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

Programming cheaper dependable systems

Owen Rees

Abstract

Services must be dependable if they are to deliver the benefit that justifies the cost of deploying them. Services must be sufficiently dependable for their purpose, but no more costly than is necessary to achieve that level of dependability.

This document describes some work on identifying programming concepts that can be exploited to reduce the cost of dependable systems. This includes both the reduction of operating cost by identifying where cheaper mechanisms are acceptable, and reducing the cost of developing dependable systems by providing guidance to programmers and tools to support the proposed programming strategy.

The main theme is to explore the issue of mutability in the context of dependable systems. The objective is to identify potential benefits of a better understanding of mutability, in particular, the benefits of a better ability to propagate information about mutability.

Once the benefits have been identified, it will be possible to specify the mechanisms required to achieve those benefits. It will then be possible to develop guidelines for the design and implementation of systems so as to make the best use of the mechanisms. These will form part of the methodology for dependability.

The work is being conducted as part of ANSA Phase III task D2

APM.1122.00.05 **Draft** 13 April 1994

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

Distribution:
Supersedes:
Superseded by:

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Programming cheaper dependable systems

Programming cheaper dependable systems

Owen Rees

APM.1122.00.05

13 April 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

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Architecture Projects Management Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

Page	Section	Title
1	**1**	**Objectives and strategy**
1	1.1	Objectives
1	1.1.1	The need for dependability
1	1.1.2	Cost-effective dependability
1	1.2	Design and programming for dependability
2	1.3	Strategy
2	1.4	Prototyping environment
2	1.4.1	Prototyping objectives
2	1.4.2	Technology options
5	**2**	**Scope for improved efficiency**
5	2.1	Overview
6	2.2	Transaction systems
7	2.3	Active replication
7	2.3.1	GEX Server group mechanisms
10	2.3.2	Exploiting knowledge of mutability at the server
13	2.3.3	Exploiting knowledge at both client and server
14	2.3.4	Groups as clients
16	2.3.5	Exploiting distributed database research
16	2.4	Passive replication
17	2.5	The ANSA atomic activity model
18	2.6	Conclusion of initial study
19	**3**	**Exploiting knowledge of mutability**
21	**4**	**Analysis of failure characteristics**
23	**5**	**Automation and language issues**
25	**6**	**Notes**
25	6.1	Background
25	6.1.1	Transaction systems
26	6.1.2	Groups
28	6.2	Possible benefits
28	6.3	Relationship to previous work
28	6.3.1	Atomicity infrastructure
28	6.3.2	Groups
29	6.4	Other issues
29	6.4.1	Sensors and actuators
29	6.5	Programmer tools
29	6.6	Activity identification
29	6.7	Failure model
30	6.8	Link to scenario
30	6.9	Link to engineering

30 6.10 Type issues

1 Objectives and strategy

1.1 Objectives

1.1.1 The need for dependability

Any service that is used in support of any business has some dependability requirement. There is a cost associated with using the service, and the service will not be delivering the appropriate benefit unless it is available and operating correctly.

Making a service more dependable increases the cost of providing the service. Even where the cost of operating the service is not increased, making the change will involve a cost.

Services must be sufficiently dependable for their purpose, but the users of the service will be reluctant to pay for more dependability than they require.

1.1.2 Cost-effective dependability

The overall objective of the work described in this document is to reduce the cost of making systems dependable. This can be broken down into two sub-goals.

1. Identify where and when the required dependability can be achieved with mechanisms less expensive in resources that those required for the general case.
2. Identify how to reduce the effort needed to design, implement and maintain dependable systems.

This document describes the programming issues associated with these objectives.

1.2 Design and programming for dependability

The objective of the work described in this document is to identify the programming concepts that can be exploited to reduce the cost of making systems dependable.

This work will contribute to the methodology for dependability:

1. Design and programming guidelines for structuring applications so that they can exploit efficient dependability mechanisms
2. Specifications for infrastructure components needed to exploit application properties to provide efficient dependability.
3. Assessment of the potential for tool support for design, programming and configuration, ideally with prototypes to demonstrate the principles.

To achieve these objectives, it will be necessary to understand of how to identify exploitable properties of an application and how to propagate this information to where it is needed.

1.3 Strategy

In order to provide a focus for the work, the plan starts from the hypothesis that identifying and propagating knowledge about the use of mutable state can lead to savings in the design, construction, and operation of dependable systems. The exploration of this hypothesis, for a selected example (to be taken from the information publisher scenario [OSKIEWICZ 94]), has been broken down into four stages:

1. identify specific benefits that might be achieved
2. identify how analysis of mutability might achieve those benefits in an example selected from the information publisher scenario
3. analyse dependability characteristics of example
4. identify tool and language issues in realising the benefits

Once the guidelines, components, and tools appropriate to one example have been developed, more general results will be developed by studying examples that introduce additional issues.

1.4 Prototyping environment

In order to carry out the work, it will be necessary to construct some prototypes which can be used for exploring the problem space.

1.4.1 Prototyping objectives

The objectives for prototypes in this work are:

1. Exploration and demonstration of basic principles
2. Demonstration of relationship to target technology

In the earlier stages of the work, the emphasis is on establishing and describing the principles, and so the first of the objectives for prototypes applies.

In the later stages, the emphasis moves towards explaining how the principles apply to currently used languages and infrastructures.

1.4.2 Technology options

1.4.2.1 Exploration and demonstration of principles

For this work, flexibility and rapid development are the first priorities. Being able to demonstrate the principles clearly is also important. Tcl with its RPC and graphics extensions is a good match to these requirements.

Tcl is simple interpreted script language, designed to be easy to extend. The particular extensions relevant to this work are Tk which provides a very easy to use interface to the X Window system, and Tcl-DP which provides RPC over TCP connections.

The ease of use, and ability to alter programs dynamically makes this a good candidate as an exploration tool. The graphics support makes it a good candidate for a tool to demonstrate the principles visually.

Tcl and Tk are already in everyday use at APM supporting the documentation system. Effort required for installation and training is minimal.

Tcl and its extensions are freely available with no usage restrictions or fees.

1.4.2.2 Relationship to target technology.

The target technology appears to be C++, perhaps with extensions, over a CORBA platform, perhaps also using DCE.

Possible candidates are the Orbix platform available from Iona, and the extended C++ and DPE platform being developed by TINA-C.

2 Scope for improved efficiency

2.1 Overview

The purpose of this chapter is to identify where knowledge of properties of an application can be used to reduce the cost of making that application more dependable.

A great deal of research has been done in the context of transactions, and in particular, transactions on replicated and distributed databases. Some of this research is described in §2.2.

This chapter also describes some replication and transaction mechanisms which have been the subject of earlier work in ANSA. It shows how knowledge of the effect of operations on mutable state can be exploited to identify the cases where simpler mechanisms can be used than are required for the general case.

Table 2.1 summarises some of the issues that arise if replication or atomicity mechanisms are to avoid the introduction of faults when the underlying mechanisms are operating correctly. It also includes some issues concerned with being prepared to deal with a failure, should one happen. These issues apply to the very simple case of an operation that does not invoke a nested operation, and are presented as an indication of the starting point for the investigation.

Table 2.1: Summary of issues in normal operation

Kind of service invoked	Kind of access to mutable state		immutable
	Update	Observe	Read
Active replica, all invoked, all respond	Must update all copies	May observe any one	May read any one
	Updates as if in same order	- no update -	- no update -
	Outcomes as if in same order	Any order between updates	Any order
Passive replica client invokes one (chosen how?), one responds	Must propagate to all copies	May observe any one	May read any one
	Updates as if in same order	- no update -	- no update -
	Outcomes as if in same order	Any order	Any order
Atomic service (nested transactions)	write lock	read lock	- no lock needed -
	release lock on termination	release lock on termination	- no lock -
	establish recovery mechanism	- no recovery -	- no recovery -
	commit/abort state	- no state change -	- no state change -

The objective of this chapter is to explore both the kinds of service, and the significance of their use of mutable state. The cases in table 2.1 are only three of many possibilities, chosen to show that the use of mutable state is worth exploring.

The particular examples chosen to show the scope for improved efficiency are:

- GEX – an active replication mechanism, implemented in ANSAware, and following the principles set out in [OSKIEWICZ 93]
- The infrastructure for the ANSA Atomicity Model set out in [WARNE 93].

2.2 Transaction systems

A great deal of research has been done on reducing the cost of transactions for replicated and distributed databases.

Garcia-Molina and Wiederhold [GARCIA-MOLINA 82] discuss queries on a replicated distributed database. These are transactions that do not modify the database, and are thus read-only. They pay particular attention to being able to satisfy a query with purely local information, thus eliminating the communication costs. The various algorithms they present give different degrees of consistency, at varying cost. The focus is on distinguishing read-only transactions from updating transactions on the grounds that making this distinction is likely to be feasible, wheras finer distinctions of transaction type will be harder to make.

Weihl [WEIHL87] considers the situation where the data is distributed, but not replicated, and there will be actions that read a substantial part of the distributed system state. With conventional locking, such an action would block updates for long periods. The longer the action, the more likely it is to be aborted, either due to a failure, or to resolve a deadlock, and the less likely it is ever to complete. Weihl develops the idea of having multiple versions and using timestamps so that the read-only action can have a consistent view of the system without blocking the updates that are effectively in its future.

Herlihy [HERLIHY87a] also considers an approach based on multiple versions and timestamps, but goes beyond the “conventional” distinction of read and write operations, to consider additional semantic information for typed objects. The idea is to define a *minimal serial dependence* relation that defines the minimal set of conflicts that lead to delays or restarts, and use that to determine which version is accepatable. Herlihy states that this approach can be readily integrated with replicated databases that use a quorum consensus replication protocol, in which the quorum intersection relation for operations must be a serial dependence relation if histories are to remain legal [HERLIHY86]. Herlihy also shows that the more concurrency can be permitted with larger quorum sizes, which requires that more of the replicas be available, thus reducing the availability of the group as a whole.[HERLIHY87b]

A later paper by Weihl [WEIHL89] develops the idea of atomic objects encapsulated within atomic abstract data types. The specification of an object is broken down into a serial specification, which describes the permissible behaviour of the object in the absence of concurrency and failure, and the behavioural specification, which describes how the object responds to concurrency and failures. The paper goes on to discuss local atomicity properties of object specifications; if every object in the system has the local atomicity property, then the whole system has atomic behaviour.

A major goal of most of this work is to permit as much concurrency as possible. Herlihy and Weihl bring their ideas together [HERLIHY 91] to develop a hybrid locking algorithm which permits more concurency than previous locking algorithms.

Defining serial specifications of objects, and deriving dependency relations from those specifications, is an attractive approach. It is not clear how easy it would be to do this in practice, nor how expensive it would be to mechanise a theoretically optimal dependency relation. These issues merit further study.

2.3 Active replication

Active replication may be used to provide high availability for an application, and that is the use considered here. In particular, in an environment where hosts can become unavailable, active replication can be used to ensure that services continue to be available.

In an active replica group, all members of the group receive requests, do the work, and send responses. This means that all members have a copy of the state. The group must have mechanisms to ensure that the responses of the members are consistent, and that in any invocations made by the group, the requests are consistent. This requirement imposes the constraint that the behaviour of the members must be consistent, and that constrains the states of the members to be consistent.

Active replication has been studied in ANSA [OSKIEWICZ 93] and also implemented as part of ANSAware. That implementation – GEX – is used here as an example to illustrate the kind of mechanisms that are needed.

In the examples that follow, some failure cases are considered. In most cases, only crash failure of group members and omission failures are considered. A more detailed analysis of the failure characteristics is planned for a later part of the work, and will be described in chapter 4.

2.3.1 GEX Server group mechanisms

2.3.1.1 Assumptions

GEX adopts a black-box approach to the operations in the service to be replicated. It assumes that, given the same external stimuli, and provided that only one invocation is being processed, all group members will have identical externally visible behaviour.

GEX makes no attempt to identify or exploit properties of operations that would allow them to be processed either concurrently, or in different orders by different group members, without violating the consistency constraints. In the terminology briefly introduced above, it assumes that every operation is dependent on every other operation.

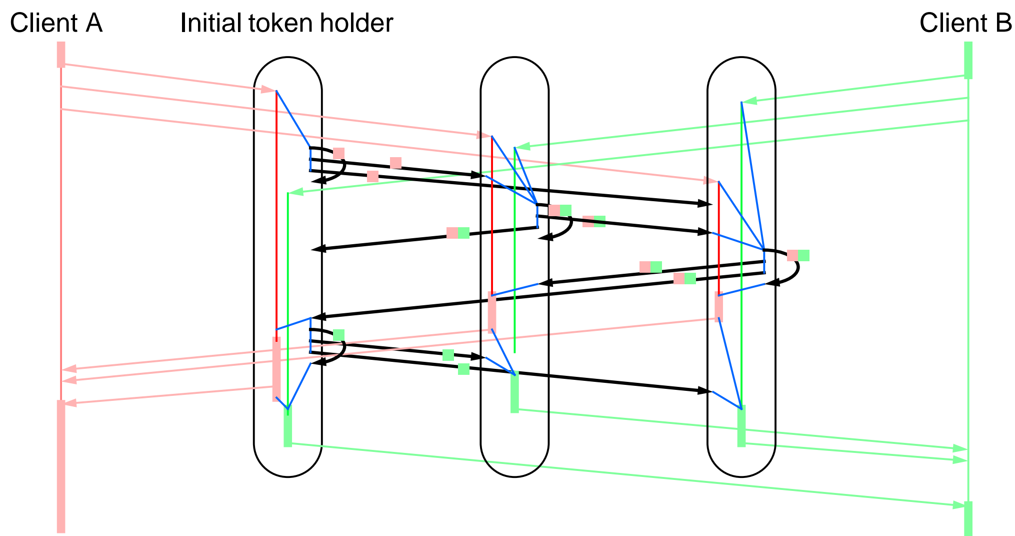
An important advantage of the approach adopted by GEX is that the replication is completely transparent to the programmer of a client that invokes a replicated service. The programmer of the server need only declare the state that needs to be transferred to a new member; this is the same as would be needed to passivate and activate the object. The programmer of the server must ensure that the behaviour is determined only by incoming invocations and responses to invocations made by the server; this means avoiding any direct use of the local environment.

The disadvantage of this approach is that the infrastructure must make worst case assumptions about the nature of operations. In particular, it must act as if every invocation of every operation both observes and updates common state, and delivers an outcome dependent on that state. It must also assume that every member must deliver the same outcome to the client in order to accomodate the most strict collation policy.

2.3.1.2 *Ordered evaluation*

Given the lack of knowledge about what operations do, GEX must ensure that all group members evaluate invocations in the same order. This is achieved by an ordering protocol that, in GEX, is combined with a quorum and liveness protocol. Figure 2.1 illustrates the protocol in the case where two singleton clients invoke a three member group concurrently, and the quorum that must have the invocation request before it is evaluated is also three.

Figure 2.1: Quorum and ordering protocol in GEX



The order of evaluation is chosen by the current holder of a notional token that is circulated around the group members. In this example, the invocation request from client A arrives at the token holder before the invocation request from client B, and is therefore selected as the next to be evaluated.

The token holder sends a message to every member of the group, including itself, indicating that:

1. the request from client A has been appended to the evaluation order
2. one group member is known to have the request
3. the token is passing to the next member of the group

In the example, the token message is shown arriving at the next token holder after the request from client A. This means that the new token holder can pass on the token with updated information. Since the request from client B has also arrived, another evaluation can be scheduled to follow that requested by client A, and this additional information can be passed in the same token message. The new token message indicates that:

1. the request from client B follows the request from client A

2. two members are known to have request A and one to have request B
3. the token is passing to the next member of the group

The next token holder has received both requests and can pass on this information with the token immediately. This brings the count of members that have request A up to the quorum of three and therefore the evaluation can proceed as soon as this token message arrives.

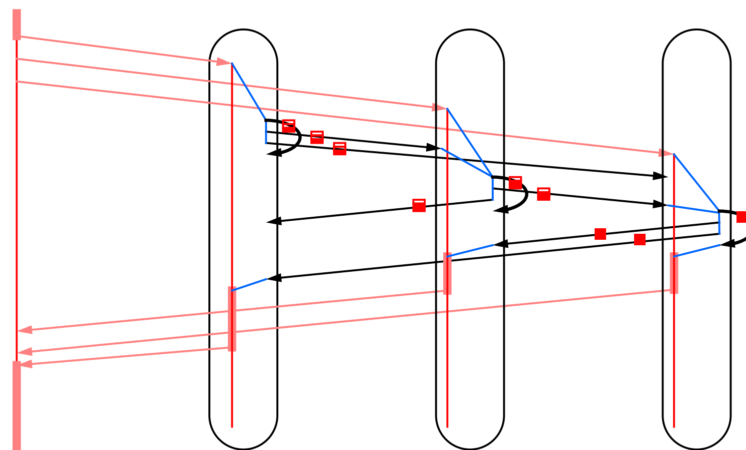
The token is now back at the initial token holder which has received request B since it passed on the token. It can therefore pass on the token again with the information that request B has also reached the quorum of three, thus allowing the evaluation to proceed.

2.3.1.3 *The cost of an invocation*

In the example in figure 2.1, the overhead of the quorum and ordering protocol was shared between the invocations. It can be seen that the cost will remain fairly constant no matter how many invocations are arriving concurrently.

In order to provide a starting point for studying the potential benefits of knowledge of mutability, figure 2.2 shows a single isolated invocation which will be used to analyse the cost of the ordering protocol.

Figure 2.2: Cost of a single invocation



In an invocation from a singleton client to a three member group using GEX as implemented in ANSAware 4.1, there will be 15 messages, three evaluations of the body of the operation, and a latency due to the five message critical path between the initiation of the invocation and its completion. This also assumes that there are no failures during the processing of this invocation, and that the invocation request messages arrive at group members before the ordering protocol messages that mention that invocation request. There will be additional messages if either of these assumptions is false.

2.3.1.4 *In the event of a failure*

In the event of a group member becoming unavailable, the GEX mechanisms will ensure that the service continues to be available.

The group members expect to receive token passing messages, and each expects to become the token holder from time to time, by receipt of the token passing message from its predecessor. If a group member fails in such a way as

to not pass the token, one or more other group members will detect the failure and initiate a reformation of the group. GEX includes a group reformation protocol which ultimately leads to a reformed group having a new membership.

Evaluations that have already taken place at one or more non-faulty members will be completed by all non-faulty members before the reformation. Invocations that had not yet been evaluated will be rejected, and the clients will be informed that their view of the group membership is out of date, by means of an incarnation number which is incremented each time the group reforms.

When the client is told that its view of the group membership is out of date, it must update its view of the group membership, and,if appropriate, attempt the invocation again.

If a group member fails to receive the message informing it that it is now the token holder, the token will cease to circulate and the reformation will be initiated.

Failure to receive a message indicating that some other member is the new token holder has little adverse effect. The group member will catch up as soon as it receives another token pass message.

Failure to receive an incoming request message will eventually lead to retransmission by the REX protocol on which GEX was built. This will also detected when the group member receives an updated evaluation order referring to a request it has not received. In this case, it requests a copy from the member who last sent out a token message, who must have a copy of the request in order to have sent the token message. This recovery mechanism will be triggered if the token message arrives before the original request, thus adding additional cost to the protocol but having no other adverse effects.

Failure by the client to receive a response message will be handled by the retransmission mechanisms of the underlying REX protocol.

Another possible failure is that the server group members receive requests from the client that are valid individually, but not the same. GEX as implemented has no mechanism to detect this case. The analysis of GEX using the ANSA failure model [EDWARDS94] revealed this case, and that this failure could be detected by including a checksums of the invocation requests in the token messages.

2.3.2 Exploiting knowledge of mutability at the server

2.3.2.1 Updates must be ordered

The purpose of the ordering protocol in GEX was to ensure that all members of the group deliver the same outcome. Updates must be made in the same order to all members to ensure that this will remain true for the outcomes of all future invocations.

The step taken here is to assume that all updates are dependent on each other, rather than assuming that all operations, including read-only operations, are dependent on each other. This leaves the question of dependencies of read-only operations, and this will be considered in the absence of other knowledge.

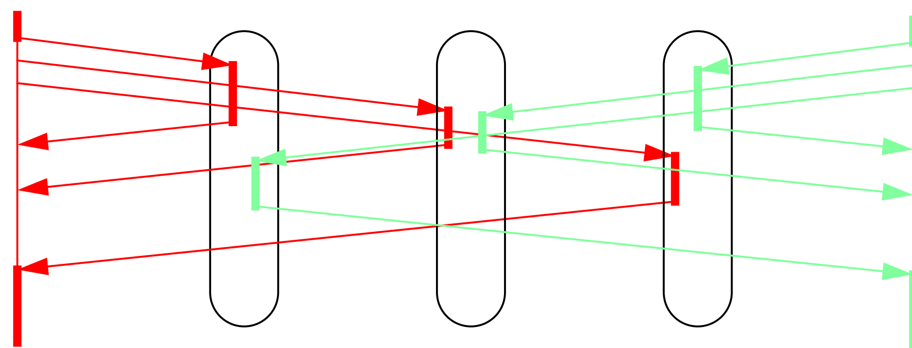
2.3.2.2 In the absence of updates

The precautions taken by the ordering protocol are not necessary for operations that do not update the state. Observe only operations will deliver the same outcome, in whatever order, and with whatever concurrency they are evaluated.

Weihl gives two definitions of what it means for an operation to be read-only [WEIHL89], and from the definition in terms of futures, is can easily be shown that no read-only operation is in the range of any minimal dependency relation. In other words, no subsequent operation is affected by a read-only operation.

Figure 2.3 illustrates a server group where the members procceed immediately with operations that do not update the state.In this case, both clients will receive consistent responses from the server group members.

Figure 2.3: Observe only – server knows

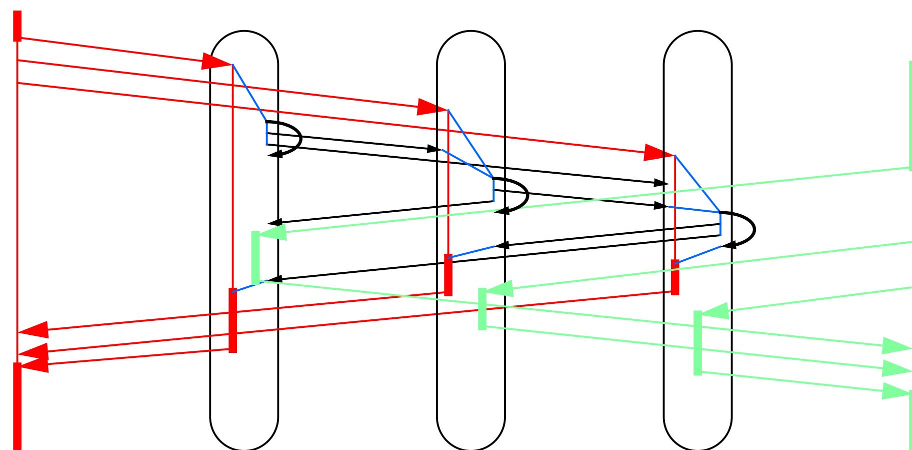


Avoiding the ordering protocol has reduced the cost of each invocation. The number of messages has reduced from 15 to six, the number of evaluations has remained the same at three, and the latency has reduced from a five message path down to a two message path.

2.3.2.3 Update concurrent with observation

If an updating operation is invoked concurrently with an observe only operation, and the observation is allowed to proceed immediately, there is a conflict as illustrated in figure 2.4.

Figure 2.4: Conflict between observe and update



In this case, one member of the group processes the observation before the update, the other members process it after. In the absence of a better dependence relation between these operations, the observation must be assumed to be dependent upon the update. If the outcome of the observation is dependent upon the state updated, then the responses will differ.

If the client has a collation policy that requires that all outcomes be identical then the collation will fail. There are other possible collation policies that would succeed.

For the case illustrated in figure 2.4, a majority policy would succeed. A majority policy would not always succeed for a server group of this kind. If two updates took place, the observation might deliver three different results.

If the group members were to keep a counter of updates since the group was formed, and the current value was delivered with the response, the client would be able to use this additional information when collating the results. The client would know which responses ought to match, and which were derived from the most recent state.

This is moving towards a multi-version protocol but with the version information is used in the collation, rather than having multiple versions at the server, and having the client identify which versions are acceptable.

2.3.2.4 *Effect of failures*

If a group member has failed, and the group has not yet reformed, the client invoking an observe only operation will receive fewer responses than it was expecting. Assuming an RPC protocol like REX, the client will be informed of a communication failure for one of the attempted invocations. The client can have a collation policy that allows it to proceed in this case. If the client's collation policy can cope with effects of a concurrent update described in §2.3.2.3, then there is little additional work to be done.

The client does not necessarily have any means to inform the group mechanisms that it failed to receive a response from one group member. This means that the client cannot initiate any kind of reformation of the group. Although there is some suspicion that the client's view of the membership is out of date, the client cannot do anything about this until the group itself resolves the problem of the failed member.

Provided that the invocation is using a conventional RPC mechanism like REX, individual lost messages will be dealt with by retransmission. If the client expects the group to preserve its own consistency, then it need not attempt to ensure that all its requests were delivered, provided that it receives enough responses for its collation.

If the RPC protocol allows the maximum number of retransmissions to be specified, then it is tempting to minimise the effort expended by the RPC mechanism and retry at a higher level just enough to achieve the collation requirement. The problem with this idea is that the "at most once" guarantee provided by typical RPC protocols will no longer give the matching property at the application level.

Since read-only operations have no effect on the future of the object, the "at most once" guarantee in not necessary for such operations. The problem is that it is the client that needs to know that the operation is read-only if it is to exploit this property, and the assumption for this case was that the server has

this knowledge but the client does not. It will be possible to consider this approach when knowledge can be propagated to the client.

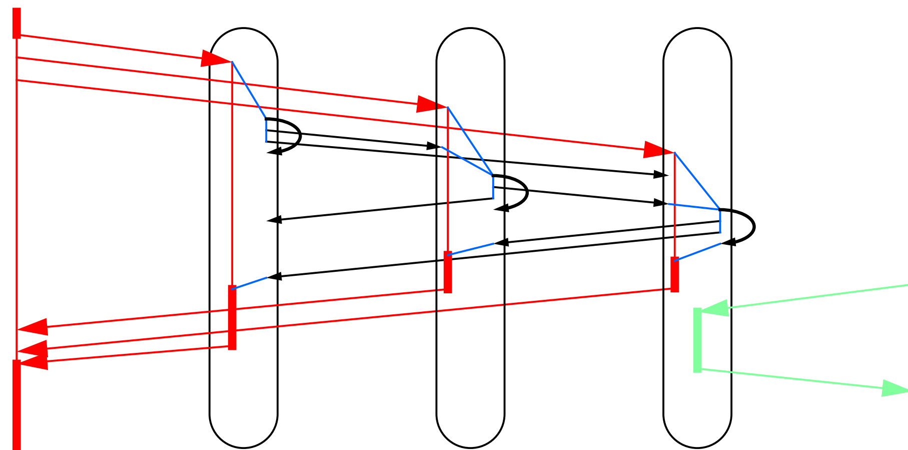
2.3.3 Exploiting knowledge at both client and server

As became apparent at the end of the last section, there are benefits to be gained by propagating knowledge about the operations to the clients that may invoke them. This section considers some of the benefits that might be gained by propagating the knowledge that certain operations are read-only.

2.3.3.1 Any one will do

If both client and server know that the operation does not update state, then the client can invoke only one member of the group without violating the consistency of the group. Being read only means that the future of the group member that performs the operation is the same as the futures of the members that do not. The way this knowledge is exploited is illustrated in figure 2.5.

Figure 2.5: Client and server know operation is observe only



The cost of the observation has now been reduced to two messages, one evaluation, and a message latency of two. This is the same cost as invoking a singleton server.

The server must be prepared for invocations of observations on individual members. In particular, it would cause problems if it were to add the request to the schedule to be processed through the quorum and ordering protocol, and act as if the client had sent messages to all members but the others had been lost.

In this case, only one member of the group is prcessing the request, so the group is no longer behaving as an active replica group for observe only operations.

This strategy is similar to a special case of the quorum consensus protocol [HERLIHY86] where the (initial and final) quorum for all the updating operations is the size of the group and the (initial) quorum for the read-only operations is one. This quorum assignment is consistent with the assumed dependency relation - all operations depend on all updating operations.

2.3.3.2 *In the event of a failure*

If a group member has become unavailable, but the client invokes a non-faulty member, then the client will receive its response as normal and will not be affected by the failure.

Provided that the invocation is using a conventional RPC mechanism like REX, individual lost messages can be dealt with by retransmission.

If a group member has become unavailable and that is the member invoked by the client, then it will receive no response, no matter how hard an underlying RPC mechanism has tried. In this case, the client can try invoking a different member of the group.

Since the client knows that the operation is read only, it also knows that the server operation may be invoked multiple times with no adverse effects. In this case there is no problem in specifiying a minimal effort RPC and invoking the group members in turn until a response is received.

2.3.3.3 *However many the client likes*

Invoking only one group member gives the least cost when there are no failures. In the event of a failure, there will be a delay if the client waits and retries when the response has not arrived after some time.

For an observe only operation, the client may invoke any number of members without affecting the consistency of the group. This gives a direct trade-off between the cost of normal operation and the delay in the event of a failure, whether of a group member, or just the loss of a message.

2.3.3.4 *Detecting value failures*

Invoking a single member implies that the quorum for collation by the client is one. The disadvatage of this is that the scope for detecting value failures is limited to any redundancy there may be in the response delivered by an individual server group member. If a server group member gets out of step with the rest of the group, for example by processing updates in the wrong order, then it may deliver a plausible but incorrect response.

If the client requires a larger quorum then it may invoke enough group members to establish the quorum. The client may still guard against delay in the event of a failure by invoking more than the collation quorum – provided that the quorum is less than the size of the group.

This will introduce the problem of observing results that depend on updates that are being processed concurrently as described in §2.3.2.3.

2.3.4 Groups as clients

The discussion above has considered singleton clients invoking group servers and has shown that there is scope for significant savings if knowledge of the kind of access to mutable state is made available.

When a computation is being performed by an active replica group, that group may invoke an operation. The replication mechanisms must coordinate the activities so as to achieve the correct effects in these cases.

All of the issues involved in invoking a group server still apply, but some additional issues are introduced.

2.3.4.1 Client end issues

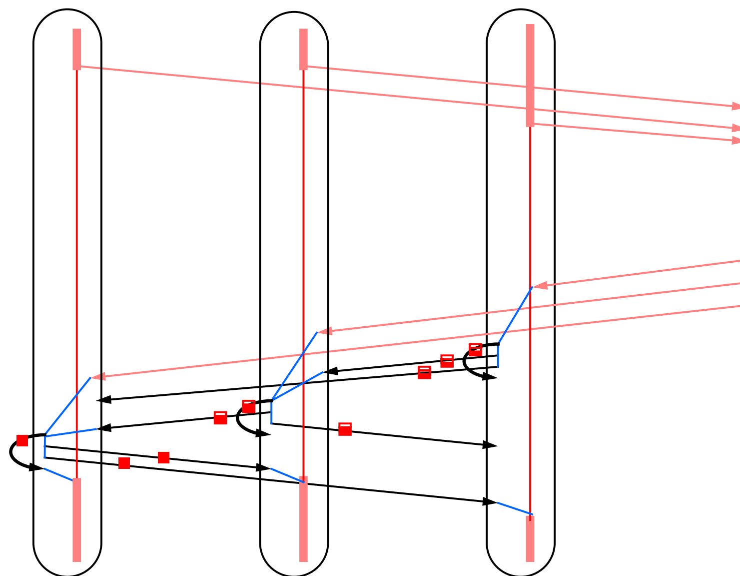
If the future of the client depends on the response to an invocation, then all members of a client group must receive the same response. Clients usually depend on the results of read-only operations, since otherwise the invocation may be omitted completely without affecting either client or server. It is more common for updating operations to deliver nothing in their result other than the indication that they have completed, and in these cases, the client depends only on the existence of the response.

GEX will put requests from group clients into the evaluation order only when a quorum of requests has arrived and been collated by the server group member that holds the token at the time.

The GEX infrastructure for client groups uses the same quorum and ordering protocol for responses to clients as for requests to servers. It can be argued that this is not necessary, but it does allow clients to request copies of responses from other group members rather than the server. If the token protocol is extended with checksums as described in §2.3.1.4, then the consistency of the responses can be verified.

An invocation by a group client is illustrated in figure 2.6. The server in this case is shown as a singleton in order to simplify the diagram.

Figure 2.6: Group client invoking singleton server



There seems to be less scope for exploiting knowledge of mutability for invocations from client groups. Even if the operation is a pure update that delivers no result values, or the clients ignore any values in the result, there is a synchronisation effect to be considered. Attempting to devise a scheme where only one client makes the request also raises the difficult question of how the requester is to be chosen.

2.3.4.2 Additional constraints

The requirement that all client members receive the same response imposes some constraints on the the ideas for reducing the cost of invocations of server groups that were presented above.

Where the client is a singleton, there is a single point where the decision about which server members to invoke can be made. If the client groups members send requests to fewer than the whole membership of a server group, do they have to pick the same set of members?

If the server group members collate the incoming requests then they must receive enough requests to do the collation. For any quorum larger than one, if the client members do not make a consistent choice of server members, then there can be a server member that receives requests but not a quorum. If the server group members treat incoming requests for read-only operations as distinct invocations, then the evaluations may interleave with updates and the client members receive different responses.

This suggests that client group members must make a consistent choice of server group members, and that the server group members must collate the incoming requests from a group and give the same response to all members of the client group.

If all the responses are received correctly, and all the client group members apply the same collation policy, then all client group members will use the same result values, even if different server members processed the invocations in different orders with respect to concurrent updates - as in the example in §2.3.2.3.

There may be savings that can be made in this case, but there are many more factors to consider than for singleton clients, and a thorough analysis is required to determine the failure properties.

2.3.5 Exploiting distributed database research

The problems of exploiting knowledge in order to reduce the cost of providing and using arbitrary replicated services are very similar to those studies in the research on replicated and distributed databases discussed in §2.2. This is particularly true of the later work which studies atomic abstract data types.

Combining the results of that research with the experience of implementing active replication that is transparent to the application programmer, is a promising area for further study.

2.4 Passive replication

Passive replication has not been studied in detail in ANSA, but there are some basic issues which can be seen from the discussion of active replication above. One of the features of suggestions for more efficient mechanisms is that fewer members of the group do the work. This is tending towards an approach that combines features of active and passive replication.

The conditions that require that updates be performed one at a time and in order are as applicable to passive replica groups as to active replica groups. This means that it is not acceptable, in general, for different members of a passive replica group to perform updates and then attempt to update the state of the rest of the group.

The conclusion is that there must be a distinguished member of the group that performs the updating operations in some order that it chooses by itself, and then propagates the state changes to the other members. This distinguised member must receive all updating operation requests, either directly or forwarded by other group members.

Read-only operations may be performed by any group member. The situation is essentially the same as for active replica groups; if clients are aware that the operation is read-only, they may invoke any member of the group without adverse effects, and the invoked member may process the request and reply without consulting the other group members.

If being the distinguished member circulates around the group then the behaviour of the group is very similar to the active replica groups described above. There must be a similar means for passing responsibility, but in this case, the distinguished member would pass on the state change caused by the next operation rather than just an indication that it is the next operation to be performed.

If the distinguished member is fixed, then this information could be passed to the clients, so that they could invoke only that member for updating operations, this reducing the cost of such operations. The problem with this approach is that dealing with failure of the distinguished member becomes more difficult.

The most significant apparent benefit of passive replica groups over active replica groups is that nested invocations are made by a single member. This means that nested invocations of read-only operations can exploit this knowledge as described above.

Suppose that the distinguished member fails after invoking a nested updating operation, but before updating the state of the other members, or before replying to the initial invoker. The typical recovery strategy is for some other member to take over as the distinguished member, and perform the failed operation again. This will cause the nested update to be performed again, unless there is some mechanism to defend against this possibility. Either there must be some atomicity mechanism to undo the effect of the nested update, or it must be possible to recognise the second invocation, which comes from a different source, as being a retry of an earlier invocation. In the latter case it must be possible to reproduce the outcome of the invocation without unwanted repetition of its effects. This raises the question of how long such outcomes should be retained.

Knowledge of the effects of operations provided by passive replica groups can be exploited to reduce the cost of invoking them. The discussion above suggests that this may affect the relative merits of active and passive replication for particular purposes.

2.5 The ANSA atomic activity model

The ANSA atomic activity model [WARNE 93] is based on nested transactions and a nested two phase locking protocol. It makes the conventional distinction between reads and writes, and locks are acquired in either read mode or write mode. Further work is being done on an extended transaction framework [WARNE 94], and current indications are that obtaining and propagating knowledge of mutability will also be relevant to this extended framework.

Part of the work on the atomic activity model was to determine whether or not the need for locks, and the kind of lock needed, could be deduced from application code as part of a process that automatically added the atomicity infrastructure mechanisms where they were needed. The results of that work indicate that this is feasible with appropriate language support, although the

applicability of the technique to commonly used languages is still an open question.

The main benefits of knowing what kind of access to mutable state is involved have been summarised in table 2.1.

The importance of knowing which operations do not access mutable state in any way became clear during the prototyping of the transformation system. The lack of such information made it necessary to invoke operations such as adding integers, with all the additional mechanisms that were needed to keep track of locks, and the mechanisms that would be called upon to revert the state in the event of an abort. Although it would be straightforward to provide special case treatment of types built-in to a language, there is significant scope for constructing applications in such a way that significant parts contain no mutable state.

2.6 Conclusion of initial study

This initial study of the exploitation of knowledge of mutability in transaction systems, and replicated services, suggests that there is significant scope for reducing the cost of the mechanisms.

Research on this topic has gone beyond the distinction of read and write operations, but there are many questions to be answered in understanding how to exploit that research.

Even with the simple distinction between read-only and updating operations, there is scope for substantial benefit, but much still to be done to discover how to realise that benefit.

The next stage of the work will be to apply these ideas to a small scale but realistic example. This will reveal the practical issues in realising these benefits, and act as an example to demonstrate what can be done.

3 Exploiting knowledge of mutability

Note: Estimated timescale for writing: Start April 1994, End July 1994

4 Analysis of failure characteristics

Note: Estimated timescale for writing: Start August 1994, End September 1994

5 Automation and language issues

Note: Estimated timescale for writing: Start October 1994, End January 1995

6 Notes

This chapter is a collection of notes of general relevance to the exploration of mutability and its role in the design and implementation of dependable systems.

6.1 Background

Any mechanism for dependability must be based on redundancy. Redundancy is necessary in order to detect deviations from the requirements, and it is also necessary in order to take corrective action.

The redundancy may be in processing, in storage, and in communication; it also has time, space and value characteristics. These various factors can be combined in different ways to provide a range of dependability mechanisms with different characteristics.

Mechanisms proposed for dependability tend to involve replicating state, replicating the processing of state, replicating communication, or some combination of these. Replicating of processing tends to imply replicating of state, but the strength of this link needs to be investigated.

6.1.1 Transaction systems

Transaction systems have a mechanism to recover to a consistent state in the event that the original goal cannot be reached. A typical ACID system will revert to the old state if the transaction aborts. A simple strategy is for there to be two copies of the state while the transaction is in progress, and for the work to be done on one of the copies. If the transaction commits, the modified state becomes the final state, if the transaction aborts, the original state becomes the final state.

The abstract view of the state is that it is the specification of the observable future behaviours. The clients of the transaction system have some token which can be used to provoke a response; this token is an invocation name in ANSA naming model terminology. That name is bound to some resource that holds, perhaps indirectly, a representation of the state.

If there are two copies of the state with one being modified, there must be two sets of resources used to represent those states. A common strategy is for one set of resources to be used to represent the state both before and after the transaction. This simplifies resource management since it avoids the need to claim a new set of persistent resources for each transaction, and recycle the old set of resources when the transaction completes.

If this common strategy is adopted, there is a permanent set of resources used for the initial and final states, and a transient set of resources used while a transaction is in progress. Either set of resources may be used as the working copy. If the transient copy is used as the working copy, the permanent copy will need to be updated on commit. If the permanent copy is used as the working

copy, it will need to be restored from the transient copy in the event of an abort. Both strategies are valid but have different resource usage profiles.

It is only the state that might be updated that need be copied. It would even be possible to arrange that the state is copied only if it is actually updated - similar to the copy-on-write that is supported by some virtual memory systems. In either case, only the mutable state need have copying mechanisms.

Operations that do not update state can proceed concurrently without violating serializability - the distinction between read and write locks is a well known strategy for exploiting this.

6.1.2 Groups

6.1.2.1 Passive replica groups

The general idea here is that one member does the work, and then updates the state of the other members.

There must be a copy of all of the state for each member, but only the mutable state need be considered for transfer.

If no state is updated then there is no need to do anything to the other members.

6.1.2.2 Active replica groups

In this case, all members do the work.

There must be a copy of all of the state for each member, and the usual assumption is that each member updates its own state, and there is no explicit state consistency mainenance. Inconsistencies may eventually be detected by collation; either of outcomes from the group, or of its invocations of other services (which may themselves be groups).

If the group does its own collation then it is in a position to initiate recovery action without involving other parties.

If an invoked service detects an inconsistency, it will apply some policy of its own to resolve the problem. If its collation policy allows it to construct a collated value from inconsistent inputs (e.g. majority) then it can proceed and can return a response which may, or may not, include a report that it detected an inconsistency. If a collated value cannot be constructed then the invocation cannot proceed and an error response is appropriate. Such an error response may, or may not, report the inconsistency.

If an invoker detects an inconsistent response from a service provided by a group, it applies some policy to choose how to proceed.Whether or not it can construct a collated value, it may choose to proceed without informing the server that there was a problem. For the client to inform the server of the problem, it must be provided with the means to do so. If the client has only the normal service interface, then no such facility is available. Thus there is a question of what ought to be in the interfaces provided to a client collator by group members for reporting of detected failures.

Even if the inconsistency is detected and reported to the group (specifically something that is responsible for restoring the group to a consistent state), there is no guarantee that the cause of the fault can be traced. The fault may have been dormant within the group for an arbitrarily long time.

Active replica groups could be built with a more aggressive consistency policy. Where information about ordering of pending invocations is already circulating, there is an opportunity to circulate information about the state of the members for checking by an internal consistency mechanism.

At first it appears that mutability has little impact on normal operation, but the serialization of invocations need not be so strict for operations that do not update state. Such invocations can be allowed to proceed in arbitrary orders at different members without any externally visible differences.

The consequence of this is that the ordering protocol is not needed for observe only operations to protect the state of the group. If the ordering protocol is not used, the responses from group members may differ

6.1.2.3 *Something in between*

Operations that do not update state need not be performed on all members of a group. Such operations can be performed with the group acting as a passive group, with updating operations being performed as an active group.

For this to be workable, the client must be capable of accepting a single response despite its multiple requests. Unlike the other options, it is not clear how this can be described using the concepts in the ANSA Computational Model. The level at which the replication is transparent can be described, as can the very detailed level of the mechanisms. For this strategy, there appears to be an level of abstraction between these which cannot be described in terms of tree structured activities and invocations.

If the client sends the request to all members but only one will process it, the group itself needs to have some policy for picking the member that will respond.

A further refinement is possible if the client is aware that the operation is non-updating. In this case it need send its request only to one member. This requires that the group be capable of dealing with requests that arrive at only one member.

The absence of collating in these cases means that the non-updating operation can be performed by the chosen member as soon as its local concurrency control permits. It need not be related to any updating operations that are in the process of being ordered by the group. Performing a non-updating operation cannot cause the state of the groupt to become inconsistent. For the request to have arrived, its activity must be concurrent with all activities which have outstanding invocations, and in the absence of any guarantees of relative progress of activities, it could have arrived at a time that would cause it to be inserted into the processing order in the place where it is actually performed.

Note: There needs to be a discussion about the absence of some sort of generalisation of a pipeline effect. It seems to be a common assumption that an invocation initiated in some non-blocking way by an activity must be done before another invocation initiated later by the same activity. It is usually unclear what it means for the invocation to be 'done' when this assumption is made, it sometimes implies completion, and sometimes implies some inadequately specified notion of 'started'.

There is also the possibility of the client invoking some arbitrary subset of the members. Is it possible for the server members to perform the requests immediately, and for a collation mechanism to resolve any inconsistency that arises from some, but not all of the server members having already performed an updating operation. If each member counts updating operations, the count

could be used to determine which responses should contain the same values, and which reflect the latest state.

6.1.2.4 *A common issue*

When a new member is added to a group, of whatever kind, that member needs to be brought up to the current state of the group. Once again, it is the mutable state that is significant.

6.1.2.5 *Other issues for active replica groups*

One of the things you get for 'free' from GEX-like token passing is the ability to insulate the group from client crashes which result in partially received invocations. This only matters for updates - another reason for not bothering with the token passing, quorum and ordering protocol for pure observations.

6.2 Possible benefits

All of the foregoing has been hinting at the idea that being able to identify mutable state, and operations that update it, can make it possible to use cheaper mechanisms for non-updating operations.

It seems likely that distinguishing

- operations that update state
- operations that observe mutable state
- operations that neither update nor observe mutable state

would make it possible to exploit simpler mechanisms.

It is not yet clear whether or not there is a significant difference between direct and indirect access to mutable state.

Design and implementation strategies to exploit this knowledge are addressed below.

6.3 Relationship to previous work

6.3.1 Atomicity infrastructure

The work on atomicity transformations included some simple exploitation of information about mutability of state. This was done in the context of a language that explicitly distinguished between mutable and immutable state, thus making the job easier than it would be in other languages.

In the course of that work it became clear that it would be a significant advantage to know which operations did not make any kind of access to mutable state. Making such information available requires further work on the type system.

6.3.2 Groups

The mechanisms to bring new member states up to the current state for the group were based on the mechanisms that had been developed for passivation and activation of objects. This mechanism is driven by programmer declarations, and has significant tool support although if falls short of the fully automated analysis approach prototyped in the atomicity work.

6.4 Other issues

6.4.1 Sensors and actuators

These are entities, interfaces, or perhaps most suitably, operations. There is some element of uncontrollable mutability about them. The significant issues seem to be that the ‘state’ behind a sensor changes without reference to the computation under consideration, and once an actuator has been invoked, state changes go out of the control of the computation. This is starting to look like mutability across a federation boundary, so the similarities and differences need to be explored.

This is related to the “hidden channel syndrome” where actuators affect sensor results through effects on the environment that are not visible in an analysis of the program.

6.5 Programmer tools

Suppose immutability is the thing to look out for; what do you tell the programmers?

Given that you know you want immutability, how do you check that you have achieved it?

How does this relate to detecting where you have immutability and exploiting it to select a cheaper mechanism? It seems likely that the analysis would be common to both of these - which makes the question of being able to do the analysis doubly important.

6.6 Activity identification

GEX will deadlock if a group attempts to invoke itself - it does not have the information to recognise that the invocation is nested, and will therefore treat it as a concurrent incoming request which must wait until the active invocation completes.

Is this also true of ANSAware? Will a single task ANSAware server deadlock if it attempts to invoke itself?

Some of the work is about ensuring that things are done so that they appear to have been done in the same order.

Where access is indirect, does this have any impact on requirements to identify activities?

6.7 Failure model

How is this work related to the failure model?

Is mutability related to a particular class of expectations or how expectations [do not] change over time?

Use the failure model to explore the real impact of proposed alternative mechanisms. What additional failure modes are introduced, what assumprions are being made.

6.8 Link to scenario

The principle of designing the system to isolate the mutable state has already been applied to the scenario.

Where does this get written up? How is this design better than the alternatives.

6.9 Link to engineering

Some of this work will propose alternative protocols, and related processing strategies. Where is the boundary between programming and engineering, does it matter provided we do not leave a gap?

6.10 Type issues

Propagation of mutability characteristics: how does invoking a bound-in interface compare with invoking an interface passed as a parameter. Can an operation adopt the mutability characteristic of its argument(s) so it becomes a per-invocation issue?

Does this make the type inferencing too hard? Fallback position is to lose information in a conformance relation. Operation is seen as the worse case, and the immutability of operations in the argument is ignored.

Similar issue for returning results.

The scenario is starting to suggest that indirect updates can also be made cheaper.

References

[EDWARDS94]

Edwards, N. J., Rees, R. T. O, "A Model for Failures in Distributed Systems", APM.1143.01, March, 1994, APM Ltd, Cambrigde, UK.

[GARCIA-MOLINA 82]

Garcia-Molina, H., Wiederhold, G., "Read-Only Transactions in a Distributed Database", ACM Transactions on Database Systems, Vol. 7, No. 2, June 1982, Pages 209-234.

[HERLIHY86]

Herlihy, M., "A Quorum-Consensus Replication Method for Abstract Data Types", ACM Transactions on Computer Systems, Vol. 4 No. 1, February 1986, Pages 32-53.

[HERLIHY87a]

Herlihy, M., "Extending Multiversion Time-Stamping Protocols to Exploit Type Information", IEEE Transactions on Computers, Vol. C-36 No. 4, April 1987, Pages 443-448.

[HERLIHY87b]

Herlihy, M., "Concurency versus availability: Atomicity mechanisms for Replicated Data", ACM Transactions on Computer Systems, Vol. 5 No. 3, August 1987, Pages 247-274.

[HERLIHY 91]

Herlihy, M.,Weihl, W. E., "Hybrid Concurrency Control for Abstract Data Types", Journal of Computer and System Sciences, Vol. 43, No. 1, August 1991, Pages 25-61.

[OSKIEWICZ 93]

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

[OSKIEWICZ 94]

Oskiewicz, E., Edwards, N.J., "IPS - An Information Publishing System", APM.1171.00.05, APril 1994, APM Ltd., Cambridge U.K.

[WARNE 93]

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

[WEIHL87]

Weihl, W. E., "Distributed Version Management for Read-Only Actions", IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, January 1987, Pages 55-64.

[WEIHL89]

Weihl, W. E., “Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types”, ACM Transactions on Programming Languages and Systems, Vol 11 No. 2, April 1989, Pages 249-282.

[WARNE 94]

Warne, J. P., “Extended Transaction Framework: Technical Overview”, APM.1060.00.02, February 1994, APM Ltd, Cambridge UK.