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

MED: A CORBA-based Management Engine for Dependability

Nigel Edwards, Ed Oskiewicz

Abstract

The basic technology for open distributed processing is now understood. The challenge now is to manage open distributed systems to deliver the right set of non function guarantees such as reliability. Until this challenge is met the deployment of open distributed processing technology will be slow.

This paper shows how to integrate the management of a service's dependability with other aspects of service management including naming and trading. The intention is that service management can be specialized by building further services on top of MED.

The underlying paradigm presented to the programmer is that each service has associated with it a management service. Clients use the management service during trading to find the right service instance and can appeal to the management should a service appear to be faulty.

APM.1203.00.03 **Draft** 25 May 1994

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

Distribution:
Supersedes:
Superseded by:

1 MED: A CORBA-based Management Engine for Dependability

1.1 Overview

The basic technology for open distributed processing is now understood. The challenge now is to manage open distributed systems to deliver the appropriate non-functional guarantees (e.g. reliability, availability and performance), and to be able to integrate existing services and systems into this world of open dependable distributed computing. It is important to realize that there will not be one set of non-functional guarantees which are appropriate to all applications; any solution must allow the selection of guarantees to match different application requirements. Until this challenge is met, open distributed computing will not be used in business critical applications [EDWARDS 94c].

The design proposed in this paper is intended to test the hypothesis that the precise nature of consistency requirements in dependable applications will be application specific [CHERITON 93], [OSKIEWICZ 94]. However, it is possible to factor out several generic aspects of service management which are common to many applications, including management of available copies of services.

This paper shows how to integrate the management of a service's dependability with other aspects of service management including naming and trading. The intention is that service management can be specialized by building further services on top of MED.

The underlying paradigm presented to the programmer is that each service has associated with it a management service. Clients use the management service during trading to find the right service instance and can appeal to the management should a service appear to be faulty.

The intended audience for this paper is system engineers and designers interested in building open dependable distributed systems.

This paper is structures as follows:

- The remainder of this section presents the motivation and aims of the work, setting out the underlying assumptions and terminology
- §1.2 introduces the various components of MED: these are further explained in the following sections
- §1.6 describes an example of MED in use
- §1.7 explains how this work relates to that of others
- Finally §1.8 presents some conclusions, some architectural consequences and some thoughts on further work

1.1.1 Caveat emptor

The ideas presented in this paper have not yet been implemented or tested; the aim of this paper is to document some early design ideas and to stimulate some discussion. Although “CORBA” [OMG 91] appears in the title, it is intended that MED could be built on any distributed object oriented platform supporting RPC and threads.

1.1.2 The technical objectives

This design presented in this paper was developed in the context of DIPS: a distributed hypertext system in which access to information objects is chargeable [OSKIEWICZ 94]. In this system the services need to be replicated for availability. To preserve resources, objects which are not in use may passivate themselves. To maintain availability of a service, the failure of service instances needs to be detected and failed instances replaced. Failure detection should not depend on the service being in use. Otherwise the failure of services subject to bursty and varying load characteristics could go unnoticed during times of light or no loading with disastrous consequences when the load starts to rise.

A protocol which relies on heart beat or keep alive messages circulating between group members is not suitable for this kind of system: objects would always be busy and never passivate themselves. In addition in a large system such heart-beat messages can impose a significant loading.

The failure of a persistent store will compromise the availability of a service: all passive objects it contains fail too. So a mechanism is needed which can detect the failure of a persistent store, map this on to failures of passive service instances, and instantiate new service instances on new nodes if the availability has dropped below a certain pre-determined level.This also requires a service which keeps track of the available nodes.

The same mechanisms should also be able to deal with the failure of active services.

This paper presents a set of services, collectively called a Management Engine for Dependability (MED) which addresses these problems. The functions of these services are:

- To keep track of the available nodes on which services can be instantiated
- To detect a service instant failure (capsule level — see §1.1.3)
- To detect the failure of a node and map this onto failures of service instances on that node (both active and passivated instances)
- To detect the failure of a persistent store and map this on failures of passive service instances
- To instantiate a new service instant on an appropriate node, if one is needed to maintain availability after a failure
- To instantiate new service instances on requests
- To keep track of available service instances (a membership service)
- To provide clients with a reference to a suitable instance (trading and naming)
- To arbitrate in the event of client reporting a service instance to have failed.

MED is not a replication protocol: it makes no attempt to enforce consistency constraints between service members; it is up to the service instances themselves to implement such consistency constraints. The system which MED is designed to support (DIPS) has many different replicated services, each with different consistency constraints. MED makes no assumptions about the nature of the communication between clients and servers (i.e. one-one versus many-many) — this is a matter which is application specific.

One possible alternative solution which was considered is based on an underlying replicated persistent store. Services which are not being used (i.e. only passing heart-beat messages between themselves) could passivate themselves. The persistent stores would run pass heart-beat messages between themselves.

There are several disadvantages to this approach which reflect the underlying problem of coarse grained management.

- The availability of the service cannot be varied: it is restricted to that of the replicated persistent store.
- If an individual persistent store fails and a new one must be added to maintain availability, the entire contents of the store must be cloned, even though this may not be appropriate for all applications, because their availability is still adequate.
- There will be more network traffic: active service instances and persistent stores will be passing heart-beat messages between each other.

1.1.3 Terminology

Although the design discussed in this paper is intended for implementation on a CORBA based platform, CORBA [OMG 91] lacks the vocabulary needed to discussed the problems associated with dependability. Hence this paper uses the ODP [ODP 93] terms of Node, Capsule, Object and Interface.

“A node is a configuration of objects forming a single unit for the purposes of location in space, and which embodies a set of processing, storage and communication functions” [ODP 93]. An example of a node is a workstation; it is assumed that at least some of the nodes in a system have a persistent object store.

A capsule is a unit of encapsulation, in particular it is a unit of failure. A capsule may contain one or more objects. An example of a capsule is a virtual machine (or process).

An interface is the unit of service provision. Under the assumption that an object has only one interface, ODP and CORBA objects are the same thing. In this paper it is assumed that objects have both a management interface, through which the object is managed, and a service interface through which “service” is delivered to clients. This enables the separation of the concerns: many different service types may have the same type of management interface.

1.1.4 Assumptions

This paper assumes bounded communication delays. It is hoped that modern networking technology like ATM will be able to provide statistically bounded communication delay. The work of the ANSA performance group will show how to build a binding between objects with bounded communication delays

and message delivery (in particular timed RPC) [WAI 94]. The precise bounds on communication delay is something which cannot be determined until the binding is made; calculating it will require knowledge of the binding structure. Such assumptions are often referred to as the synchronous model of distributed computation [SCHNEIDER 93] (as opposed to asynchronous).

This paper assumes no partitions. This means the network needs to offer multiple routes between objects. This could be implemented by redundant networks (e.g. dual LANS) or multiple point to point links with intelligent routing.

It is assumed that there are two levels of service. One in which a single message is sent out; this can fail. A second in which the message is sent by all known routes, duplicates are suppressed, but the message is guaranteed to arrive with some known bounds (greater than the first method). This is very similar to the assumption made by Cristian for his FAA work [CRISTIAN 91].

These assumptions do not preclude an object disconnecting itself from the network. It just means that this should be regarded as a failure of the object, rather than a failure of communications.

The advantages and disadvantages of this set of assumptions compared with alternative sets of assumptions are discussed in [EDWARDS 94b] and §1.8.

1.2 The levels of management

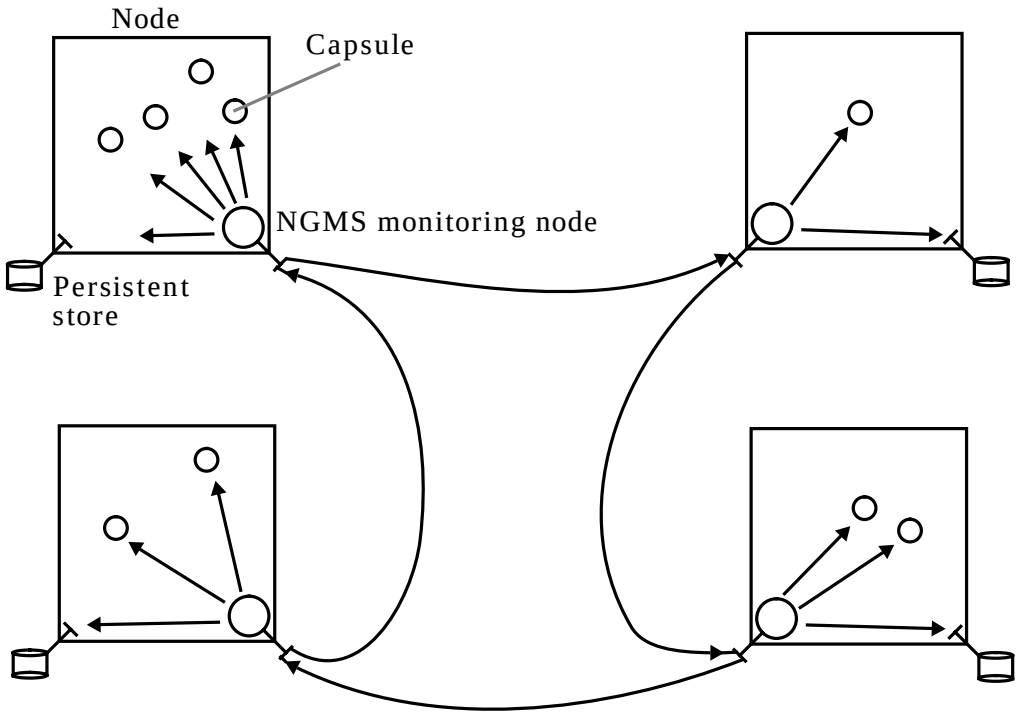
This section summarises the function of each level of management. The following sections describe the architecture of each level in more detail. There are three levels of management:

1. A Node Group Membership Service (NGMS) keeps track of which nodes are available and which services are on those nodes. It informs the appropriate service manager(s) if it detects the failure of a service, or the failure of a node. Other services (service managers) use this service to find out which nodes are the most appropriate on which to establish new instances of a service.
2. Service Managers (SMs) manage instances of services. Service instances are collected together into a number of non-intersecting groups, but no assumptions are made about the consistency constraints between members of the group. A service manager is responsible for: establishing and maintaining the membership; arbitrating in the event of a failure report; and instantiating new members of a group. The group members themselves are responsible for maintaining any state consistency guarantees (e.g. by running a protocol which propagates any necessary state changes).
3. Management of service managers and Node group membership services is the third level of management. These services are self managing they are responsible for maintaining their availability and detecting and tolerating their own failure.

1.3 A Node Group Membership Service

Figure 1.1 shows a system comprising of four nodes. Each node is running an instance of the NGMS. The NGMSs run a membership protocol to agree on who is present i.e. which nodes are available and in the system. The protocol

Figure 1.1: Node Group Membership Service



also has to cope with reconfigurations caused by nodes failing and new nodes joining the system. A suitable protocol is the Attendance List protocol described by Cristian in [CRISTIAN 91] which guarantees bounded processor failure detection and join delays in the present of any number of concurrent join and failure events.

This protocol makes assumptions consistent with the synchronous assumptions described in §1.1.4. If this assumption is not valid, there are protocols which can determine membership in an asynchronous environment (see §1.8), one possible piece of future work would be to investigate deploying NGMS using a different protocol. The following describes the behaviour of the NGMS independently of the membership protocol which gives at least some confidence that it could use a different membership protocol which makes the asynchronous assumptions.

Note: Do we need to add a description of the protocol in appendix. Cristian's protocol also has the advantage that it is much easier to implement than most (all) of the asynchronous alternatives: important if we want to build a prototype quickly!

Each NGMS allows clients to register interest in services which reside on its node, much like the notification service in ANSAware [ANSA 92a]. In principle any arbitrary client of a service instance could register an interest with the NGMS. However, this is not the way the NGMS is intended to be used. The intention is that its clients should be the Service Managers (SMs) responsible for managing that service instance. Ordinary (application-level) clients would not be aware of the NGMS; if they detect a possible failure they complain to the appropriate SM (see §1.4).

Once an SM has registered an interest in a service the NGMS works out which capsule or persistent store the service is in. If that capsule or store fails, the NGMS will notify the registered SMs of all services in the capsule or store that their service has failed. How an NGMS detects a failure is dependent on the

facilities provided by the underlying operating system. In Unix it could associate processes with capsules and persistent stores, when it is run it could check that these processes are still around. Ideally, the kernel would run the NGMS whenever the process table changed, this would make failure detection as fast as possible. Unfortunately, this ideal situation is probably not possible in most current releases of Unix (assertion needs checking). An alternative strategy is to use pipes: if the NGMS leaves a pipe open to each process it is interested in, it would get an interrupt when the process dies.

Once an NGMS has notified all interested parties of a service failure, it must purge the registration information from itself and other NGMSs in its group.

If the service migrates, passivates or activates the NGMS needs to be informed of this. The details of this protocol need to be worked out (transactions?). In particular there is a danger that the NGMS could be notified of a move and a failure occur before the move takes place; or a failure could occur after a move, but before the NGMS is notified.

A node failure means that all services residing on that node will have failed. Unfortunately this also includes the NGMS on that node! When this happens the other NGMSs in the system need to notify all registered SMs of all services which resided on the failed node. To avoid repeated notifications, the NGMS instances are ranked; only the highest ranking NGMS notifies the registered parties of service failures, purging the registrations from itself and the other NGMS instances left when it does so. Should the NGMS fail before it has completed this, the next highest ranking NGMS will restart: repeated notifications will not damage the SM service.

This means that the NGMSs will need to propagate their registration information to all the other NGMSs in the system. This can be done by piggybacking the membership information onto the messages sent between NGMS when they run their membership agreement algorithm. When an NGMS receives a registration request it could initiate the agreement algorithm and in so doing, propagate the registration request to all other NGMSs. Once the agreement algorithm had terminated it would send a reply back to the SM acknowledging successful registration.

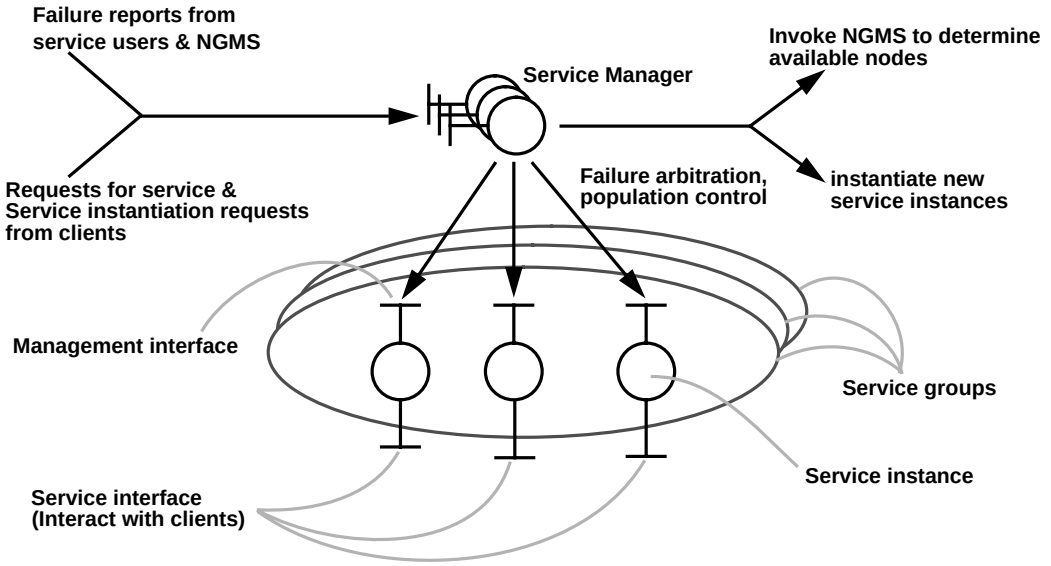
One possible approach of scaling the NGMS to manage large systems is to divide a system up into groups of say three or four nodes. Each groups would be managed by a different NGMS group. There is nothing to stop a service manager using NGMS in different groups: the grouping of service managers is independent of the grouping of NGMSs.

The managing of NGMS groups within a federation is postponed for further investigation. Although some initial thoughts on the problem are given in §1.5.

1.4 Service Managers (SMs)

Figure 1.2 shows an SM, managing a service consisting of three instances which would be sited on different nodes, possibly in different groups. The service instances managed by a particular SM form a number of non-intersecting groups. Each service instance in a particular group is of the same type. However, the service manager imposes no requirement for any consistency between the members, in particular they are not required to be replicas.

Figure 1.2: Service Manager

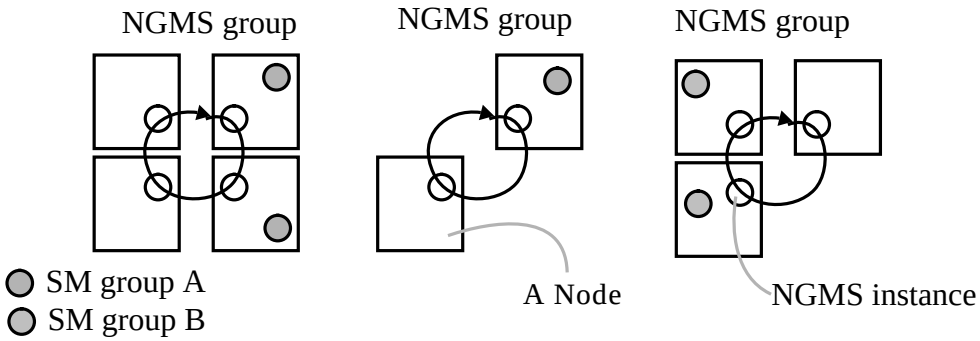


An SM is responsible for:

- Responding to a client's request for service with suitable service instances or groups
- Instantiating instances of a service when requested to do so
- Determining the membership of groups and arbitrating in the event of failure reports from clients.
- Instantiating instances of a service to maintain availability when failures occur
- In service upgrades
- Naming and location of collections

Figure 1.3 shows a possible relationship between three NGMS groups and two SM groups. Note that the topology of the SM groups is independent of the topology of the NGMS groups.

Figure 1.3: Node Group Membership and Service Management Services



1.4.1 Responding to requests for service

A client may go to the trader to find the SM responsible for the particular kind of service it wants to use. It then asks the service manager for an instance of

the service. In effect it is engaging in more trading with the service manager. The service manager will try to provide it with the service which best fits the client's requirements. (A constraint language, like the one used for the ANSAware trader [ANSA 92b], could be used to describe the client's requirements). If there are no service instances available which will satisfy the client's request, the SM will return a "failed" status.

If the SM finds a matching service, it needs to exercise some choice over whether or not it returns a single service instance or an entire group. For example, it may return a single service instance, if the service is immutable. If the service is mutable, it may return the whole group. So in the terminology of [OSKIEWICZ 93], in the first case the client would receive a singleton interface, whilst in the latter it would receive an interface group. Whether a singleton or an interface group is returned is an attribute of the service type. These ideas are explored in more detail in [REES 94].

Note: We need to think about how this would work with passive replication, in particular how does the client know which is the active member? If we expect clients to multicast to all members of a passive replica group, then the passive members merely ignore the invocation, and which is the passive member is private to the group. Otherwise either the group itself or the SM needs to nominate the active member. If the SM nominates the active member, either it participates in some of the internal reconfiguration protocols of the group (something which the design is trying to avoid), or it nominates the member based on some other criterion (e.g. using a constraint language description of the service). The alternative seems to be to have the active member invoke the SM, to tell it to return its interface in response to an invocation.

1.4.2 Requests to instantiate a service

A client may also explicitly request an SM to instantiate a service. A constraint language needs to be provided to allow the client to constrain its requests and also to specify certain QoS aspects (e.g. minimum population or performance requirements). The SM will use the NGMS service to determine which nodes are available and instantiates service instances on suitable nodes to satisfy the client's requests. Once the SM has instantiated the service it will return the interface to the client. Again whether or not it returns an interface group or a singleton is a specific attribute of that service.

1.4.3 Membership and failure arbitration

One of the underlying paradigms of MED is that each service has associated with it a management interface to which clients of that service can report failures. An implementation may require the client to remember this association; an alternative is to embed the management interface into the service interface. The latter approach would make it easier to automate the invocation of the management interface on detection of a failure, rather than relying upon the application programmer to do it.

If the SM receives a failure report, it responsible for deciding whether to believe that report or not. If the failure report is from the NGMS there is no point in conducting any further arbitration. However, if the failure report is from a client, the SM may elect to invoke an audit routine on the suspected server, if the service provides one. This means that clients in another federation have a service to which they can appeal to resolve failures, and the services themselves are protected from clients over whom their organisation has no control.

If the results of invoking the audit routine indicate that the service is correct, the SM may take no further action. However, if the results indicate that the service has failed, the SM excludes the service from the group and destroys the service instance. Initially it may try the gentle approach of invoking the destroy operation provided by the service itself. If this fails, more violence might be required: the NGMS asks the host operating system to kill the service (e.g. “kill -9”).

Note: This deals with the problem of a very badly behaved service, but what if the node itself exhibits similar bad behaviour? There is scope for investigating how the NGMS (and perhaps other services) could collaborate (with the network) to isolate a failed node, some relevant ideas are presented in [COAN 94] and [EDWARDS 94b].

The SM may receive repeated failure reports, when a single service instance fails. This can arise if the service has multiple clients, or if more than one NGMS failure occurs (see §1.3).

In the same way as an NGMS allows clients to register interests in services for which it is responsible, the SM allows clients to register an interest in a service group. This means that the SM will inform clients of any change in the population of the groups in which they have registered an interest. This allows clients to monitor the health of services on which they depend and also ensures that they can maintain their binding to the current group membership.

Group members can register an interest in their own group. This is useful if they need to be aware of the membership. For example, they may need to run a replication protocol amongst themselves. The group itself would be responsible for ensuring that the membership change notification satisfied any necessary ordering constraints (see [OSKIEWICZ 93] or [BIRMAN 87] for discussion of suitable constraints). Recently, our work has suggested there are many kinds of replication which do not require this [OSKIEWICZ 94], so it should be optional rather than something the group members are forced to deal with.

1.4.4 Maintaining availability by instantiating new instances

If a client detects that the population has dropped below some minimum level, then they could request the SM to instantiate more instances into the group. This is an alternative to negotiating a QoS agreement which would require the SM to maintain the population at some minimum level. The latter approach offers better encapsulation, whilst the former approach offers the potential for more dynamic management.

Note: I have doubts as to whether the former approach is consistent with the architectural principles. Perhaps it would be better to say it offers the client a chance to renegotiate the QoS with the SM.

1.4.5 In service upgrades

SMs need to support in service upgrades. So far only very simple cases have been identified, see [OSKIEWICZ 94] and also §1.6. These simple cases can be satisfied by **`withdraw`** and **`replace`** operations.

The **`withdraw(oldservice)`** operation makes the SM to stop providing the service interface in response to requests for service. However, it will still continue to manage the service (e.g. failure arbitration and determination of membership).

The **`replace(newservice, oldservice)`** operation will cause the SM to offer the newservice where it would have offered the oldservice. There are some type issues which need careful consideration here. The SM needs to insist that the newservice conforms to the oldservice (see [REES 93] for definition of conforms to).

The **`replace`** operation allows incremental state transfer to take place between old and new service instances, before switchover occurs.

1.4.6 Naming and location of collections

Service managers group services together into collections or sets. In many distributed applications there is a need to name collections of objects or to name and locate objects which satisfy particular attributes [EDWARDS 94a].

Service Managers provide some of the foundation technology for implementing such naming schemes. Other relevant ANSA work which can be exploited to solve this problem is: the ANSA Naming Model [LINDEN 93], The Model for Interface Groups [OSKIEWICZ 93] and The Remote Data Base Query Model [THOMAS 94]. The Naming model shows how to build federated naming schemes. The interface group model shows how to access a collection of objects through a single interface. The Remote Data Base Query Model shows how to manipulate sets of objects contained in a database.

1.4.7 Specialisation of Service Management

There will be some aspects of service management which are specific to the implementation of that service. One option would be to allow specialisation of SMs to make them specific to the service which they are managing. This would mean that programmers had to program an SM as well as the service itself; it also does not fit well with the principle of object encapsulation.

An alternative approach (which is used in §1.6) is to provide generic facilities in the SM, and make the service instances themselves responsible for application specific management. Examples of application specific management include consistency protocols and initialisation of state, some examples are given in §1.6.

Another approach, also use in §1.6, is to build application specific management services on top of the basic facilities provided by SMs and NGMSs.

1.5 Managing the Mangers

Both the NGMS and SM are replicated and self managing. Instances of NGMSs are grouped together and run a membership protocol to agree membership (see figure 1.3). Some mechanism is needed to bootstrap the NGMS and introduce new nodes into the groups. The least that is needed is a tool or configuration manager which can instantiate NGMSs on nodes and group them together. The configuration manager itself should be reliable and persistent: it should be possible to restart it and add new nodes to existing groups. For the moment it is assumed that sufficient reliability can be achieved by assuming the tool stores its persistent state at multiple sites and that it can be restarted at any of these sites. Initially a simple text based service will be provided, but there is scope to provide a graphical representation to make it easier to manage the system.

One aim is to make sure that the NGMS and SM are not dependent on services such as traders; rather it should be an objective to be able to use them to make the trader highly available.

Note: A highly available trader, built using this technology, might be an interesting deliverable/demonstration. It doesn't have to be the ANSAware trader, a tcl/tk based one would suffice? Comments invited as to whether this is worth pursuing?

Once administrators have established a number of groups running the NGMS service, the next step is to establish one or more SM groups. The same configuration manager could be used, assuming it remembers which groups it has set up and where the NGMSs are. It can use the NGMS to determine which nodes are available and allow the administrator to select on which nodes to place SMs (recall that all members of an SM groups do not have to reside within the same NGMS group — see figure 1.3).

Each time an SM is instantiated it must be joined to a group, unless it is the first member. So the configuration manager broadcasts the new member's identity to the group. The result sent by each member of the group to the configuration manager is its current view of the membership (which should be identical and if it is not, the group members should be destroyed). Each time a new member joins the group, the existing members register with the NGMS on that member's node; the new member has to register its interest with the existing members' NGMSs.

Once a sufficient number of SMs had been added to the group (typically envisioned to be two or three) the administrator can start to tell the SM which service types to manage: where the templates are, what parameters are needed etc. In a new system this would include the trader.

When the SM group has been initialised it can be set running. From this point it takes responsibility for controlling its own membership and failure arbitration (in the event of NGMS and user reported failures). Such a group is shown in figure 1.4. In a new system at this point the SM would be asked to instantiate a trader service and put a reference to itself in it.

1.5.1 Consistency requirements of SMs and NGMSs

NGMSs need to maintain the consistency of their registration information. It is not necessary for each NGMS in a group to see changes in registration in the same order, it is sufficient that they will all eventually converge to having the same registration information. This can be done by piggybacking registration changes onto the membership protocol, as explained in §1.3.

SM groups use a variant of the Leader-Follower protocol [BARRETT 90]. Operations are divided into updates and reads: the former are always invoked on the leader, whilst the latter can be invoked on any member — it is assumed that there is a high ratio of reads to updates. Figure 1.5 shows what happens when a client invokes an update operation and explains the basic protocol for a three member group.

Although the protocol can be extended to cope with any number of replicas, it is envisaged that there will be very little need to have more than three replicas. It is also a very simple exercise to reduce the protocol to cope with only two replicas.

The leader is responsible for invoking operations on the followers which update their state, it alone is responsible for invoking third party services. In the case of service managers the only time the leader invokes an operation

Figure 1.4: Management of an SM group

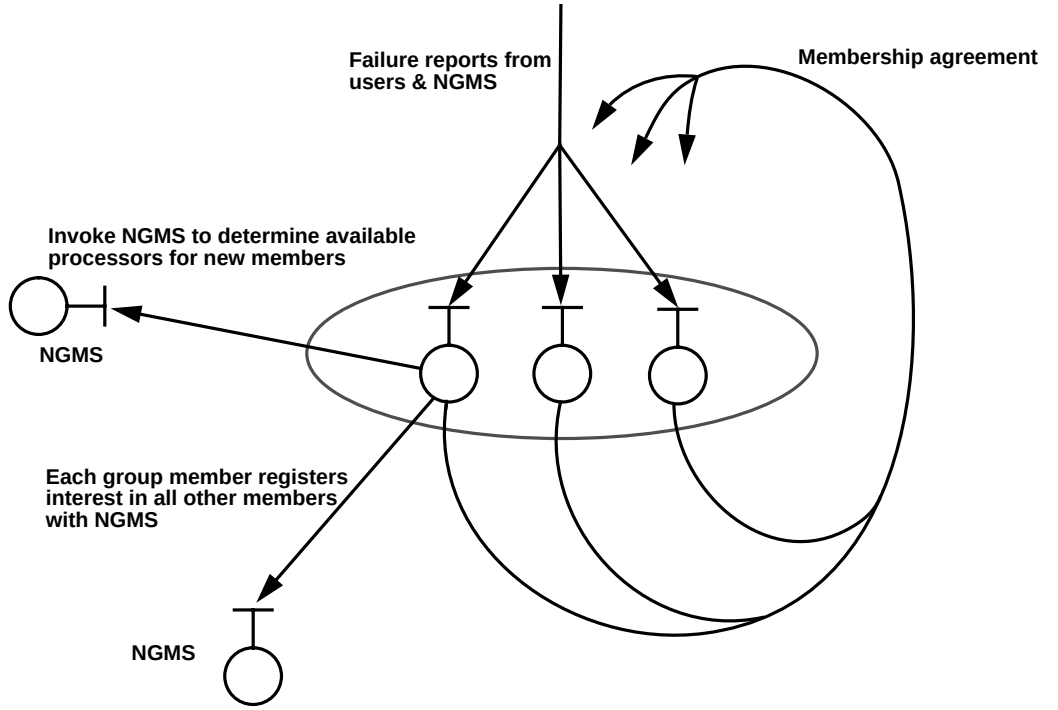
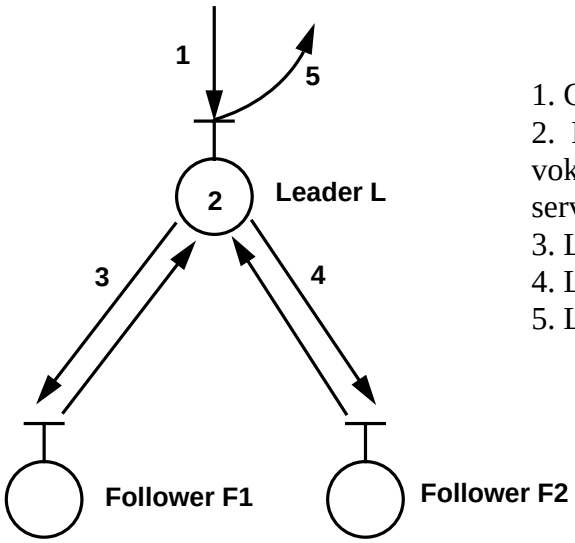


Figure 1.5: SM group being updated



1. Client makes invocation
2. L executes operation (may invoke third parties to instantiate new services)
3. L sends update to F1
4. L sends update to F2
5. L sends reply to client

which updates state is when it needs to instantiate a new service. At this point there is a danger that if the leader where to fail the SM group would not know about any state changes which their leader had made to the outside world.

Application semantics can be exploited to resolve this problem. Each SM instance will have the same list of NGMSs which it can use. So recovery involves the new leader asking each NGMS service which services have been created by the SM group (i.e. which services has it got the SM group registered as an interested party). Any services which the new leader does not know about are orphans which must have been created by the old leader which failed before it updated the its followers. This means that the act of service

instantiation must include registering the SM group's interest with the local NGMS; registration and instantiation needs to be one atomic action to avoid orphans. This can be most easily achieved by making the NGMS the factory responsible for service instantiation.

It is assumed that all state update operations are idempotent: withdrawing a service twice has the same effect as withdrawing it once. The only exception is service instantiation, which involves state changes outside of the group: the SM group needs to query the environment (the NGMS instances) to recover it state if it fails while executing this operation.

The remainder of this section examines what happens if a double failure occurs including the failure of the leader (the worst possible scenario which a three group SM can withstand). The discussion assumes that the SMs are fail silent.

When a double failure occurs, the surviving SM group member will receive a failure report from an NGMS or a client of the SM service. It then initiates a membership agreement protocol (e.g. [CRISTIAN 91]). Suppose L and F2 fail, the group recovers as follows.

- F1 promotes itself to leader and creates another two followers, using the NGMS service.
- The new followers' state is cloned from F1.
- F1 queries the NGMS to see if there are any orphans. If it finds any, it updates it own state and sends the state change to the new followers. If it finds none it does nothing.

Orphan services (services which the SM does not know about) can occur if the failure occurs after 1 or 2 (see figure 1.5). In this case the leader created new services instances, but failed before telling the rest of the group about them.

Eventually the client may retry the failed operation. If the operation did not involve updating third party servers, there is no problem: SM update operations are idempotent. If the operation was to instantiate a service the SM needs to recognise that it has already instantiated the requested service.

If a single failure (of the leader) occurs the new leader needs to ensure that all members of the group are in the same state. (In particular if a failure occurs after 3 in figure 1.5, the two followers will be in different states.) This presents no difficulty because SM state update operations are idempotent: the new leader merely resends the last update to both of the followers. If the original follower rexecutes an idempotent operation it will cause no damage.

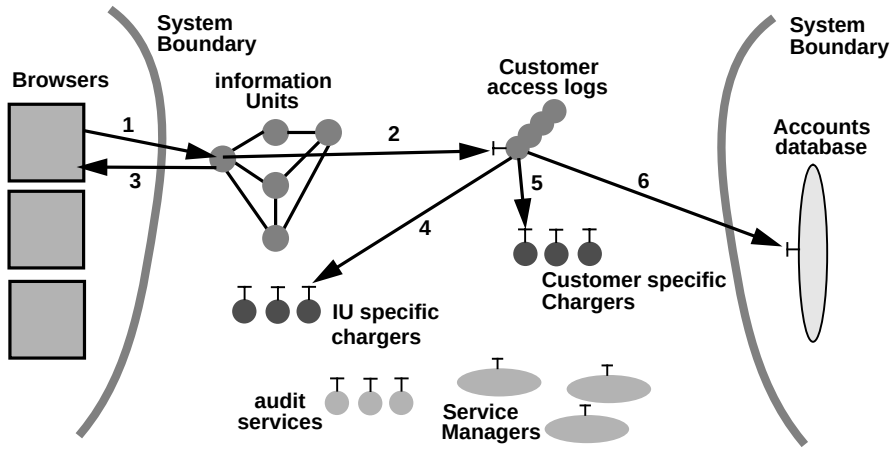
If a failure occurs during the recovery period, recovery is restarted. If the new leader fails after a double failure, before all group members are in a consistent state, it is equivalent to a triple failure and the service will fail.

The task of distributing state changes and the recovery algorithm is very simple, because the application semantics have been exploited (e.g. the operations are idempotent). The most complex part of the SM service is the membership agreement algorithm; something that does not have to be done for applications which use the SM service, since it provides them with a membership service.

1.6 Case Study: MED in DIPS

DIPS is a information publishing system in which access to information is chargeable. Figure 1.6 is a block diagram of the major components of DIPS. The following is a very brief and much simplified description of how DIPS works. More information (e.g. how browsers can get quotes) is given in [OSKIEWICZ 94]. The remainder of this section looks at how service managers provide support for DIPS, the specialisation of service management for DIPS, and the MED support for lifecycle in DIPS.

Figure 1.6: The main components of DIPS



1.6.1 System overview

The information units are CORBA wrapped World Wide Web Objects. A browser accesses an information unit by asking it to display itself (1). An information unit first logs this in the customer's access log (2), before displaying itself on a customer's browser (3).

Some time later a customer's access log calculates the charges due from the customer. It does this by first consulting the information unit's charger which applies the charges for accessing the server (4) — the server specific charging policy. It then consults the customer's charger which applies the customer specific charging policy (discounts) (5).

When the customer's credit is exhausted, or the charging period has expired, the charges are downloaded to the accounts database (6). The latter is a legacy system: DIPS cannot control its commit, effectively introducing a system boundary.

The browsers reside on customer's machines and are therefore outside of the control of the system. Hence there is a boundary between the browsers and the rest of the system. The system needs to minimise the potential for customers to damage it; in particular it is assumed that browsers may fail at any time (e.g. customers gets bored and switch their machines off).

Any service in DIPS can passivate itself if it has not been used for a certain time.

1.6.2 Service managers in DIPS

The information units, IU chargers and customer chargers are read only services: they are not updated, only replaced. However, they are replicated for availability and under the control of service mangers. In the event of a failure of a capsule, persistent store or node containing a service, the appropriate service manager will be notified by an NGMS or client. The service manager may then try to instantiate another service instance: it uses NGMS to determine available nodes and asks a service instance to clone itself onto the node identified.

Clients only ever invoke one of these service instance at a time. If a client detects that any instance of a replicated service has failed it complains to the service manager which is responsible for failure arbitration (and instantiating new service instances). The client then retries another instance of the service. The one exception to this rule is if a browser detects a failure of an information unit: it must invoke the audit service (see below) associated with the customer's access log.

The customer's access log is also replicated for availability and is under the control of a service manager. Each time its customer accesses an information unit the access log is updated. The information Unit multicasts the update to all instances of the customer's access log. No attempt is made to order the updates: since a customer can only do one thing at a time using a browser there can be no concurrent updates. (DIPS was designed so that each customer had a separate access log to avoid the problem of coordinating concurrent updates.) If an information unit detects the failure of an access log (because it fails to acknowledge that the access has been logged), it will complain to the service manager.

The service manager may ask an access log instance to clone itself onto a node identified using NGMS to maintain availability. Once the membership has changed the group needs to force all clients to rebind to it to get the correct view of the membership. This can be done using incarnation identifiers [OSKIEWICZ 93]: clients presenting the wrong incarnation identifier in an update request are forced to rebind. If a leader-follower (§1.5.1) or active replication protocol were used, this would not be necessary: the leader distributes updates to the current membership rather than clients. However, such protocols would introduce extra latency.

1.6.3 The audit service — specialisation of the service manager

The interaction involving the browser, information unit and access log is vulnerable to failure. If the information Unit fails before displaying itself on the customer's browser, how does the customer know if the charge has been registered or not? If it was not for the boundary between the browsers and the rest of the system, a natural and easy way to avoid this would be to make the interaction an atomic action. However, (in its most simplistic form) this would mean that the browser would be responsible for committing the atomic action.

The browser is not on a trusted machine: if a customer switched the machine of at a critical time, a large part of DIPS could be locked up waiting for a commit or abort which was never going to happen.

To avoid this scenario, if a browser or an information unit detects a failure in the browser/information unit interaction, it invokes the audit service. The audit service is responsible for removing any access which may have been

logged in some instances of the access log (it may not have been registered in all instances of the customer's access log).

The audit service can be regarded as a specialisation of the service manager: browsers use it instead of the service manager for information units. However, it uses service managers for a large part of its functionality: forwarding requests for service instances to the information unit's service manager. Substituting the audit service interface for that of the service manager preserves the paradigm of providing information unit clients with a single interface for service management.

When a browser reports an information unit failure the following occurs.

1. The audit service invokes the information unit's service manager reporting a failure. The latter begins failure arbitration to determine whether the information unit has indeed failed.
2. The audit service invokes a compensating operation on each instance of the customer's access log. The compensating action needs to cancel the effect of access registration whether it has occurred or may occur some time in the future.
3. The browser is trusted to ignore any information returned by a slow running information unit, which it has previously reported to have failed.

Note that at step 2, even if the access log was a totally ordered active replica group [OSKIEWICZ 93], charge registration could not be guaranteed to preceded the compensating action. Some notion of causality is needed from the message passing system, the natural causality provided by RPC having been broken by the "failure" of the information unit. Whether the cost of building causality into the message passing system merits the simplification to the application programmer is an open question.

This scenario shows that rolling the group forward is not always the right thing to do. Suppose an information unit fails mid-way through updating the access log instances for a particular client, but after updating the majority. If the replication protocol was based on a majority quorum, the group would be rolled forward, so the charges would be registered.

There is a double failure scenario to which access logs are vulnerable: simultaneous failure of browser and information Unit when some of the access logs have been partially updated. This could be tolerated by having the service manager periodically check the state of access logs. (It would not cope with the scenario in which the information unit and browser failed when all access logs had been updated, but before display occurred).

The audit service itself is under the management of a service manager for availability.

1.6.4 MED support for service lifecycle in DIPS

Customer access logs demonstrate how MED can support the service lifecycle. The service manager creates a number of instances of an access log for a customer on various nodes using the NGMS. When they are first instantiated by a service manager, they invoke an external credit agency (not shown in figure 1.6) to ascertain a suitable credit limit for the customer. This computation needs to occur once and only once. The access logs behave like an active replica group: each access log invokes the agency and the agency only responds if it receives a predetermined quorum of requests. The access log service is now ready for use by clients.

When the customer's credit is exhausted the access logs **`withdraw`** their service from the service manager and download the charging information to the accounts database. Again they behave as an active replica group, since they must not fail while interacting with the database. Finally the access logs destroy themselves. The next time the customer accesses an information Unit a new access log service will have to be created by the service manager for the customer.

1.7 Related Work

The work reported in this paper has been influenced by the work of the Isis Project and also that of Cristian. The contribution of the work reported here is to show how group management can be fully integrated with other aspects of service management, throughout the lifecycle of service provision, including: activation, passivation, trading, instantiation, naming and location, load balancing and in service upgrade. This is achieved by separating out failure detection and group management (including membership and population control), rather than to mixing them together. In addition the approach seems to offer the potential to scale and federate easily.

The service manager is able to use the service of the NGMS to monitor the availability of third party services on which its service depends. Hence it monitors the availability of the persistent store on which it service instances depend. More experience of using this facility is needed to understand its usefulness and limitations. For example, should a service manager monitor other services in this way or in general is it better to engage in a QoS negotiation and agreement with the third party service manager.

This idea is further extended to allow clients to register interests in particular service groups with the SMs managing those groups. This can be used to build active replica groups and also allows clients to dynamically renegotiate QoS. Again more experience of using it is required.

Finally the failure assumptions in this paper are not restricted to fail silence [BIRMAN 87] or crash failure [CRISTIAN 91]. Rather we assume that managed services can be observed to suffer omission failures and value failures, as well as crash failures. The infrastructure will suppress more severe failure behaviour (see [EDWARDS 94b] for how this can be done). The NGMS and SM services themselves are assumed to be fail silent — we argue that this is a reasonable assumption for small well debug “system services” services.

1.7.1 Isis

In [BIRMAN 87], Birman and Joseph describe a site view management service in Isis. This provides much of the basic functionality required of the NGMS service. The NGMS presented in this paper also copes with activation, passivation and migration. This integrates failure detection more fully with the object-lifecycle.

If the Isis site view management service detects a failure it reports this failure to other members of the group. The approach taken in this paper is to report the failure to a management service responsible for the group. If the management service is the group itself then the two approaches are equivalent. However, the approach presented here allows the notification of other interested parties too. In addition, by factoring out the fault management of the group it is possible to achieve better integration with the

object lifecycle management and also to deal with issues of scaling and federation.

1.7.2 Cristian

In [CRISTIAN 91], Cristian describes a processor group membership service and server group membership service. The processor group membership service keeps track of which processors are available in the system. A server group membership service can be built out of this by having the processor group membership service keep track of the services running on each node and associating each service with a group. Service activation and passivation and other aspects of object lifecycle management are not considered.

The approach described in this paper separates the notion of failure detection from membership and group management. Unlike the server group membership service, the NGMS in this paper has no notion of membership: this is provided by a separate entity — the service manager. Since service managers are specific to a particular service type, this makes it easier to integrate fault management with other aspects of managing a service such as trading, instantiation and upgrading as discussed above. Cristian does not consider failure arbitration.

In [CRISTIAN 92], Cristian describes an availability manager. This uses the server group membership service to keep track of the number of instances in a system using the processor group membership service described above. The availability manager has some similarities with service managers. The main difference is that it assumes that service instances are replicas and only deals with instantiating new instances of a service when a failure occurs; no other aspects of service management/lifecycle (load balancing is identified as being important in practice). It also relies on another service to maintain a notion of membership (the processor group membership service), unlike the SMs discussed in this paper.

1.7.3 Event services

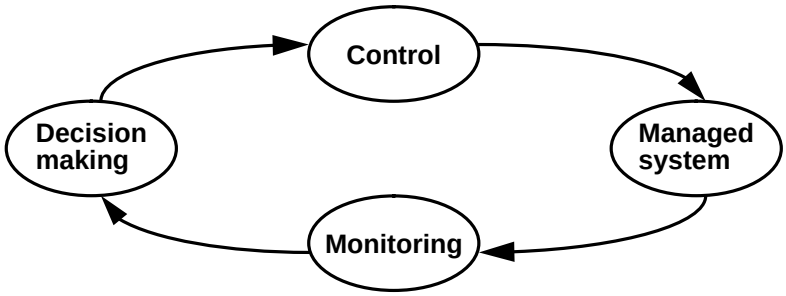
The communication patterns in MED between the NGMSs and SMs have strong analogies with the kind of communication patterns which can be supported by an event service. The NGMSs acts as a set of collaborating event servers, sending out event notifications when they detect failures. It may be possible to deploy MED using an event service (such as that envisaged in [OMG 94]) as part of the communication sub-system. Initial investigations suggest that it would be necessary to modify the event server which runs on each node so that registration information is distributed between groups of servers. Failure detection code would also need to be added: each time an SM registered an interest in a service, the event server would have to start monitoring that service. The event server will also have to be modified to use a membership protocol to agree on NGMS membership and cope with joins and leave. Further investigation is needed. Failing that some notion of multicast would be helpful.

1.7.4 The ANSA management model

The idea and motivation for having a separate management interface for services is explored further in [ANSA 93]. The generic ANSA model of management is shown in figure 1.7. In MED monitoring is carried out by NGMS and clients: they are responsible for detecting and reporting failures.

Decision making and control is exercised by the service managers: they are responsible for failure arbitration and service instantiation.

Figure 1.7: The generic model of management



An ANSA principle is that objects should managed themselves. This is entirely consistent with MED: a service manager cannot force a service to do anything it does not want to do. For example in §1.6.4 the managed service only delegates what is necessary and convenient to the service manager, managing most aspects of its lifecycle itself.

1.7.5 Industrial management platforms (HP OpenView)

This section considers how MED relates to management platforms provided by industry. At present only Hewlett-Packard's OpenView [HP] is considered, because the authors do not have ready access to information on other platforms. OpenView provides support for SNMP, CMIP and provides much functionality which will appear in the OSF's DME. The following is the result of a preliminary study, an implementation of OpenView would be needed to confirm these suggestions.

NGMSs and service managers could run in OpenView as (XMP based) Agents and be started using OpenView's "robust startup facility". OpenView provides facilities which can discover what machines are in the network. This would allow the administrator to choose on which nodes to instantiate NGMSs. Once a number of NGMS services has been installed, the administrator could use OpenView to interrogate NGMSs and set up service managers.

OpenView contains several facilities for managing and maintain MIBS. Both the NGMS and SM effectively contain MIBS of the objects which they manage and monitor. It would be very useful if OpenView could be used to manipulate these MIBS, thereby providing the administrator with a convenient interface to service managers and NGMSs.

It may be possible to use OpenView's event facilities for communicating failure events to service managers, but it is not clear if they could provide adequate performance. (An objective of MED is to provide failure diagnosis and instantiation of new services in the order of a hundred milliseconds.) However, service managers could be made to generate events which OpenView understands when significant lifecycle events occur, such as failure. The membership algorithm run by NGMSs and service managers would be private to those services.

Finally the graphical user interface and "map" paradigm of OpenView would be an extremely convenient way of visualising what was going on in a MED network.

1.8 Conclusions and some architectural consequences

An underlying assumption is that clients are expected to go to the service managers to ask for the appropriate service instance. This means that trading is on service managers and not on the services themselves. A consequence of this is that once the nature of the service required has been established the service manager can take over the more detailed aspects of trading (e.g. QOS). More work is needed to investigate the consequences of this.

The service manager also acts as a relocator (in the terminology of [OSKIEWICZ 93]); it is the place to which clients go to get the up to date view of group membership.

One of the underlying assumptions is that every interface has embedded in it an interface reference to a management interface to which the client can report a failure of the service instance being used. For the three levels of management identified in this paper the interface references are as follows:

- Failure of application service instance: service manager;
- Failure of service manager instance: service manager (another member of the group);
- Failure of NGMS: NGMS (another member of the NGMS group).

This reflects that service managers and NGMSs are self managing.

Although it is assumed that groups of NGMSs and SMs do not span federation boundaries, the computations involving SMs may well span federation boundaries. If a client detects a failure in a service, it may report that failure to the SM managing that service (or in the case of the SM the whole SM group). The SMs perform failure arbitration on the services they manage and also on themselves, rather than the clients of these services which may be in different federations.

There are broadly two sets of assumptions which are commonly made when building membership protocols: synchronous or asynchronous [SCHNEIDER 93]. The assumption made in this paper are that the system is synchronous: bounded time RPC. In an asynchronous system no assumptions are made about bounded time message delivery. Those who advocate making such assumptions argue that they match more closely the behaviour of the systems in use today.

The assumptions made in this paper were made to simplify the underlying protocols; they reflect what the authors believe will be possible in future systems. MED has been designed so that its synchronous assumption is made only in a few places: in particular the use of Cristian's Available List protocol. In principle it should be possible to replace this with an asynchronous protocol for example: [BIRMAN 87] or [MACEDO 94]. This has not been tested.

1.8.1 Possible future extensions

We need to figure out the right way to build passive replica groups or leader follower groups using MED. In particular, how does the SM get to know which group member is responsible for processing updates?

There is scope for investigating how the NGMS (and perhaps other services) could collaborate (with the network) to isolate a failed node, some relevant ideas are presented in [COAN 94] and [EDWARDS 94b].

More work is needed to understand how to use SM to name and locate collections (see §1.4.6).

More experience is needed in specialising the management facilities provided by MED; are the methods suggested in §1.4.7 sufficient? The audit service in §1.6.3 can be thought of a specialisation of the Service Manager. Its detailed design is for further study: it should not be able to fail in such a way which leaves access logs in an inconsistent state. Active replication is a possibility: the audit service itself has no long term state which needs replicating, but the audit computation needs to be dependable (through replication). The audit service itself should use MED to make itself dependable.

At low loads the NGMS could activate and exercise the passive objects running audit routines to make sure that had not failed.

Using NGMSs and service managers to instantiate services on demand to meet peak load levels.

Note: This needs to be aligned with any work BNR have done on load balancing using nodemanagers in ODS.

The use of service managers to implement more complex naming and location schemes requires further study (see §1.4.6).

1.9 Acknowledgements

The authors are grateful to Ian Domville (BNR), Andrew Herbert (APM Ltd.) and Bill Noakes (BNR), for some useful discussions.

References

[ANSA 93]

"Management in Object-Based Federated Distributed Systems", TR.39.00, APM Ltd., Cambridge, U.K., February 1993.

[ANSA 92a]

"ANSAware 4.0 System Programmer's Manual", APM Ltd., Cambridge, U.K., February 1992.

[ANSA 92b]

"ANSAware 4.0 Application Programmer's Manual", APM Ltd., Cambridge, U.K., March 1992.

[BARRETT 90]

Barrett, P.A., Hilborne, A.M., Verissimo, P. , Rodrigues, L. , Bond, P.G. , Seaton, D.T., Speirs, N.A., "The Delta-4 Extra Performance Architecture (XPA)", in Proceedings of the 20th International Symposium on Fault-Tolerant Computing, Newcastle, June 1990, pp481-488

[BIRMAN 87]

Birman, K.P., Joseph, T.A., "Reliable Communication in the presence of Failures", ACM Transactions on Computer Systems, 5(1), February 1987, pp47-76.

[CHERITON 93]

Cheriton, D., Skeen, D., "Understanding the limitations of causal and total ordered commmunication", ACM SIGOPS 27(5) pp44-57, (December 1993), Proc 14th ACM Symposium on Operating Systems Principles, Dec 5-8 1993.

[CRISTIAN 92]

Cristian, F., "Automatic Reconfiguration in the Presence of Failures", Proceedings of the International Workshop on Configurable Distributed Systems, Imperial College, UK, March 1992, IEE, pp4-17

[CRISTIAN 91]

Cristian, F., "Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems", Distributed Computing, 4(4), April 1991, pp175-187.

[COAN 94]

Coan, B.A., Heyman, D.,"Reliable Software and Communication III: Congestion Control and Network Reliability" , IEEE Journal on Selected Areas in Communications, (12), 1, January 1994, pp40 - 45.

[EDWARDS 94a]

Edwards, N.J., "Commercial Information Services in the World Wide Web", APM.1220, APM Ltd., Cambridge U.K., May 1994.

[EDWARDS 94b]

Edwards, N.J., Oskiewicz, E., “Generic Hazards for DIPS”, APM.1182, APM Ltd., Cambridge U.K., April 1994.

[EDWARDS 94c]

Edwards, N.J., “Open Dependable Distributed Computing”, APM.1145.0, PM Ltd., Cambridge U.K., February 1994.

[HP]

“HP OpenView Technical Evaluation Guide”, Release 3.1, Hewlett-Packard.

[LINDEN 93]

van der Linden, R.J., “The ANSA Naming Model”, AR.003.01, APM Ltd., Cambridge U.K., February 1993.

[MACEDO 94]

Macêdo, R.A., Ezhilchelvan, P.D., Shrivastava, S.K., “Newtop: A Fault-Tolerant Total Order Multicast Protocol Using Causal Blocks”, Department of Computer Science, Unversity of Newcastle upon Tyne, 1994.

[ODP 93]

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

[OMG 94]

Common Object Services Specification, Volume I, OMG Document Number 94-1-1, March 1994.

[OMG 91]

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

[OSKIEWICZ 94]

Oskiewicz, E.P., Edwards, N.J., “DIPS — A Distributed Information Publishing System”, APM.1171, APM Ltd., Cambridge, UK, April 1994.

[OSKIEWICZ 93]

Oskiewicz, E.P., Edwards, N.J., “A Model for Interface Groups”, AR002.001, APM Ltd., Cambridge, UK, 1993.

[REES 94]

Rees, Owen, “Programming cheaper dependable systems”, APM.1122.00.05, APM Ltd, Cambridge U.K., April 1994.

[REES 93]

Rees, R.T.O. “The ANSA Computational Model”, APM.1001.01, APM Ltd., Cambridge, U.K., February 1993.

[THOMAS 94]

Thomas, G., van der Linden, R., “Remote Database Queries in Object-Oriented Distributed Systems”, APM.1138, APM Ltd., Cambridge, U.K., February 24 1994.

[SCHNEIDER 93]

Schneider, F.B., “What Good are Models and What Models are Good?”,, in Distributed Sytems, 2nd Edition, Sape Mullender (ed.), ACM Press, 1993.

[WAI 94]

Wai, F., Otway, D., Howarth, N., Herbert, A., “A Performance Framework”, APM.1137, APM Ltd., Cambridge, U.K., March 1994.