Workshop On Distributed Computing

S.K. Shrivastava

Abstract
This report is a record of the second Newcastle-Rennes workshop on distributed computing held from 25th to 28th March, 1985 at Linden Hall, Longhorsley, (near Newcastle upon Tyne). About half of the participants were from Newcastle and Rennes, and the rest were invited from various European research establishments. Topics covered included operating systems, reliability, programming, communication primitives and architecture.

Series Editor: M. J. Elphick

© 1985 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details


Second Newcastle-Rennes Workshop on Distributed Computing.
Edited by S.K. Shrivastava.

Newcastle upon Tyne: University of Newcastle upon Tyne,
Computing Laboratory, 1985.

(University of Newcastle upon Tyne, Computing Laboratory,
Technical Report Series, no. 203.) Workshop held 25th - 28th

Added entries

SHRIVASTAVA, Santosh Kumar.
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

This report is a record of the second Newcastle-Rennes workshop on
distributed computing held from 25th to 28th March, 1985 at Linden
Hall, Longhorsley, (near Newcastle upon Tyne). About half of the
Participants were from Newcastle and Rennes, and the rest were
invited from various European research establishments. Topics covered
included operating systems, reliability, programming, communication
primitives and architecture.

About the author

Dr. S.K. Shrivastava joined the Computing Laboratory in August 1975,
where he is a lecturer.

Suggested keywords

ARCHITECTURE
COMMUNICATION
DISTRIBUTED SYSTEMS
FAULT-TOLERANCE
OPERATING SYSTEMS
RELIABILITY

Suggested classmarks (primary classmark underlined)

Dewey (18th): 001.64404
U.D.C. 519.687
PREFACE

The second Newcastle-Rennes workshop on Distributed Computing was held from 25th to 28th March 1985. The workshop was a follow-up to the successful event on the same topic held from 24th to 28th September 1983 at Mont St. Michel. These workshops form part of an EEC sponsored programme of scientific collaboration between the Computing Laboratory University of Newcastle upon Tyne and the Institut de Recherche en Informatique et Systemes Aleatoires (IRISA) which is a laboratory jointly managed by INRIA, CNRS, Universite de Rennes and INSA (Institut National des Sciences Appliquees de Rennes). The programme is jointly managed by Professor B. Randell (Newcastle) and Professor J.P. Banatre (Rennes).

The workshop was held at Linden Hall, Longhorsley, about 15 miles North of Newcastle. It is a magnificent Georgian Country House set within 300 acres of park and woodland. It provided an ideal setting for the workshop. In addition to Newcastle and Rennes people (numbering 15), thirteen more European research workers, both from industrial and academic research establishments, took part in the workshop.

Like the previous workshop at Mont St. Michel, the sessions at the workshop were intended for discussions on topics on common interests rather than presentations of papers. The session topics were: operating systems, reliability and fault tolerance, programming, communication primitives, the Gothic project and architecture. Due to the informal nature of the workshop, all sessions generated very useful and enjoyable discussions and served the important purpose of bringing the participants up to date with the latest research work on distributed systems in Europe's foremost research establishments. This report summarizes the contributions of the participants.

While I was nominally in charge of the organization for the workshop, I received invaluable help from my colleagues, in particular, Drs. Fabio Panzieri (Newcastle) and Michel Banatre (Rennes). The secretarial work of the workshop was in the capable hands of Carol Reynolds.

S.K. Shrivastava
# LIST OF PARTICIPANTS

### 1. Newcastle:
- B. Randell
- S.K. Shrivastava
- L.F. Marshall
- T. Anderson
- A.S.J. Linton
- S. Pfleger
- P. Ethilchelvan
- F. Panzieri
- R.J. Stroud

Computing Laboratory
Claremont Road
The University
Newcastle upon Tyne
NE1 7RU
ENGLAND

### 2. Rennes:
- J.P. Banatre
- M. Banatre
- M. Raynal
- R. Mari
- M. Jegado
- F. Ployette

IRISA/INSA
Universite de Rennes
35042 Rennes Cedex
FRANCE

### 3. Others:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Lister</td>
<td>Department of Computing Science</td>
</tr>
<tr>
<td></td>
<td>University of York</td>
</tr>
<tr>
<td></td>
<td>Heslington</td>
</tr>
<tr>
<td></td>
<td>YORK YO1 5DD</td>
</tr>
<tr>
<td>K.H. Bennett</td>
<td>Department of Computer Science</td>
</tr>
<tr>
<td></td>
<td>University of Keele</td>
</tr>
<tr>
<td></td>
<td>Keele STAFFS</td>
</tr>
<tr>
<td></td>
<td>ST5 5BG</td>
</tr>
<tr>
<td>Y. Paker</td>
<td>School of Engineering Science</td>
</tr>
<tr>
<td></td>
<td>Polytechnic of Central London</td>
</tr>
<tr>
<td></td>
<td>115 New Cavendish Street</td>
</tr>
<tr>
<td></td>
<td>LONDON W1</td>
</tr>
<tr>
<td>Y. Deswarte</td>
<td>CNRS</td>
</tr>
<tr>
<td></td>
<td>7 Avenue du Colonel Roche</td>
</tr>
<tr>
<td></td>
<td>31077 Toulouse Cedex</td>
</tr>
<tr>
<td></td>
<td>FRANCE</td>
</tr>
<tr>
<td>M. Seifert</td>
<td>IBM Scientific Centre</td>
</tr>
<tr>
<td></td>
<td>Tiergartenstrasse 15</td>
</tr>
<tr>
<td></td>
<td>6900 Heidelberg</td>
</tr>
<tr>
<td></td>
<td>WEST GERMANY</td>
</tr>
<tr>
<td>M. Varneresi</td>
<td>Istituto di Scienze de’Informazione</td>
</tr>
<tr>
<td></td>
<td>Universita degli Studi di Pisa</td>
</tr>
<tr>
<td></td>
<td>Corso Italia 40, 56100 Pisa</td>
</tr>
<tr>
<td></td>
<td>ITALY</td>
</tr>
</tbody>
</table>

J.M. Foster and S. Wiseman
Royal Signals & Radar Establishment
St Andrews Road
Malvern
WORCS
WR14 3PS

D. Brownridge
High Level Hardware
PO Box 170
Windmill Road
OXFORD OX3 7BN

J. Andre
CNET
Route de Tregastel
BP 40 22301 Lannion Cedex
FRANCE

G. Wendome
Centre de Recherche Bull
c/o IMAG BP 66
38402 St. Martin D’Herres Cedex
FRANCE

E. Best and E. Mott
GMD/ISF
Postfach 1240
Schloss Birlenghoven D-5205
WEST GERMANY
# Table of Contents

**Operating Systems**

1. Topology Updation in UNIX United Internetworks  
   K.H. Bennett  
   Page No. 1

2. Heterogeneous Transparently Distributed Systems  
   L.F. Marshall and B. Randell  
   Page No. 4

3. Wormos - Self Managing Distributed Processes  
   Y. Fakar et al  
   Page No. 7

4. Galaxie: A Distributed Operating System  
   J.P. Andre et al  
   Page No. 10

5. Garbage Collection in Distributed Systems  
   S. Wiseman  
   Page No. 16

**Reliability and Fault Tolerance**

6. Reliability Analysis of a Fault Tolerant System with Shared Pools of Spares  
   R.A. Marie  
   Page No. 20

7. PROFEMO - The Main Goals and the Implications on its Design  
   E. Nett  
   Page No. 24

8. Foundations of Self Reconfiguration in NMR Systems  
   F. Eschelchavan  
   Page No. 29

9. Fault Tolerant Levels of Execution in Distributed Applications  
   S. Pfleger  
   Page No. 31

**Programming**

10. Concurrent Programming and Decentralized Systems  
    M. Vanneschi  
    Page No. 34

11. Some Aspects of LC³, a Language for Parallel Programming  
    J.P. Banatre and P. le Carten  
    Page No. 37

12. Concurrent Behaviour: Sequences, Partial Orders and Programming Languages  
    E. Best  
    Page No. 40

13. A Distributed Algorithm to Prevent Mutual Drift Between N Logical Clocks  
    M. Raynal  
    Page No. 43

14. Towards a Better Understanding of Distributed Algorithms  
    M. Raynal  
    Page No. 50
Communication Primitives

15. Communication Primitives in Distributed Systems
   A. Lister
   58

16. Remote Service Request and Object Sharing with Merkur
   M. Seifert
   61

17. A Remote Procedure Call Mechanism Supporting Orphan
    Detection and Killing
   F. Panzieri and S.K. Shrivastava
   65

Gothic Project

18. About the GOTHIC PROJECT
   J.P. Banatre
   69

19. A note on the design of a stable storage facility
    for GOTHIC
   M. Banatre and F. Ployette
   70

20. Proposal for the Design of a Stable Object Manager
    N. Jegado
   74

21. A Note on the Nesting of Activities
    J.P. Banatre et al
   78

Architecture

22. Workstations in Distributed Systems
    D. Brownbridge
   82

23. Flex: ESRE's capability computer and its remote capabilities
    J.M. Foster
   84

24. Introduction to the "Saturne" Project - A Fault-
    and Intrusion-Tolerant Distributed System
    Y. Deswarte et al
   87

25. Distributed Academic Computing
    M. Seifert
   91

26. The ROSE and CSA Project
    G. Vandome
   94
Topology Updating in UNIX United Internetworks

K. H. Bennett

Department of Computer Science,
University of Keele

1. Introduction

The Newcastle Connection[1] is a software subsystem written at the University of Newcastle upon Tyne, which may be added to each of several stand-alone UNIX [2] systems which are physically interconnected. The resultant system is known as UNIX United.

The construction of UNIX United namespaces from the namespaces of component UNIX (or UNIX United) systems potentially allows very large scale distributed systems to be established relatively easily. The hierarchical namespace is indefinitely extensible, and name clashes must only be resolved within single directories. All naming is relative to one of two standard UNIX contexts, the working directory and the root directory. The implementation of the Connection subsystem will accommodate different network technologies and network interfaces, using the universal datagram service [3].

This offers the potential of allowing the underlying networking to change, without being visible to the user or applications level. For example, a personal workstation may be moved from one site, and be connected to a different network at a remote site. The user of the workstation will still see exactly the same naming hierarchy as before the move.

The problem described in this paper has two major components:

i) How do we permit the underlying network topology to change while keeping the logical (user) namespace constant?

ii) How do we route through the underlying networks at a remote operation?

In section 2, we shall discuss the characteristics of a desirable solution, and explain the assumptions we feel able to make. In section 3, we shall define more carefully the problem itself.

2. Characteristics of solution

2.1 Assumptions

We shall assume that the UNIX United system may be very large, and certainly capable of further expansion. This will involve the addition and removal of hosts, gateways and networks. Any of these three may also fail spontaneously at any time. At the physical level, a UNIX system may be connected to one, or more than one, physical networks. Any single given network will be regarded as fully connected. Matters concerned with routing and congestion within these networks are of no concern to the current study. A UNIX system which connects to more than one network will be termed a gateway. Gateways between networks will always
be UNIX systems running the Newcastle Connection.

2.2 Characteristics of a solution

The principal characteristic of a solution is that it must enable a binding between the namespaces of separate UNIX systems to be set up and subsequently to be maintained in the face of changes to the underlying network topology. This includes the moving of UNIX systems between connection points. Moreover, the binding must be secure against unintended events such as the reception of messages intended for machine X at address Y, when the machine at address Y is now W. Also the binding should be maintained in the face of deliberate attempts at breaching security, such as workstation substitution, although we shall not initially consider this problem.

Any solution must be based on a firm theory of naming, such as that put forward by Saltzer [4].

The solution must work for very large scale UNIX United systems involving many networks, gateways and hosts. Moreover it should permit the extension of the UNIX United system, for example by uniting the namespaces of several existing large UNIX United systems.

Any component UNIX system does not have knowledge of all the other UNIX systems, but only of its name neighbours (at the naming level). It is highly desirable that a component UNIX system retains network addresses which refer to the network(s) to which it is connected. If addresses percolate widely, they are difficult to change, and they are difficult to keep unique. A solution should not assume "global knowledge" or a central coordinator but be based on a distributed algorithms.

The performance of the eventual system, both in terms of the overheads and delays at each remote operation, and at setting up after a move, is a vital characteristic. The rate at which the algorithms can respond to changes, and the resultant congestion, will be important determining factors in the overall performance. We would like to have some indicators for how near optimal the routing is. The fraction of control messages from the routing algorithms is also a useful metric. Finally, in a large system, the size of the routing tables must be kept feasible.

Any solution must clearly identify the various issues involved, and the implementation should clearly separate these issues (example: naming, addressing and routing).

3. The problem revisited

We shall regard the UNIX United system as being constructed from component namespaces; the minimum unit of namespace in which we shall be involved is that residing on one logical disc volume. We shall consider four mappings:

a) A namespace (on one volume) is mounted in another namespace (on another volume). If the first namespace is a root filestore, this is a UNIX United "unite" operation. If it is a standard UNIX filestore subtree then this is a UNIX "mount" operation. A child has only one parent; a parent may have many children in this mapping. A namespace on a volume may also be free (not mounted under either of the above operations; it still has some identity.

b) A volume, containing a namespace, is mounted on a UNIX host. It is not necessarily mounted on the host hardware on which the physical disc is actually spinning (remote mount).
c) A host is mapped on to one or more physical connection points to one or more networks.

d) Networks are interconnected to form internetworks; gateways are always UNIX systems running the Connection. Logically, gateways have no filestore.

This is a somewhat simplified model; for example, details of the mappings are conventionally stored within the filestore. We can now define the problem more carefully: the namespace mapping must be preserved in the face of alterations in the other three mappings. Additionally, the solution must handle the cases in which any or all the three mappings become degenerate. This implies that the mappings must be implemented separately, and not confused.

5. Conclusion

We have stated the assumptions made for the problem of network topology updating, and investigated desirable properties of a solution. Currently we are working on a solution which employs a distributed nameserver to map logical names onto internetwork addresses; a distributed adaptive routing algorithm then routes messages between hosts on an internetwork. No assumption is made that the mapping or routing is correct, so end-to-end authentication is employed.

Acknowledgements

The ideas described in this paper are the result of discussions with several colleagues, especially Pearl Brereton, Paul Singleton and John Grant at Keele, and Prof. Randell at Newcastle. The financial support of the Science and Engineering Research Council is acknowledged.

References


THE UNIVERSITY OF NEWCASTLE UPON TYNE
COMPUTING LABORATORY

Heterogeneous Transparently Distributed Systems

by

L.F. Marshall
S. Randell

Computing Laboratory, University of Newcastle upon Tyne
The term 'transparently distributed system' is normally used for a system which provides to its users and their programs the functionality of a conventional centralized computing system, and so hides the fact that it actually comprises multiple component computing systems, joined together by network links. In many cases such a system mimics the functionality, and is constructed from multiple instances, of some existing centralized system, such as UNIX or VM/CMS, to the convenience both of the users and of the constructors of the system. In other cases, such as the APOLLO DOXAH system, the system, and its functionality are specially designed. However, in each case the system is essentially homogeneous, both in appearance and in construction, even though some component computing systems may have only reduced functionality (e.g. diskless workstations), and there may be special purpose components (e.g. print servers).

The purpose of this brief note is to speculate on the problems of constructing distributed systems from a heterogeneous collection of component computer systems. The conventional (e.g. OSI) approach to the linking of heterogeneous computing systems uses various specialised high-level protocols (e.g. for file transfer, mail, job submission, etc.) This does not provide any network transparency, since the fact that there are multiple component computing systems, remains obvious, and users and/or their programs have to make explicit use of the protocols. Moreover this approach is based on the invention of generalisations of typical system objects, such as files, jobs, devices, etc. Thus the users of each component computing system must be aware of two sets of system objects - the local system-dependent ones, and the generalised ones - even though they need not be aware of the details of the objects supported by other types of component computer.

Our own work on distributed UNIX systems using the Newcastle Connection is based on the use of component interactions at the UNIX system call level, so that just one set of system objects - those supported by the UNIX kernel - is involved in all such interactions. However we have made some modest attempts to relax the requirements for homogeneity amongst the component systems. First of all the Newcastle Connection contains basic provisions for coping with variations between different versions of UNIX. It also, for all data items whose type it can be sure of, copes with differing data representations (e.g. due to word length and byte order differences). However there is a strict limitation to what can be achieved in this regard. This is because the UNIX system call level does not ensure complete type safety. For example, one can store a set of integers in a file, and retrieve them (perhaps accidentally) as if they were floating point numbers. Thus the Newcastle Connection, lacking any information about the types of the data items held in files, cannot make any appropriate changes to their data representations when transmitting a file block to a different type of computer (in other words, the Connection cannot make use of a "presentation-layer" protocol).

More significantly, we have explored the incorporation of non-UNIX systems into a UNIX United system, by mapping their interactions into UNIX system calls. The most extensive experiments (carried out by Dick Snow and Harry Whitfield) have concerned a (non-UNIX) terminal concentrator. From the messages it exchanges with other component systems the terminal concentrator appears to them to be a (very limited) UNIX system. This work takes advantage of two valuable characteristics of the UNIX system call interface, namely the simple yet general
provisions for naming and accessing files and devices, and the fact that there is a standard means of indicating that a system call has not succeeded. Thus the terminal concentrator appears to support a UNIX-like name space, but consisting just of one directory and a set of terminal names, and returns failure exceptions for all but a few of the different types of system call that could be sent to it, possibly in error, by other component computer systems.

However we now realise that, at least in principle, there is no reason to insist on using the same single set of system calls in all interactions between component systems, and, more importantly, no reason to regard the functionality of one particular type of component system as playing a primary role. For example one might construct a transparently distributed computing system out of multiple VMS and UNIX systems by linking them together and arranging for them to interact at their respective system call levels - VMS systems would exchange VMS calls, and UNIX systems would exchange UNIX calls, with mapping only taking place when calls were exchanged between VMS and UNIX systems. To a UNIX user the entire distributed system would be a UNIX United system; to a VMS user it would be, in effect, a VMS United System. One thus would have a transparently distributed computing system which, despite its underlying heterogeneity, provides an essentially homogeneous appearance to each of its users.

This approach is, admittedly, an essentially pragmatic one. And many systems do not possess an easily-intercepted and sanitary system call level, supporting a set of system calls which can readily be mapped into those corresponding to other systems of interest. Moreover many systems have a functionality which would greatly limit the use that could be made from them of the rest of the distributed system (e.g., UNIX could provide a much more complete view of VMS than would be possible from VMS/DOS) or of the services that they could provide to the system. However we would conjecture that, at least for the limited subset of system calls necessary to support the sort of interactions that are being entered for by proposals for standard OSI high level protocols, system call mappings are likely to be feasible for a number of current systems. It is then a matter of taste whether such mappings are attempted directly, between all relevant system pairings, or via one or more "standard" intermediate system call languages. (Our speculation is that it will be easier to produce reasonably complete mappings between various pairs of systems of interest, than to produce a sensible single intermediate language suitable for mapping to and from a wide range of systems).

There still remains the problem that few current systems are inherently type-safe. Thus this approach is likely to be most effective for connecting systems whose processors use similar data representations, and/or for system calls which involve transferring just a single type of data item, such as ASCII characters. Nevertheless we believe that this approach could provide a very worthwhile useful degree of network transparency in many situations. Moreover experiments with the approach should provide useful guidance to recently initiated efforts elsewhere aimed at defining standard protocols, to augment or replace existing OSI high-level protocols, for facilitating the construction of coherent, if not transparently, distributed computing systems.
Self Managing Distributed Processes

Project Team:
Yakup Paker, Rod Ellis, Tim Kindberg, Ali Vahit Sahiner

"No, Mr Sullivan, we can’t stop it! There’s never been a worm with that tough a head or that long a tail! It’s building itself, don’t you understand? Already it’s passed a billion bits and it’s still growing."


Introduction

A Science and Engineering Research Council (SERC) funded project is in progress at the Polytechnic of Central London to develop both practical and theoretical understanding of "self-managing processes" by exploring ideas first experimented by Shoch and Hupp at Xerox PARC, namely the so called "Worm" programme [1].

A "Worm" is simply a computation which lives on one or more machines. "Worm" programmes span machine boundaries and also replicate themselves in idle machines. A "Worm" is composed of multiple "segments" each running on a single machine. These ideas were first tried out in a rich but fairly homogenous computer environment at the Xerox Palo Alto Research Centre including 100 Alto computers interconnected by an Ethernet local network.

The project at PCL is based on an environment including five nodes of Motorola 68000 microcomputers interconnected by means of a Cambridge Ring. One of the nodes is a Unix node containing back-up storage whereas the other nodes consist of bare processors (no file support).

A distributed operating system called WORMOS is being developed to be replicated in all the nodes to support the Worm programmes. The Unix node is seen to provide development
support for the project and secondary storage for file handling. In the other nodes, no Unix support is assumed.

Worm Support Mechanisms

In order to support a Worm programme the following facilities need to be provided:

1. Segment propagation
2. Segment population monitoring
3. Joint decision making

The head of a Worm is assumed to be launched at the Unix node. There are a number of options whereby the growth of a Worm can be achieved. If the nodes are assumed to be in a certain order, the segment propagation could take place according to node availability following the ring order. In this model, each node needs to find the next available node for segment copying. The growth terminates when a specified number of segment population is reached.

A centralised approach could be adopted by maintaining the image of each processor status around the ring at, say, the Unix node. This node, then, could copy the segment of a Worm onto available processors.

A more disorganised solution to the same problem could be for each segment to copy itself onto the first available processor chosen by random trials.

Once a certain segment population is achieved it is necessary that this number is monitored for the duration of a Worm. Thus, if one segment becomes inactive, for example, due to processor failure, a new copy needs to be generated at another free node. This implies a certain amount of failure detection and recovery.

Worm implementation also requires mechanisms of joint decision making. One example is the termination of a Worm. This implies the availability of a voting system. Life cycle of a Worm, then, can be seen as follows:

1. Birth (loading the original head)
2. Growth to maturity (replication of segments)
3. Adult life (normal operation)
4. Death (termination)
During the adult life phase, communication support might be needed between segments.

**WORMOS Layers**

At each node WORMOS will have the following layers:
1. Kernel: implemented as a subset of XINU
2. Inter-Process Communication: Process-to-Process communication via the kernel
3. Existential (machine independent): to support process migration and position-independent communication between processes
4. Worm: to support the Worm programmes

**Application Areas**

Beyond the obvious intellectually stimulating aspect of the Worm concept, the following important application areas are clearly serious candidates for the approach adapted by this project:

- Broadcast environment support: Clock display, broadcast message display, etc.
- Parallel Computation: Exploit multiple processors available around a LAN for parallel computation
- Performance monitoring: Design a special performance Worm which can live permanently in the system to gather performance data
- Maintenance: to periodically unleash a maintenance Worm to exercise different parts of the system for fault finding
- Fault detection: this Worm is activated when a fault occurs to locate the faulty component
- Fault tolerance: Exploit the survival capability of a Worm in the face of a failure

**References**

GALAXIE: A DISTRIBUTED OPERATING SYSTEM


Centre National d' Études des Télécommunications
Route de Tregastel. -- 22301 LANNION (FRANCE)

I. Introduction

GALAXIE is a general purpose Distributed Operating System (DOS). However, as the application to telephone switching systems was in mind, some functions have been enhanced. Yet, the solutions we have adopted are applicable to a large variety of applications.

A telephone switching system may be characterized by three factors:

- **Grade of service**: the overall availability of the system must be very high. No more than a total of two-hour failure is tolerated over a 40-year-life span. However, parts of the system may fail as long as it does not prevent the system from functioning (in a degraded mode).

- **Life span**: an average switching-system life span is likely to be several decades. Thus, we must necessarily cope with the evolution of the service that is offered to the subscribers. Extension and even change of software has to be possible. Maintenance has to be simple.

- **Size**: the software size and the database size is very large (one million statements, tens of mega bytes).

Thus the architect of a DOS for switching systems must strive to reach the following goals:

- **1** - The DOS must be **reliable**. It should also facilitate the design of reliable applications.

- **2** - The DOS should provide mechanisms that allow easy reconfiguration.

- **3** - It should also provide an environment that encourages and enforces well-structured applications, that is highly modular software structures.

The solutions that are to be adopted should be as general as possible so as not to restrict the range of application. This should be the case as the aforementioned goals are desirable features for many applications. Note that the GALAXIE DOS is not intended to address
crucial real-time problems like high-throughput switching because of their performance requirements. These problems should be solved by hardware technology.

II. GALAXIE's Principles

The goals were achieved by focusing on the following concerns:

- Allow dynamic system-driven reconfiguration: when a software or a hardware component fails, the objective is to place the system back into a consistent state. This addresses the reliability goal (Goal 1).

- Protection: prevent an error from spreading. This is done for system errors. This has also to be possible for application errors (Goal 1).

- System language, application language: they must enforce reliable programs and encourage structured programming (Goal 1 and 3)

- Programming rules (Goal 3 - but does help for Goal 1)

- Allow easy application-driven reconfiguration: adding, replacing or deleting components should be easy (Goal 2).

III. Dynamic System-driven Reconfiguration

The system components have to be replicated so that reconfiguration is possible. This advocates for distribution.

One major novelty of GALAXIE lies in its InterProcess Communication (IPC) scheme which implements dynamic system-driven reconfiguration. This IPC is based on the service notion.

Instead of providing IPC between two single processes (the client and the server) as most system does, GALAXIE provides IPC between a set of client processes and a set of processors where server processes may be created.

When a client process generates a request, the system dynamically chooses a processor that is able to support a server process. This choice is based on criteria like processor availability, load balancing, performance and so on. The request is forwarded to a processor where a server process may be created. When the request is serviced, the server process is deleted by the system.

A processor may support several server processes at the same time. The maximum number of server processes for each processor is decided by the application.
This approach has several advantages. First, as server processes are created on demand, there is no idle server process and server creation may be done according to the processor load. We shall see later an additional reason for the dynamic creation of server processes.

Second, if one server processor fails, client requests will be directed automatically to the remaining processors, thereby increasing the system robustness. Note that this scheme includes the common form of one-to-one IPC because the application may decide that there is only one server process at a time. This latter form of IPC is sometimes necessary. For instance, if there is only one printer, there will be only one printer server.

IV. Protection

GALAXIE uses well-known protection techniques. The fundamental principle is based on the closed world system, that is, no action is permissible unless explicitly authorized. This implies the principle of error confinement because it limits the risk that errors do much damage before being detected. The error confinement principle is a major argument for distribution and for a loosely-coupled distributed system (a network) because the failure of one logic function in a given node will not affect any other node.

Let us give a brief description of the protection techniques used in GALAXIE. At the system level, each processor is controlled by an MMU, and the domain protection concept is used. Each system module has access only to its own code and data. There is a distinction between a privileged supervisor state and a nonprivileged application state.

What is more original is how GALAXIE enforces protection at the application level by extending the closed world principle to application programs. All the inter-processor and intra-processor interactions between application software modules (called abilities) have to be explicitly declared to the system. IPCs are allowed only between the abilities that are "linked" by such a declaration. This is the only way abilities may interact with one another because abilities do not share any data or code.

V. Language

The language used at the system level should produce more reliable programs. PASCAL was chosen because of its strong typing (and because of its availability). This is in accordance with the closed world principle as PASCAL restrictions were intended to encourage the development of reliable programs by enforcing a programming discipline. PASCAL makes it difficult to access memory locations outside its data area.
The language used at the application level is an extended PASCAL (that is PASCAL plus some primitives). We thereby combine a powerful high language and operating systems features. Note that the language used at the application level is not an essential element of the GALAXIE DOS as GALAXIE was designed to be compatible with several languages.

VI. Programming Rules

GALAXIE enforces a discipline of programming at the application level. Programs must be structured in abilities. As we have already seen, abilities do not share data or code with one another. All the interactions between abilities have to be explicitly declared.

Abilities are formed with functions which are the entry-points of an ability. The only possible interactions between abilities are through these entry-points. Note that functions are reentrant because reentrant code is a fundamental need in switching systems. This is another reason why servers are dynamically created.

VII. Application-driven Reconfiguration

It should be easy to replace, add or delete the application components. Such actions should not imply the modifications of the whole system specially if it is a very-large-size software. Moreover, these modifications must be done without stopping the system.

Therefore, the places where the interactions between modules are expressed in the program should be as few as possible, and more importantly, their locations should be known.

For instance, a system where an interaction between Module A and B is implicitly declared is unacceptable because it would be very difficult to replace B by C without changing A. Thus, A should not call B by its global name, but by a local name that only designates a partner. The link between this local name and the global name should be declared elsewhere. This local name scheme is a well known addressing technique and the local name is commonly called port.

We have already seen that the interactions were to be explicitly declared to the system for the sake of protection. Now, we know that this interaction is in the form of a link between a local name—the port—belonging to a function of an ability and a function of another ability.

Another major novelty of GALAXIE is that this declaration is the only place where the interaction is declared. To change an interaction, it suffices to declare a new interaction and this can be done dynamically.

Therefore, the application reconfiguration has been facilitated by making the distinction between the writing of application software modules and their configuration. This configuration is done by means of
declaration primitives provided by the GALAXIE DOS.

To prevent the system from being stopped when an application modification is to be done, GALAXIE provides dynamic loading primitives. This possibility is another original feature of GALAXIE. It allows not only modifications without generation, but it also allows easy maintenance by allowing the dynamic addition of spy modules.

VIII. Performance

The service notion used in GALAXIE IPC allows the use of various algorithms for the server choice. For instance, a server-choice algorithm may take into account performance criteria.

GALAXIE offers the datagram service because a more sophisticated communications may be costly and useless as the application may wish to build its own protocol. However, GALAXIE does provide an automatic end-to-end protocol to the application if needed. GALAXIE's communication protocols were structured in a manner close the ISO model. However, some modifications were introduced to the ISO model for performance reasons.

IX. Current Status

GALAXIE is currently available on a network limited to two sites (based on the M68000 microprocessor). The implementation is going on with a more complex network.

X. Expected Insights

Further studies are to be done on the following aspects:

- server-choice algorithms,
- dynamic application-configuration algorithms,
- performance.

The following people have actively taken part in the project: H. Derriennic-Le Corre, P. Desclaude, J.P. Ollivier, J. Proust, M. Serge, P. Stephan, G. Douéin, C. Rougerie.

Publications:


- Dynamic software reconfiguration in a distributed system (GALAXIE), J.P. André, J.C. Petit, H. Derriennic-Le Corre, ICC 82, June 82, Philadelphia.

- Structure of a distributed application software for GALAXIE, J.P. André, R. Kung, J.C. Petit, AFCET, November 82, Lille.


(15)
1. Introduction.

If resources can be dynamically allocated in a system, then the question of garbage collection arises. Explicit deallocation is fine if the resources are used in a fairly static manner, but as complexity rises it becomes safer to use an automatic garbage collector. However, in the past garbage collection, usually of a program’s heap, has been a notoriously slow process. If the same algorithms are used in a distributed system, then performance degrades intolerably, simply because of the scale of the system.

A garbage collector is proposed which is hierarchical in nature. At one level, disjoint areas of the address space are garbage collected independently. These garbage collections, which are performed in parallel with normal processing, do not consider pointers between areas. This is done at the next level in the hierarchy. In distributed systems this hierarchical structure can reflect the geographical organisation of the network.

2. The Memory Organisation.

The memory is viewed as a set of memory blocks, rather than as one large linear array. Each block is an array of words, where the words are either integers or pointers to blocks. A pointer points to a block as a whole; to address an individual word a pointer and an offset must be supplied. This is illustrated by Fig 1.

- Address Space is Divided into Blocks
- Blocks Contain Integers & Pointers to Blocks

![Address Space Diagram]

FIG 1. Blocks and Pointers.

Such memory organisations are found in computers which use capability-based addressing [Fabry74], where the pointers are primitive capabilities, in LISP systems [McCarty60] and Smalltalk systems [Engar74].

The registers of the computer can contain pointers and non-pointers. Pointers stored in registers form the root of the accessible structure that is in memory. Any block that cannot be reached by following a chain of pointers starting at the root can never be accessed. These inaccessible blocks can therefore be recovered and used to make new blocks. It is the responsibility of the garbage collector to identify and recover these inaccessible blocks.

3. Existing Garbage Collection Techniques.

There are many techniques which can be used for garbage collection. [Cohen71] and [Miseman85] give comprehensive surveys. Garbage collectors work either by garbage collecting the entire memory in one operation with normal processing suspended, or by proceeding gradually in pseudo-parallel with normal processing; that is garbage collection "on the fly".
There are two main types of on-the-fly garbage collector. The mark-scan type, first proposed by [McCarthy60] and formally described by [DiJKstra.et.al.72], gradually scans the accessible structure and marks all the blocks that are accessible. The storage occupied by the inaccessible blocks, which are those that are not marked, can then be reclaimed.

The copying type of garbage collector, first proposed by [Hamsay79] and refined by [Cheney79], divides the memory into two spaces. Only one space is used at a time. It is garbage collected by copying all the accessible blocks from it into the other space. Then all of the accessible blocks have been copied from it, the roles of the spaces are reversed. A similar algorithm is used by [Ungar80] in a Smalltalk system.

In a distributed system, stopping the entire system while garbage collecting is clearly unacceptable, so an on-the-fly method is needed. However, both the copying and scanning garbage collectors take a long time to complete a cycle, during which time no garbage is recovered. This means that very large amounts of redundant memory are required to ensure that the system does not run out of space.

Another technique often proposed for garbage collection is Reference Counting. Here a count is kept with each block of the number of pointers that refer to it. A block is deallocated if its reference count ever drops to zero. The main drawback of this method is that it cannot recover all garbage. Blocks in cyclic structures which become inaccessible still have positive reference counts and so are not deallocated. In addition, maintaining the reference counts can be very time consuming.

4. The Hierarchical Garbage Collector.

It is proposed that the distributed address space is divided up into disjoint areas, illustrated in FIG 2. Each area is garbage collected independently using a scanning garbage collector. These areas are much smaller than the entire address space so the garbage collection cycle is much shorter. Therefore garbage is recovered more quickly and less spare memory is required.

![FIG 2. Address Space is Divided into Areas](image-url)
Inter-area pointers are implemented through two indirection entries. The pointer refers to an outward entry. This describes the location of an inward entry which in turn gives the location of the block. The inward entry allows a block to be moved even if it is referred to by another area. The outward entry holds the information required to locate the inward entry, including any addressing information needed in the distributed system.

The garbage collector for an area must assume that any block that is referred to by an inward entry is still required by some other area. It must also keep all blocks that are accessible from such blocks. The garbage collector can find inaccessible outward entries and discard them in the same manner as inaccessible blocks. However, to recover inaccessible inward entries a higher level garbage collector which takes a global view of the areas is required.

It is proposed to go further than this simple two-level garbage collection system and structure the areas recursively, while still retaining the single address space of the distributed memory. Therefore, an area at one level will consist of several areas at a lower level, and will be part of a higher level area. A block will belong to some area at each level in the hierarchy. Each area will have a (virtual) garbage collector which is responsible for recovering garbage that spans the boundaries of its component areas.

The recursively structured areas are illustrated by Fig. 2. Here the inter-area pointers are shown with their outward and inward indirection entries placed within the appropriate boundaries to indicate which level of garbage collector is concerned with them.

![Levels](image)

**FIG 2. Recursively Structured Address Space.**

The hierarchical mark-scan garbage collectors are not implemented as separate processes. Instead there is a separate garbage collector process for only the lowest level areas. These are responsible for the garbage collection of their areas, but also contribute to the garbage collection of the higher level areas above them. This means that the amount of scanning and marking is no greater than if one system wide garbage collector was used. Garbage collection of the lowest level areas can proceed as often as required as the processes do not need to be synchronized.

An area is garbage collected by scanning accessible blocks for pointers, which in turn point to more accessible blocks. It is necessary to remember the state of each block in the scan: not found, found but not scanned, or found and scanned. Once there are no blocks waiting to be scanned, the scan terminates. Then any block that was not found is discarded because it is inaccessible.
Since each block is a member of one area at each level, its state in each of the scans needs to be remembered. This is too great an overhead to be practical. Fortunately it is necessary to only remember the state of a block in the highest scan which has found it.

Reference counting can be used as a useful optimization in recovering inward entries. A count can be kept of the number of outward entries that refer to a particular inward entry. If this count drops to zero then the inward entry can be discarded. Note that this scheme will fail to recover inaccessible cyclic structures that cross area boundaries, but does cater for the vast majority of cases.

4. Conclusions

A method has been suggested which allows timely garbage collection in a distributed system. The problems with this approach are general ones of distributed computing: agreement must be reached between parallel garbage collectors about termination and counts must be correctly maintained in the distributed environment. Furthermore, the garbage collector must operate in an error prone environment, coping with line failures and node crashes.

5. References

C.J. Cheney
- A Nonrecursive List Collecting Algorithm.

J. Cohen
- Garbage Collection of Linked Data Structures.

  Coop. of the ACM, Vol 21, Num 11, Nov 1978, pp666..675

R.S. Fabry
- Capability Based Addressing.
  Coop. of the ACM, Vol 11, July 1976, pp403..412

H.J. Hansen
- Compact List Representation: Definition, Garbage Collection, and System Implementation.
  Coop. of the ACM, Vol 12, Num 9, Sept 1969, pp599..607

J. McCarthy
  Coop. of the ACM, Vol 3, 1960, pp101..105

0. Ungar
- Generation Scavenging: A Non-disruptive High Performance Storage Reallocation Algorithm.
  ACM SigPlan Notices, Vol 12, Num 5, May 1984, pp157..167

R.W. Wiserman
- On the Garbage Collection of Block Structured Memories.
RELIABILITY ANALYSIS OF A FAULT TOLERANT SYSTEM
WITH SHARED POOLS OF SPARES

Raymond A. MARIE
IRISA/INRIA - RENNES
Campus universitaire de Beaulieu
35042 RENNES CEDEX
FRANCE

ABSTRACT

A method of deriving the reliability expression of large fault tolerant systems using redundancies is proposed. Each redundant block consists of a series of modules. The reconfiguration process for a block corresponds to the exchange of the broken down module if a spare still exists. The spares are supplied only at the beginning (t=0) and in a limited amount.

INTRODUCTION

This presentation deals with the reliability evaluation over a period (0,T) of a system consisting of a set blocks in parallel so that it processes k out of n type redundancy. Each block consists of a series of modules. The reconfiguration process for a block corresponds to the exchange of the broken down module if a spare still exists. A fixed number of spare modules are supplied at the beginning of the period (0,T) and they cannot be resupplied before the end of the period.

This problem contains essentially two major sources of difficulty. The first one comes from the fact that the repair consists of the exchange of a module (and not of the exchange of a block). The second difficulty comes from the fact that we cannot simplify the model by assuming that the spares are unlimited. Such an assumption would make impossible the elaboration of a spare policy.

These potential difficulties become real ones when each subsystem is large because of the number of blocks in parallel (e.g. 20 blocks) and because of the distinct number of module types (e.g. 200 kinds of modules).
NOTATIONS AND HYPOTHESES

Let us consider a system consisting of \( n \) blocks. Each block consists of \( q \) modules. A block \( j \) is said to be down as soon as one of its modules, \( i_j, i=1,...,q \), breaks down. The device is said to be down if there are more than \( r \) blocks down simultaneously. When a type \( i \) module breaks down, the failure is assumed to be detected immediately. If a spare exists, it is then exchanged. This exchange constitutes the repair process. The probability distributions of the lifetime and of the repair time of any type \( i \) module are both exponential with respective rates \( \lambda_i \) and \( \mu_i \).

Let us introduce the following conventions: any module of subsystem is in one of the three following states:

- \( e_1 \), if the module is up,
- \( e_{21} \), if the module is down and it is possible to replace it (temporary-down state), and
- \( e_{22} \), if the module is down and there is no spare to replace it (definitive failure).

CONSTRUCTION OF A RECURSIVE PROCEDURE

Recursive formulae

Let \( B_i \) denote a fictitious block consisting of modules of types \( 1,2,...,i \). Let \( D_i \) denote the fictitious redundant system consisting of \( n \) fictitious blocks \( B_i \). For a given time \( t \), let us define the following random variables

\[ N_{a}^{i}(t) \] number of fictitious blocks \( B_i \) in state \( E_{a} \), at time \( t \).

where \( a=1 \) if the block is up
\( a=0 \) if the block is down.
\[ Y^i(t) \triangleq \text{number of type } i \text{ modules whose states are in subset } \{ e_{21}, e_{22} \} \text{ at time } t. \]

Thus we always have
\[ N^i_1(t) + N^i_2(t) = n \]

The relationship which will now be pointed out will be valid for any time \( t \) and we will now omit the parameter \( t \), up to the end of this section.

For \( i \geq 1 \), we have by the theorem of total probability:
\[
P( N^i_2 = m ) = \sum_{j=0}^{m} P( N^i_2 = j ) \cdot P( N^i_2 = m | N^i_2 = j )
\]
\[
P( N^i_2 = m ) = \sum_{j=0}^{m} P( N^i_2 = j ) \cdot \left( \sum_{y=m-j}^{m} P(Y^i = y) \cdot P( N^i_2 = m | N^i_2 = j \text{ and } Y^i = y) \right)
\]

For \( i=0 \), we have,
\[
P( N^0_2 = 0 ) = P( N^0_2 = 0 ) \cdot P( Y^i = 0 )
\]

If \( i=1 \), we have,
\[
P( N^1_2 = m ) = P( Y^1 = m ), m=0, \ldots, n
\]

**Determination of the conditional probabilities**

From the theory of Probability (cf. the hypergeometric distribution), we have:
\[
P( N^i_2 = m | N^i_2 = j \text{ and } Y^i = y ) = \binom{j}{y} \binom{n-j}{m-y} \binom{n}{y}
\]

**Determination of the reliability of the real system**

The reliability of the real subsystem is equal to:
\[
R(t) = \sum_{j=0}^{P} P \{ N^q_2(t) = j \}
\]
PROBABILITY DISTRIBUTIONS OF $Y^i(t)$

Let $S_i(t)$ denote:

i) The number of spares of type $i$ available at time $t$, if $S_i(t) > 0$,

ii) The number of shortages relative to the type $i$ modules at time $t$, if $S_i(t) < 0$.

Let $M$ denote the value of $S_i(t)$ for $t=0$, i.e. $M=S_i(0)$. Since the failure and repair processes are exponentially distributed with respective rates $\lambda_i$ and $\mu_i$, the process $(S_i(t), Y^i(t))$ which takes its values on $(M, M-1, \ldots, -n) \times (0, 1, \ldots, n)$ is Markovian and homogeneous.

In order to simplify the written expressions, we omit the index $i$ in the remaining part of this section. Thus the Markovian process is now denoted $(S(t), Y(t))$ and the rates are denoted $\lambda$ and $\mu$.

Let $(s, y)$ be a state of the process $(S(t), Y(t))$.

Let $Q$ be the infinitesimal generator matrix of the process. Its non null coefficients $q(s, y; s', y')$ are defined as follows:

$$q(s, y; s-1, y+1) = (n-y)\lambda$$

$$q(s, y; s, y-1) = \begin{cases} yu & \text{if } M>s>0 \\ (y+s)u & \text{if } s<0 \end{cases}$$

$$q(s, y; s, y) = 1 - (q(s, y; s-1, y+1) + q(s, y; s-1, y))$$

The initial state is $(S(0), Y(0))=(M, 0)$.

The Markov Chain is acyclic and it is possible to obtain the transient solution in a tractable way. Moreover, it is easy to construct approximate solutions, with lower computing time, by taking the fact that $\lambda \ll \mu$ into account.
The major objective of our project is the design of a fault-tolerant distributed system architecture and its implementation as a prototype. In order to meet this requirement, the design provides operating system-level support for nested transactions operating on shared data in a distributed object-oriented environment. This constitutes a major departure from conventional transaction implementations: First, instead of being above an OS with the corresponding functionality to draw on, our transaction facility is beneath, inside the OS kernel. As a consequence, both the services and structure of the OS are substantially affected by the availability of nested transactions as kernel primitives. Rather than handling simple data base objects such as records and files, it must accommodate the far more complex, abstract and dynamic data types found in an OS. This is essentially virgin territory. Secondly, we had to take into account that the overhead of the transaction mechanism, though it is always a concern, becomes of eminent importance in a distributed OS environment. To make this clear, we first have to look at the most significant features and advantages user should expect from a DCS. The answer is essentially twofold: Enhancement of performance by exploiting the inherent potential concurrency and higher reliability and availability due to the inherent redundancy. Unfortunately, however, predominantly with respect to concurrency control and recovery being two major issues for designing a reliable DCS there exists a trade-off meaning that you cannot combine optimal solutions for both: achieving a high degree of concurrency and having an efficient recovery strategy at the same time because implementing efficient recovery methods inevitably implies restricting the potential concurrency. The only question is how much? Presupposing that a successful completion of a transaction should be the rule while aborts due to failures should be the exception and not vice versa we concluded that a viable nested transaction concept has to meet the following postulates:

1) No sacrifice of potential concurrency in favour of an easier way of implementing recovery

2) Introduction of concurrency between such functions contributing to the progress of computational activities and those being in charge of transaction and recovery management

3) Both, the object model and the transaction mechanism must be supported by dedicated hardware

Recalling the typical features of conventional transactions, the two most far-reaching concept changes we made are the following:

- Separating successful completion of a transaction from its commitment. Conventionally, a transaction may end up in two different states: either it is committed, i.e. it has completed successfully and the permanence of its effect are guaranteed, or it is aborted due the occurrence of failures or other undesired effects. We have introduced an additional state called completed that only indicates the successful termination of the transaction without guaranteeing permanence of effect, i.e. its effect is revocable.

- Permission of the premature release of those objects that are no longer used by the still ongoing transaction.

This means that we do not implement the feature 'prevention of cascading aborts' which implies that, to the contrary, we admit the possibility of cascading aborts. To achieve this property, the system must be able to keep track of these still 'uncommitted' objects so that in case of a subsequent abort of the releasing transaction all further modifications can be revoked. These dependencies are maintained locally at each site in the recovery graph. This is a distributed data structure maintained by the recovery managers of each site. The nodes of this graph are the...
so-called recovery units. They refer to the activity of a transaction at a single site. Arising dependencies between recovery units due to information flow by sharing global objects are detected at the hardware level and then are recorded in the recovery graphs of the affected sites meaning that the local parts of the recovery graph residing in the various sites mainly act independently of each other. Only in case that the dependencies cross site boundaries they have to cooperate in executing two specific protocols either
- to establish a global recovery line accomplished by adopting the idea of the ‘chase protocols’ (MeR) in case of a transaction abort or
- to perform the two-phase commit protocol.

In the following the main advantages are discussed.

Basically, for providing backward error recovery there exist two major approaches mirroring the trade-off between the exploitation of concurrency and consistency issues [And]. The first one is that of designing a system so that an appropriate recovery line is always available and known to the system in advance. Thus, it is based on the use of planned recovery lines requiring constraints to be imposed on the communication between processes as well as enforcing some form of synchronization on them. Transactions are an example for this approach where the recovery lines are delineated by the boundaries of the transactions. Its advantages are:
- it ensures the existence of appropriate recovery lines
- relatively simple mechanisms can be used
- the overhead incurred is bounded and under the control of the system designer

Against these benefits must be set the degradation of system performance which results from imposing severe constraints to concurrency, communication and resource utilization.

These drawbacks are mostly avoided by the unplanned recovery line approach. By this, in case of a failure the recovery mechanism must search for an appropriate recovery line. Hence, the recovery mechanism is in charge of monitoring interprocess communication and of recording the information flow in the system so that a consistent recovery line can be determined. The corresponding drawbacks to this approach, however, are:
- an appropriate recovery line may not even exist.
- a complex mechanism is needed to locate recovery lines, generally coupled with a costly mechanism for monitoring communication between processes
- The possible occurrence of the domino effect may result in an excessive loss of system activity and beyond the control of the system designer.

Our approach in a certain sense uses a combination of those mentioned above thus considerably alleviating the disadvantages of both of them. This is because of the mechanisms provided by our design we are much more flexible in tuning the nasty trade-off. By allowing transactions to access still uncommitted objects they are already waiting for we enhance system performance and resource utilization.

The point is that first, this is done without giving up the existence of the appropriate recovery line which is represented by the state of the accessed objects at the beginning of the releasing transaction. Secondly, the abortion of additional transactions due to the premature release of objects by the failing transactions does not mean any loss of system activity compared to the conventional transaction implementations because there all these additional transactions could not have been performed any way. Some analogy may be seen compared to pipelined architectures using look-ahead strategies where, in order to gain efficiency, several possible outcomes of a conditional jump are considered.

The situation slightly changes in case of a site failure. Then, in our approach the results also of the already completed transactions are lost which normally would have been saved on stable storage by starting the commit procedure after the successful termination of every transaction. However, this loss is always under the control of the system designer since the commitment can be initiated on request either by the system itself or by each participant of a completed transaction.

From that discussion we can derive straightforwardly the following positive consequences:
1. The elaborate commit procedure can be initiated optionally or if necessary. The most popular commit procedure is the 2-phase-commit protocol. However, according to accomplished measurements and the resulting experimental data, this protocol causes the paramount costs in running transactions. Now, in our approach, this performance degrading factor becomes less significant since it is no longer necessary to commit every single transaction. Thus, the commit intervals generally will be longer yielding a much more efficient ratio computational activity versus commitment.

2. Reduction of performance losses due to waiting for the commit signal. We allow participants of a transaction already to proceed with subsequent activities using the obtained results of that not yet committed transaction.

3. The amount of data to be saved on stable storage is reduced. We are able to commit several transactions all at once which implies that we only have to save the most recent versions of the affected objects.

4. The latency problem is alleviated. The latency interval denotes the time between the occurrence of a failure and its detection leading to a transaction abort. The commit interval refers to the time between the beginning and the commitment of a transaction. It follows immediately from the very definition of the state 'committed' (effects not revocable!) that, in order to recover from a transaction failure its latency interval must be located within the commitment interval of the affected transaction. Obviously, the longer the commitment interval, the more likely it is that this requirement is fulfilled.

5. The potential degree of concurrency is considerably enhanced leading to a broader range of applications.

Fig 1 delineates the layered structure of our system design. It also illustrates the correlation between the transaction concept and the object design.
The underlying paged virtual memory provides two completely disjoint address spaces for the so-called representation objects and the recovery segments, respectively. The representation objects may be seen as the 'guarded containers' for the typed objects made available at the next level. They are based on capability protected segments and their format constitutes a header followed by a capability and a data part. A recovery segment comprises the recovery information associated to each global object. It consists of a sequence of recovery points each associated to a transaction that manipulates the corresponding object.

Above this the transaction mechanism is implemented in 3 layers. The save/restore management maintains the recovery points for the representation objects. It saves object states and restores objects to prior states. The next layer offers recoverability of system and user defined objects (created by means of the type extension mechanism in the corresponding level of the object architecture) based on the recovery graph. Most important functions offered here are the establishing of a global recovery line and the global commitment. Thereby it relies on the object file which manages the passive object versions on a background medium. Since the passive object space entirely holds committed object versions, every update is controlled by the recovery management.

The top level offers the transaction management. Its main functions are the concurrency control and the managing of the action trees of nested transactions. For this task, the global object manager provides two essential services, the transfer of objects and the remote procedure invocation. In our first implementation concurrency control is performed by applying a two phase locking strategy. If, however, this mechanism is not appropriate e.g. for a specific application, because of our layered design it would be possible to replace it by a more suitable mechanism or to shift this task to the application level and continue to further use the services of the recovery management.

Fig. 2 depicts the hardware architecture of one node of the first version of a prototype to be implemented next. Particularly, it illustrates our emphasis on dedicated hardware support.

![Fig. 2: NODE ARCHITECTURE](image)
The processing subsystem and the recovery subsystem constitute the most interesting part of the node design. The CPU of the processing subsystem is realized by a 32 bit off the shelf microprocessor with coprocessor interface. By that we are able to implement our design of a MMU called MUTABOR (Mapping Unit for The Access By Object References) as a coprocessor. MUTABOR realizes a virtual object space in which the sharing of global objects is protected by using a capability mechanism. In particular it provides

- capability protected representation objects
- virtual memory management by paging
- up to 4 Gbyte virtual address space
- up to 4 K objects per process directly accessible whereby the size of an object may range between 1 byte and 4 Mbyte
- save/restore support by providing a completely disjoint recovery space and by executing the restoration of a recovery point
- detection of dependencies between recovery units.

We also intend to realize the implementation of system types and extended types as what we call an OS-coprocessor in a future version. The tasks of this coprocessor would include

- machine recognized system types
- type extension mechanism
- invoke mechanism for operations on extended types
- protection by type specific rights check
- tree structured inheritance

In our first prototype, however, this will be done in software.

The recovery subsystem totally is in charge of the recovery management. It is autonomous and acts concurrently to the processing subsystem. To fully exploit the resulting concurrency potentialities we provide an additional internal bus between processing and recovery subsystem. This recovery bus mainly has to transfer information about dependencies arising due to the sharing of global objects. The main tasks of the recovery subsystem are

- logging of recovery information
- maintenance of the recovery graph
- systemwide control of the recovery line
- commitment of transactions

The actual state of the project is that we have just completed the design which is fully documented in two technical reports coming out in these days. Next we are going to implement a LAN-3-node system prototype based on a 68020 VME system and including the development of VLSI components for the translation look aside buffer and the cache controller.
INTRODUCTION

In this talk, mentioned are the design principles to transform an N-modular redundant system, NMR system for short, into a system that, on identifying a faulty component, autonomously reconfigures itself into an all-perfect state. The reconfiguration is performed by replacing a faulty component by a spare one and, as a result, the NMR system is required to be complemented with a set of spare units which can be taken to be nonfaulty. With this essential requirement, our design achieves the transformation with no stronger a failure criterion than the one that would be necessary to ensure the correct operation of any simple NMR system with no static redundancy.

2. SYSTEM STRUCTURE

Our (hybrid NMR) system has two disjoint sets of nodes - the set N of active nodes and the set S of cold spare nodes. The nodes are fully connected by the communication links. Nodes and links form the system components. Though our design can be reasonably extended for the classes of faults described in [1],

![Diagram: A -> B -> C; N = A, B, C; S = S1]

Fig. 1. HYBRID TMR SYSTEM

Here, for simplicity, we assume that the components suffer only from omission faults which result in no response from the faulty component.

3. SELF RECONFIGURATION STRATEGY

The fundamental NMR requirement is:

R1
There must exist a set G of correctly functioning nodes s-t. G ⊆ N and |G| ≥ m+1.

The nodes in G, called G-nodes, are used to replace the faulty component. This means that all the G-nodes or at least (m+1) of them must correctly and consistently identify a fault. On having so identified a faulty component (the how-abouts described later), they take identical and appropriate steps towards its replacement. Note that since the G-nodes work identically with a common aim, even the presence of faults more serious than the assumed omission faults will not matter at all in replacing the identified faulty component.

4. FAILURE OBSERVATION AND CONSISTENCY ALGORITHM

The operation of an NMR node alternates between computation and voting. The next stage computation will start after confirming the correctness of the previous stage computation through majority voting. This implies the following second NMR requirement.

R2
Every node g in G will have a set Sg of at least m other G-nodes from which it will correctly receive messages.
Note that a G-node, by executing the voting scheme, will be able to observe the failures from the results sent to it by the other nodes. Will the observation be identical for all the G-nodes? Need not be. In figure 1, let G={A,B} and C fail in such a way that it correctly responds to B but not to A. In this case, B won't observe the failure but A will. To have identical failure information, the nodes execute the following algorithm, called the Interactive Consistency (IC) algorithm which will have the following (m+1) steps.

**STEP 1**
Multicast about failures observed directly.

**STEP i** 2 ≤ i ≤ m+1

Multicast the failure information newly received at the (i-1)th step.

Due to R1 and R2, the execution of the above algorithm by each NMR node will result in all the nodes in G obtaining
1. each other's failure information, and
2. identical failure information.

The correct and consistent identification of faulty components will be guaranteed by results 1 and 2. We also observe that, when links fail unidirectionally i.e. with correct transmission being in only one direction, at least a majority of G-nodes will meet, due to R1 and R2, the above two results that fulfill the self reconfiguration requirements.

CONCLUSION

The IC algorithm is found useful in other NMR applications such as voting over non-identical results and clock synchronisation. The algorithm is currently being studied under a more elaborate fault classification than the one described in [1].

REFERENCES

[1] CRISTIAN, F., ACHILI, H., STRONG, R., DOLEV, D.
FAULT TOLERANT LEVELS OF EXECUTION IN DISTRIBUTED APPLICATIONS

S. Pfleger

University of Newcastle upon Tyne, United Kingdom

The structuring of distributed applications by fault tolerant functional levels of execution is a refinement of the structuring concept of "generalized fault tolerant component" [And 81, Ran 83] that extends the construction of "fault tolerant components" by providing a fault tolerant interaction between components and supporting a hierarchical error control during the parallel execution of independent, competitive and co-operative processes:
- internally, in each component by the internal handler
- locally, in each functional level by the local handler
- globally, in the global handler (for distributed systems with centralized coordination) or by distributing the global error control functions into the local handlers (for distributed systems with a decentralized co-ordination).

A distributed application is regarded here from the user's point of view, as a hierarchically structured system with multiple layers of execution:
- hardware equipment layer (local processors and network hardware),
- operating system and communication software layer,
- software application layer,
- user interface layer (man-machine or machine-machine).

The activities in a layer may be:
- functional, for performing the specified function under the assumptions of correctly developed and correctly operating levels (ideal levels) or
- corrective, for providing fault tolerant capabilities to the latent design and operating faults (also known as abnormal, or error control activities).

Several functional levels of execution can be distinguished in the software application layer:
- execution of a program block into a program unit,
- execution of a program unit into a subsystem (the subsystem is a group of program units)
- execution of a subsystem into a hierarchical higher subsystem (the higher subsystem is a group of subsystems)

A fault tolerant functional level of execution has error control capabilities for detecting and tolerating:
- faults in the level itself (faulty components, faulty interfaces and faulty interaction between its components),
- faults in its server levels (failure of the expected service)
- faults in the client levels (invocation of its component outside of their standard domain by a component of the client level).
The relationship between the components is time-dependent; each component may perform services for another component as a server component at a given time and may become a client component by requesting services from another components, at another given time.

A component may perform functional activities in the functional flow as a functional component, or may perform error control activities in the correctional flow as a manager component (local manager if it performs error control activities between the levels).

The manager component has supervisory functions in the execution flow, it detects interface exceptions between the functional components belonging to the same level, identifies the propagated exceptions from other managers of the server-levels, isolates and identifies the propagated exceptions from its functional components. The handling of the raised and propagated exceptions based on a consecutive replacement of the related faulty components and is performed by the manager component using an exception handler, called local handler. The unhandled exceptions are propagated as "level failure" exceptions to a global handler, that may be implemented in:
- the local manager component of the client level, if the system has a decentralized coordination between subsystems, or
- in a global manager component, if the system has a centralized co-ordination between subsystems.
The concept of fault tolerant functional level of execution is proposed here as a structural support in the construction of distributed software applications with "self repair" capability during run time, by using a successive and hierarchical replacement of the faulty components. The "self-repair" capability during execution is needed in distributed applications with the requirement of long-term preservation of on-line state information.

References


1. Introduction

Our group is working on the definition and evaluation of a design methodology for decentralized systems, i.e. systems where no singularity point exists at any abstraction level and object management is based upon cooperation and consensus among entities with equal authority [1].

Decentralized systems are potentially able to exploit high degrees of parallelism and fault tolerance, as well as proper trade-offs between them. Our approach is based upon the definition of a concurrent programming language, where the features for achieving the mentioned objectives are naturally integrated into the cooperation mechanisms.

Typical applications of this approach are:
- massive parallelism architectures implemented by regular, very large structures of VLSI components;
- fully distributed operating systems and related architectural supports.

Current experience is based on the ECSP concurrent language.

2. Features of ECSP concurrent language

ECSP [2] is a message-passing, high-level language, whose semantics is an extension of the CSP model. A complete programming environment and some versions of the run-time support have been developed.

The language features, relevant here, are the following.

a) Communication (expressed by i/o commands) is asynchronous. Processes can be referred to by "processname" variables, so that the rights to utilize a communication channel can be assigned/revoled dynamically.

Nondeterminism is controlled by guarded commands (output guards are not allowed), where each guard may be associated a variable priority.

b) Computations are structured in a hierarchical-parallel fashion by means of nested parallel commands. Import/export lists are used to communicate between a process and the processes it activates.
c) Exception handling is expressed by the on fail clause, that takes account all the possible conditions for which a command may fail. For an i/o command they are: "partner terminated", "channel disconnected", "run-time support error"; for a parallel command, the possible combinations of termination-with-failure conditions.

Distributed implementation schemes for ECSP are described in [3], with emphasis on communication and termination. The only assumption about low level protocols is that the diagnostic mechanisms are reliable, so that the termination conditions of a command are uniquely distinguishable. No ambiguity arises due to the language semantics: e.g. the outcome of an output command (with success/with failure because of partner termination/etc.) and the behaviour of a remote partner (termination/etc.) are communicated by distinct messages at the run-time level.

3. ECSP programming of decentralized policies

We summarize now some main concepts and techniques for programming decentralized policies in ECSP [4,5,6].

a) Consensus in cooperation.
In the cooperation between a pair of processes (e.g. a client C and a server S) both autonomy and agreement on decisions must be granted (e.g. S must be informed that C considers it faulty because a reply did not arrive within a time-out interval). Disconnection of dynamic channels, non-forced termination and exception handling through the onfail clause are the main ECSP mechanisms for achieving this.

b) Parallelism vs. nondeterminism.
Often nondeterminism is used to simulate a concurrent behaviour. By transforming nondeterminism into parallelism (e.g. a guarded command into a parallel command) it is possible to achieve not only better performance but also finer granularity in mutual controls and cooperation (e.g. to be informed promptly about the termination of one partner process). Nesting of parallel commands is often used in place of guarded commands, without affecting the programs of partner processes.

c) Atomic actions and backward recovery.
The proper combination of parallel commands, import/export lists, dynamic channels and onfail clause allows us to easily express nested atomic actions and backward recovery, without specific mechanisms and with additional degrees of freedom in achieving the desired trade-offs between parallelism and fault tolerance.

d) Decentralized object management.
When the internal state of a computation is partitioned/replicated into parts managed by distinct processes, an high degree of parallelism is in general possible only by allowing "temporary
inconsistencies" in their values. Efficient and robust mechanisms must be provided for detecting and recovering them. In ECSP inconsistencies are modelled as failures in process interactions, so they are detected through the exception handling mechanisms and recovered through the mechanisms mentioned in point c). Several examples (mutual exclusion, schedulers) have been evaluated by simulation in order to investigate the parallelism/fault tolerance trade-off, locality of communications and scalability. The results encourage us to pursue this research line.

4. Acknowledgement.

All the ideas summarized here derive from the joint work with Fabrizio Baiardi. The research is sponsored partly by ECC Esprit project 415, and partly by Ministry of Education Project on Decentralized Systems.

5. References.

1. Motivations.

The language LC\(^2\) is currently developed at IRISA in relation with the CNRS C\(^2\) project which aims in promoting research in different fields of parallel processing. It appeared very soon among the C\(^2\) research groups that there was a need for a language facilitating experimentation of new ideas in the field of parallel programming. For example, it should be easy to evaluate new cooperation schemes by extending the language with them. During the design phase of LC\(^2\), two ideas were put forwards:
- LC\(^2\) should be the extension of a widely used sequential language. C was chosen.
- Program structures for parallelism should be minimal and powerful enough to express new synchronization schemes.

The next sections of this note describe the main aspects of the language.

2. Main aspects of LC\(^2\).

We only present aspects of LC\(^2\) related to parallelism.

2.1. Processes.
Processes may be defined and dynamically created.

2.2. Communications.
Processes communicate via sequences accessible through channels.
Sequences are bounded or not and contain elements of the same type.
Channels are typed e.g. a channel int can only receive or send integer values. Channels are associated to sequences via the operation "open". Several channels may be opened simultaneously on the same sequence and processes may share channels.
A sequence is built by using the operation \texttt{write}(s,v) which adds the new element \( v \) to the sequence \( s \).

A sequence is used through the operation \texttt{read}(\text{channel}) , where the channel parameter is assumed to be opened on the right sequence. Let us point out that the \texttt{read} operation is non-destructive.

The \texttt{write} operation is non-blocking, i.e., a process using it is not delayed. The \texttt{read} operation may be blocking if the information to be read is not yet produced. As soon as this information becomes available the process is resumed.

2.3. Families of Atomic Actions.

Families of atomic action allow the definition of sequences of instructions (actions) that a set of processes has to execute atomically and simultaneously. While executing a family of atomic actions, the concerned processes may exchange informations between them, but cannot communicate with processes outside the family. All actions constituting a family begin simultaneously and terminate simultaneously.

Example 1: Rendez-vous without information exchange.

Let us express a rendez-vous between three processes without information exchange.

\begin{verbatim}
action A = ();
fax (p_1.A, p_2.A, p_3.A), #family of atomic action #
process q
{
...
!!!A;
...
}
process master
{activate p_1 = q();
 activate p_2 = q();
 activate p_3 = q();
}
\end{verbatim}

The master process creates three instances of the \( q \) process. A process \( p_i \) (\( i = 1, 2, 3 \)) can execute its action \( A \), only when the two other processes belonging to the family are ready to do so.
Example 2: Rendez-vous with information exchange.

This example deals with the rendez-vous of two processes with information exchange.

\[ action A_1 = (\text{write}(X, 0);) ; \]
\[ action A_2 = (x = \text{read} C ;) ; \]
\[ \text{Faa}(q_1, A_1, q_2, A_2) ; \]
\[ \text{sequence \ int}(I) \ X ; \]
\[ \text{process} \ p_1 \]
\[ \{ \ldots \]
\[ \quad !! A_1 ; \]
\[ \quad \ldots \]
\[ \} ; \]
\[ \text{process} \ p_2 \]
\[ \{ \text{int} \ x ; \]
\[ \quad \text{channel \ int} \ C ; \]
\[ \quad \text{open}(X, C) ; \]
\[ \quad !! A_2 \]
\[ \quad \ldots \]
\[ \} ; \]
\[ \text{process \ master} \]
\[ \{ \text{activate} q_1 = p_1( ) ; \]
\[ \quad \text{activate} q_2 = p_2( ) ; \]
\[ \} \]

Processes \( q_1 \) and \( q_2 \) synchronize on the actions \( A_1 \) and \( A_2 \) which are executed simultaneously and atomically.

3. Implementation.

A first centralized implementation of LC\(^3\) is presently working on UNIX. Preliminary studies for a decentralized implementation have been undertaken.

References.

D.B. LOMET "Process structuring, synchronization and recovery using atomic actions'.

P. LE CERTEN "LC\(^3\), Un langage de programmation parallèle'.
Extended Abstract

Two ways of describing the behaviour of concurrent systems have widely been suggested: arbitrary interleaving of atomic actions and partial orderings between atomic actions. Sometimes the latter has been claimed superior because concurrency is represented in a "true" way. On the other hand, some authors have claimed that, at least as far as practical purposes are concerned, the former is sufficient.

Petri net theory offers a framework in which both kinds of semantics can be defined formally and hence compared with each other. The usual notion of firing sequences corresponds to interleaved behaviour. The process notion developed in //2,3,1// can be used to capture partial order semantics within net theory. For concurrent programming languages such as CSP or shared variable languages, interleaved semantics is widely used but little work exists on partial order semantics.

The present work deals with four problems. Firstly: In general, does the partial order approach yield behaviour which is essentially different from the behaviour obtained in the interleaving approach? The general answer to this question is affirmative. For example, the behaviour described by the following ("infinitely broad") Petri net cannot be described by interleavings (since no linearisation of this net is order-isomorphic to N).

§ This work was done in collaboration with Raymond Devillers (Université Libre de Bruxelles) //1//
However, if one excludes infinite concurrency structures of the following form then, indeed, the classes of behaviours described by partial orders essentially coincide with those described by interleavings:

Thus, in the absence of this concurrency structure, partial orders give no essential new powers. However, the question still remains (and this is the second problem considered in this work) exactly what is "lost" if one knows the set of interleavings as compared to if one knows the set of partial orders. To answer this question, a relation $\equiv$ on sequences may be defined (meaning that sequences differ only in the order of concurrent transition occurrences) and a relation $\equiv$ on processes may also be defined. A theorem states a 1-1 relationship between $\equiv$-equivalence classes of sequences and $\equiv$-equivalence classes of processes. For the important special case of 1-safe Petri nets, $\equiv$ reduces to identity on processes, so that one may state the "equality"

$$\text{Processes} \quad = \quad \{\text{Sequences} + \equiv\}.$$ 

In other words, in the 1-safe case, $\equiv$ is the exact piece of information lost
when sequences instead of processes are being considered. Moreover, it is known that \( = \) may be generated by a simpler relation called \text{indep}.

Turning to concurrent programs, the third problem concerns the question whether the above deliberations and results can be transferred. The (perhaps surprising) answer is that both CSP programs and concurrent programs with shared memory can be viewed as special cases in two respects. Firstly (unless one admits, unrealistically, infinitely many variables or communications between an infinite number of partners), the concurrency structure mentioned above is absent. As a result, partial orders give no essentially different behaviour than do sequences. Secondly, concurrent programs are analogous to the 1-safe case in that one may derive an "equality"

\[
\text{Processes} \quad = \quad (\text{Sequences} + \quad =),
\]

where \( = \) can again be derived from a simpler relation \text{indep}. Moreover, \text{indep} has an intuitive meaning: Two actions \( a_1 \) and \( a_2 \) are in relation \text{indep} iff \( a_1 \) and \( a_2 \) do not contain any common variables and concern communications between disjoint sets of partners.

The fourth and last problem concerns the question whether or not something can be gained in terms of the practical aspect of program proving if one considers partial orders instead of interleavings. Alternatively, the question is whether one can use the \text{indep} relation in program proofs. The answer is again affirmative because, for instance, if one has the two sequences

\[
\text{abcd}, \quad \text{acbd},
\]

and if it is known that \( b \text{ indep } c \) then it suffices to reason about one of them if one wants to, say, prove some invariant about the states before "a" and after "d". This may pay especially if there are lots of independent actions. One may wonder if it is just "similar" reasonings that may be saved in this way. However, there are examples where the reasoning concerning \text{abcd} differs from the reasoning concerning \text{acbd} and yet, one of them may be saved.

1. INTRODUCTION

Numerous distributed algorithms whose aim is to control a computation are based on logical clocks. Each process is endowed with such a clock which behaves as a counter initialized to zero and strictly increasing (LAM 78). Each time an event is produced by a process (e.g. sending or receiving of a message) that process clock is incremented and the resulting value stamps the event. So the event is uniquely identified by this timestamp coupled with the process name (all the process names are supposed to be distinct).

A priori the logical speeds of the processes are independent (by logical speed we mean the growth of logical clocks). Logical clocks can drift ones with respect to the others and two attitudes are possible according to the algorithms in which they are used : either a clock is reset when its drift with respect to an other is too large or a global control of logical clocks pulsations is ensured in such a way that their mutual drift does not exceed a predefined threshold ; it is nothing but the two classical control techniques : the a posteriori one (detection and then correction) and the a priori one (prevention).

An a priori technique for controlling logical clocks pulsations is examined in this paper. Carvalho and Roucairol have proposed such a distributed algorithm in the case of two processes meeting the following requirement : \( h_i \) \((i = 1, 2)\) being the logical clock of the process \( P_i \) and \( \delta \) being the authorized maximum drift between the two clocks, the relation \( |h_i - h_j| \leq \delta \) is always satisfied. After a recall of this algorithm we first proof its correctness ; then a generalization is proposed in the case of \( n \) processes. The generalization is such that :

\[
\forall i, j \in \{1, \ldots, n\}, \quad |h_i - h_j| \leq \delta
\]
The behaviour of this algorithm is then studied in various environments.

2. THE CARVALHO AND ROUCAIROL ALGORITHM

The hypotheses of the Carvalho and Roucairol algorithm (CAR 84) are the following ones. The communications between the two processes (named $P_i$ and $P_j$) are asynchronous. The communication primitives are denoted $!!$ (sending) and $??$ (receiving). Messages can be lost between the two processes and may not be delivered in the order sent.

Processes are expressed in a CSP-like syntax [HOA 78] in which non-deterministic choices are assumed to be fair (to allow the progression of processes). Moreover a guard can contain a sending or a receiving statement [BUC 83].

Each process $P_i$ is endowed with the variable $h_i$ which represents its logical clock; the variable $\max_i$ indicates the maximum value that $h_i$ can reach without violating the invariant $|h_i - h_j| \leq \delta$ ($i=1$ or $2$ and $j=3-i$).

\[
P_i := (h_i := 0 ; \max_i := \delta ; x_i := 0 ; \)
\]

\[
\alpha^x \quad (h_i < \max_i) \quad \rightarrow \quad h_i := h_i + 1
\]

\[
\beta \quad \Box P_j ?? h_i \quad \rightarrow \quad \text{skip}
\]

\[
\gamma^1 \quad \Box P_j ?? x_i \quad \rightarrow \quad (\max_i - x_i < \delta \Rightarrow \max_i := x_i + \delta
\]

\[
\gamma^2 \quad \Box \max_i - x_i > \delta \Rightarrow \text{skip}
\]

The process $P_i$ can increment its clock and use it as long as the clock does not pass beyond the maximum authorized value $\max_i$. (line $\alpha$). At every moment $P_i$ can either send its clock value to the process $P_j$ (line $\beta$) or receive a clock value from $P_j$ (if any) (line $\gamma$); in that case $P_i$ computes its new authorized maximum as a function of the received value $x_i$ and of the $\delta$ parameter (maximal drift).

**Characteristics of the algorithm**

This algorithm tolerates message loss if the communication line is fair (i.e. every re-emitted clock value will arrive eventually at least once). The clock values sent by a process form a non-decreasing sequence. As message can
overtake each other the sequence of received values has not this monotony property, but messages which have been overtaken (and so carry old clock values) are ruled out (line γ²); consequently the sequence of the received values causing the update of $max_{i}$ is an increasing sequence.

3. PROOF OF THE ALGORITHM

Initially $h_{1} = h_{2} = 0$, so $|h_{1} - h_{2}| \leq \delta$. The invariant being initially verified the proof consists in showing it is left true by the updates of the $h_{i}$ variables.

Let $h'_{i}$ be value of the clock $h_{i}$ communicated by $P_{i}$ to $P_{j}$. The lines α and β ensure that we have at every instant:

$h'_{i} \leq h_{i} \leq max_{i}$
and
$h_{j} \leq h'_{j} \leq max_{j}$

From these formulas we can deduce:

and $h_{i} - h_{j} \leq max_{i} - h'_{j}$
and $h_{j} - h_{i} \leq max_{j} - h'_{i}$

Initially $max_{i} - h'_{j} \leq \delta$ and $max_{j} - h'_{i} \leq \delta$ and then these inequalities are preserved by the updates of the $max_{i}$ and $max_{j}$ variables (indeed the two streams of values causing the updates form two increasing sequences). Therefore we have the invariants $max_{i} - h'_{j} \leq \delta$ and $max_{j} - h'_{i} \leq \delta$; from these we can conclude that:

$h_{i} - h_{j} \leq \delta$ and $h_{j} - h_{i} \leq \delta$

i.e. $|h_{i} - h_{j}| \leq \delta$ with $i=1$ or 2 and $j=3-i$. QED.

4. GENERALIZATION IN THE CASE OF $N$ PROCESSES

The aim of our generalization is to offer a distributed algorithm ensuring pulsations control of $n$ logical clocks in a way such that:

$\forall i, j \in (1..n) \quad |h_{i} - h_{j}| \leq \delta$
The generalization does not rely on an instantaneous global state; it relies only on the knowledge that a process has of the previous local states of the other processes: the monotony property of clocks ensures consistency of that knowledge (BOC 79).

The hypotheses are the same as previously except that there is no overtaking of messages (this limitation will be next removed).

From a methodological point of view it is necessary to introduce two special device to generalize the algorithm. First, between the processes: each process must be able to send its clock value to all the other processes, so a broadcast (or a multicast) device is necessary. Secondly, inside each process: each process $P_i$ must be constrained in such a way that

$$\forall j \in \{1..n\} \quad |h_i - h_j| \leq \delta$$

so an array $\text{last}_i$ instead of the variable $x_i$ is introduced. The variable $\text{last}_i(j)$ records the last value sent by $P_j$ and received by $P_i$.

With these two devices the generalized algorithm is analogous to the previous one. When $P_i$ receives a clock value from $P_j$, it records it in $\text{last}_i(j)$; but now the update of the variable $\text{max}_i$ is done (not with respect to the last received value but) with respect to the smaller one of the values received from each of the $n$ processes. (So it is the process with the slower clock progress or whose clock values are transmitted the more slowly which get into step the variables $\text{max}_j$ of the other processes).

\[ P_i := (h_i := 0; \text{max}_i := \delta; \forall j \in \{1..n\}, j \neq i : \text{last}_i(j) := 0) \]

\begin{align*}
\alpha & \quad \{ h_i < \text{max}_i \}_{\rightarrow} h_i := h_i + 1 \\
\beta & \quad \forall j \in \{1..n\}, j \neq i \quad P_j \text{!!} h_i_{\rightarrow} \text{skip} \\
\gamma & \quad \forall j \in \{1..n\}, j \neq i \quad P_j \text{??} x_i_{\rightarrow} \text{last}_i(j) := x_i \\
& \quad \text{y} := \min \{ \text{last}_i(k) \}_{k \neq i} \\
& \quad (\text{max}_i - y \leq \delta \rightarrow \text{max}_i := y + \delta) \\
& \quad (\text{max}_i - y > \delta \rightarrow \text{skip})
\end{align*}
Proof of the algorithm

Establish the proof consist in showing that the relation:
\[ \forall i, j \in \{1, \ldots, n\} \mid h_i - h_j \leq \delta \] is invariant. The proof is similar to the previous one; it is based on the following inequalities (from the lines a, b):

\[ \forall i, j : \min_{k \neq i} \{ \text{last}_i(k) \} \leq \text{last}_j(i) \leq h_j \leq \max_{k \neq j} \{ \text{last}_j(k) \} \leq \text{last}_i(j) \leq \max_i \]

from there inequalities we extract:

\[ \forall i, j : h_i - h_j \leq \max_j - \min_{k \neq i} \{ \text{last}_i(k) \} \]
\[ h_j - h_i \leq \max_i - \min_{k \neq j} \{ \text{last}_j(k) \} \]

moreover the modifications of the variables \( \max_i \) are done at the line \( \gamma \) and then we have:

\[ \forall i \max_i - \min_{k \neq i} \{ \text{last}_i(k) \} \leq \delta \]

We can therefore conclude that:

\[ \forall i, j \mid h_i - h_j \leq \delta \text{ and } h_j - h_i \leq \delta \]

i.e. \( \forall i, j \mid |h_i - h_j| \leq \delta \) Q.E.D.

5. CHARACTERISTICS OF THE ALGORITHM

This generalization of the two processes solution has been naturally made thanks to the introduction of two devices. This algorithm tolerates the loss of the messages: when a clock value is broadcasted by \( P_i \), either all the other processes, or only a subset of them, or even any, receive it. The medium has been assumed to be fair.

When the messages may not be delivered in the order sent between two processes the statement \( \text{last}_i(j) := x_i \) at the line \( \gamma \) must be replaced by:

\[
\text{if } \text{last}_i(j) < x_i \text{ then } \text{last}_i(j) := x_i \]

in order that the sequence of values sent by \( P_j \) and received by \( P_i \) is an increasing sequence; in this manner overtaken messages are ruled out.

The algorithm can be very easily adapted to the case where the processes are not fully connected. A symmetric neighbourhood relation is then defined between processes; it associates to each process the set of its direct neighbours. The algorithm ensures:
\( \forall P_i, P_j : \text{neighbours } (P_i, P_j) \iff |\dot{h}_i - \dot{h}_j| \leq \delta \)

For any two processes the maximal drift between their clocks is then given by:

\( \forall P_i, P_j \mid |\dot{h}_i - \dot{h}_j| \leq m \cdot \delta \)

where \( m \) is the length of the shortest path between \( P_i \) and \( P_j \). In the particular case of a ring of \( n \) processes the maximal drift between two arbitrary clocks is \( \left\lfloor \frac{n \cdot \delta}{2} \right\rfloor \) (\( \lfloor x \rfloor \) denoting the greatest integer less than or equal to \( x \)).

If some applications require particular maximal drifts \( \delta_{ij} \) between every pair of neighbour processes \( P_i \) and \( P_j \), the algorithm can be easily fitted up to take this characteristic into account. The maximal drift between any two clocks is then obtained by weighting each edge \( (P_k, P_l) \) of the neighbourhood relation graph by the associated \( \delta_{kl} \). Then:

\( \forall i, j \mid |\dot{h}_i - \dot{h}_j| \leq \min_{u \in S} \left\{ \sum_{(P_k, P_l) \in u} \delta_{kl} \right\} \)

where \( S \) is the set of elementary paths between \( P_i \) and \( P_j \).
REFERENCES

(BOC 79) BOCHMANN G.V.
Distributed Synchronization and Regularity.

(BUC 83) BUCKLEY G.N. and SILBERSCHATZ A.
An Effective Implementation for the Generalized Input-Output
Construct of CSP.

(CAR 84) CARVALHO O. and ROUCAIROL C.
Algorithme de Contrôle d'horloges entières entre deux sites.

(HOA 78) HOARE C.A.R.
Communicating Sequential Processes.

(LAM 78) LAMPORT L.
Time, Clocks and the ordering of events in a Distributed System.
1. The different classes of distributed algorithms.

A distributed algorithm is a parallel algorithm made of a finite set of communicating sequential processes exchanging messages (fig.1) (RAY 85b).

Two classes of distributed algorithms can be distinguished:
- distributed algorithms that perform a computation (i.e. implementation of an abstract function).
- distributed algorithms that control a computation (i.e. interpretation or controlling part of a system).

These two classes of distributed algorithms do nothing but extend a distinction already existing in sequential and parallel programming (fig.2). In the following we shall focus on the second class of distributed algorithms.

\[
\text{Distributed Algorithm} = \{\text{processes}\} + \text{messages}
\]

fig.1
2. The nature of the distribution.

The specificity of the distribution with respect to parallelism is that the interaction between processes is done through exchanging messages. This kind of communication medium affects the distributed algorithm itself according to two distinct features:

- the topology of software communication lines defined between processes, and
- the property of the lines concerning reliability (e.g., lost or erroneous messages), transmission rate, messages serialization (overtaking), etc.

Besides the topological and behavioural features of communication lines, a distributed algorithm can be characterized by the absence of knowledge of a global state. A process can only capture partial (and previous) states of its communicating processes via the information carried by the messages. (that is why distributed algorithmic has often be compared with relativist theory.)
Note: we are only interested in distributed algorithms with distributed control (sometimes called decentralized control). We consider distributed algorithms made of communicating processes with a central coordinator process as a distributed implementation of a centralized control: the failure of the coordinator being fatal to the algorithm.

3. Distributed control problems.

The types of control found in distributed algorithms can be classified into four classes as following:

3.1. Privilege allocation

The problem is to allocate a privilege to one and only one process among a set of distributed processes. According to the forms this allocation can take (permanent or temporary) one will speak of an election or mutual exclusion algorithm (DOL 82, RAY 85a).

The main problem encountered here is to (fairly) order the parallel requests sent by the processes to the privilege allocation device, in this decentralized context.

3.2. Verifying some process property

Deadlock and termination detection problems belong typically to this class of control problems. They consist in:
- either controlling that a property is verified in a given set of processes (termination problem),
- or extracting from a set of processes the subset which verifies a given property (deadlock problem).

3.3. Data transfer

Protocols are another example of distributed algorithms. The parts of processes involved in the implementation of such data transfer protocols are generally dyssymmetric (e.g. a sender and one or several receivers).
Languages such as CSP or ADA which provides the user with high level transfer primitives, rely on run time support built with such protocols.

In unreliable contexts the implementation of a reliable transfer can be of great significance: the processes must reach a common agreement (e.g. Byzantine general problem (LAM 82)).

3.4. Data duplication

In a distributed system it is a common use to duplicate at each node global objects in order to speed up the access time and to increase fault tolerance. The problems which arise with duplicated objects is to maintain the consistency of duplicates (i.e. every update must be broadcasted through the network in a consistent way) (BER 81, TRA 82).

Moreover if we make the assumption that the set of processes can be partitioned into disjoint subsets in case of failures, the underlying problem is to detect the mutual inconsistency and then to merge the disjoint subsets of duplicates to form a consistent object (PAR 83).

Distributed algorithms field is large. In the next part we try to sketch basic structures for such algorithms.

4. Towards basic structures for distributed algorithms.

In sequential context, program design and programming methodology advances have given rise to data and control structures well-suited to express and resolve problems. In the context of parallelism based on shared objects, structuring concepts, such as the monitor (MOA 74), have also emerged.
In distributed context it is very important to search for (and then define properly) basic structures:

- in the communication topologies, and
- in the control structures associated with these topologies.

The studies we have performed on some distributed algorithms allow us to consider as basic structures the following ones.

4.1. Topological structures

i) The uni or bi-directionnal ring is a communication structure in which each process is constrained to communicate only with its neighbours; the set of processes forming a loop in which each process has only two neighbours.

ii) The tree. (the tree can be a spanning tree over the whole graph formed by all the processes and all the communication lines). In this structure the root process is a privileged process. This structure takes into account most of the hierarchical communication schemes.

iii) The fully connected structure, in that case each process can communicate directly with each other; there is no privileged process.

4.2. Communication disciplines

The ring and tree topologies allows to visit easily all the processes; consequently they are well-suited for privilege allocation and property verification problems (CHA 83, DIJ 83, RAN 83). With the ring the discipline of a special message going through is associated: the ring token (LEL 77, RAY 85c). With the tree the discipline of the diffusing computation is associated (DIJ 83).
Fully connected topology is used in algorithms where each process state modification must be broadcasted to all other processes. Causality relations are to be preserved (e.g. a send must precede the corresponding receive); so each process is endowed with a logical clock; each message carries a clock value and a timestamping mechanism ensures clocks consistency. This discipline is particularly used in mutual exclusion (LAM 78, RIC 81, HEH 81) and data consistency algorithms (BER 81, TRA 82).
References

(BER 81) BERNSTEIN Ph, GOODMAN N.
Concurrency Control in Distributed Data Base Systems.

(CHA 83) CHANDY K.M., MISRA J., HAAS L.
Distributed Deadlock Detection.

(DIJ 80) DIJKSTRA E.W., SCHOLTEN C.S.
Termination Detection for Diffusing Computations.

(DIJ 83) DIJKSTRA E.W., FEIJEN W.H.J., VAN GASTEREN A.J.M.
Derivation of a Termination Detection Algorithm for Distributed Computations.

(DOL 82) DOLEV D., KLAWE M., RODEH M.
An O(nlogn) Unidirectional Algorithm for Extreme Finding in a Circle.

(HEH 81) HEHNER E.C.R., SHYAMASUNDAR R.K.
An Implementation of P and V.

(HOA 74) HOARE C.A.R.
Monitor : an Operating System Structuring Concept.

(LAM 78) LAMPORT L.
Time, Clocks and the Ordering of Events in a Distributed System.
(LAM 82) LAMPORT L., SHOSTAK R., PEASE M.
The Byzantine Generals Problem.

(LEL 77) LELANN G.
Distributed Systems: Towards a formal approach.

(PAR 83) PARKER D.S. et al.
Detection of Mutual Inconsistency in Distributed Systems.

(RAN 83) RANA S.P.
A Distributed Solution of the Distributed Termination Problem.

(RAY 85a) RAYNAL M.
Algorithms for Mutual Exclusion.

(RAY 85b) RAYNAL M.
Algorithmes Distribués et Protocoles.
Ouvrage à paraître (Oct. 1985), Eyrolles.

(RAY 85) RAYNAL M.
A Mutual Exclusion Algorithm for a Logical Ring Structure.
(in French)

(RIC 81) RICART G., AGRAWALA A.K.
An Optimal Algorithm for Mutual Exclusion in Computer Networks.

(TRA 82) TRAIGER I.L. et al.
Transactions and Consistency in Distributed Data Base Systems.
COMMUNICATION PRIMITIVES IN DISTRIBUTED SYSTEMS

Andrew Lister
Department of Computer Science
University of Queensland
St. Lucia
Queensland 4067
Australia *

1. Introduction

The problems of constructing distributed computing systems can be tackled from at least two different viewpoints:

(1) the application or programming viewpoint, via proposed models of distributed computation and high level communication primitives

(2) the low level communications viewpoint, via proposed protocols and interconnection architectures.

In recent years considerable progress has been made from both viewpoints, but it is not clear that the groups involved have had each other's interests at heart. In particular, little attention has been paid to the demands made by high level communication primitives on the underlying communications systems which must support them. These demands can have significant impact on the feasibility of implementation, but they are rarely explicitly stated.

The aim of the work reported here is to bring such demands into the open, and to examine their implications. Specifically, the objectives are

(1) to examine the implications of various high level communication primitives (e.g. CSP[3], DP[1], Ada, PLIFS[2]) for the underlying communications subsystem.

(2) to explore the suitability of different primitives for the construction of robust distributed software.

Our method has been to develop a testbed [4] which incorporates

(1) a model of communication and a set of primitives into which all the others can be (more or less) conveniently mapped

(2) some distributed hardware on which to implement the model.

2. The Stream Model

The model developed is that of asynchronous message passing via streams[6]. Any number of processes may send messages on a stream, and any number of processes may receive messages from a stream (but each message is received at most once). Senders and receivers need not be aware of each others' identities, they may be dynamically created and

* Current address: Dept. of Computer Science, University of York, Heslington, York Y01 5DD.
deleted, and their physical location is immaterial. The message passing primitives are

\[
\text{SEND}(\text{message}, \text{stream}) \quad \text{message} := \text{RECEIVE}(\text{set-of-streams}, \text{timeout}, \text{desire function})
\]

where set-of-streams corresponds to the sources from which a message is requested, timeout is anything between zero and 'forever', and the desire-function expresses the selectivity which the receiver wishes to exercise on the messages available (eg. selectivity by type). We claim that all communication primitives can be mapped into this model. Detailed justification of this claim[9] is beyond the scope of this paper, but its plausibility is supported by the following observations.

- Asynchronous communication can readily model other degrees of synchronisation.
- Streams can be placed in one-to-one correspondence with any abstract message source or destination, such as processes, ports, links or mailboxes.
- Arbitrary timeouts, including zero and 'never', are allowed for.
- The desire function can model arbitrary forms of selectivity (eg. by type, transaction key, sequence number, etc).

Implementation of the model is distributed in the sense that status information about a stream is held in any processor containing source or destination processes for that stream. Since processes and streams may be created and deleted there may be temporary inconsistencies in the information held in different processors. Such inconsistencies never lead to communication failure, and can if necessary be resolved through the offices of a "coordinator" for each stream. The coordinator is analogous to the "current synchronisation site" used to control file access in LOCUS[10]. Details of the implementation are described in[5,9].

3. Hardware

The system is implemented on 5 M6809 processors linked by 9600bd serial lines. The 6809s are currently being replaced by 68000s. One processor has a floppy disc drive attached, and can act as a file server. The software is written in C on a Unix host, and downloaded.

4. Results

The communication primitives of CSP, Ada, DP and PL/I/S have been mapped into our model, and various qualitative results obtained[5,8,7]. Generalising a little, the results show

(1) the inclusion of apparently innocent semantic "frills" in a communication primitive can add substantial overhead to its implementation cost, or, worse, lead to ambiguous semantics

(2) models of communication and models of computation do not always display the orthogonality one would perhaps wish for.
5. Future Work

Attention is now being directed at identifying primitives suitable for building robust distributed software. A Remote Procedure Call mechanism has been built on the stream model, and various notions of transaction are being studied.

6. Acknowledgements

It is a pleasure to acknowledge the contributions to this work of Kerry Raymond, Tim Segall, Peter Barnes and Brian Hicks. The project has been supported by the Australian Research Grants Scheme under grants F8015880 and F8215057, and by the Australian Computer Research Board.

References


Remote Service Request and Object Sharing

with Merkur

M. Seifert

IBM, Heidelberg Scientific Center
Tiergartenstr. 15, D-6900 Heidelberg

1 INTRODUCTION

This presentation shortly describes the Merkur facility of the Distributed Academic Computing (DAC) project.

Position in the DAC Architecture
The major goal of the DAC project is to provide the user community of an academic environment with a "compatible" view of services and resources in a network of interconnected heterogeneous computing systems. Within the DAC architecture, Merkur is the basic cooperation mechanism that is used to implement distributed computing applications. It comprises all necessary primitives to perform remote service requests in a unified way, despite of heterogeneous communication architectures and operating systems.

Relation to the OSI Reference Model
According to the ISO OSI Reference Model Merkur is located at the bottom of the application layer. It belongs to the category of "Common Application Service Elements", like the proposal for 'Commitment, Concurrency, and Recovery' (ISO/TC 97/SC 16 N 1713).

It is also comparable to the Remote Operations concept of the CCITT Message Handling System proposals (CCITT X.410, Message Handling Systems: Remote Operations and Reliable Transfer Server).

Relation to Remote Procedure Call
Merkur is similar to the Remote Procedure Call concept, except for the following points:

- A client can pass objects for sharing to a server with specified rights.
- A client retains the control over the shared objects. There is an explicit operation to retract single shared objects, or to retract a whole request thus terminating it prematurely.
- A service request is not blocking the client until completion. There is an explicit operation to wait for completion.
- A server may interleave the execution of requests, since it is not executed as a procedure, but as an independent process.
- A server may refuse to execute requests from some clients, or terminate a request abnormally by deleting the request.
2 OVERVIEW

The Merkur facility essentially allows a client to request a service from a server in a transparent and unified way despite of an underlying heterogeneous computing environment.

It provides a collection of object types and operations on these objects. The supported object types are: \texttt{process, set, signal, event list, port, carrier, account, window, subwindow, notice}.
The operations can be grouped into operations for object and process management, event handling, object sharing and service binding, request processing, message sending, exception handling, and debugging.

The \texttt{process} is the object which issues operations on other objects. A \texttt{set} is used by a process to keep the objects it owns in a group together.

The most relevant objects used in a basic request processing cycle are the \texttt{port} and the \texttt{carrier}.
A client assembles some objects (e.g. data and synchronization objects) and a message in a carrier. The carrier is sent to a particular port and the client process may continue executing.
At the destination port the carrier is enqueued. A server may dequeue the carrier, and then gain access to the objects and read the message included in the carrier.
For the duration of the request execution the objects in the carrier are shared between the client and the server.
On completion of the service the server returns the carrier and looses the access to the objects.

In the following the provided facilities are described in more detail.

2.1 Object Management

For each object type there are several operations available to create or delete an object, and to gain information about existing objects. There is one special operation to get access to a whole set of objects.

2.2 Process Management

Processes can be started and stopped, and a \texttt{halt} operation allows a process to raise an exception.

2.3 Event Handling

Port and carrier are event-type objects which have an implicit synchronization effect for a waiting process.

For explicit synchronization between processes there is a \texttt{signal} object available.
A signal allows one process to indicate the occurrence of an event, and one or more processes to wait on it. Mutual exclusion between processes must be accomplished by a lock, which can be constructed out of one or more signals. For joint progress of some processes again a collection of signals can be used.

Signals, ports, and carriers can be combined into an \texttt{event list} to handle them collectively, allowing a process to issue a multi-wait operation.
2.4 Object Sharing & Service Binding

The processes in Merkur may share objects during some cooperation. The sharing is performed either explicitly by defined offer and share operations, or implicitly by transfer (objects received in a carrier) or inheritance (objects placed in a set).

For explicit sharing, the interested partners can be supported by a directory service (DS) in form of DS agents and directories. A server may offer an object by asking a DS agent to register it with an external name in its directory, and by subsequently issuing an offer operation. A client can now retrieve information about this object by asking a DS agent (with the same external name). If the object is available, then the DS agent delivers a server-related address, where the client can ask for the object by a share operation. The offer can be revoked by a retract operation on the object. An offered object may also be deleted by the offerer, which results in an implicit retract operation.

This object sharing can be used for service binding between a client and a server. This is accomplished by the offering of a port by a server, and the sharing of this port by a client.

2.5 Request Processing

After the establishment of a service binding, several steps are performed to get a request processed:

- Preparation
  The client inserts a message and some objects into a carrier. Possibly, a special account is set.

- Initiation
  The carrier is sent to a port. A server dequeues the carrier, and gains access to it.

- Execution (with Data Transfer and Accounting)
  The client and the server have to adhere to a common way of usage of the objects. The server reads the message and extracts the objects from the carrier.
  The objects are shared and data is read or written. (See below "Data Transfer"). While performing the service the carrier account or some other account is used. (See below "Request Accounting").

- Termination
  On normal completion of the request the server returns the carrier and looses the shared access to the objects. The client waiting for the carrier receives a notification. The deletion of the carrier by the server results in abnormal termination of the request for the client. The retraction of the carrier by the client results in premature termination of the request for the server.

The actions of multiple requests to the same or different servers can be interleaved by clients and servers, since a carrier is used only in one request at a time.
Data Transfer

To exchange data, a window object is provided. The window is a contiguous piece of shared storage, created and owned by one process and shared by other ones according to the granted rights (read, write, read/write). A subwindow object allows a process to define sections in a window.

Request Accounting

Merkur supports resource usage monitoring and recording through an extra account object. The account contains a protected account-number, and allows installation-determined accumulation of service units to configured fields. The automatic update of those resource usage fields in the account object and some other objects, which are reasonably maintained by Merkur itself, is provided.

Every process is bound to exactly one account object at a time. The initial account object is bound by Merkur, or by a process's creator at CREATE(process) time. Also, each carrier refers to an account when being used in a request. A (server) process may switch from one account to another one via received carriers. When such an account is withdrawn, the process suffers an exception and falls back to its created account object.

2.6 Message Sending

An object notice allows a process to send some message to a port without any object sharing. The notice can be dequeued from the port by another process, which can then read the message. For multicast and broadcast purposes the notice may be sent to several ports in multiple operations without blocking.

2.7 Exception Handling

Each process owns an exception carrier. If a process causes an exception to be handled externally, then the process is halted and the carrier is sent to an exception handling port where another process can wait to handle such exceptions. After the handling of the exception the carrier is returned, and the halted process resumes execution.

2.8 Intercepting

Merkur provides the intercept mechanisms which may be used by a process to trace accesses to objects, including deletions thereof. Any attempt to access or withdraw the intercepted object will cause the accessing or withdrawing process to suffer an exception. The exception carrier shall be sent to the interceptor's exception port to handle this intercept exception, and return it subsequently to the intercepted process.
A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing

F. Panzieri and S.K. Shrivastava
Computing Laboratory,
The University,
Newcastle upon Tyne, UK.

Introduction

This report summarises the recent work done at Newcastle on the design and implementation of RPCs (for further details, see [1]). An important issue in an RPC design is the choice of the semantics of the RPC:

(i) At least once semantics: a normal termination implies one or more executions at the called server.

(ii) Exactly once semantics: a normal termination implies exactly one execution at the called server.

Both of the above semantics say nothing about what happens if a call does not terminate normally and it is assumed that zero, partial, one, or more executions (for type (i)) or zero, partial or one execution (for type (ii)) are a possibility. A 'stronger' semantics is specified by the third type given below:

(iii) At most once semantics: same as exactly once, but in addition, calls that do not terminate normally do not produce any side effects.

Choosing appropriate fault tolerance capabilities and the semantics is indeed one of the most important decisions to be taken in an RPC design. We will next consider the problem posed by orphans. Orphans are unwanted executions that occur due to failures such as node crashes. We will use the term abnormal termination to refer to a call that does not terminate normally. Network protocols typically employ timeouts to prevent a process waiting for a message from being held up indefinitely. Assume that a client process waiting for results from the called server has a timer set (or equivalently, some other protocol dependent mechanism that signals the client if no reply is received after some duration). If the call terminates abnormally (the timeout expires) then there are four mutually exclusive possibilities to consider: (i) the server did not receive the call message; (ii) the reply message did not reach the client; (iii)
the server crashed during call execution and either has remained crashed or is not resuming the execution after crash recovery; and (iv) the server is still executing the call in which case the execution could interfere with subsequent activities of the client, as depicted in Figure 1 below.

```
<table>
<thead>
<tr>
<th>NODE A</th>
<th>NODE B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIENT K</td>
<td>SERVER X</td>
</tr>
<tr>
<td>(1st call)</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>* Time out *</td>
<td>&quot;call&quot;</td>
</tr>
</tbody>
</table>
| (2nd call) | "call" | Interference | "work 2"
```

Figure 1: Example of interference caused by a time out.

The client K at node A issues a call to server X at node B that executes the requested work ("work 1" in Figure 1), and the call terminates abnormally before X completes the work. The client then issues another call to some server Y at node B ("work 2" in Figure 1). If the computation by X is still in progress, and "work 1" and "work 2" have data in common, then these computations can interfere with each other. Note that the concurrency depicted in Figure 1 must be regarded as undesirable, since the execution of a sequential program should give rise to a sequential computation characterised by a single flow of control. Concurrency control techniques (e.g. locking) are normally intended to prevent interferences between different programs under the assumption that each program will invoke a sequential computation.

The interference depicted in Figure 1 might also occur in the case of a crash of the client node A. If the client resumes execution after recovery by reissuing the call, or by making a new call to the same node, then we have a similar situation as before. We will refer to unwanted computations (e.g. "work 1") as orphans.

RPC Design

The following primitive has been made available to clients for invoking a remote call (where parameters and results are passed by values):
\texttt{rpc(server:...; call:...; timeout:...; retry:...)}
\texttt{\hspace{1em} var reply:...; var rpc\_status:...};

The "rpc\_status" variable can assume one of the following values:

\texttt{rpc\_status = (OK, NOTDONE, UNABLE);} 

The second parameter contains the name together with the relevant parameters of the operation to be performed by the server whose address is in the first parameter. The retry parameter indicates the number of times the call is to be retried (default value being zero). Let for some call, \( n \) be the value of the retry parameter and \( t \) be the timeout value. Then, if after issuing the call, no reply is received within duration \( t \), the call will be reissued; this process is repeated a maximum of \( n \) times. So, the worst case normal completion time for a call will be at most \( (n+1)t \) units of time. The semantics of the call under status OK, NOTDONE and UNABLE is given below:

\texttt{rpc\_status=OK:} The specified call has been executed exactly once by the server; the result is available in 'reply'. This represents a normal termination of a call, with the call taking at most \( (n+1)t \) units of time to complete.

\texttt{rpc\_status=NOTDONE:} The call has not been executed. This response is obtained when some communication failure prevents the call message from being transmitted to the server; the response is obtained in less than \( t \) units of time.

\texttt{rpc\_status=UNABLE:} At most one execution may have taken place at the called server; 'reply' does not contain any results. This case represents an abnormal termination, with the call taking at most \( (n+1)t \) units of time to complete. It is guaranteed that any computation the call may have generated has also terminated. So, referring back to the previous example (fig. 1), when the first call from the client \( K \) terminates, there will be no ongoing computations for that client at node \( B \).

Finally, we provide the following orphan handling capability to cope with client crashes. Consider a node that crashes and after recovery makes a remote call to some node \( C \). Then, if \( C \) has any orphans because of the caller's crash, they will be aborted before execution of the call starts at \( C \). What if the node remains crashed or after recovery never makes calls to \( C \)? In this case it is guaranteed that any orphans on \( C \) will nevertheless be detected and killed within a finite amount of time. For implementation details of this design, the reader is referred to [1].
Comparison with other RPCs

First of all we claim that our RPC closely approximates the behaviour of local calls. For a sequential program, when a local call terminates either normally or abnormally (an exceptional return is obtained), we do not expect any ongoing activities at the callee: the same behaviour is modelled by our RPC. In a single node system, a crash halts all the ongoing computations. This behaviour is approximated by our RPC as follows: a crash of a node does not 'instantly' stop all the remote calls initiated from the node, rather, when post crash calls are made, any orphans on the called node are first aborted.

The first detailed study of RPCs appeared in the Ph.D. dissertation of Nelson where among other things, a variety of orphan killing techniques were presented but not implemented. The subsequent Cedar implementation also has not addressed the issue of orphan treatment. Cedar RPC supports exactly once semantics, and like our RPC, does not permit a call to terminate normally in the presence of server crashes. However, an abnormally terminated call does not guarantee that the computation invoked (if at all) at the callee has terminated - so no guarantee of freedom from interference for subsequent calls can be given. The same is true when a crashed node, after recovery makes remote calls. Of a few commercially available RPCs [SUN, Courier], the SUN RPC does not specify the call semantics to be supported and has no provision for orphan treatment. Similarly, Courier RPC appears to support exactly once semantics, but its description is not precise about its fault tolerance capabilities and no support for orphan treatment is provided.

We are not aware of any working distributed implementations of RPCs supporting 'at most once' semantics. As far as we can tell it is hard (or pointless) to implement such RPCs without the support of atomic actions, so comparisons with simple general purpose RPCs discussed earlier is not meaningful.

Reference

About the GOTHIC project.

J.P. Banâtre
IRISA/INSA Campus de Beaulieu
35042- Rennes (France)

As some papers in the following refer to the GOTHIC project, it seems appropriate to give some informations about this project.

In recent years, our activity aimed in the design and construction of an application-oriented distributed system: ENCHERE. Because of the specificity of the application this system was not fully general. The main purpose of GOTHIC is the construction of a general purpose distributed operating system from our experience in ENCHERE. Actually, the main characteristic of GOTHIC is the implementation of "nested atomic activities" on a pluri-processors machine.

The three following papers describe some of our present investigations: design of a hardware stable storage, virtualization of a stable storage and nesting activities.
A note on the design of a stable storage facility for GOTHIC

M. Banâtre,
F. Ployette.

IRISA/INRIA-Campus de Beaulieu,
35042- Rennes (France).

1-Introduction.

A stable storage is viewed as a memory device with the following properties:

i) the physical storage is non-volatile,
ii) read and write operations are atomic.

The first stable storage implementation is due to [LAMP-81]. It is based on disk devices for non-volatility and atomicity of read and write operations is implemented by software.

Such an implementation is suitable to store information like files but problems arise when small objects (variables, process states,...) are to be managed since access time is too long. Due to this remark, we have proposed for the ENCHERE system another implementation made out of two components:

i) a stable RAM memory, (SR memory), made of coupled banks, which is a part of the machine address space,
ii) a disk unit which is used to store long-termed information, (files).

The main design problem of the SR memory concerns protection. As this memory belongs to the machine address space, it is vulnerable to any erroneous behaviour of the processor which can result in hardware or software faults. A complete description of the hardware and software protections mechanisms which have been realized can be found in [BANA-85].

However, these two proposals can only manage fixed sized and contiguous objects (i.e. pages) and are not well adapted to the management of typed objects. This lead us to review the design of the stable storage,
considering that an object concrete representation is composed of \( n \) elements (or word) which can be contiguous or non-contiguous in case of structured objects as described in the next section.

2-Structure of the stable storage for the GOTHIC.

We had three goals in mind during the design phase of the stable storage of the GOTHIC system:

i) to build a large stable storage,
ii) to provide short access time (equivalent to a ordinary RAM memory),
iii) to provide objects management implemented in hardware.

Disks seem to be desirable devices according to point (i) but their access time is not acceptable (point (ii)). We propose two approaches:
- The building of a stable storage only from non-volatile RAM memory,
- The building of a virtual stable storage (stable RAM plus disks),

These two complementary approaches can be used in the same GOTHIC architecture, for example, the first one may be integrated to diskless workstations, the second may be used in a mass storage server.

In this paper, we are only concerned with this first approach, the second is described in a companion paper within this report.

Before describing the hardware mechanisms, let’s define more precisely our fault hypothesis:

i) a memory device may decay over time. As a consequence, each object has two copies on two decay-independent banks.

ii) In case of crashes, information stored in a memory bank might be corrupted, (bad addresses, ...). Therefore, we have to ensure that all elements (words) belonging to the updated object, and only these elements, can be addressed by the processor.

Here below is a description of the hardware structure of the stable storage memory.
Stable storage architecture.

On this figure,
- B1 and B2 are coupled memory banks, each of which contains 512K bytes.
- AUT is the hardware automaton which is responsible for the access control on B1 and B2.
- LT is a link table which contains logical links between the different elements of each object located in (B1,B2) as explained later.
- AT is an access table which characterises the elements of the current object Obj under modification. AT[i] is set if \( i^{th} \) element of the current object is being modified.

The two tables LT and AT are only accessed by the automaton AUT. They are not visible from the processor.

When an object is created in stable storage the automaton is put in the initial state "object creation" and memory is allocated on B1 and B2. The LT table is updated such that, \( k \) and \( k+1 \) being two consecutive elements of the object located at real addresses \( p \) and \( q \) respectively, LT[q] contains a logical link to LT[p].

The object update operation follows four phases:
1) The automaton is put in the initial state "object update".
2) The object is read and the two copies are compared in order to detect any eventual decay. The access table is set, so as to
enforce that the only words of stable storage that can be
modified by the processor are those marked in the access table.

iii) The object is written on the first bank element by element.
When the j\textsuperscript{th} element of the object is accessed, it is checked
that the previous written element was the j-1\textsuperscript{th} one. This
mechanism implies that, by the end of the update operation, it
has been controlled that the object has been fully written on
the first bank.

iv) The object is then written from the first to the second bank,
element by element with the same checks as described in step
(iii).

Recovery schemes are provided in case of errors during the
processing of these phases.

3-Concluding remarks.

By now, a full specification of the hardware architecture of the
stable storage is available. As far as performances are concerned, a first
estimation indicates that the time needed to update a stable object is three
times more than the one needed to update a standard object in RAM memory.

References:

The Design and Building of ENCHERE, a Distributed
Electronic Marketing System.

[LAMP-81] Lampson B.,
Atomic transactions.
1. Introduction.

Our decision to build a Stable Object Manager (SOM) for the GOTHIC system is motivated by the fact that stable objects are a useful tool to implement atomic activities or transactions.

Basically, a stable object is represented by two copies. When the object is updated, both copies are written. However, the second copy is modified only when it is known that the first one has been written correctly. Stable objects aim at providing tolerance to crashes and decays.

Traditionally, stable objects are fixed size objects represented on disk volumes (Lampson 81). One originality of the ENCHERE system (Banâtre 83) is to provide a stable-RAM memory made out of two non-volatile memory banks to represent stable information in order to improve object access speed.

The aim of our proposal is to couple such a stable-RAM memory with an auxiliary memory (disk volumes) so that arbitrary size objects can be maintained and accessed in a convenient and efficient manner.

2. Physical resources.

The stable-RAM provides the ability to access atomically stable zones possibly noncontiguous.

The auxiliary stable memory is built out of conventional disk volumes and provides the ability to access atomically stable blocks. Stable volumes are named by volume-ids and can be mounted/unmounted.

The physical resources are summarized in the diagram below, the role of the SOM being to manage these resources at best implementing an appropriate object swapping policy.
3. Object management.

Objects are defined as a contiguous sequence of fixed size pages of bytes (e.g. 512 bytes) and are named by a single OUID ("Object Unique Identifier"). The OUID is a capability.

We require the operations available at the SOM interface to be atomic and as fast as possible idempotent. This latter property is a useful feature for the FOMIC distributed system which makes use of the SOM facilities since operations can be repeated in case of doubt (e.g. time-out expiration) without causing any harm.

Here below are a subset of the intended operations provided by the SOM.

3.1. Object creation.

Object creation takes as input parameters: the object-class, a volume-id and the initial length of the object - the length can be changed dynamically.

The object is chosen out of the standard stability class or the weak stability class. The standard stability class provides tolerance to crashes and decays while the weak stability class provides tolerance to crashes only. It is believed that decays might be very rare and therefore non critical objects can be created in this class. The SOM uses this information to minimise the storage amount in auxiliary memory to represent the object.
The volume-id names the volume which is used both for the object swapping and long-term storage. The SOM applies the principle that an object has a single representation in auxiliary memory so as to optimise auxiliary storage needs.

Some applications need to access few stable objects, generally small, very quickly (e.g. process state). This demand is captured by attributing a volume-id to the stable-RAM itself. An object created on this "fictive" volume will reside permanently in core and will not involve any I/O operations when accessed.

The creation of an object returns the OUID and a short object handle. This latter allows further object accesses to be mapped more efficiently by the SOM: This feature is quite conventional in accordance with file systems usage for instance.

3.2. Accessing objects.

A single page portion can be accessed atomically in reading or writing. Providing atomic access to a sequence of pages, although nice in concept, might turn out to be expensive and not necessary at the SOM interface. Indeed, if atomic access to several pages is needed, this service can be built at a higher level by using some logging technique for instance.

4. Swapping policy.

The stable-RAM is divided into fixed size page frames. When an object page is referenced, the page might be resident or absent. In the latter case, a clean page frame must be found so that the corresponding stable block can be loaded.

The page replacement algorithm we introduce is derived from the clock algorithm implemented in the MULTICS system. Clock approximates a global Least Recently Used replacement algorithm. Moreover, it allows page cleaning to be processed as an asynchronous activity. This latter point is quite important since it enforces the application to be delayed at most for a single I/O when accessing an object.
5. Conclusion.

This proposal for the stable object manager is currently being precised. Many issues such as concurrency control (object locking), garbage collection have not yet been dealt with and it is not known whether the overall performances will meet our expectations in the GOTHIC system environment. However, we believe that our basic decisions such as "virtualising" the stable-RAM and providing a simple interface where objects are seen as a sequence of pages and named by OUIDs constitute a sound basis to build upon.

Bibliography.
(Lampson 81)


(Banâtre 85)

A NOTE ON THE NESTING OF ACTIVITIES

J.P. BANATRE
M. BANATRE
F. PLOYETTE
I.R.I.S.A.
Campus de Beaulieu
35042 RENNES CEDEX

1. Introduction.

Several authors claim that their system implements the concept of "nested action" or "nested transaction" (1, 2, 3). It is clear that "nesting" does not mean the same thing for everybody. In this note, we try to clarify the situation and we define the nesting of activities as implemented in ENCHERE and used in GOTHIC.

2. Classified nesting.

2.1. Procedural nesting.

Procedural nesting may be viewed as a general form of block nesting. If a block B is "nested" within the block A, the following execution scheme is followed:

(1) (A₁ ; (B) ; A₂)

The beginning of A (A₁) is executed, then B is executed and finally A is resumed by execution of A₂. A₂ can access the context set up by A₁. The execution of a procedure A calling a procedure B can be described by (1).

2.2. Parallel clauses nesting.

Several languages allow the following structure:

(A₁ ; ((B₁) || (B₂) || ... || (Bₙ)) ; A₂)

Block A (A₁) is initiated, then the parallel clause B₁ || ... || Bₙ is executed and after completion A (A₂) is resumed. A₂ can access the context set up by A₁.
Procedural nesting is qualified of 1-1 nesting (one caller—one callee) and parallel clause nesting is qualified of 1-p nesting (one caller-p callees). Next paragraph generalizes 1-n to n-p nesting (n callers-p callees).


The nesting of a parallel clause within another parallel clause may be figured as follows:

\[
\begin{align*}
(A_1 & / A_{12} ) \\
(B_1 & ) \\
(A_2 & / A_{22} ) \\
(B_2 & ) \\
& \\
& \\
& \\
(A_n & / A_{n2} )
\end{align*}
\]

The parallel clause \((B_1 \mid \cdots \mid B_p)\) is nested within the parallel clause \((A_{11}/A_{12}) \mid (A_{21}/A_{22}) \mid \cdots \mid (A_{n1}/A_{n2})\). Execution may be described as follows:

\(A_{i1}\)'s sequences of instructions are initiated, and when they have all reached the /, \(B_j\)'s \((j \in [1,p])\) are executed. Upon termination of all \(B_j\)'s, \(A_{i2}\)'s are resumed, with the property that \(A_{i2}\) may access the context defined by \(A_{i1}\).

This form of nesting is the most general and one can realize that 1-1 and 1-n nestings are particular cases of this n-p nesting.

4. Nesting of activities in ENCHERE and GOTHIC.

Activities in ENCHERE may be seen as parallel clauses. In ENCHERE literature (5, 6), they are described as groups of cooperating processes. The ENCHERE system uses the most general form of nesting in order to implement the basic transaction mechanisms of the application (4). However the notion of nested (atomic) activity has not been fully studied and integrated to the system programming language of the application.

In GOTHIC, we have first defined a computational model based on multi-functions which may be seen as the abstraction of the parallel clause, in the same way as procedures are abstractions of blocks. It is possible to call a
multi-function from a procedure but also from another multi-function, thus providing the general form of nesting. Activities à la ENCHERE are implemented by multi-functions.

Here follows the flavour of the structure of multi-functions and of the way they are used. Actually, these notions have been introduced as an extension of PASCAL.

5. Extended PASCAL.

A multi-function definition is described as a generalization of a PASCAL procedure or function:

\[
\text{multifunction } \text{simple}(x : \text{integer}; y : \text{boolean}; z : \text{real}) : (u : \text{boolean}; v : \text{integer}; w : \text{string}) \]

\[
\begin{align*}
\text{var} \\
\langle \text{declarations} \rangle \\
\text{cobegin} \ (x,y)u : \text{begin} \ldots \text{return}(u) \text{ end}, \\
\quad (z)v : \text{begin} \ldots \text{return}(v) \text{ end}, \\
\quad (y,z)w : \text{begin} \ldots \text{return}(w) \text{ end} \\
\text{coend} \; ;
\end{align*}
\]

Multi-functions are the executive objects of our system. A multi-function accepts an input parameter when called and returns an output parameter. Input and output parameter are tuples of values.

The body of a multi-function is a parallel clause cobegin ... coend which can be inclosed in a context defined by the declarations following the multi-function header.

The stagement

\[(l, m, n) := \text{simple}(a, b, c)\]

is the notation given to the call of the multi-function simple. In this example, it is a 1-3 nesting scheme.

It can be generalized to a 3-3 nesting as followed:

\[
\begin{align*}
\text{cobegin} \\
\quad (...) \ldots : \text{begin} \ldots (l, m) := \text{simple}(x=5),(w,v) \ldots \text{ end} \\
\quad (...) \ldots : \text{begin} \ldots (n) := \text{simple}(y = \text{true}),(w) \ldots \text{ end} \\
\quad (...) \ldots : \text{begin} \ldots \text{simple}(z = 8.2) \ldots \text{ end} \\
\text{coend}
\end{align*}
\]
As the multi-function call is "distributed", the input tuple parameter is scattered over the caller parallel clauses. Each part of the actual parameter must be preceded by the corresponding identifier of the formal parameter (as in ADA). The result parts are identified by the dot notation following the input parameter part. The result parts are assigned to the variables of the left hand side of the assignment statement.

One can note that calling a nested multi-function define a synchronization point in the caller multi-function.

We are now implementing these extensions into PASCAL. The resulting language will be the GOTHIC operating system base language. Some more details about multi-procedures can be found in a technical paper under preparation (7).

References.
(1) MOSS E.
 Nested Transactions and Reliable Computing.
  2nd. Symp. on Reliability in Distributed Software and Database Systems,

(2) MUELLER E.T., MOORE J.P., POPEK G.J.
 A Nested Transaction Mechanism for LOCUS.

(3) LISKOV B., SCHEIFLER R.
 Guardians and Actions : Linguistic Support for Robust Distributed Programs.
  ACM TOPLAS, Vol. 5.3, June 1983.

(4) BANATRE J.P., BANATRE M., PLOYETTE F.
 Construction of a Distributed System Supporting Atomic Transactions.
  Proc. of 3rd Symp. on Reliability in Distributed Software and Database Systems,

(5) BANATRE M.
 Le système ENCHERE : Une expérience dans la conception et la réalisation
d'un système réparti.
  Thèse d'Etat, Université de Rennes, Mars 1984.

(6) BANATRE J.P., BANATRE M., LAPALME G., PLOYETTE F.
 The Design and Building of ENCHERE, a Distributed Electronic Marketing System.
  Communication of ACM (to appear).

(7) BANATRE J.P., BANATRE M., PLOYETTE F.
 An Overview of the GOTHIC System.
Workstations in Distributed Systems

David Brownbridge

High Level Hardware Ltd.
PO Box 170, Windmill Road,
Oxford,
England.
OX3 7BN

(dave@hh.AC.UK)
[...mcvax!ukc!hh!dave]

A workstation provides powerful processing and graphics facilities and is usually dedicated to a single user. Communication between workstations is important to allow resources to be shared and information passed between users. At the lower levels, standards are rapidly emerging for local networks. Higher level standards for file and document transfer are under development but there still remains the problem of integrating these facilities into the workstation. The challenge is to use standard communications protocols but to hide the network from the workstation user.

The origins of the concept of workstation are in the Xerox Alto computer. This was one of the first to provide personal computing coupled with powerful graphical display facilities. Such a single user system provided uniform response when the majority of contemporary systems suffered extremes of response time as the time-sharing load varied. The graphical display provided an interface much superior to conventional display terminals. Another essential feature was the provision of a high speed local area network. This allows expensive peripherals (printers etc.) to be shared and for the workstations to communicate with each other. Later systems have improved on the technology and facilities of the Alto but the basic workstation concept remains the same.

I would suggest that the important feature of workstations is the performance and facilities delivered to the user. It is not so important that the whole workstation be build as a single unit or even installed in a single room. This is especially true when current technology requires (potentially) noisy fans or discs in order to achieve good local performance. There is no reason why a workstation should be allowed to intrude more than any other office equipment.

This makes workstation design a very flexible balance between purely local facilities which appear (say) on the user's desk, dedicated but non-local facilities nearby but isolated from the user's environment, and shared resources not dedicated to any one user. The communications between workstations and resources can also affect the balance. If communication is slow compared to processing resources will tend to be concentrated locally. If communication is rapid and reliable it is easier to disperse the system elements.
The user interface is of vital importance; workstations provide "window managers" and other software to make them easy to use. These allow new styles of programming and interaction never previously possible. For example the Blit terminal appears to be very good for UNIX programming because of the good match between the multiple windows on the screen and the multiple activities being performed by the system. In general, UNIX does not provide a good base for this kind of programming because of the strong separation between 'processes' and 'files'. This makes it difficult to cleanly encapsulate a data item and its access routines in the way that can be achieved in other systems (e.g. the Smalltalk environment).

The High Level Hardware ORION workstation provides high processing speed (equivalent to a VAX 11/750) and high performance graphics (40M pixel per second rasterop) together with Ethernet or Cambridge Ring for data communications. This represents a design oriented toward responsive UNIX performance and high resolution graphics. The display appears similar to the Smalltalk or Macintosh environment; transparent distributed processing is provided by the Newcastle Connection.

To conclude, workstation design is a challenge of balance between locality performance communications and cost. As technology advances the balance will change and have a profound affect on the facilities available in a single user workstation system.

Bibliography


"Smalltalk-80 The Language and its Implementation" A. Goldberg and D. Robson Addison-Wesley, Reading Mass. 1983
Flex: RSRE's capability computer and its remote capabilities

J. M. Foster

Introduction

Flex is a new computer architecture which has been developed at RSRE. It is based on capabilities. A number of implementations of the architecture have been made, including one using the ICL Perq2 hardware, and this version is available for evaluation outside RSRE. Flex has been in use for more than four years, and a considerable amount of software exists. It supports a multi-language environment, and compilers exist for Algo168 and Pascal. An Ada compiler is being completed.

The essence of all architectures based on capabilities is that the hardware or the microcode controls access to data at the finest level of detail so as to make sure that only legal operations are performed. This ensures that only operations of the right kind are applied to the data and that only data which is legally accessible can be reached. A main feature of such computers, therefore, is the ability to control access to data. It is clearly important to be able to arrange that certain data can only be be read by the appropriate people, and equally important to arrange that it can only be altered by them. But the idea of capabilities can go much further than that. It is possible to arrange that data can only be used in the appropriate way, that is through the use of particular procedures, special to the kind of data. So we may be able to ensure that information is updated consistently, or only at particular times, or only by people having access to other information, and indeed any kind of check or control that can be programmed can be applied. Furthermore, we can make sure that this is the only way to reach the information. Earlier capability computers, such as the Plessey PP258 (1), the Cambridge CAP computer (2), the Intel iAPX 432 (3) and the IBM System 38 (4), did not offer these possibilities in full.

Flex has extended the notion of capabilities in these earlier machines in three ways. First, the idea of capabilities is used on backing store as well as main store, so that all the kinds of structured object which can be held in main store can also be held on any of the backing stores with the same degree of protection. There are two points here; the data is structured, and it is protected by the capability mechanism. On the Flex file store we can hold any kind of data with the same kinds of data structuring as are available in Algo168 or Pascal; this includes the ability to hold procedure values on the backing stores. Such a file store is in contrast to one which holds only vectors of characters, unstructured binary and directories. It gives Flex a powerful means of expressing database programs. The backing store data also has the
capability protection, including the ability to access data only through procedures, which may be themselves stored on the file store. Flex has an explicit idea of many file stores, it treats each store not as an extension of the main store but as a separate object in its own right.

The second extension of capabilities is to use them across a network of Flex computers, so that capabilities for data in one machine may be passed to and held in another. Once again Flex allows structured data to be passed across the network, again using the same kinds of data structure as in Algol68 or Pascal and again allowing procedure values to be passed. This means that Flex can not only use a remote procedure call protocol like Courier (5), but also pass new procedures around, as parameters or results, so that new dynamically created procedures can be remotely called by the same protocol.

Third, Flex uses true procedure values in the sense of Landin (6). Procedures in most machines are not free values, they can only be used in a particular context or environment and are not defined outside it. This means that the full benefits of capabilities are not achieved, for, though a procedure may control access to data, the environment necessarily also has access to it. In a system using true procedure values, where the procedure can be cut loose from an environment, this no longer applies, and the procedure can contain the only pointer to an object in the computer. So this allows Flex fully controlled access to data. In fact, Flex also uses procedure values as the mechanism for providing an object-oriented machine architecture. Indeed Flex does not use the notion of programs, but uses procedures throughout, programs being merely the special case where a procedure has no parameters and no result.

There is an additional benefit of using capabilities for addressing which helps the designers of interactive systems. It is possible to arrange that the data for many procedures should coexist in the store and to allow values from one to be used freely in the other, provided that the capability rules are met. So in the middle of running an interactive procedure, the user can pick up some of his data, apply some completely unanticipated procedure to it in order to investigate or modify it, and deliver results which can be incorporated in his original data. It is the essence of interactive programming that the course of action is not totally foreseen when the user starts, so this consistency of data addressing combined with the security given by the capabilities is one of the most noticeable features to someone who is using, rather than programming, Flex.

On top of this architecture has been built a programming support system of considerable size and power. About three hundred modules are available to users, varying from large ones, like the compilers, to small ones, like the conversion of integers into their character representation.
This programming support uses a universal system of types, similar to but more powerful than that of ML. These types are sufficient, not only to describe all the values of the compiled languages (Algo68, Pascal and Ada) but also accomplish the much more challenging task of describing the values used in the command language. So all operations are type checked. The same system of types controls the displaying of values, that is, any value can be examined and displayed in a standard way. This is particularly useful in run_time diagnosis, where any value can be picked out of a run_time environment and displayed or otherwise used.

The same types again control the movement of data across the network between computers. As in Courier, Flex uses a remote procedure call protocol and can send structured data across the network. But the sending of the structures in Flex is controlled by the Flex type system, so the objects which can be transmitted consist of those which can be used in the command language and a fortiori those which can be used in Algo68, Pascal or Ada.

References


(2) M.V.Wilkes, R.M.Needham "The Cambridge CAP computer and its operating system" North_Holland 1979

(3) Symposium on Architectural Support for Prog. Languages and Operating Systems. Sigplan Notices Vol 17, No 4


(5) "Courier: The remote procedure call protocol" Xerox Corporation, XSIS 638112, December 1981

(6) P.J.Landin "The mechanical evaluation of expressions" Computer Journal Vol 6, No 4, p308-320 (Jan 1964)
INTRODUCTION TO THE "SATURNE" PROJECT

A FAULT- AND INTRUSION-TOLERANT DISTRIBUTED SYSTEM

Yves DESWARTE  Jean-Charles FARRE  Joni FRAGA  Jean-Claude LAFFIE  David POWELL

LIGANTRIERE D'AUTOMATIQUE ET D'ANALYSE DES SYSTEMES
L.A.S. - C.N.R.S.
7, avenue du Colonel Roche
31077 TOULOUSE CEDEX
FRANCE

INTRODUCTION

The spectacular development of local area networks (LAN's) is focused today on three main areas:
- Industrial LAN's which fit the geographic distribution of operations required for process automation, process control, flexible manufacturing, ...
- LAN's for software engineering, connecting personal computers or more powerful computers, providing large scale storage, efficient computing, input-output facilities, ...
- LAN's for office automation improving internal communications within companies.

Each of these three domains leads to specific problems and requirements. However, all of them share a common need of data integrity, which is the aim of the SATURNE project. The causes which may affect data integrity that are considered here are (accidental) faults and (malicious) intrusions.

The need for computing systems which are reliable (with respect to faults) and secure (with respect to intrusions) is becoming more and more apparent. An example is given by the 'Newcastle Connection' [RAN 84]. Our project is aimed at contributing to the solution to this problem by exploring joint fault- and intrusion-tolerance.

I. REQUIREMENTS FOR LOCALLY-DISTRIBUTED SYSTEMS

1.1 Fault-tolerance

While the importance and the complexity of distributed applications or LAN's are increasing, the penalty due to faults in the computer or in the communication system becomes more and more important, either if data is corrupted or if the computer goes down. In more 'critical' applications, this fact becomes an imperative concept: even in case of a failure of one or more of the computing elements in the system, processing must be carried out correctly and the results transmitted within bounded delays: the system must be fault-tolerant.

According to the environment previously described, the most convenient fault-tolerance technique is error compensation or masking: processing and communication must be carried out with sufficient redundancy to prevent delivery of an incorrect result without using specific procedures to detect and recover from a fault. Majority voting mechanisms and correcting codes are examples of the error masking technique. The major drawback of error masking is the increased complexity of the processing and communication overhead continuously involved even in the absence of any effective errors. For instance, majority voting on N identical copies of one process leads to exchange in the order of N² messages (for each functional message).

The notion of error latency must also be taken into account when designing the system. Error latency can be defined as the delay between the occurrence of the fault and the resulting error. Many studies [CAS 81, YVE 82, ...] show the important correlation between fault occurrence and system load, either for hardware or software faults. The main explanation is that any hardware fault which occurs during a period of low activity has a high probability of resulting in an effective error when the system is highly loaded. In the same way, software errors have a high probability of appearing when the system is highly loaded. Actually, it is when you need most of the power of the system that it has the highest probability of failing. It is not necessary to invoke Murphy's law to explain this fact. Reduction of error latency improves the trust you may have in the system.

1.2 Intrusion-tolerance

Whereas more and more "sensitive" applications are being developed (i.e. concerning confidential data), the increasing number of users of large scale networks and LAN's make it more difficult and less efficient to use intrusion avoidance mechanisms like identification, access control, protection... It is thus necessary to develop intrusion tolerance techniques: rather than prevent a malicious or unintentional intruder from accessing confidential data, it is better to make the data meaningless for any intruder.

Cryptography is a classic example of such a technique, giving a substantial overhead in terms of processing and data size. In the same way as for fault handling, avoidance and tolerance mechanisms are not exclusive, but are complementary.

II. OBJECTIVES OF THE PROJECT

The aim of our project is to evaluate both of these techniques: fault tolerance and intrusion tolerance.

The technique used for fault tolerance is what we call SATURNE (where the name of the project: SATURNE = SATURATION NETWORK) i.e. a technique for the creation of multiple copies of an active process in the system so that the following criteria are satisfied:
- every site in the network runs at most one copy of a process,
- the most critical processes are executed with the highest number of copies,
- masking is obtained by voting on the exchanged data and remanent information.

Obviously, the number of activated process copies varies dynamically according to the creation and completion of other processes. This technique presents two major advantages:
- at any time the redundancy level of each process is maximum, according to the number of sites available in the system; so, masking mechanisms reach optimum efficiency,
- since every processor runs one process, error latency is minimum, for hardware faults as well as for software faults.

The second technique we want to evaluate is called fragmentation and scattering which is a confidentiality technique proposed for message transmission in meshed networks [KOC 82]: each message is split into several fragments sent to the receiving nodes by various routes. Only the receiving node receives all the fragments and is able to reconstitute the original message. A similar technique might be used on local networks such as rings or buses: each copy

ACKNOWLEDGEMENT: The SATURNE Project is a joint L.A.S.-I.N.R.I.A. venture and is partially supported by the "Cooperation, Concurrency and Communication" C3 program of C.N.R.S.
of one process creating only one fragment of any message; for example, this fragmentation operation should be done so that only the receiving process is able to reconstitute complete messages. Our aim is to extend the fragmentation principle to data storage policies on different sites; with such a technique, logical files will be fragmented and the fragments will be scattered among several archive sites [FA 85].

These two techniques should lead to an important overhead if a broadcast network is used. The same would not be true using a meshed network.

The system interface presented to application programmers must be simple: the management of process copies will be hidden at the programming interface level. This facility means that we must ensure transparency for process creation, communication, process removal, etc. The number of copies of one process will change dynamically from time to time according to processor load and process criticality (greater criticality for more critical processes). This leads us to define a robust distributed scheduling algorithm with multiple criteria (criticality, system load, efficiency, processor specialization).

A close collaboration with the SCORE project at INRIA should be fruitful, due to the results which were obtained in similar topics (SICMA action on PPPLI, [SED 82]).

To express parallelism on a network we decided to cooperate with the CHORUS team at INRIA (a group designing a distributed architecture model) for the following reasons:

- CHORUS is quite easy to implement and to validate (there are few primitives), CHORUS has been implemented on different machines in many environments…
- several complex applications have been developed using CHORUS, these might be used to evaluate our system,
- the CHORUS team has already investigated fault-tolerance and obtained some results in this area [BA 85]; our system might be seen as a fault-tolerant version of the CHORUS system.

The next section gives a brief overview of CHORUS.

III. OVERVIEW OF THE CHORUS DISTRIBUTED SYSTEM

In the CHORUS architecture [BA 85], a distributed system is a set of autonomous active entities, called actors, distributed across a network of computers. An actor is like a sequential process, and includes code, data, and a context.

The entire system is modular. The operating system itself is made of actors ("system actors"), in charge of managing the logical and physical resources. An application installed on CHORUS is also made of actors.

The system is flexible. Actors are highly independent: they run concurrently, they do not share data and they interact by the asynchronous exchange of messages. The use of logical ports as the only communication interface ensures that processing (within actors) is independent of communication. Ports are dynamically attached to actors. A message is exchanged by two actors in the following way: the first actor sends it from one of its ports to a port of the second actor. Ports are bidirectional and may be used both for sending and receiving. The sending actor addresses its messages to the destination port and does not need to know the identity of the destination actor. Port names are globally unique; they are related neither to the location of the port, nor to the name of the actor owning this port.

The CHORUS architecture was designed to make the implementation of fault-tolerant services easy. An actor operates as a sequence of execution granules, called processing steps. A processing step is triggered in an actor on reception of a message on one of its ports. Conversely, messages prepared by an actor are transmitted only upon completion of the processing step; no message is transmitted if the processing step does not complete. A processing step is sequential and the actor performs processing steps one after the other.

The CHORUS operating system consists of a set of actors. On each site are present at least those actors managing local resources and a kernel which supports the execution of actors and handles local communications. An application is built as a collection of actors which utilise services provided by the kernels and by system actors. This homogeneous structure is shown pictorially in figure 1.

IV. FRAGMENTATION-AND-SCATTERING MECHANISMS

The fragmentation-and-scattering technique has been studied within the context of a fault- and intrusion-tolerant distributed file system [FA 85]. We shall summarise here the description of this system.

The application of intrusion-tolerance is being considered for a distributed system comprising a local area network of personal workstations (user sites) and specialised computers (service sites). The individual workstations correspond to the "physical areas" of their users. A user has full authority over the objects stored in his machine.

The system contains two sorts of specialized computers: archive sites (with mass storage devices) and security sites (figure 2). The archive sites and the security sites contribute to the formation of a confidential and dependable distributed file system. This distributed file system is essentially made up of four services: the file management service, the fragment service, the directory service and the authentication service. This organization and the use of physical isolation permit an increase in the availability and the security of the stored data.

V. SATELLITE PROJECT ORIENTATIONS

First of all, we must say what we do not intend to do:

- we do not intend to design new techniques for ensuring database consistency within a multi-access environment,
- we do not intend to design a general method for transaction scheduling, with atomicity, consistency, ...

but, our fault-tolerance techniques will make these problems easier to solve: the error-masking mechanisms enable the design of more efficient and simpler protocols to achieve these aims.
We have already mentioned that a broadcast network will be more efficient for implementation of the saturation and fragmentation-and-scattering mechanisms. We thus assume the availability of such a network, but we do not require this network to provide reliable broadcasting, i.e., we assume that:

- a broadcast message may be badly received, or not received at all, by some (few) destination sites,
- a surplus (fictional) message may be received by some (few) sites.

As a matter of fact, we think that our error-masking mechanisms will make us tolerate this kind of fault as well as processing faults. That means that, as long as only a small number of sites is concerned, processing and/or transmission errors will be masked.

However, some transmission errors are liable to defeat this masking policy, such as:

- network blocking,
- network partitioning into several isolated subnetworks,
- network overload, due to a faulty site,
- ...

These kinds of errors must be tolerated by specific means, at the network architecture level and at the protocol level, such as double-bus, double-ring, by-pass switches, ...

Our techniques intend to provide a facility, based on majority voting, for processing and transmission with such a high quality that it will be possible to simplify the design of higher level services: distributed data bases, transaction management, ...

Moreover, we shall provide such mechanisms that will make an eventual intruder unable to get any significant information, even if he manages to access one or several sites.

V.1. Allocation of sites

As we have already mentioned, we have chosen the CHMHS model for the design of distributed applications. In the SATURN system, several copies of actors on different sites execute the same processing step, and the information messages transmitted by the processing step (at its completion) are voted according to a threshold majority voting scheme.

Actors consist of several sequential programs (which are to be executed as 'processing steps'), which may have permanent data. Updating actor permanent data must induce the same updating in all copies of the actor, so as to preserve the same execution of the actor for the next message whatever copies are replicated in this new executed. This permanent data updating is done through specific messages, transmitted at the end of the processing step on which voting is applied, just as on information messages.

The allocation of sites to execute sequences must obey several rules:

- as much as possible, at each time, all sites must execute the same processing step (saturation), so that the number of copies of each active sequence can vary during the execution, but will be as great as possible (highest possible redundancy),
- the minimum number of copies of each processing step is given by a parameter of the message initiating the processing step; the 'criticality' of the processing step; a processing step can be executed only if the number of its copies is greater than or equal to its criticality,
- as much as possible, a load balance must be achieved so that the number of copies of each processing step is proportional to its criticality,
- error latency will be minimized by using detection and diagnosis mechanisms (discrepancy detection by vectors, self-test programs running on idle sites, error-logging on observer sites [NYA 82], ...),
- conclusion between the various processing steps of an actor must be obeyed, so that, if a processing step is active on certain copies of an actor, no other processing step can be launched on other copies of that actor.

- the software involved in the site allocation must be distributed among all sites, so as to prevent "hard core", but the execution of the various copies of this allocation software must be consistent; the consistency is realized through voting mechanisms.

The sites are divided into several classes such as user sites, archive sites, security sites, specialized processing sites, ... Every site of any class possesses a copy of every actor of the class. When a message sent to an actor of the site class is transmitted on the broadcast network, the receiving device of the site takes it and gathers it with the other messages of the same kind (identical messages transmitted by various copies of the same processing step). When the number of identical messages is over the threshold, the local 'scheduler' is activated to decide if the site must execute the corresponding processing step or not, depending on the activity of the site and of the other sites of the class, and of the criticality of the new message and the criticality of the current processing step.

The different phases of a processing step are:

- Reception of the message
- Voting / Scheduling
- Processing with multiple copies
- Message transmission

The two possible states of a site are 'idle' and 'active' (Figure 3). At the moment of its connection to the network, a site is in the idle state. When a message is transmitted on the broadcast network for an actor of the same class as that of the site, and when enough identical copies of the message are received (threshold voting), the site becomes active and executes the corresponding processing step.

Figure 3

Site state transitions

When a new message, aimed at an actor of the same class, is voted, the local scheduler must decide either to stop the current processing step and to take the new one, or to continue the current processing step (depending on the scheduling rules which have already been given). When the current processing step completes, if there is another waiting processing step, or if the finishing processing step sends new messages for the same class, the site remains active and runs a new processing step. If there is no candidate, the processing step, the site becomes idle.

While idle, a site is runs self-test programs, so as to detect latent errors which are unlikely to be detected by application programs (higher addresses of memory or of disk space, infrequent instructions, ...).

To describe the global scheduling rules, let us take a simple example (Figure 4). Let us suppose that at time 0, all the n sites are idle (we consider only one class). At time 1, a message is taken into account (voted). All the sites become active and execute processing step 1. At time 2, message 2 arrives, with the same criticality as message 1 (criticality: i.e., i/n). Then, half the active sites (running process 1), must abort their current process and execute process 2 (load balance). At time 3, message 3 arrives with the same criticality. Then i/n sites must execute each of the three processing steps. Process 2 then completes without sending any new messages. Since there is no pending messages, the i/n sites which were executing process 2 become idle. At
time 4, message 4 arrives, with a criticality greater than n/3, such that:

\[ \text{criticality} (\text{mess.1}) \times \text{criticality} (\text{mess.3}) \times \text{criticality} (\text{mess.4}) > n \]

It is impossible to run all three processing steps so message 4 must wait until there are sufficient computing resources. This occurs when process 3 completes without sending any new messages. All the idle sites and all the sites which were executing process 3 will now run process 4. When process 1 completes (without sending messages), all the corresponding sites become idle. When message 5 arrives, with a criticality nearly equal to the criticality of message 4, such that:

\[ \text{criticality} (\text{message 4}) \times \text{criticality} (\text{message 5}) = n \]

then all the idle sites and some of the sites running process 4 will execute process 5.

![Saturation scheduling (example)](image)

In this example, we can see that load balance is not always achieved (e.g., when process 1 and 4 are running concurrently). This is due to the fact that idle sites can only take new messages, and that the requirement for "saturation" (all the sites must be as active as possible) is stronger than the requirement for "load balance".

In order to maintain consistency in the local scheduler decisions, remanent scheduler data are voted at every decision date. Initiation of a processing step, end of a processing step, queuing of messages, etc.

V.2. Application of fragmentation-and-scattering

The study of the fragmentation-and-scattering technique that we have presented in section IV will continue in the SATURE project but particular attention will be focussed on three points:

- more than three site classes;
- access to parts of files;
- use of a broadcast communication system.

In the design of the fragmentation-and-scattering technique, only three site classes were considered: user sites, security sites, and archive sites. In the distributed computing environment of SATURE, we must consider other special site classes. User sites will be the only "accessors" of the files, and the rights to read/modify/fill files could be transmitted from user sites to other sites (e.g., number-crunching sites). So, the "scattering" cannot be the privilege of the user sites; it should be "dexterous in action!"

The fragmentation-and-scattering technique was first designed with an object "granularity" (a access element) equal to the file. This constraint was reasonable in a highly protected system with little concurrency and sharing of objects. This is not allowed for highly concurrent distributed systems. The granularity of archived objects should be much smaller: we introduce a notion of "working set" as being the transfer element between archive sites and user or computing sites. Each working set should be handled in the same way as the complete (logical) file was handled in section IV: each working set has to be fragmented and scattered among the various archive sites. But, of course, the directory service operates on the whole file (not on the working set). Two opposite policies can be adopted:

- the working set depends on the structure of the records of the file; the access time is reduced, but the (intruder) observation of the transfers can give information on the file structure so confidentiality is sub-optimal.
- the working set size is fixed, whatever the structure of the file; in this case, confidentiality is nearly the same as with a granularity equal to the whole file but the access time is sub-optimal.

The use of broadcast communication system should greatly reduce the overhead of the replication of the fragmentation-and-scattering technique for instance, a single transfer of a fragment could transmit all the replicates to all the destination archive sites. For the scatterings of files, we contemplate the possibility of using algorithms similar to those used for scheduling (see paragraph V.I); when writing a fragment, the write requests can be received by all the archive sites, each then having to locally decide whether or not to store the fragment, depending on the "state" of the site, and of the knowledge of what the other archive sites will do. In order to respect dispersion and concealment constraints (as defined in [FRA 85]), the archive sites could be divided into several classes so as to prevent any single archive site from storing all the fragments of a working set. On the other hand, the cryptographic keys should not be linked to "static" port associations (one user port - one archive site port), but must be made flexible (made less confidential): one unique transfer should transmit all the fragment replicates to all the destination sites. Another key system will have to be designed.

CONCLUSION

The SATURE project will experiment new techniques to tolerate both accidental faults and malicious intrusions. The fragmentation technique should reduce error latency and provide the highest possible redundancy at any time, and then achieve efficient error-detection through threshold voting. We hope to reduce the high number of messages induced by these techniques, by using broadcast networks.

The fragmentation-and-scattering technique should provide a high resistance to intrusions of archive sites, and the use of broadcast communication should reduce the overhead it implies. But the cryptographic technique cannot be as efficient on a broadcast communication system as on a point-to-point system.

REFERENCES


BAN 85 J.S.BASHING, J.G.FABRE, J.M.GUILLEMITOT, G.KIRKEST, M.KOGIER: "Some fault-tolerance issues in the CHARS distributed system", Proc. 5th Int. Conf. on Dist. Comp. Systems, Denver (USA), May 1985


FRA 85 J.P.FRAGA, D.P.CHELL: "A fault and intrusion-tolerant file system", Proc. 3rd Int. Conf. on Comp. Security (IFIP/Sec'85), Dublin (Ireland), Aug. 1985


RAN 84 B.RANDEL: "Fault tolerance and system structure", Proc. 4th Jerusalem Conf. on Information Technology (JICT), May 1984, Jerusalem, pp.192-197

Distributed Academic Computing
M. Seifert
IBM, Heidelberg Scientific Center
Tiergartenstr. 15, D-6900 Heidelberg

1 INTRODUCTION

The activity presented is a joint research project between the University of Karlsruhe and the IBM Heidelberg Scientific Center. A prototype of a university network consisting of IBM 43xx computers, DEC VAX computers, and a large set of (intelligent) workstations will be installed on the campus. The systems will be connected by a backbone net (token ring) and some local nets.

The computers and workstations will initially run under the operating systems VM/SP, VMS, VM/PC, and PC DOS. As part of the project activity, these operating systems will be extended to provide general resource and service sharing facilities in the university network.

The prototype system will be used in various departments of the university to make experiments in supporting real applications.

A main purpose of this project is to contribute to the understanding of the requirements of a computing environment in a university.

This environment can be characterized as follows:

- The computing resources are dispersed over large computing centers, computers in the departments, and workstations in offices and classrooms.
- Due to the geographic distribution, the computers and workstations are connected by various local area networks.
- The installed hardware and software is heterogeneous and will stay so.
- The "owners" of the computers are mostly autonomous in operating their systems.
- There is some demand for communication and resource sharing over the whole campus and with external partners.
2 OVERVIEW

In the course of the project a prototype of a network operating system and some network applications will be built to allow a user to request services and to share resources in a unified way, despite of the underlying heterogeneous computing systems.

Though standardized interfaces and protocols will be used as far as possible in the implementation, the user should keep a "compatible" view of the new services and resources, and most not learn a completely new set of commands and system calls. Instead, after issuing some set-up commands or calls, the user should be able to access network resources or services in the same way as the local ones in the environment he was always working in. This results in a homogeneous and transparent view of a heterogeneous environment.

The development of the prototype encompasses at present the following components.

2.1 Kernel Services

The kernel services encompass process management and event handling functions to provide an unified interface for all DAC components to the diverse local operating systems.

2.2 Transport System

The interface offered by the transport system is a "reliable datagram" service. It allows processes to communicate in a connection-less mode and still have sequencing and retransmit characteristics for the messages.

At present, the system is based on the DoD TCP and IP protocols for inter-computer communication, but will change to ISO OSI transport protocols in the future.

2.3 Presentation Services

Due to the heterogeneous systems, presentation services are provided to transform the data to the required local representation during a communication.

The services are a connection less variation and extension of the proposed OSI presentation facilities.

2.4 Remote Service Request Facility

This facility is the basic cooperation mechanism that is used to implement distributed computing applications. It comprises all necessary primitives to perform remote service requests in a unified way, despite of heterogeneous communication architectures and operating systems. (See the companion abstract.)
2.5 Directory and Name Service

The basic directory component will provide functions to register services and network objects and to ask queries about them. The necessary authentication and authorisation functions will be supported too. Later extensions will provide user guidance and information management for the whole DAC network.

2.6 Global File System

The DAC virtual file system offers network wide access to files and provides file maintenance services. There are different views of the virtual file system, each being a natural extension of some local file system. So, the user has a compatible view of all files, no matter if they are local or kept by some file server on a heterogeneous computer.

2.7 Distributed Programming Environment

The Remote Service Request facility can be used easily to provide remote procedure call services in Pascal, Fortran or C. The implementations will use the "stub" generation technique.

3 PROJECT STATUS

The project has been started in May 1984. Three IBM 43xx (1 x 4341, 2 x 4361) and a VAX 750 have been installed and connected. A number of IBM PCs and PC/XTs have been distributed over the campus, and are used for computer-aided courses. A prototype version of the Zurich token ring is used to connect PCs and a 4361.

The overall DAC architecture has been designed, and first versions of the kernel services and the transport system are under test on IBM (43xx and PC) and VAX computers. The most other components are designed and preversions are coded.
Gerard Vandome  
Bull Research Center  
c/o Imag  
BP 68 - 38402 Saint-Martin d'Hères  
France  
tel (33) 76514857  

During the Newcastle Workshop I have presented two Esprit Projects in which Bull is involved. The first one, the Rose Project, is now an operational network. The second one, the CSA Project, is just starting and I am not allowed to give informations on it. So I just give my personal view of the project, rather than the Project Point of view... These two projects are briefly described below.
1. ROSE Project

The ROSE (Research Open System for Europe) project aims at a maximum connectivity of potential users through the use of Open system Interconnection ISO services and protocols, and starts with an implementation under the UNIX operating system.

The project is performed by a consortium of six partners: Bull, GEC, ICL, INRIA Olivetti and Siemens. It implements an operational network providing message passing, file transfer, remote login and remote job execution between the contractors.

Initial connectivity is provided between UNIX or UNIX-like systems by UNIX communications facilities (cu and uucp) used over X25 networks. In parallel an OSI session service is developed, relying on ISO session and transport protocols over X25 and Ethernet networks. This will give to UNIX the visibility of an ISO Open System.

ISO and CCITT defined applications such as file transfer and message handling system are implemented over the session interface. This will enable communication with non-UNIX systems having implemented OSI protocols. New applications are planned as soon as the corresponding protocols become available.

Asynchronous terminals can be connected through PSTN or the PAD service. PTT telematic services will be directly connected up to the point where they support OSI protocols. Connection to non-OSI proprietary network architectures will be possible through specific gateways.

The ROSE Architecture is strictly derived from the OSI Reference Model, with some implementation choices, mainly for Local Area Networks and Wide Area Networks interconnection. The ISO class 4 Transport protocol is used on LANs and the ISO classes 0, 2 and 3 on X25. LAN-WAN interconnection is provided by a transport relay which masks all protocols at, and below, the transport layer in the LAN. Only the protocols operating on the LAN above the transport layer are externally visible.
2. The CSA Project

The Communication Architecture Project (Esprit Project No 237) will define an Architecture which will provide for integrated service communications including voice, text, data and image for both intra and inter-office communications.

This Project will define the use and the interconnection of emerging networks (PABXs, LANs, ISDN...), and will provide interworking between existing heterogeneous Operating Systems (UNIX, MS/DOS, CP/M...). The Project doesn't intend to define new communication medias, and will use as much as possible existing protocols.

The project is performed by a consortium of 5 partners: Plessey (main contractor), Bull with INRIA as subcontractor, MARI, Philips and SG2.

The proposed Architecture will be logical and will specify communications within and between sets of shared resources and systems. The logical architecture will be based on the concept of a domain which consists of a number of interconnected sub-systems under the responsibility of a single administrator. The sub-systems in a domain are not necessarily located in the same geographic location and may share a common subnetwork, or a domain may utilise a number of interconnected sub-networks: there is no mandatory relationship between the logical and the physical topologies.

In the Project intra-domain communications (communications inside a distributed end system) are distinguished from extra-domain communications. The extra-domain communications can be between CSA domains (inter-domain communications) or with the outside world (OSI or non-OSI based extra-domain communications).

This Project is a two years duration project which must define detailed specifications in order to be able to produce a prototype in an additional 3 years phase.