binding object template defined in [ODP].

```
MechPolRecord : TYPE = RECORD [
     mechanism-interface: Mechanism_InterfaceRef,
     policy-interface : Policy_InterfaceRef];

MechPolPairs : TYPE = SEQUENCE OF MechPolRecord;

GroupTemplate : TYPE = RECORD [
     service-interface : Service_InterfaceRef,
     management-interface: Management_InterfaceRef,
     mechs-policies : MechPolPairs];
```

Instantiating a group interface involves the group factory taking a group template and a list of interfaces which are to be combined into a group. The group can then be invoked either by its service interface or its management interface. The behaviour of a group will be defined by its service type and sequence of mechanism-policy pairs. If a template does not specify a full set of mechanism-policy pairs, the infrastructure will provide default mechanisms and policies for those which are not specified.

Some mechanism-policy pairs will influence the quality of service provided by a group, e.g. the failure handling mechanism (see chapter 4).

Binding to a group involves providing a set of mechanism-policy pairs. The binder can check that these mechanisms or requirements are consistent with the mechanism-policy pairs specified by the group template. For example if the server group's template specifies a null collation mechanism then a client group binding to the group needs to provide a consistent distribution mechanism (e.g. only one member of the client group will send the invocation). Otherwise a bind failure occurs.

3.7 Outstanding issues

Identifying and specifying the interfaces of an appropriate set of mechanisms and policies to support interface groups is still a research topic: the “group run-time support” object in figure 3.1 warrants further decomposition. Ultimately the aim is to have a structured kit of parts using rules and recipes to select the appropriate mechanisms and policies to satisfy specific transparency requirements.

More work is needed to understand how to ensure consistent choices of policy and mechanism are made at the client and at the server. There may be a need for direct negotiation between clients and servers over which policies and mechanisms to use, but no such need has been identified yet.

Engineering groups involves building n-m interaction structures n>1 and m>=0: it is possible for n invocations to generate m terminations. If groups are transparent this n-m interaction structure appears as a single invocation resulting in a single termination. Relaxing transparency may involve some applications explicitly handling all invocations and terminations from groups. For example, a client may invoke a group and receive multiple terminations. It is not clear how best to introduce this into the computational model.

Much more work is needed to understand the implications of interactions between multi-threaded clients and/or group members as some of the synchronisation and recovery issues become very complex.

It is not clear how to synchronise a log or snapshot with the state of client groups which generate activity internally. Some notion of quiescence might be useful.

A better understanding of the semantics of a group template is needed. Does it fix the mechanisms and policies for all time?

4 Quality of service

This chapter looks at the implications of group management. The reason for managing groups is to ensure that they deliver a stated **quality of service**. Hence first the chapter discusses what quality of service is and its relationship to group templates. Next it looks at what aspects of quality of service are particularly important to active replica groups and why a rigorous failure model is needed. Finally there is a discussion of what factors can affect quality of service; these factors need to be considered by management operations.

4.1 Quality of service

Quality of service statements (QoS) describe non-computational attributes of a service, for example the attribute: “a service can survive up to three fail-noisy node failures”. Such statements are constraints on choices of mechanisms and policies (and hence transparencies); a four member group using the mechanisms and policies of the group execution service (GEX) [OSKIEWICZ 93] satisfies the above requirement. Quality of service statements can be considered to define types, subtyping relationships can be defined by explicitly defining a QoS hierarchy.

Group templates also define group types. They are a way of making *explicit* choices of mechanism and policies, and *implicit* QoS statements. The QoS type of a group template will be the strongest QoS statement which can be satisfied by the template.

4.2 Quality of service and replica groups

Perhaps the most important QoS attribute for replica groups is the class of failures which can be tolerated. Much of the degree of tolerance is determined by the protocol which is used, two examples are:

- the GEX protocol can tolerate fail-silent and fail-noisy behaviours [LAPRIE 85], but not Byzantine behaviour [LAMPORT 82], the resiliency of the group determines how many of these failures the group can tolerate
- the lightweight GEX protocol [WARNE 92] and most passive replication protocols can tolerate fail-silent behaviours

A replica group is unlikely to continue to satisfy its QoS specification if an unanticipated failure occurs: e.g. a byzantine failure in a group using GEX.

Dynamic reconfiguration may or may not affect the QoS of a replica group. For example adding a new member to a group and placing it on a node which already has an existing member will not increase the group’s ability to tolerate node crashes. Moving a member from one node to another may also affect the QoS.

It may not be possible for an application or an infrastructure to satisfy a quality of service statement at all times. For example, a replica group with three members can tolerate up to two failures; this could be stated in a QoS statement. If a failure occurs and a new member is not created, the QoS statement can no longer be satisfied by the application, since it can tolerate only one failure. A termination needs to be generated and propagated to client infrastructures when they attempt to invoke it. The infrastructures can take various actions such as: rebinding to another server; ignoring the termination; passing the termination to their clients.

The foregoing shows that if QoS requirements are applied to a service then QoS error terminations become possible. This shows how non-functional requirements can have functional implications.

4.3 A failure model

A failure model is a rigorous taxonomy of failure classes: it identifies a set of classes and describes the relationship between them. Examples of failure classes are “fail-silent”, “fail-stop” and “Byzantine-failure”. It is essential to have such a model before QoS statements can be made about failures. The model will allow comparison of the quality of service provided with that required.

A failure model gives a framework for relating different mechanisms which can be provided to compensate for different classes of failures. There are likely to be trade-offs in the classes of failures which can be tolerated and the (real-time) performance of services. It may be desirable to provide a QoS framework which makes this explicit to the programmer.

4.4 Managing groups to satisfy a QoS statement

Any operation which configures a group may affect the QoS provided by the group. The QoS statements which can be made about the service provided by an application depend on the following: the QoS provided by the infrastructure, that provided by third party services, that built into the application, and that provided by the environment. Configuration operations need to take these into account.

4.4.1 QoS provided by third parties

Section 3.2 describes how trading can take place on mechanisms and policies. This may be manifested as trading on QoS: when an application attempts to trade for a third party service, it is making QoS statements about the service to which it wishes to bind by declaring a service type. The binder will check that the appropriate policies and mechanisms are present to deliver the stated QoS when binding occurs.

For example, an application attempting to bind to a group built using the lightweight GEX protocol [WARNE 92] must have the policies and mechanisms used by this protocol so that it can communicate with the group.

If an application rebinds to (another) service the QoS provided by that application may change.

4.4.2 QoS provided by the underlying infrastructure

Giving the programmer the ability to state different QoS requirements on an infrastructure implies selective transparency: different choices of policy and mechanism will give a different QoS. For example, a programmer might specify the minimum number of threads an infrastructure must provide. Chapter 3 shows how transparencies can be engineered. The mechanisms and policies provided by this engineering will limit the choices which can be made when selecting transparencies. This will in turn limit the different kinds of QoS statements which can be made.

4.4.3 QoS provided by the environment

The application's environment will also affect its QoS: an application running on fault-tolerant hardware may inherently be more reliable than if it were running on non fault-tolerant hardware. Hence the environment in which each member of a group is placed (e.g. the node it is on) will affect the QoS of the group.

4.4.4 QoS built into the application itself

Programmers can build QoS into an application. For example, programmers can use a replication transparency provided by the infrastructure or build their own protocol into the application.

4.5 Outstanding issues

As noted in section 4.3, it is essential to develop a rigorous failure model so that QoS statements can be made about failures.

QoS statements may be made in very concrete terms such as: “a service can survive up to three fail-noisy node failures”. However, this does not say anything about the availability of the service in the absence of information about the availability of nodes on which it is running. It is desirable to make statements about availability such as: “this service will be available 99.9998% of the time”. The mapping between such statements and specific policies and mechanisms may be very complex. In addition extra services will be required to monitor the availability of hardware.

The types defined by group templates need to be QoS types. One possible approach to this would be to generate group templates from QoS statements using transformers [AR.004]. This also requires the architecture to provide a set of mechanisms, policies and recipes which can be used to implement a given QoS.

In addition to QoS factors which need to be considered when configuring a group. Further work is needed to understand the interaction of the QoS provided by the infrastructure, that provided by third party services, that built into the application, and that provided by the environment. This will enable tools to be designed which configure and manage groups to deliver a stated QoS.

The whole area of providing an overall QoS composed from many QoS requirements taken together is not well understood because the trade-offs are complex. A high QoS in one area may imply a diminution of QoS in some other area(s), e.g. reliability versus cost or performance.

It may be possible to derive QoS statements about an application from statements about the infrastructure, the environment and services used by the application. It is not clear how to determine the QoS engineered into an application; programmers may need to make assertions about the QoS they engineer into their own applications.

5 Engineering model requirements for groups

This chapter describes a set of requirements for an engineering model for interface groups, this is largely a communications-oriented discussion because groups place new requirements upon the communications infrastructure. For example, support for interaction with and between groups is assisted by facilities such as multi-casting and relocation. Transparency introduces other requirements such as collation and distribution while maintaining synchronisation between members requires communication about invocations received between members. The present chapter discusses the general principles of these requirements while a companion document [OSKIEWICZ 93] discusses a particular implementation.

5.1 Communications infrastructure requirements

This section discusses general communications facilities needed to support RPC interaction with groups. These are multicast communications which facilitate efficient communications with a group and relocation which masks membership changes.

5.1.1 Multicast communications

Multicasting exploits the ability of some hardware technologies such as asynchronous transfer mode (ATM) switches to enable multiple recipients to be addressed by a single network address. This enables senders to communicate with groups with minimal overhead to the sending system or the network and is especially beneficial for intra-group communications (see 5.3).

Note however that multicasting only optimises transmission to a group, if each member generates its own reply these will arrive individually. Also multicasting is best where hosts are equipped with intelligent network interfaces, otherwise if any sort of broadcasting is used the interrupt overhead on other hosts may be a problem.

Unfortunately there are drawbacks in trying to use multicast protocols as an RPC substrate. It can be difficult to decide which member needs a retransmission in the presence of lost messages, a similar problem arises when attempting to utilize a fragmentation protocol to hide network packet size limitations. Recovery strategies under these circumstances are either expensive (e.g. retransmit to everyone) or involve switching to a second interaction protocol to do point-to-point retransmission to individuals. This is still an area which is undergoing investigation.

Although potentially more efficient, multicasting is not mandatory as it can be simulated by point-to-point RPC, where this is the case then [WARNE 92] may be particularly useful as it attempts to minimise network traffic.

5.1.2 Relocation

Fortunately it is not necessary to introduce special mechanisms to notify clients of changes in the membership of a server group. Relocation [AR.007] is a service which enables clients to refresh interface references which have become out of date (e.g. due to migration or group population changes). This relieves servers of the (impossible) burden of remembering and notifying all existing clients whenever changes occur.

Relocation is initiated when a client receives some form of error termination which provides a hint that an invocation failed because the interface reference may no longer be valid. Following successful relocation the invocation may be re-attempted. A more efficient approach may be to have the group notify the client directly thus bypassing the relocation process so long as some members are reachable.

The relocation service (described in [AR.007]) is provided by a chain of relocators, each of which has a wider (i.e. less localised) scope of knowledge than it's predecessors, this knowledge must be provided by the capsule(s) supporting the relocated interface(s). In order to relocate, a relocator is given the existing interface reference and asked to return the most recent version it knows about. If there is no change then the next relocator is consulted until one of them returns a newer interface reference for the relocated service or until all have been consulted.

Clearly, some relocators need fixed locations (e.g. co-located with the trader) and unless they are themselves replicated may constitute a potential single point of failure. Note however that as relocators are only consulted when there has been some change or failure, they are not necessary to the operation of a group with a static population of non-failed members. The means by which routine relocation after population change may be facilitated is described in 5.2.1.

5.2 Inter-group communications

Under normal working, groups will invoke operations upon each other, either to request a service or to effect population changes, e.g. to add/remove members. Transparency requires that clients need not be aware of the group's population and members can be ignorant of the consequences of their membership. This requires that the members are addressable in a transparent fashion and that requests and replies are subject to distribution and collation.

5.2.1 Group references

Group references enable client applications to have a uniform invocation style for groups and singletons (i.e. have computational transparency), this is done by masking all underlying network addressing. For this to occur transparently the group reference must change whenever the population of the group changes so that new members can be addressed and ex-members are no longer addressed, i.e. the client always addresses the current group membership.

One way to facilitate this is to incorporate an incarnation id within the group reference, this is maintained by the group and may be realized as a counter incremented whenever the population changes. Clients always cite the incarnation number from their group reference so that it may be compared against the current incarnation number. A mismatch indicates that a

membership change has occurred after the client was bound to the group and a termination is returned to the client causing it to relocate. Upon acquiring the new group reference, the client has the opportunity to decide whether to continue depending on whether the new group still satisfies it's original QoS requirements (this may be done by the client's infrastructure to preserve transparency).

5.2.2 Distribution and collation

Distribution masks whether a client is invoking (or a server replying to) a group by causing a request or reply to be sent on each distinct address within the group. Because of this, each address may be on a different network or use a different protocol, this enables the simultaneous use of multi-cast and serial point-to-point transmission methods. Note that the two types of distribution are implemented quite differently, client distribution is based on interpreting of the contents of an interface reference while server distribution is based on replying to invokers.

Collation masks whether a server is invoked by (or a client replied to) a group by reducing multiple requests/replies from group members to a single request or reply (see 2.3 and 3.3).

Collation and distribution are ubiquitous because all capsules in the system must be prepared to be invoked by (and respond to) groups.

5.2.3 Invoker identifiers

There is a need for unambiguous identifiers so that clients of a group can be distinguished from each other and so that they may be referred to when group members need to exchange information about invocations received.

These invoker identifiers should be sufficiently unique so that all simultaneously active clients (whether group or singleton) of a group can be distinguished from each other. To permit client groups, it is necessary that all members use the same invoker ID which must contain the client's cardinality, this also provides the basis by which request collation can associate all the requests from a client group with each other. This is still an unresolved issue though there are links with the work in [AR.004].

5.3 Intra-group communications

The purpose of intra-group communications is to enable the members of a group to communicate with each other to facilitate their operation as a group, e.g. to enable the group to synchronise itself and to carry out other management activities such as reconfiguration — these operations are invoked on an internal group management interface.

5.3.1 Quorum and sequencing

In the presence of unreliable communications it may be necessary to prevent some members processing messages which other members have not received to stop them getting potentially out-of-step with the rest of the group or perhaps confusing their client. One way to prevent this is to insist that a certain number of group members possess the message before acting on it, such a message is said to be quorate.

Note that it may not be always necessary for members to execute operations in lock step, what is important is they execute it in the same order. Even this requirement may be relaxed if it is known that some operations have no side effects (e.g. are read only).

5.3.2 Recovery of lost messages

A member detecting a missed message will try to recover by getting a copy from a member which has it. This requires the use of some 'clock rewinding' technique such as a history list of requests, some possible techniques are described in 5.4.3.

5.4 Population change considerations

The construction of group introduces no new concerns into the computational model. However there are engineering model issues which need resolving.

5.4.1 Making a new group

There are many approaches to this, [OSKIEWICZ 93] describes the approach used to fit within the limitations of ANSAware. This section describes a more ideal approach.

A new group requires initialising with management information such as resilience policies, it also requires a suitable management infrastructure to be nominated or created.

The initial step is the creation of a management interface to the new group. This can be exported to the trader and used by intending members to add themselves to the group. After the first member has joined, the group service reference can be exported so that clients can import the group.

5.4.2 Making a new member

Each member is actually a member of three groups and is associated with three references: one for the service itself; one for internal management invocations; and one for external management invocations. As the service interface will already exist, a wrapper must be deployed around it to intercept invocations sent to the group so that they may be subject to group vetting processes (e.g. quorum determination) before evaluation. The two management interfaces are created at join time.

Joining the group involves acquiring the external management reference, e.g. from the trader and invoking its `addMember` operation naming the interface to be added[1]. The result of successful addition is a copy of the group management and service references.

If accepted as a member the new member must first establish its initial state. In active replica groups, this must be done before the member evaluates invocations as a member of the group as it is required that all members be deterministically equal, i.e. they behave identically in response to a given invocation. For non-trivial applications this usually requires that the members contain identical internal state reflecting identical execution histories. This requires state synchronisation.

1. The major difference between this discussion and the ANSAware implementation is that in ANSAware `addMember` is provided in the service interface.

5.4.3 State synchronisation techniques

State synchronisation ensures that all externally visible parts of a member's state are the same as all other members of the group. This enables a new active replica member to appear to have an identical execution history to all the other members. State synchronisation is also useful for other types of group such as passive replica groups.

Two common methods for achieving this are state transfer and message replay, i.e. transferring the checkpointed state from some up-to-date member or simply sending it the invocations which would have caused the state change, the message list is also known as a log. Note that the two methods are not identical because replaying invocations from a log may involve interaction with third parties.

In both cases it must be possible to synchronise state with messages: the log and snapshot must be consistent with the internal state of existing members when a new member joins. For server groups the join operation can be scheduled as a normal group operation, synchronising the snapshot and log with the state of each member. The sequencer must ensure that no other operations are running when the join operation is running [OSKIEWICZ 93].

If the log is not to be a single point of failure, and is to be up-to-date, an active replication scheme should be used to organise it. Thus members of a passive replica group may form a subset of themselves into an active replica group for the purpose of logging and recovery. Input messages being processed only by the primary replica.

5.4.4 Controlled membership changes

Controlled membership change requires that the change be seen at all members so that all share a common view of the population. This requires interaction between members identifying the candidate or departee and choosing when to implement the change, e.g. between invocations. The incarnation number is updated to let extant clients update their group reference. Also the group relocator is notified of the new membership.

Special action needs to be taken if the membership rises to one or shrinks to zero. The first member to join a group may need to consult an archive to establish its initial state, other new members will need to establish synchronised internal state. After the last member has left the group, it will not be possible to invoke the service interface, but it is possible to invoke `addMember` on the external management interface.

5.4.5 Recovery from member or communications failures

Failures can result in apparent uncontrolled membership decrements. These may be caused by a member crash, so that it no longer participates in the sequencing process, or by a network crash so that some connectivity is lost. The recovery strategy is quite complex. Firstly, the remaining members of the group must establish what the optimal surviving configuration is, ensure that all remaining members are synchronised (i.e. they share a common view of the population and have identical evaluation histories) and then behave as though a controlled change had taken place. [OSKIEWICZ 93] contains one possible strategy.

If network partition occurs and the group breaks into two or more subgroups then things can get very difficult indeed. The remaining survivors may each

assert they are the reformed group which requires arbitration. The problem continues after the partition is removed because the separate subgroups may need to be resynchronised, this is still subject to investigation

5.5 Outstanding issues

Efficient exploitation of multicast based RPC still needs further work.

Lightweight approaches to GEX [WARNE 92] could exploit the ability to forward replies to members of a client group, even though some of those members may not have issued the original invocation.

Currently we can not have multi-threaded active replica group members because of the need to preserve determinism. This is because we cannot guarantee that the parallel threads within each member would generate side effects in the same order. To fix this we need some predicates to determine whether the operations affect common state, clearly if there is no overlap on state then they are logically separate and could run in parallel. There are links to the atomicity tasks here [AR.004] which need to explored.

In many cases, request or reply collation criteria will be application specific and need to be performed after unmarshalling, this could still be automatically generated.

It is not known how to generate invoker IDs. In particular, what names to use for multi-threaded clients where distinct but parallel invocations on a server group need to be distinguished. Again there are links here with [AR.004].

Checking type conformance is a major outstanding problem, this is necessary to provide guarantees about the members of a group. The computational model allows for signatures to be checked (or rather it will ensure that if I invoke `addMember` on a group that I must be have the right signature), however there is no provision for the semantics of the candidate member to be checked. In ANSAware it is all down to programmer discipline as any random interface can invoke `addMember` on any group without any checking being possible.

A potential problem exists with groups which shrink to zero members and then grow in size again as the group management state would evaporate. In the ANSAware implementation this is overcome by storing these parameters in the group relocator. This is not a good general solution.

Network partition causes problems with the possibility of independent sub-groups forming, this potential divergence would inhibit eventual reformation after the partition is lifted. This is still a research area.

Is there is a fundamental difference between a member and a non-member or are they both interfaces? Is there a fundamental difference between a group interface and a non-group reference?. If not can a group become a member of another group?

6 Summary and conclusions

6.1 Summary

An interface group is a collection of one or more interfaces which are accessible externally through a single service interface. If the interaction with a (server or client) group is indistinguishable from an interaction with a singleton, the group is said to be transparent. Non-transparent interaction is also possible.

A transparency is implemented by an orchestrated set of mechanisms. These mechanisms provide services through interfaces. Policy objects partially define the behaviour of these mechanisms; different policy objects can define different behaviours. A number of mechanisms are necessary to support transparent group interaction: distribution, collation, sequencing and quorum processing.

Groups need to be managed so that they deliver a stated quality of service. The most important aspect of quality of service for active replica groups is the class of failures which they can tolerate. A rigorous failure model is needed to compare different kinds of groups delivering different quality of service guarantees.

Implementing groups raises a number of issues for the communication infrastructure. Multicast communication is useful, but error handling is not straightforward. As well as inter-group communications it is necessary to provide a high degree of intra-group interaction so that the group may manage itself. Handling population changes transparently requires a relocation service so that a distributor can obtain an up to date view of the membership. It is necessary to synchronise new members when they join a group.

6.2 Conclusions

It is possible to build support for groups above an RPC based infrastructure, this may be made transparent to application programmers. However, a more efficient implementation would be to build the group protocols independently, taking advantage of a nucleus which provides support for modular composition of protocols. Further work is needed to understand how to structure group protocols as a set of mechanisms and policies.

Interface groups can provide fault-tolerance transparently through active replication. In addition groups can provide a powerful abstraction for building decentralised applications such as bulletin boards [BIRMAN 86] and conferencing applications. Building decentralised applications requires selective transparency: the full functionality of an infrastructure supporting active replication is not required (e.g. state synchronisation). Further work is needed to understand how to configure a consistent set of mechanisms and policies to satisfy specific transparency requirements.

Groups can be managed to deliver a stated quality of service, but further investigation is needed to development a rigorous failure model and

understand how to configure the infrastructure to deliver a given QoS. Satisfying abstract QoS statements may require the introduction of services to monitor the availability of hardware.

Declarative QoS statements can be made and mapped onto available protocols by tools (e.g. transformers). This requires a type system over the QoS attributes of interest. In addition the architecture must provide a set of mechanisms, policies and recipes which can be used to implement a given QoS. Further study is needed to understand this.

A prototype has been built [OSKIEWICZ 93] but many areas require further investigation, in particular: exploiting multicast technology efficiently, dealing with network partition and coping with multi-threaded clients and servers.

References

[ANSA 91]

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

[AR.001]

The ANSA Computational Model, AR.001, APM Ltd., Cambridge U.K., February 1993.

[AR.004]

ANSA Atomic Activity Model and Infrastructure, AR004, APM Ltd., Cambridge U.K., February 1992.

[AR.007]

The ANSA Interface Reference: An Engineering Model, AR.007, APM Ltd., Cambridge U.K., February 1993.

[BIRMAN 85]

Birman, K., "Replication and Fault-tolerance in the ISIS system", *10th Symposium on Operating Systems Principles*, **19** (5), 79-86 (December 1985).

[BIRMAN 86]

Birman, K., Joseph T. A., Schmuck, F., Stephenson, P., "Programming with Shared Bulletin Boards in Asynchronous Distributed Systems", *TR 86-772*, Department of Computer Science, Cornell University, New York (1986).

[BIRMAN 87]

Birman, K., "Exploiting Virtual Synchrony in Distributed Systems", *ACM Operating Systems Review*, **21** (5), 123-138, (November 1987).

[CHANG 84]

Chang, J. & Maxemchuk, N., F., "Reliable Broadcast Protocols", *ACM Transactions on Computer Systems*, **2** (3), 251-273 (August 1984).

[CHERITON 84]

Cheriton, D., "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, **1** (2), 19-43 (April 1984).

[CHERITON 85]

Cheriton, D., "Distributed Process Groups in the V Kernel", *ACM Transactions on Computer Systems*, **3** (2), 77-107 (May 1985).

[COOPER 85]

Cooper, E., "Replicated Distributed Programs", *ACM Operating Systems Review*, **19** (5), 63-78 (December 1985).

[CORBA 92]

"The Common Object Request Broker: Architecture and Specification", Object Management Group, Inc, Framingham, MA, USA, and X/Open Company Ltd., Reading, UK, 1992

[CRAFT 83]

Craft, D. H., "Resource Management in a Decentralized System", *ACM Operating Systems Review*, **17** (5), 11-19 (October 1983).

[DOLEV 87]

Dolev, D., Lamport, L., Pease, M., & Shostak, R., "The Byzantine Generals", *Concurrency Control and Reliability in Distributed Systems*, 348-369, Van Nostrand Reinhold (1987).

[DESCHREVEL 93]

Deschrevel, J.-P., , "The ANSA Model for Trading and Federation", AR005, APM Ltd., Cambridge England, February 1993.

[HUGHES 88]

Hughes. L., "A Multicast Interface for UNIX 4.3", *Software Practice and Experience*, **18** (1), 15-27 (January 1988).

[HUTCHINSON 91]

Hutchinson, N.C., Peterson, L.L., "An Architecture for Implementing Network Protocols", IEEE Transactions on Software Engineering, **17** (1), 64-76 (January 1991).

[KAASHOEK 91]

Kaashoek, M.F. and Tanenbaum, A.S., "Group Communication in the Amoeba Distributed Operating System",in Proc 11th International Conference on Distributed Computing Systems, Arlington, Texas, USA, May 20-24, 1991, p222-230.

[LAMPORT 78]

Lamport, L., "Time, clocks, and the ordering of events in a distributed system", *Communications of the ACM*, **21** (7) 558-565 (July 1985).

[LAMPORT 82]

Lamport, L., Shostak, R., Pease, M., "The Byzantine generals problem", *ACM Transactions on Programming Languages*, **4** (3) 382-401 (July 1982).

[LAPRIE 85]

Laprie, J., "Dependable Computing and Fault Tolerance Concepts and Terminology", *Proceedings 15th Annual International Symposium on Fault Tolerant Computing*, Ann Arbor, Michigan, USA, 2-11 (June 1985).

[LIANG 90]

Liang, L., et al, "Process Groups and Group Communications", *IEEE Computer*, **23** (2), 56-66 (February 1990).

[MAGEE 87]

Magge, J., Kramer, J., Sloman, M., "Constructing Distributed Systems in Conic", *Imperial College Research Report*, Deparment of Computing, London, U.K., March 1987.

[HENNESSY 85]

Hennessy, M., Milner, R., “Algebraic Laws for Nondeterminism and Concurrency”, *Journal of the ACM*, **32** (1) 137-161 (Jan 1985).

[MULLENDER 89]

Mullender, S., ed. *Distributed Systems*, ACM Press, New York (1990).

[ODP]

Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing - Part3: Prescriptive Model, ISO/IEC JTC/SC21/WG7.

[OSKIEWICZ 93]

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

[PETERSON 89]

Peterson, L., L., Bucholz, N., C., & Schlichting, R., D., “Preserving and Using Context Information in Interprocess Communication”, *ACM Transactions on Computer Systems*, **7** (3), 217-246 (August 1989)

[POWELL 91]

Powell, D., (editor) “Delta-4 A Generic Architecture for Dependable Distributed Computing”, *Springer-Verlag, 1991.*

[SCHLICTING 83]

Schlichting, R. D. & Schneider, F. B., “Fail stop processors: an approach to designing fault tolerant computing systems”, *ACM Transactions on Computer Systems*, **1** (3), 222-238, (August 1983).

[SCHMUCK 88]

Schmuck, F., B., “The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems”, *TR 88-928*, Department of Computer Science, Cornell University, New York (1988).

[SHIMIZU 88]

Shimizu, K., et al, “Hierarchical Object Groups in Distributed Systems”, *Proceedings of the 8th International Conference on Distributed Computing Systems*, IEEE, 18-24, (1988)

[WARNE 92]

Warne, J., Oskiewicz, E., Edwards, N., “A Lightweight Group Execution Protocol”, *RC.439, APM Ltd., Cambridge, U.K., October, 1992.*