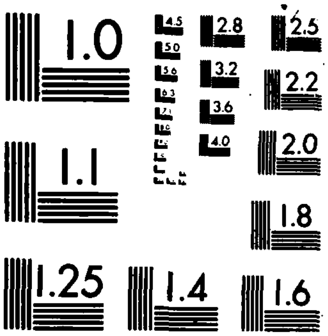
AD-A141 501   SOFTWARE SUPPORT FOR FULLY DISTRIBUTED/LOOSELY COUPLED   1/3
PROCESSING SYSTEMS..(U) GEORGIA INST OF TECH ATLANTA
SCHOOL OF INFORMATION AND COMPUT..   P H ENSLOW ET AL.
UNCLASSIFIED   JAN 84 GIT-ICS-82/16-VOL-2   F/G 9/2   NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-83-238, Vol II (of two)
Final Technical Report
January 1984

SOFTWARE SUPPORT FOR FULLY DISTRIBUTED/LOOSELY COUPLED PROCESSING SYSTEMS -Appendix - Selected Papers

Georgia Institute of Technology

Philip H. Enslow, Jr.; N. J. Livesey; Richard J. LeBlanc
and Martin S. McKendry

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-83-238, Vol II (of two) has been reviewed and is approved for publication.

APPROVED: Thomas F Lawrence

THOMAS F. LAWRENCE
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER RADC-TR-83-238, Vol II (of two)	2. GOVT ACCESSION NO. AD-A141 507	3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
SOFTWARE SUPPORT FOR FULLY DISTRIBUTED/LOOSELY COUPLED PROCESSING SYSTEMS - Appendix - Selected Papers

5. TYPE OF REPORT & PERIOD COVERED
Final Technical Report
20 Aug 81 - 31 Dec 82

6. PERFORMING ORG. REPORT NUMBER
GIT-ICS-82/16

7. AUTHOR(s)
Philip H. Enslow, Jr.
N. J. Livesey
Richard J. LeBlanc
Martin S. McKendry

8. CONTRACT OR GRANT NUMBER(s)
F30602-81-C-0249

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Georgia Institute of Technology
School of Information and Computer Science
Atlanta GA 30332

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
31011G
R2440101

11. CONTROLLING OFFICE NAME AND ADDRESS
Rome Air Development Center (COTD)
Griffiss AFB NY 13441

12. REPORT DATE
January 1984

13. NUMBER OF PAGES
220

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
Same

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Thomas F. Lawrence (COTD)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed System Support Capabilities
Fully Distributed/Loosely Coupled Processing Systems
Software Development Tools

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The development and operation of very loosely-coupled distributed processing systems presents several new challenges. These "Challenges", or differences from the techniques applicable to centralized systems, result primarily from the environment that is involved -- a multiplicity of logical and physical resources that are very loosely-coupled, a highly distributed and decentralized control system, and the autonomous and asynchronous operation of the various components. This report identifies

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

the system support capabilities necessary to support the design, analysis, implementation, operation, utilization, and management of fully distributed, loosely-coupled, data processing systems. These system support capabilities are divided into three categories - software development support tools, distributed system design facilities, and operational support capabilities. Selected support capabilities are described, together with the rationale for the services that they will provide. Estimates are presented for the resources and facilities required to design and implement selected support capabilities. Also provided is a development priority for the support capabilities. ~~The~~ appendix to the report (Volume II) contains several papers providing in-depth discussions of various support capabilities and their features.

ACKNOWLEDGEMENTS

Support for the preparation of this report and most of the research on which it is based was provided by the U.S. Air Force, Rome Air Development Center under contract F30602-81-C-0249. In addition, general support for the Georgia Tech Research Program in Full Distributed Processing Systems has been provided by the Office of Naval Research under contract N00014-79-C-0873 as part of the ONR Selected Research Opportunities program. Dr. Enslow has also received support as a consultant in the area of software tools for embedded distributed systems under RADC contract F30602-81-C-0142, "Distributed Processing Tools Definition Study," with General Dynamics, Data Systems Division, Fort Worth, Texas.

VOLUME 2

TABLE OF CONTENTS

VOLUME 2

LIST OF FIGURES

LIST OF TABLES

VOLUME 1

TABLE OF CONTENTS

VOLUME 1

LIST OF FIGURES

LIST OF TABLES

APPENDIX A

EXTENDING FILE SYSTEMS TO DISTRIBUTED SYSTEMS

N. J. Livesey

A.1 INTRODUCTION

This paper examines the so-called 'meta-system' approach to Distributed System construction; that is, constructing a distributed system by utilizing existing local operating systems. In particular it looks at some of the file system problems that may be encountered when such an approach is followed. These problems are simplified if one has the opportunity to rewrite the underlying local operating system (see [Oppen 81), for example], but typically, this is not the case. Since it is impractical to look at all existing local operating systems, I focus on a particular local operating system, Primos, with its overlying user interface, the Georgia Tech Software Tools Subsystem. However, these comments do not apply to just this environment; much of what is said is probably true for many existing local operating systems.

A.2 GENERAL PROBLEMS

The problem with most existing local operating systems is precisely that they do preexist the design to distribute. Although a local operating system should have some autonomy, it should also have been designed with an eye to integration, if it is to be useful in a distributed system. There seem to be two classes of problems in extending existing operating systems to new purposes:

"Deficiencies" in the existing local operating system can make it very difficult and involved to perform new functions in a reasonable way. There are very few things that any operating system will make it impossible to do, but if the system was originally built without them in mind, it can lead to contortions, and contortions lead to inefficiencies.

"Biases" in the design of an existing system can often lead one to extend it in certain ways, without fully exploring alternatives which might be equally valid, and the ease or difficulty of adding features to an existing structure can close off debates on the best new features to add.

The solutions suggested in this paper are intended to avoid these problems at minimum cost, rather than to produce radical, but expensive, solutions.

A.2.1 Naming and Addressing

"Meta" distributed operating systems are produced by introducing a network operating system on top of previously separate local operating systems. This network operating system must at least make it possible to allow a job on one machine to access files on another, and it should preferably allow a user on one machine to run a job on another without having to log on to the second machine.

File naming is a problem area in meta-distributed systems because the naming 'space' of these systems is usually the union of the naming spaces of their component local operating systems. In order to address resources in the total system, one needs to introduce mechanisms to allow the user to address outside the local system on which he may be running.

Ideally, one would like a single name space for the entire system, rather than connected individual name spaces. Why is this not easy in a meta-system?

A.2.1.1 File System Naming

We need first to allow across-machine file access. This is easily achieved by running a server process in the system which accepts requests from jobs running on one node to access files on another node. This may be a central process or a distributed one.

Usually this server is capable of dealing with both local names, which are interpreted in the name space of the current machine or of a local directory, and global names, which are interpreted in a space consisting of all the machines which are currently operating. From the user's point of view, there are also relative and absolute names. For example:

- The unadorned pathname:

  macros

  might be relative to my current directory (set by the 'cd', change directory command), and returns a file called 'macros', if it exists, in the directory to which I am attached.

- The absolute pathname:

  /uc/jon/macros

  returns 'macros' which is on node C, if node C is accessible, irrespective of which machine I am currently working on. The pathname element /uc simply indicates machine C in the network.

- The relative pathname:

  //jon/macros

  is a pathname relative to the current system, and returns the "nearest" file called 'macros'. "Near" is determined by the way in which the logical disks are ordered by the systems administrator. Local disks are always "nearer" than remote disks.

File systems are only special cases of name-to-address mapping mechanisms. At each directory level in a file system, you can tell a directory the **name** (of a file or directory) in its name space, and it will respond with an **address**, leading you to the file, or to a directory whose address space contains the rest of the pathname. So,

/ub/jon/macros

is interpreted (mapped) first by a top-level directory which strips off '/ub' and maps you to a directory 'jon' on a particular node, node B, where a directory strips off 'jon' and maps you to a file 'macros' in its address space. Graphically, this can be represented as a tree, where each path through the tree leads to one leaf:

```
                 root
               /      \
             /          \
           /              \
          ub               uf
        /     \              \
      /         \             \
    fred          jon          jon
    /  \          / \           /  \
   /    \        /    \       /     \
other macros other macros other macros
```

Here, /ub and /uf point to the roots of the file systems on machines B and F.

A relative pathname starts at any given internal node of the tree (determined by the last 'cd', change directory, command) and an absolute pathname starts at 'root'. Since there is only one path from a given internal node to a given leaf, there is no ambiguity, once we know at which internal node to start.

A.2.1.2 Aliasing

In addition, I may also have a *template* or *alias* file which will perform **transliteration** or aliasing of file names allowing one file name to masquerade as another, but which will not perform any interpretation (i.e. mapping). The interpretation is performed after the aliasing.

One can imagine a routine *expand*() which performs filename aliasing before passing the expanded pathname on to an *open*() routine which performs directory searching to find the actual file intended. The alias filename must appear surrounded by '=' signs and if the alias file contained the line:

```
| macros    /ub/jon/progs        |
```

then =macros= would be transliterate into /ub/jon/progs by expand() before interpretation.

I might decide that **=macros=** should translate

```
into any one of:
/uc/jon/progs
//jon/progs
progs
```

and the file I finally get would depend on the interpretation which is performed by open(), **after** *=macros= is transliterated*, by expand(). In other words, it would depend on the current directory if we transliterate into a relative pathname, and not otherwise.

Templates introduce a new mapping which is not a tree and which does not map from the internal nodes of the file system, set by 'cd', but from the alias or template file of **'current user'**, set by **'login'**. A template file is attached to an account not to a directory, and does not change as a user works on a given node of the system. It is set as he logs on.

However, a user may have accounts on more than one machine. Since template files are per-account, and not per-user, they are a local-system concept which has to be extended in some way in extending the system.

Suppose 'jon' has two template files, one on logical disc ub, and one on logical disc uf.

/ub/jon/template:

```
| macros //jon/macros          |
| other /ub/jon/other          |
|                              |
```

/uf/jon/template:

```
                ____________________________
                | macros  //jon/macros     |
                | other   /ub/jon/other    |
                |                          |
```

Then when 'jon' is logged on to machine B, the following mapping will be superimposed onto the file system by templates:

```
                               -  root  -
                              /          \
                            /             \
                          /                \
                        ub                  uf
                     /      \                \
                   /          \               \
                 fred         jon            jon
                 /  \         /  \          /   \
               /     \      /      \      /       \
            other macros other macros other macros
                           |       |
                           |       |     __________
                           |       '------ macros |
                           |             |--------|
                           '-------------- other  |
                                         |________|
```

while if 'jon' is logged onto machine F, the mapping changes to:

```
                               - root -
                              /        \
                             /          \
                           /             \
                          ub              uf
                        /      \           \
                     /          \           \
                 fred         jon           jon
                 /  \          /  \         /   \
               /      \      /      \     /       \
            other macros other macros other macros
                            |                    |
                            |     __________     |
                            |     | macros ------'
                            |     |--------|
                            '------ other  |
                                  |________|
```

The mixing of transliteration (templates) and interpretation (directories) leads to some unexpected 'features'. For example, if the text formatter program 'fmt' which reads text source files and formats them for printing, also allows me to read in a file containing formatter macro definitions by including a line:

```
.so filename
```

in my formatter source file, then I can make up a text formatter source file 'file.fmt' which calls another file 'macros' using the text formatter's '.so' feature:

```
        file.fmt
        ___________________________________
        | .so =macros=                    |
        | rest of file                    |
```

Now if I format the file, using the command line:

```
        fmt file.fmt
```

the formatter will first read file.fmt and on the first line find the reference to file =macros=. In attempting to open =macros= it will first get it transliterated according to my templates file, then read the =macros= file (whichever one is finally opened by expand() followed by open()) and use its contents to format the rest of the document. Supposing that I am on machine B, and that my templates file on that machine contains the line:

```
        macros //jon/macros
```

then it will be transliterated to //jon/macros, interpreted according to the machine I am currently on, and finally read in as file /ub/jon/macros.

If I were logged in on machine F, with an identical templates file, and read in the same formatter source file 'file.fmt', then although expand() performed the same transliteration, the macros file finally read would be /uf/jon/macros! This might or might not be an identical file.

In other words, there are two levels of potential confusion:

1. You can have dissimilar template files in your accounts on different machines, and then the same template may be transliterated differently.
2. Even if the template is transliterated identically on the machines, the fact that you may be in different directories on the different machines may lead to the interpretation of the transliterated template being different.

Clearly, to this, and to many of the subsequent difficulties I shall raise, there are purely administrative solutions.

I can avoid the first difficulty by having identical templates on all machines, and undertake to keep them all identically updated.

I can avoid the second difficulty by making all templates transliterate into absolute pathnames:

```
        macros /ub/jon/macros
```

However, the fact that I have to solve these very obvious problems by user action suggests that there is some underlying problem which is not being solved at the operating system level.

Even if I adopt these administrative solutions, there remains a problem. For example, if I grant READ permission on 'file.fmt' to another user 'fred', when fred reads 'file.fmt', the formatter will try to transliterate =macros= according to fred's template file, which may contain no line for =macros=, or worse, a line:

```
macros //fred/some_other_file
```

in which case fred will format my formatter source file with an unintended macros file which may be grossly inappropriate, leading to a formatter crash; or it may be very subtly incorrect, leading to a successful run of the formatter and the production of an incorrect but plausible document.

Fred ought to have access to my template file when formatting my documents. In fact, Fred ought to be me, or rather, be me in this project. In order for the operation of formatting the document to be carried out correctly, interpretation and transliteration ought to be carried out per-project rather than per-user.

The main problem is the mixing of two operations, transliteration and interpretation, which look similar but are quite different. Transliteration is always carried out in the context of the account (not the user, since he may have accounts on several machines), while interpretation is carried out in the context of the file system subtree identified by the pathname.

In summary, if you tie together several existing file systems by connecting their roots, then a single user's files will no longer be in one file system subtree, but in several, one on each machine. This leads to inconsistent file pathname mapping.

Operations are performed per-account (such as template handling) which ought rather to be performed per-user.

If this is resolved by administrative measures, such as the use of absolute pathnames, the user will be forced to be aware of his physical location in the system, rather than of his logical location.

A.2.2 File Storage Structures

Given that file system structures should reflect some logical relationship between the files that they contain, some other questions arise.

There does not seem any good reason why a user should not be able to generate and use a file system subdivision (not a subtree) which crosses disc boundaries, and even overlaps with other subdivisions. Directories should be able to contain entries pointing to directories or files on other disks.

A.2.3 Distribution

Now we can take a look at the consequences for a distributed system. One finds that in a distributed system the concepts of transliteration and interpretation change in subtle ways. Even in a centralized system, we have template mappings which change according to login account, as well as file system mapping which do not change if absolute pathnames are used, or which do change, if they are relative pathnames. The only reason that relative pathnames change their meaning is that 'login' implies 'cd'.

In a distributed system the problem is complicated because jobs can run on one machine, with the file system mappings appropriate to that machine, but with the template mappings which have been imported from the machine on which the user is logged in. In effect, we have a 'login' which does not imply a 'cd' (unless we *want* to do a remote 'login', in which case it is not clear what happens).

One thing that a distributed system should try to do is to allow a user on, say, B, to run commands on remote machines, using a syntax such as:

```
fmt@F file.fmt
```

In order to achieve this, the command line is sent to the command interpreter on machine F. Along with the command line is sent the 'current context' (including the template file) of the user. In fact, what is sent is the current context of the account of that user on B, and is potentially quite different to his context on machine F. (He might, for example, have no account on F or might not be logged on there at present).

Now we have the potential for a job run on F from B to produce different effects from that same job run on *either* B or F directly.

Suppose we have the same source file 'file.fmt' on /ub.

```
file.fmt
 _________________________________
| .so =macros=                    |
| rest of file                    |
```

This file will *not* be the file accessed by the command line:

```
fmt@F file.fmt
```

instead, the command line would have to be changed to:

```
fmt@F /ub/jon/file.fmt
```

Now we can send the command line to F, along with the current context, and see what happens. The remote job executes on F *as if* executing on behalf of /ub/jon, as far as transliteration is concerned, since we sent the template file of /ub/jon along with it. When the formatter on F reads /ub/jon/file.fmt he sees:

```
file.fmt
 ___________________________
| .so =macros=              |
| rest of file              |
```

transliterates =macros= into //jon/macros, for example, and now interprets //jon/macros in the context of the machine he is running on, attempting (or succeeding) to read a file /uf/jon/macros. In effect the job has run partly (transliteration) as if belonging to /ub/jon, and partly as if belonging to /uf/jon. This raises some interesting questions about location information, and how much of it the user has to be aware of.

And it is not always an answer to suggest that interpretation should also be carried out in the account context sent with the command line to the remote machine, since some users will want their remote command lines to access within the context of the machine that they are on. In particular, someone who wants to use the remote command line execution feature as 'remote login' wants precisely that, to transliterate according to the machine he is on, and access files according to the remote machine he is remotely logged into. Or does he? Some of these problems can be avoided by simply using absolute pathnames at all times, but then you solve the immediate problem at the cost of giving up the entire template and file pathname interpretation system.

As usual, you can kludge these problems away by totally rigidifying the system. For example, adopting the rules

- all templates map into absolute pathnames.
- all remote command lines contain absolute pathnames.
- all remotely executed source files call absolute pathnames.

But maybe this is not what is wanted in a distributed system. Maybe I want to get "The nearest file called 'macros', or a further away one if the nearest node is down". This would amount to allowing the transliteration of a given template to vary from time to time during the execution of a job. This would require the file system to behave much more like a database, recording, for example, if two files called 'macros' are identical copies, or separate objects. And I would like to have at least some facilities equivalent to relative pathnames and templates.

A.3 SOLUTIONS

In this section we attempt to present solutions to some of these problems in a fairly general way. In some cases these solutions call for simple modifications to the local operating system file system.

A.3.1 A Domain Structured File System

In order to avoid some of the difficulties listed in file system addressing I am suggesting that we use a file system which is basically domain (or capability) based.

The underlying structure of the file system will be unchanged, and will be tree-structured, but users will be able to set up '**domains**' at any point in the file system to which they have access, which will allow them to address any files they want in a '**one step**' fashion. The files to which a user has access will be determined by owned capabilities rather than by access list. We will change the routines expand() and open() to implement the new file system structure on top of the old.

A.3.2 Description and Implementation

It seems that all the tools that you need in order to construct a domain-based file system already exist in most local operating systems, since my proposal basically uses only an extension of the '**template**' mechanism.

A domain structured file system allows a user to set up a context for himself in the file system at any point. In order to do this he uses local template files. At present, template files are per user (actually per account, since there is an individual template file on each machine for each account) and the merit of templates is that they allow very simple '**one-step**' file name transliteration.

Typically, one now keeps all the files for a particular program in a single directory or subtree, except for central files used by several projects, and once you have done a 'cd' to that project directory, you can then refer to those files using very short relative pathnames, typically only one element long; the name of the file. This only works for files in that directory; hence templates, which allow one-step file naming for files in other directories.

One can achieve the same effect in a more consistent way by abandoning direct use of pathnames altogether, and relying totally on indirect use of pathnames through templates. In effect we redefine a directory to _be_ the template file at some position in the file system.

Consider a user 'jon' who has several projects under development concurrently. Each project has a program source file; prog1.r, prog2.r, etc, and a documentation file doc1.fmt, doc2.fmt, etc. In addition each program calls an 'include' file; def.i, which contains some common program definitions used by both programs. When run, each program will read data files data1, data2, etc.

```
                                  ---  root  ---
                               /               |
                             /                 |
                           /                   |
                          ub                   uf
                          |                    |
                         jon                   |
                  __________________           |
                 /     |      |    \           |
               /       |      |      \         |
            docs     proj1   include proj2     |
            |         |        |      |        |
            |doc1.fmt |prog1.r |def.i |prog2.r |
            |doc2.fmt |        |      |prog2.b |
            |doc3.fmt |        |      |        |
            |doc4.fmt |        |      |        |
                                               |
                                               |
                                               |
                              _________________|
                              |
                             jon
                            /   \
                          /       \
                        proj3    data
                         |         |
                         |prog3.r  |data1
                         |         |data2
                         |         |data3
                         |         |test1
                                   |test2
                                   |test3
```

Now we can build a domain (local template file) for project1 which is suitable for program development and testing.

```
testing1:

prog.r   -  /ub/jon/proj1/prog1.r
data     -  /uf/jon/data/test1
obj      -  /ub/jon/proj1/prog1.r
prog     -  /ub/jon/proj1/prog1
def.i    -  /ub/jon/include/def.i
```

We can build a similar domain for project 3:

```
testing3:

prog.r   -  /uf/jon/proj3/prog3.r
data     -  /uf/jon/data/test3
obj      -  /uf/jon/proj3/prog3.r
prog     -  /uf/jon/proj3/prog3
def.i    -  /uf/jon/include/def.i
```

It may be worth noting a few points here:

- All pathnames in the domain file are absolute; they all have exactly the same effect no matter where on the network they are evaluated.

- In use, the domain files allow you to address each file 'in scope' by a short, one element name. For example, when project1 is in effect (for SWT, when **testing1** is the current template file) editing '**prog.r**' edits the file /ub/jon/proj1/prog1.r. Naturally, you can choose any names for files that suit you. A one-step operation of changing the domain in effect to **testing3** changes '**prog.r**' to /uf/jon/proj3/prog3.r.

- Some files appear in more than one domain file. These are shared files; for example '**def.i**'.

- The domains are similar for similar projects. This suggests that standard domains might be parameterised.

- We can produce other domains which use the same files in different ways, for example, a domain suitable for documenting project1:

  ```
  document1:

  prog     -  /ub/jon/proj1/prog1.r
  doc      -  /ub/jon/doc/doc1.fmt
  defs     -  /ub/jon/include/def.i
  defdoc   -  /ub/jon/doc/def.fmt
  ```

- A file can have different names in different domain files. When documenting, we are no longer concerned with binary and object files, so there is no longer a need to make them visible in the domain. We could have another domain for testing:

  ```
  run1:

  prog     -  /ub/jon/proj1/prog1
  data     -  /uf/jon/data/data1
  ```

- More than one file can appear with the same name in different domain files; '**data**' was a test file in testing1, but a regular data file in run1.

- Once again, at run time we are not concerned with file prog.r. Once a program has been compiled we no longer care about its source and object files.

A.3.3 Using Domains

Templates are taken from whichever domain file is in effect at a given moment, so we need a command which will transfer us from the current domain file to a new one. If we are currently testing project1, and we want to switch to documenting project3 we might type

```
cd document3
```

The effect of the cd command is to make document3 the current domain file, to make all the files listed in testing1 inaccessible, and make the files listed in document3 accessible.

We might choose to have testing1 disappear, or to have it pushed onto a domain stack, from which it can be recovered by a '**pod**' (pop domain) command. Maybe we should also have an explicit '**pud**' (push domain) command)

Of course, the new domain file used by '**cd**' must appear in the current domain file, so we shall have to modify testing1 and document3 so that we can execute the 'cd' command:

```
testing1:

prog.r    -  /ub/jon/proj1/prog1.r
data      -  /uf/jon/data/test1
obj       -  /ub/jon/proj1/prog1.r
prog      -  /ub/jon/proj1/prog1
def.i     -  /ub/jon/include/def.i
next      -  /ub/jon/domains/document3

document3:

prog      -  /ub/jon/proj3/prog3.r
doc       -  /ub/jon/doc/doc3.fmt
defs      -  /ub/jon/include/def.i
defdoc    -  /ub/jon/doc/def.fmt
next      -  /ub/jon/domains/testing1
```

Now, when testing1 is in effect:

```
cd next
```

switches us to document3, at which point the effect of:

```
cd next
```

is to take us back to testing1 If we want to do something more than switch back and forth between these two domains, one of them will have to contain the name of some other domain file:

```
testing1:

prog.r    -  /ub/jon/proj1/prog1.r
data      -  /uf/jon/data/test1
obj       -  /ub/jon/proj1/prog1.r
prog      -  /ub/jon/proj1/prog1
def.i     -  /ub/jon/include/def.i
next      -  /ub/jon/domains/document3
other     -  /ub/jon/domains/testing2
```

'**Other**' is the name of another domain file we might wish to use.

```
testing1:

prog.r    -  /ub/jon/proj1/prog1.r
data      -  /uf/jon/data/test1
obj       -  /ub/jon/proj1/prog1.r
prog      -  /ub/jon/proj1/prog1
def.i     -  /ub/jon/include/def.i
next      -  /ub/jon/domains/document3
other     -  /ub/jon/domains/testing2
out       _  /ub/jon/domains/master_domain
```

'Out' is the name of some top-level domain file containing (perhaps) the names of all the domain files we would consider there:

```
master_domain:

testing1     -  /ub/jon/domains/testing1
testing2     -  /ub/jon/domains/testing2
testing3     -  /ub/jon/domains/testing3
document1    -  /ub/jon/domains/document1
document2    -  /ub/jon/domains/document2
document3    -  /ub/jon/domains/document3
run1         -  /ub/jon/domains/run1
run2         -  /ub/jon/domains/run2
run3         -  /ub/jon/domains/run3
```

We might say that 'master_domain' is the 'root' of the domains.

We can structure our work as we structure our file system, by restricting the domains we can reach from the current domain. It may be highly appropriate to have document1 as the only domain you can get to on exit from testing1. This would enforce good habits of work.

A.3.4 Rules

File system domains should have the same rules as capability systems[Cook 79].

Every file listed in the current domain is accessible.

No file not listed, including other domain files, is accessible.

The domain file, not the directory, gives the access rights to a file. Access rights should be listed in the domain file. For example:

```
testing1:

prog.r -  /ub/jon/proj1/prog1.r       RW
data   -  /uf/jon/data/test1          R
obj    -  /ub/jon/proj1/prog1.        RW
prog   -  /ub/jon/proj1/prog1         EX
def.i  -  /ub/jon/include/def.i       R
next   -  /ub/jon/domains/document3   ED
other  -  /ub/jon/domains/testing2    ED
out    _  /ub/jon/domains/mstr_domain ED
```

gives read/write access to 'prog1,r', whereas:

```
document3:

prog   -  /ub/jon/proj3/prog3.r      R
doc    -  /ub/jon/doc/doc3.fmt       RW
defs   -  /ub/jon/include/def.i      R
defdoc -  /ub/jon/doc/def.fmt        R
next   -  /ub/jon/domains/testing1   ED
```

only gives read access to 'prog3', (You ought not to be able to modify a program while you are documenting it.)

The domain rights are:

RW	-	read/write data
R	-	read data
EX	-	execute data
ED	-	enter domain
RD	-	read domain
WD	-	write domain

Rights on domain normally only allow you to enter it (ED). You are not allowed to read or edit the current or any domain file unless you have RD or WD access to it, and this must be specified in the current domain file. For instance:

```
master_domain:

testing1   -  /ub/jon/domains/test1 ED
testing2   -  /ub/jon/domains/test2 ED
testing3   -  /ub/jon/domains/test3 ED
document1  -  /ub/jon/domains/doc1  ED
document2  -  /ub/jon/domains/doc2  ED
document3  -  /ub/jon/domains/doc3  ED
run1       -  /ub/jon/domains/run1  ED
run2       -  /ub/jon/domains/run2  ED
run3       -  /ub/jon/domains/run3  ED
testing1   -  /ub/jon/domains/test1 WD
testing2   -  /ub/jon/domains/test2 WD
testing3   -  /ub/jon/domains/test3 WD
```

allows you to enter several domains, and also to modify testing1, 2, and 3.

Otherwise, the domain files should be unwriteable to the user. He should not be able to change his current domain setup informally.

However, once you have modify rights on a domain file, you can change it with the editor.

Domain enforcement is performed by the file system 'open' routine. This routine is modified so as always to look in the 'current' domain file and translate any filename string given it according to that domain file. If the string supplied to 'open' appears in the domain file, it is located on the network file system and opened, but only with the access permissions allowed by the current domain. No file not listed in the current domain is opened, and an error is returned to the user on failure.

This is a more extensive change than simply to use templates and absolute pathnames, which can be done with the present system.

The 'current' domain is that in /ub/jon/domains/current.

The effect of the:

```
        cd new_domain
```

command is to perform a:

```
        cp /ub/jon/domains/new_domain
                  /ub/jon/domains/current
```

The effect of 'push':

```
        push_domain new_domain
```

is first to stack /ub/jon/domains/current, and then copy /ub/jon/domains/new_domain to /ub/jon/domains/current.

The effect of 'pop' is to pop the domain stack into /ub/jon/domains/current, losing the current contents of /ub/jon/domains/current.

Domains can be created when the current domain has CD (create domain) permissions. A domain file entry (capability) can be created for any object the user owns.

A user can share objects, including data files and domain files, with other users.

A user's domain file can be sent to another user, with or without changes (restrictions) on the permissions on its lines.

When new objects are created, a side-effect is the appearance of a new line for the new object in the current domain file.

This new line can be moved to another domain file if that file is accessible from the current domain (directly or indirectly).

Each domain file for a user appears in a given directory, say '**domains**'.

Then the effect of a change domain command is to copy the named domain file into /swt/vars/user/.templates

The routine '**expand**' already looks at /swt/vars/user/.template when it encounters a pathname containing =string=. Expand then has to be modified to assume that all pathnames which are given to it need to be expanded. Expand will look in the current templates file (the current domain) and perform transliterations according to the contents of the file. Strings which do not appear in the current domain will not be transliterated, and error will be returned.

A.3.5 Command files

So far we have only mentioned data files, while in fact we could also deal with commands in the same way.

In most systems, each user has a (static) '**search_rule**' which records the order in which the command interpreter should search certain directories for the commands that he runs. When the user invokes '**cmd**', the command interpreter will run the command of that name which is found earliest while searching the directories in '**search_rule**'.

In a domain-based file system, each domain file can clearly contain a new '**search_rule**' and that means in turn that while using another user's domain file, you also inherit his '**search_rule**'. To a very large extent, this means that you **become** that user, since you not only process his files, but you are constrained to process them in the same way as he, and using the same command libraries.

A.4 UNSOLVED PROBLEMS

We now consider some problems which cannot be solved without some changes to the underlying system.

A.4.1 More Binding

We mentioned above that some context problems can be avoided by using absolute pathnames at all times, but this depends on the absolute pathname feature being available. Files are somewhat unusual objects in having pathnames. Most local operating systems do not generalize naming concepts to all objects. There is no multi-level pathnames translation scheme such as file pathnames, for library programs or physical devices (although these facilities do exist in some local operating systems, such as Multics).

A.4.2 Variables

Some command languages have both local and global command variables, which raises an interesting question in a distributed environment; where should they be evaluated? Local variables are defined only in the activation of the shell program in which they are set, while global variables are associated with a user rather than a shell program (and they are stored in a variables file when he logs off). He can also save them at any time using a save command language command.

This raises the question of the 'scope' of variables. Supposing that a process on A spawns a remote process to run on B, presumably the variables file of the user is exported to the remote process. What happens if a variable is changed?

```
echo [vara]
set@B vara = newvalue
echo@A [vara]
```

What happens if a variable is changed, and the remote command line invokes a 'save' function on the variables. Does this cause the new variables to be saved? And then can the original job on A also do a 'save'?

A.4.3 Command Language Functions

Command language functions can return a single success code directly in such a way that it can be evaluated in a command language 'if' statement. This means that the command line:

```
if [function] then S1 else S2;
```

executes command language statement S1 if [function] returns TRUE and S2 otherwise.

This is going to have odd results unless [function] is executed on the same machine as the 'if'. If the 'if' assumes different context information to that assumed by the function (for example, a different interpretation of relative pathnames) then there will be an inconsistency.

A.4.4 On-Condition

Some languages allow events (conditions) external to a program to be signalled to it in the form of a software-implemented interrupt. It is interesting to speculate what would happen if we declared an on-conditon on one machine and raised the condition on another.

A.4.5 Conclusion

Other problems in extending single-machine systems to run on multiple machines are easily solved by adding extra software over the original operating system.

Some problems are not so easily solved, either because of addressing deficiencies, or because results have to be returned to an indeterminate location.

REFERENCES

[DSWT1] Myers, et al. Initial Experiences with a Distributed System. Unpublished draft.

[Andl79] Andler, S. Synchronization Primitives and the Verification of Concurrent Programs. Proc. 2nd Int. Symposium on Operating System Theory and Practice. North-Holland, 1979.

[Kier81] Kierbutz, R. B. A Distributed Operating System for the Stony Brook Multicomputer. Proc. 2nd International Conf. on Distributed Computing Systems. April 1981.

[Lind81] Lindsay, B. Object Naming and Catalog Management for A Distributed Database Manager. Proc. 2nd International Conf. on Distributed Computing Systems. April 1981.

[Need79] Needham, R. M. and Lauer, H. C. On the Duality of Operating System Structures. Proc. 2nd Int. Symposium on Operating System Theory and Practice. North-Holland, 1979.

[Oppe81] Oppen, D. C. and Dalal, Y. K. The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. Xerox Office Products. October 1981.

[Thom78] Thompson, K. Unix Implementation. Bell System Tech. Journal. Vol 57, no 6. July 1978.

APPENDIX B

INTERPRETING AND COMPILING COMMAND INTERPRETERS

N. J. Livesey

B.1 COMMAND INTERPRETERS

A command interpreter is a program which typically establishes the interface between the user and the operating system performing a translation from the input commands into the form which the operating system understands. We contend that there are, in fact, two rather different forms of command interpreters, those which execute commands directly (which we call "compiling command interpreters,") and those which, because of a gross mismatch between the world as the user sees it, and the world as understood by the operating system, have to intervene at almost every stage of the execution of the user's commands (which we call "interpreting command interpreters").

This note is a proposal for a compiling command interpreters for use on an FDPS (Fully Distributed Processing System).

It first examines the differences between interpreting and compiling command interpreter's, especially with respect to their 'computational power' and the run-time overheads that may be expected from each.

It then argues that it is possible to produce a command interpreter of the compiling rather than the interpreting kind for an FDPS and supplies a theoretical framework for such an interface.

It finally suggests a method of implementation for a fully distributed compiling command interpreter.

B.1.1 Compiling Versus Interpreting

In this section we define the two kinds of command interpreters. An interpreting command interpreter is a command interpreter which is invoked to carry out each individual command on a command line. A compiling command interpreter is a command interpreter which is invoked to scan a command line, compile it into some intermediate representation, and issue the intermediate representation to the operating system in one step.

B.1.2 Command Line Execution

In this section, we explain how the two kinds of command interpreter's would handle a simple command line:

```
fmt file.fmt | os | sp
```

This command line is a 'pipeline' (a sequence of operations in which the result of one operation is immediately "piped" to the next operation) which calls three commands in series. 'Fmt' is a text formatter, 'os' is an overstrike and underscore processor, and 'sp' is a line-printer spooler. The file 'file.fmt' is input to the formatter. The command line is a pipeline in the sense that:

1. The commands 'fmt', 'os', and 'sp' are filters, programs which will take a continuous stream of input, perform some transformation on it, and produce a stream of output which can be 'piped' to another filter.
2. The output from one command can be redirected to be the input of another filter, or to some file or device. In this example the output from 'fmt' is piped to the input of 'os', and the final output of 'sp' goes to a line-printer.

A interpreting command interpreter would handle this command line by first parsing it to find its three pipeline nodes (fmt file.fmt), (os), and (sp), and would then run the first, the formatter, directing its output to an intermediate file 'temp1', then, when the formatter terminated, run 'os' with 'temp1' as input and 'temp2' as output, and finally run 'sp' with 'temp2' as input, and the line-printer as output.

The compiling command interpreter would parse the command line in the same way, but would then cause the operating system to set up a separate process for each command line node, with inter-process communication 'pipes' between successive nodes, and allow the node processes to run concurrently, communicating through the inter-process communication 'pipes'.

B.1.3 An Aside - Processes and Processes

One reason for the difference in these two modes of operation lies in operating system structure, in particular the treatment of processes. The process primitive in a system can be identified with (at least) the user or with a task. For example, in Unix, a process is identified with a task. It consists of an identifier, an address space, input/output ports, and a process state, so that it can be separately scheduled. Unix processes can create other processes, or can arrange to run a new program in the address space of the current process, destroying the program presently running there. They can also communicate with any other process whose identifier they know. Interprocess communication can be through memory buffers. A user can have several inter-communicating concurrent processes running on his behalf. In particular, his command interpreter can create several processes from one command line to run the elements of a pipline concurrently.

In a system in which 'process' is identified with the user, a process represents the entire state of an on-line user, and might, for example, be tied to his terminal, and contain his command interpreter running in shared system space. If the user runs a new program, it is loaded and run in the address space of the 'user' process, replacing (and destroying) the previous program run there rather than being run in the address space of a new concurrent process, and there is little idea of inter-process communication, except through the file system. Successive programs running in the user process communicate by writing intermediate files, and one program has to run to completion, and close the intermediate file which contains its results, and exit, before the next program can open that intermediate file and read it as input.

B.1.4 A Stack Command Interpreter

There is an intermediate form of command interpreter, found in the SOLO operating System.

In the SOLO operating system the single 'user' process maintains a 'program stack' so that one program running in the process can invoke a second program, at which point the first program will be suspended, rather than destroyed, and its state pushed on the program stack. The second program in turn can get itself en-stacked by invoking a third program. And so on.

At some point, a program terminates, and then the top program state is popped off the stack, and that program is resumed. At any time, the stack represents the states of several suspended programs.

B.1.5 Computational Power

One can show the differing power of the kinds of shell, by considering them as computational machines. Intuitively it seems that:

1. A single process command interpreter is equivalent to a Turing machine with a one-way read-write head.
2. A UNIX (multi-process) command interpreter (called a "shell") is equivalent to a Turing machine with a two-way read-write head.

3. The SOLO command interpreter is equivalent to a Turing machine with a single pushdown.

Cases 2. and 3. are the same power, but case 1. has lesser power. In particular, case 1. does not allow pipeline elements to be connected by pipes in two directions. All single command lines have to be capable of being serialised. (Although, of course, one is allowed to execute a single command line several times by 'looping'. This involves re-executing the individual commands of the command line pipeline). From now on we forget case 3.

B.1.6 Elasped Time

There are some obvious differences in elapsed time to execute the same command line.

1. Does not allow the exploitation of concurrency (parallellism) in the pipeline.
2. Where concurrency exists, it can be exploited. In a single-processor system pipeline elements which do not depend on one another can run in any order, but concurrency need not lead to any elapsed time saving. In a multiprocessor system one can get true physical concurrency.

B.1.7 Overhead

We can try to evaluate the overhead incurred by the two forms of command interpreter's. The total overhead is made up command execution time (which should be the same in any system), shell overhead, and inter-process communication overhead.

```
total_overhead = command_execution + shell_overhead +
                 inter-process_communication_overhead
```

B.1.7.1 Command Execution

Total command execution time should not differ, whether commands are run serially or concurrently. Call this time 'C'. This might be affected by swapping and paging.

B.1.7.2 Command Interpreter Overhead

An interpretive command interpreter will be reinvoked as each element of the command pipeline terminates (or at least some component of the command execution code will be reinvoked).

A compiling command interpreter is invoked only when the command line is parsed, and when the entire line terminates. The overhead is then

1. overhead = O(n)
2. overhead = O(2)

where n is the number of command line elements.

B.1.7.3 Inter-process Communication Overhead

We can try to evaluate the inter-process communication overhead incurred by the two forms of command interpreter's. In either case, interprocess communication involves transferring each block of intermediate results from one program to the next, or from one process to another. However, in the case of successive programs running in the same process, the intermediate results must be written to a disc file, which must then be closed before the next program can open and read it.

In the case of concurrently running processes the intermediate results can be buffered, block by block, through central memory.

From previous results obtained on the MININET project, writing to the file system costs around 1000 ms per block, while buffering through central memory costs around 1 ms. per block. Then we have:

1. overhead = O(1000*m)
2. overhead = O(m)

where m is the number of blocks of intermediate results transferred.

B.1.7.4 Total Overhead

Combining the results of the previous two sections given the total overhead incurred by the two forms of command interpreter's.

1. total = C + O(n) + O(1000*m)
2. total = C + O(2) + O(m)

B.1.8 Summary

The difference in elapsed and total time is clear, even for a single processor system. For a multi-processor system we conclude that a compiling command interpreter, capable of taking advantage of the parallelism in a command line, is necessary.

B.1.9 A Distributed Command Interpreter

For purposes of comparison, here is an outline of an interpreting command interpreter for an FDPS. The basic ideas are:

1. A command file can be parsed once and for all to produce a graph whose nodes are individual processes (We take the view that processes are tasks, not users), and whose directed edges represent inter-process communication data flow.

2. The graph can contain choice (i.e. decision and/or selection) nodes and iteration. (The command file lines are not necessarily serialisable.) The graph can contain choice (i.e. decision and/or selection) nodes and iteration.

3. In order to exploit the parallelism inherent in the process graph and also present in the particular hardware configuration on which we are currently running, we must be prepared to distribute a representation of parts of the process graph to the processors on which particular processes are to be run.

4. In order to distribute the information contained in the graph we treat each node as a single unit and attach to each node (representing one process) exactly the information needed to connect that node to its neighbors in the original graph and to handle any synchronization which may arise. Its neighbors in the original process graph are those process nodes to which it was connected by inter-process communication flow edges.

5. Then we call resource allocation and work distribution in order to find out where each process is in fact to run.

6. We then send the connectivity information for each process to the processor on which it is to run.

7. Finally, distributed control will use the distributed connectivity information to set up Inter-process communication between concurrent processes in a given processor and inter-process communication between concurrent processes running in separate processors.

The rest of this document suggests ways of parsing a command file into a process graph, and ways to distribute the connectivity information so that Distributed Control can use it on physically separated processors.

The information which is distributed so that local elements of Distributed Control can run the complete command file consists of IPC tokens. Two tokens, a send and a receive token, are sent out for each edge in the original Process Graph.

We have not specified exactly what process graph edges represent, apart from representing IPC. A single edge might represent a write-read pair, a communication line, a message buffer being sent, a synchronizing signal, or the action of one process creating another. All that we require is that an edge has two end-points which are processes (We do not even require the two processes to exist at first. Perhaps one of the processes is being created, or dying).

For an edge, we distribute the send token to the process which is initiating the IPC transfer represented by the edge, and the receive token to the process which is the object of the IPC transfer.

Distributed Control uses the tokens at run-time to implement distributed system control.

B.2 PROCESS GRAPH COMPILATION

B.2.1 Process Graph Language

A command file can be represented as a graph, with processes as nodes and inter-process communication represented as directed edges. This graph can be specified in a specialised programming language, the Process Graph Language.

The translation of a process graph proceeds in six stages:

1. Specification of the process graph in process graph description language. This language is described in the next section.
2. Parsing of the process graph description language program. Detection of syntax errors in the process graph description language program.
3. Construction of the precedence graph from the process graph program. The precedence graph is a directed graph whose nodes represent IPC, and whose edges express the precedence relations between IPC. This will be explained further below.
4. Checking of the precedence graph. Detection of semantic errors in the process graph description language program
5. Translation of the precedence graph into IPC tokens. Tokens are used at run-time to enable the operating system to enforce the precedence relations between IPC which were laid down in the process graph.
6. Distribution of the processes of the process graph, along with their IPC tokens, according to instructions issued by Resource allocation and work distribution.
7. Run-time enforcement, by Distributed control, using the IPC token lists.

B.2.2 Process Graph Language Grammar

Here is a simplified grammar of the PGL. This is not intended to be a comprehensive definition of the grammar; merely a summary around which to build a description of the process graph description language compiler. For the full grammar see [Livesey 6]. The semantic actions to be taken upon the satisfaction of each production are given in the right-hand column below each production as a section number, referring to the rest of this section.

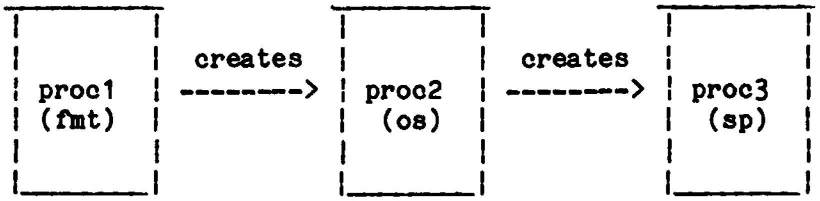
```
Production                                                   Section Number

<process graph> ::= 'begin' <pg head> <pg body> 'end'         B.2.2.1

<pg head> ::= <prog def>* <process def>* <type asg>*          B.2.2.1
              <edge def>*

<pg body> ::= 'begin' <stmt list> 'end'                       B.2.2.1

<prog def> ::= 'prog:' <prog ident> '==' <string>             B.2.2.1

<process def> ::= 'process:' <process ident> '=='             B.2.2.1
                  <prog ident>

<type asg> ::= 'trans:' <ident list> |                        B.2.2.2
               'root:'  <ident list> |
               'perm:'  <ident list> |
               'graph:' <ident list> |
               'edge:'  <ident list> |

<edge def> ::= <edge ident> ':='                              B.2.2.3
               <process ident> '-->' <process ident> |
               <process ident> 'creates' <process ident> |
               <process ident> 'dies' --> <process ident> |
               <process ident> 'signals' <process ident> |

<prog ident> ::= 's1' |..........| 'sn'                       B.2.2.3

<process ident> ::= 't1' |..........| 'tn'                    B.2.2.3

<edge ident> ::= 'e1' |..........| 'en'                       B.2.2.3

<string> ::= '"'.*'"'                                         B.2.2.3

<stmt list> ::= <stmt> | <stmt list> ';' <stmt>               B.2.2.4

<stmt>  ::= <concatenation>  |                                B.2.2.5
            < binding >  |
            <concurrent>  |
            <procedure call> |
            <choice>  |
            <iteration>  |
            <edge ident>

<concatenation> ::= <stmt> '<' <stmt>                         B.2.2.6

<concurrent> ::= <stmt> '^' <stmt>                            B.2.2.7

<procedure call> ::= <proc name> '(' <stmt> ')'               B.2.2.8

<choice> ::= 'if' <expression> 'then' <stmt> |                B.2.2.9
             'if' <expression> 'then' <stmt>
             'else' <stmt>

<iteration> ::= 'while' <expression> 'do' <stmt>              B.2.2.10
```

We begin by building a very simple process graph from a command file definition, and showing how it might be handled across several machines:

```
begin
  prog; prog1 == "=bin=/fmt",
        prog2 == "=bin=/os",
        prog3 == "=bin=/sp";
  proc: proc1 == prog1,
        proc2 == prog2,
        proc3 == prog3;
  root: proc1;
  trans:   proc2,   proc3;
  edge: edge1,   edge2;
  edge1 := proc1 creates proc2;
  edge2 := proc2 creates proc3;
  edge1 < edge2;
end;
```

In this example, we have described the creation of three processes in series. Process 'proc1', which runs the program "=bin=/fmt", creates process 'proc2', running program "=bin=/os", which in turn creates process 'proc3', running program "=bin=/sp". The '<' symbol expresses the order of creation.

The exact form of the 'process graph' does not matter, but it might look something like:



Enthusiasts of cryptic systems will protest that all this could be more succinctly expressed as:

```
fmt | os | sp
```

My only answer is that my lengthy syntax expresses what is actually going on, and that in any case, it can be collapsed to a cryptic form as needed.

Note that as yet we have only expressed process creation, and nothing about the inter-process communication between them. That will come later.

Finally, we assume that the three processes run concurrently on one machine, or on several.

In the next sections, we explain the grammar step by step, and then expand the example to be more realistic. First, we explain the actions to be taken upon parsing. The explanation is top-down, descending to greater detail.

B.2.2.1 Process Graph

In parsing we build up a symbol table, containing the definitions of the program files, processes, and IPC edges, and a *precedence graph*, which defines the precedence relations between the edges of the process graph.

Upon recognizing:

```
<process graph> ::= 'begin' <pg head> <pg body> 'end'
```

the parsing of the process graph description language program is terminated, and the precedence graph constructed (see below), is processed to produce IPC token lists.

Upon recognizing:

```
<pg head> ::= <prog def>* <process def>* <type asg>*
       <edge def>*
```

the building of the symbol table is terminated, and the parsing of the process graph body is begun. If a symbol is encountered at this stage which is not defined in the symbol table, it is treated as undefined.

In our example, the <pg head> is:

```
begin
  prog; prog1 == "=bin=/fmt",
        prog2 == "=bin=/os",
        prog3 == "=bin=/sp";
  proc: proc1 == prog1,
        proc2 == prog2,
        proc3 == prog3;
  root: proc1;
  trans:   proc2,   proc3;
  edge: edge1,   edge2;
  edge1 := proc1 creates proc2;
  edge2 := proc2 creates proc3;
end;
```

Upon recognizing:

```
<pg body> ::= 'begin' <stmt list> 'end'
```

the parsing of the process graph description language program has terminated, and the precedence graph has been built.

For the example above, the <pg body> is the single expression:

```
edge1 < edge2;
```

The symbol table contains an entry for each program file (code segment), process and edge in the process graph description language program. In this table, file system pathname strings are treated as predefined, program files are identified by their file system pathname, processes are defined in terms of their constituent program file, and edges are defined in terms of their sending and receiving processes.

For the process graph description language construct

```
        prog: prog1 = "prog_name";
```

which satisfies the grammar rule

```
        <prog def> ::= 'prog:' <prog ident> '==' <string>
```

we add a node to a linked list of program file definition nodes.

```
 ____|____
|    v    |
|  prog   |
|---------|
|  &prog  |------> "prog1"
|---------|
|  &name- |-------> "=bin=/fmt"
|---------|
|  fwd    |
'----|----'
     v
```

The node contains both a pointer to the program file identifier, and to the string name of the file in which the program file is contained.

A process definition node is added to a linked list of process definitions when the rule:

```
        <process def> ::= 'process:' <process ident> '==' <prog ident>
```

is satisfied. Every process is defined in terms of its code part; its program file:

```
 ____|____
|    v    |     (process)
|---------|
| process |
|---------|
|  &proc  |----> "proc1"
|---------|                 __________
|  &prog  |-------------> |  prog    |     (program file)
|---------|               |----------|
|         |               |  &prog   |----> "prog2"
'----|----'               |----------|
     v                    |  &name-  |-------> "=bin=/fmt"
                          |----------|
                          |          |
                          '----|-----'
                               v
```

B.2.2.2 Type Assignment

When the grammar rule:

```
<type asg> := 'trans:' <ident list> |
              'root:'  <ident list> |
              'perm:'  <ident list> |
              'graph:' <ident list> |
              'edge:'  <ident list> |
```

is recognized, process type is assigned. There are two cases; processes which already existed when the process graph was created, and processes which are created during the processing of the process graph. Pre-existing processes are of two kinds; permanent processes, which exist outside the current process graph, and the root process of the process graph, which is created by the process graph supervisor (a component of Distributed Control) in the course of process graph execution. These are declared:

```
perm: process1, process2;
root: process3;
```

Typical examples of pre-existing processes are resource processes, which are permanently in existence, and can be included in any process graph authorized to access them. Pre-existing processes do not need to appear as the receiver of a create edge. All transient processes appear as the receiver of some create edge.

Created (or 'transient') processes come into existence during the execution of the current process graph. They are declared as:

```
trans: process4;
```

Every created process must appear as the receiver of some create edge in the current process graph before it can be used as the sender or receiver process in a non-create edge. The program file pointed to by the 'prog' field of the definition node of a created process is used at run-time to create that process:

```
 ____|____
|    v    |
|---------|
|process  |
|---------|
|  &t --  |----> "proc2"
|---------|                  _________
|  &prog  |--------------> |   prog  |
|---------|                 |---------|
|         |                 |  &prog  |----> "prog2"
'----|----'                 |---------|
     v                      |  &name- |--------> "=bin=/os"
                            |---------|
                            |         |
                            '----|----'
                                 v
```

Until the edge which has the creation of this process as side-effect appears in a precedence expression, the process is marked as 'not-yet-created' in the symbol table. If it is used in another edge before it is created, a compile-time diagnostic is issued. After the creation of process 'proc2' from program file prog2, 'proc2' can be referred to in process graph description language statements as any other process.

A node in a process graph may be another graph, rather than a process, and a mesage or signal may be sent to a graph in the same way as to a process.

B.2.2.3 Edge

There are also two cases for edge. In the first case, the edge represents a send to an existing process, and in the second, a send which creates a process as a side effect. The first is called a send-append, and the second, a send-create.

A send-append is the transmission of IPC from one already existing process to another. The node which defines the edge therefore points to two process definition nodes, which must both refer to existing processes; either pre-existing processes, or created processes which have already been created in some edge *in the current process graph program*. The grammar rule to be satisfied is:

```
<edge def> ::= <edge ident> ':='
               <process ident> '-->' <process ident>
```

The edge e4 is from process2 to process3, both pre-existing processes, and so the format of this edge node will be:

```
 ____|____
|    v    |
|  edge   |
|---------|
|  &e  --|------> "edge4"
|---------|                                    _________
|  proc  -|---------------------------------> |process|
|---------|              _________            |-------|
|  proc  -|------>      |process|             | proc  | --> "process2"
|---------|             |-------|             |-------|
|         |             | proc -|--> "process3" |     |
'---|---'               |-------|             |-------|
    v                   |       |             |       |
                        |-------|             '---|---'
                        |       |                 v
                        '---|---'
                            v
```

The grammar rule to be satisfied for send-create is:

```
        <edge def> ::= <edge ident> ':='
                           <process ident> 'creates' <process ident>
```

In a send-create, the receiving process is created as a side-effect of the message transmission. The edge node refers to a sending process definition node which must refer to an existing process, and to a receiving process definition node which refers to a created process which is marked as not-yet-created. At run-time, the program file referred to by the receiving processes's process definition node will be used to create the receiving process. In the example below, program file "prog2" will be used to create process "proc2".

```
 ____|____
|    v    |
|  edge   |
|---------|                              _________
| proc1- |------------------------------> |process|
|---------|                             |-------|
| proc2- |-----> |process|              | proc  |
|---------|      |-------|              |-------|
|         |      |       |
'---|---'        | proc  |               ____|____
    v            |-------|              |   v   |
                 | &prog |----->        | prog  |
                 |-------|              |-------|
                 |       |              | &prog |->"prog2"
                 |-------|              |-------|
                 |       |              |       |
                 '---|---'              |-------|
                     v                  |       |
                                        '---|---'
                                            v
```

During parsing, a send-create edge is marked in the symbol table as a 'create' edge, and later, when the edge appears in a precedence expression, the receiving process which it creates is marked 'created'. If a process not yet marked 'created' is used in an edge in a precedence expression, a diagnostic is issued.

The grammar rule for <u>send-die</u> is:

```
        <edge def> ::= <edge ident> ':='
                            <process ident> 'dies' --> <process ident>
```

This defines an edge which has the death of the sending process as a side-effect. A message is sent from the first process to the second, and, if successful, the first process dies.

The grammar rule:

```
        <edge def> ::= <edge ident> ':='
                        <process ident> 'signals' <process ident>
```

defines an edge which consists solely of synchronization information passing from one process to another in the current process graph, or from a process in this process graph to one in another. No message is transmitted.

For simplicity, we have given the definition of program file identifiers here as:

```
        <prog ident> ::= 'prog1' |..........| 'progn'
```

The process graph description language compiler can, in fact, recognize identifiers of arbitrary length, composed of alphanumeric characters, and beginning with a letter. However, in the rest of this section we shall continue to use the suggestive subset <'prog1',22. . .,'progn'>. As process identifier, we have used "process1", "process2", etc. As edge identifier, we have used "e1", "e2", etc.

B.2.2.4 Precedence Graph

Here we describe the construction of the precedence graph which is built in the course of compilation. It is analogous to an evaluation tree in conventional compilation. The precedence graph is a directed graph in which one or more nodes are present for each construct of the process graph description language, and the directed edges between the nodes express the run-time time precedence between the nodes. If node1 and node2 are connected by a directed edge:

```
        node1 ---> node2
```

then node1 precedes node2. The precedence graph may contain loops, corresponding to iteration constructs in the process graph description language.

B.2.2.5 Statement List

A statement list is a sequence of statements S1, S2,..... A statement list is represented by a series of connected nodes.

A statement is one of the following four types.

B.2.2.6 Concatenation

The construct:

```
        S1 < S2
```

which is represented by grammar rule

```
        <concatenation> ::= <stmt> '<' <stmt>
```

leads to the linking of the first node for S2, to the final node of S1. The statements S1, S2 can be single edges, or process graph description language statements.

```
 ____|____
|    v    |
|  edge   |
|---------|
|   S1    |
|---------|
|         |
'----|----'
 ____|____
|    v    |
|  edge   |
|---------|
|   S2    |
|---------|
|         |
'----|----'
     v
```

B.2.2.7 Concurrency

Each construct indicating concurrent statements, generated by the satisfaction of the grammar rule

<concurrent> ::= <stmt> '^' <stmt>

is represented by four node groups; a concurrency-begin (^) node, the two concurrent statements, and a concurrency-end (v) node. For example: S1 ^ S2

```
 ____|____
|    v    |
|         |
|---------|
|      ---|-------------,
|---------|             |
|         |             |
'----|----'             |
 ____|____          ____|____
|    v    |        |    v    |
|  edge   |        |  edge   |
|---------|        |---------|
|   S1    |        |   S2    |
|---------|        |---------|
|         |        |         |
'----|----'        '----|----'
     |                  |
 ____|____              |
|    v    |  <----------'
|    v    |
|---------|
|         |
'---------'
```

The nodes marked S1, S2, represent arbitrary process graph description language statements. The only restriction is that they should be subgraphs with a single entry and a single exit.

B.2.2.8 Procedure Call

A procedure call is the principal mechanism for communication between processes and process graphs. The grammar rule for procedure call is:

```
<procedure call> ::= <proc name> '(' <stmt> ')'
```

The representation of a procedure call in the precedence graph is:

```
 ____|____
|    v    |
| call    |
|---------|
| proc  - |---------> "proc name"
|---------|
| pars  - |------------,
|---------|            |
|         |            :
'----|----'            :
     |            _____|_____
     |           |     v     |
     |           |   edge    |
     |           |-----------|
     |           |    S2     |
     |           |-----------|
     |           |           |
                 '-----|-----'
                       v
```

The "proc" field in the node points to the symbol table entry for the procedure definition, and the "pars" field is a pointer to the subgraph representing the parameter list for the procedure call. Most process graph procedures evaluate logical values derived from edge execution, from system or hardware error checks, or passed down from processes.

B.2.2.9 Choice

The choice construct is represented by four node groups: Again, S1, S2 represent arbitrary process graph description language statements. The choice grammar rule is:

```
<choice> ::= 'if' <expression> 'then' <stmt> |
             'if' <expression> 'then' <stmt>
             'else' <stmt>
```

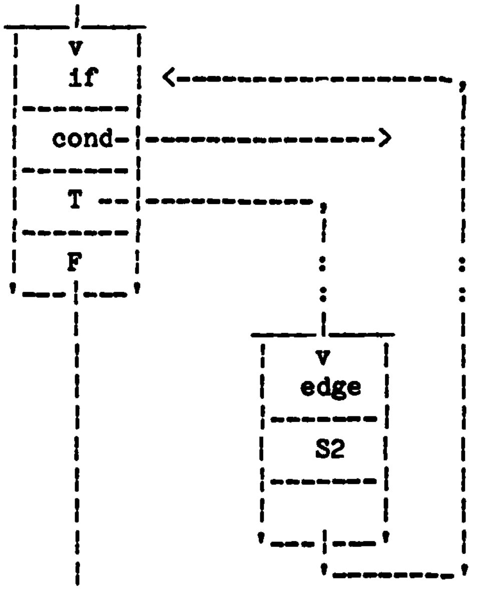
The <expression> in the if statement can itself be a process graph statement of the type: single edge, process graph procedure call, or bound expression.

B.2.2.10 Iteration

The iteration grammar rule is:

```
<iteration> ::= 'while' <expression> 'do' <stmt> 'end'
```

Iteration is a special case of choice. The iteration construct is represented as follows:



Condition and S2 represent arbitrary process graph description language statements.

B.2.3 *Translation from Task Graph to Precedence Graph*

In the task graph description language there are specification statements, assignment statements, and expressions. Specification statements have been dealt with above. Assignment statements are used in building the individual nodes of the precedence graph. The precedence relationships between edges are spcified using expressions, using the operators '^', '<'. Expressions may be nested using '(', ')'. As these edge precedence expressions are parsed, a 'precedence graph' is built which is analogous to the expression tree built during the parsing of arithmetic expressions in an algorithmic language such as Algol or Pascal.

- Edge_-_<exp>_::=_<edge>

Each time that an edge is recognized, a node is created to represent it and its address saved on both the root and leaf stacks. It will become clear why two stacks are used:

```
      case <edge>:
              /* create_node() returns the new node's address */
          node = create_node (edge_node);
          push_root (node);
          push_leaf (node);
          break;
```

We shall describe algorithms using their 'C' language representation (see, for instance [KERN\ 3]). In this language, procedure invocation has the syntax:

```
  procedure (param1, param2,....);
```

while the selection of an element from a data structure has the syntax:

```
  structure -> element
```

- Precedence_-_<exp>_::=_<exp1>_'<'_<exp3>

Each time that a precedence construct is recognized , we are required to set up connecting edges (representing time precedence) between two subgraphs in the precedence graph, one representing <exp1>, and the other representing <exp2>. As <exp1> and <exp2> were recognized themselves, their root and leaf addresses were pushed on the root and leaf stack, so at this point, the top elements on the root and leaf stacks are the root and leaf addresses of <exp2>, and the second-from-top are the root and leaf addresses of <exp1>. We have to pop these addresses, link <exp1> to <exp2>, and push the root and leaf addresses of the linked subgraph representing <exp1> '<' <exp2>.

```
      case <precedence>:
          temp = pop_leaf();
          last = pop_leaf();
          next = pop_root();
          last -> fwd_ptr = next;
          next -> parent = last;
          push_leaf (temp);
          break;
```

- Concurrency_-_<exp>_::=_<exp1>_'^'_<exp2>

An operator such as '^' requires more complicated code generation than, for instance, '+' or '*'. A '^' operator takes a task graph subgraph as its right and left-hand operands, but then these two subgraphs must be terminated by a 'v' node which unites their two lower-most leaves. The expression tree of a language with conventional binary operators is a tree; that of a task graph is a double tree. For every '^' which splits flow of control into two parallel streams, there is a later, matching 'v' node which unites these two streams again. For instance, the expression

(e1 < e2 < e6) ^ (e3 < e7) ^ e4 ^ e5

will generate the following subgraph:

```
               ^
 ______________|_______________
 |  |
 e1 _____________|______________
 |  | ^
 e2 e3 ________|_______
 |  |  |  |
 |  |  e4 e5
 e6 e7 |_________|
 |  |  v
 |  |__________________|
 |  v
 |__________________|
              v
```

The essential point is that each opening par begin '^' node (parallelism- ·begin) must be matched by a closing par end 'v' node. It follows that, as well as enstacking the addresses of the roots of expression subgraphs as they are formed, we also have to save the addresses of the leaf nodes of these subgraphs, so that we can later terminate pairs of leaves with a 'v' node. We therefore employ two stacks in precedence graph building, an expression root stack and a leaf stack.

- Choice_-_::=_'if'_<exp>_'then'_<stmt>_'else'_<stmt>

An analogous problem is encountered in building a representation of an

if P then S1 else S2;

This is solved using a similar double-stacking algorithm. In the case of 'if' however, we have to handle three expression subgraphs; those for the 'if' condition, the true, and the false exit subgraphs from the 'if'. Each of these three subgraphs is enstacked as it is created, and then, upon recognition of

```
<choice> ::= 'if' <expression> 'then' <stmt> |
             'if' <expression> 'then' <stmt> 'else' <stmt>
```

the subgraphs are de-stacked in reverse order (false, true, condition), and attached to the 'if' node. Finally, the terminating 'fi' node has to have the output leaves of the true and false 'if' exit subtrees connected to it, and so these two leaf addresses are de-stacked from the leaf stack. The code for handling if and fi nodes is:

```
case <choice>:
    node = new_node ("if");
    node -> false_exit = pop_root ();
    node -> true_exit = pp_root ();
    node -> condition = pop_root ();
    (node -> true_exit) -> parent = node;
    (node -> false_exit) -> parent = node;
    push_root (node);
    node = new_node ("fi");
    false = pop_leaf ();
    true = pop_leaf ();
    false -> fwd_ptr = node;
    true -> fwd_ptr = node;
    node -> falseparent = false;
    node -> trueparent = true;
    push_leaf (node);
    break;
```

- Iteration

The code for <iteration> is an obvious variation on that for <choice>. The true-exit subgraph connected to the 'f' node has its leaf node connected back again to the 'if' node, while the false-exit points to the next subgraph in the precedence graph after the while-do.

```
     case <while>:
         node = new_node ('if');
         node -> false_exit = 0;
         node -> true_exit = pop_root ();
         (node -> true_exit) -> parent = node;
         node -> condition = pop_root ();
         (node -> condition) -> parent = node;
         push_root (node);
         true = pop_leaf ();
         true -> fwd_ptr = nod;
         push_leaf (node);
         break;
```

APPENDIX C

THE DESIGN OF A PROGRAMMING LANGUAGE BASED ON COMMUNICATION NETWORKS

Aurthur B. Maccabe
Richard J. Leblanc

C.1 INTRODUCTION

The design and implementation of message-oriented programming languages has recently become an active area of research. The increased activity in this area is due, in part, to the increased interest in distributed processing systems. Message-oriented languages structure programs as collections of processes that communicate and cooperate using message transmission primitives. Distributed processing systems can be viewed as collections of processor nodes with independent address spaces that communicate and cooperate by exchanging messages over communication channels. Hence, there is a natural mapping of the units of a program written in a message-oriented language to the resources of a distributed processing system.

This paper presents the design of a new message-oriented language, PRONET (Processes and Networks). The goals of PRONET are to provide a high degree of process independence and a mechanism for describing process hierarchies, while obtaining information about inter-process relationships which will aid in effective program execution. In most message-oriented languages, relationships between processes must be expressed in the descriptions of the individual processes. PRONET has been developed to investigate the separation of inter-process relationships from the description of processes. This separation is expected to enhance process independence while isolating information which will aid in the distribution of processes. The initial design of PRONET concentrates on inter-process relationships that describe structural aspects of the communication environment used by processes. To this end, PRONET provides powerful features for describing the instantiation and dynamic reconfiguration of "communication networks."

*This paper was published in the Proceedings of the 3rd Int. Conf. on Distributed Computing Systems, Miami Florida, October 1982.

C.1.1 Programming Environments

PRONET was motivated by a perceived need to aid application programmers in their efforts to use the programming environment presented by a distributed processing system [Fors81]. Our view of distributed processing systems is based on the definition presented by Enslow [Ensl78] and subsequently refined by Enslow and Saponas [Ensl81]. To distinguish the systems meeting the criteria of this definition from other distributed processing systems, they have been termed "Fully Distributed Processing Systems" (FDPS).

For the purposes of this paper, an FDPS is a collection of loosely coupled processors that function in a cooperatively autonomous fashion to provide services ([Ensl78], [Clar80]). The processors are autonomous in that their activities are entirely controlled by local decision-making criteria. To avoid total anarchy, the decision-making criteria of each processor are integrated with the goal of cooperation. This cooperation is represented within each processor by a component of the network operating system (NOS). The primary function of the NOS is to provide a unified view of the resources available in an FDPS. It performs this task by imposing a layer of control above the processors which recognizes and respects the autonomy of the individual processors. The assumed existence of an NOS appears to distinguish the environment we anticipate from that assumed by other researchers. For example, upon request the NOS will provide scheduling and allocation functions based on its global view of the network. Thus, when a new process is created, a program can use the NOS to determine an appropriate physical location for the process.

C.1.2 Logical Communication Networks

Programs written in message-oriented languages may be viewed as specifying "communication networks". The nodes of these networks are the processes defined or used by the program, while the arcs between nodes represent communication links. These communication links are directed and may be used in the transmission of any number of messages. Note: this definition of "communication network" concentrates on connectivity, other definitions are possible--for instance the task graphs of [Live80] reflect a definition that concentrates on communication sequencing.

Languages that support dynamic reconfiguration of communication networks typically do so by allowing processes to create new processes and/or allowing processes to pass the names of processes (or ports) to other processes ([Kahn77], [Feld79], [Lisk79], [Hewi79]). Because the activities that control dynamic reconfiguration of the communication network are intermixed with the activities of individual processes, they are not readily available for examination by an operating system or a person attempting to understand the program.

Other programming languages/systems that support a similar separation (UNIX [Bour78], Mesa [Mitc79], Task Forces [Jone79] and PCL [Less79]) do not enforce a complete separation. In each of these languages/systems a process may specify the creation of new processes in its description. Thus, while an abstract view of the communication environment is available, neither the operating system nor a person reasoning about the program may rely on the completeness of this view. In PRONET, the conditions and activities associated with any structural modification of the communication environment (including process creation) must be stated in a network specification.

C.2 THE BASIC FEATURES OF PRONET

PRONET is composed of two complementary sublanguages: a network specification language, NETSLA, and a process description language, ALSTEN. Programs written in PRONET are composed of network specifications and process descriptions. Network specifications initiate process executions and oversee the operations of the processes they have initiated. The overseeing capacity of network specifications is limited to the maintenance of a communication environment for a collection of related processes. The processes initiated by a network specification can be simple processes, in which case the activities of the processes are described by ALSTEN programs, or they can be "composite processes", in which case their activities are described by a "lower-level" network specification.

ALSTEN is an extension of Pascal which enables programmers to describe the activities of sequential processes. During their execution, processes may perform operations that cause events to be announced in their overseeing network specification. Network specifications, written in NETSLA, describe the activities that are performed when an executing process 'announces' an event. This chapter describes the mechanisms that enable processes to announce events and the network-level activities that can be performed in handling an announced event. Two principles have influenced the design of these features: independence of process descriptions and distributed execution of network specifications.

C.2.1 The Features of ALSTEN

Pascal was chosen as a basis for process descriptions because of its simplicity, its strong type checking and the availability of an extendable compiler. ALSTEN is, for the most part, an extension of Pascal with the exception of the 'file' interface provided by Pascal. Process descriptions written in ALSTEN communicate with their surrounding environment primarily through locally declared ports which are visible to their overseeing network specification. Hence, the Pascal 'file' interface has been replaced by port declarations and message transmission operations in ALSTEN.

This section describes the ALSTEN features associated with messages transmission and process-defined events. Each message transmission initiated by a process causes an event to be announced in the network specification which oversees the operations of the process. In handling this event, the overseeing network specification determines where the message is to be delivered and how the communication environment being maintained is to be altered as a result of transmitting the message. While this provides a powerful mechanism for dynamic reconfiguration of logical communication networks and maintains a high degree of independence in process descriptions, a more flexible mechanism of transmitting information from an executing process to its overseeing network specification is often useful. In ALSTEN, this mechanism is provided by event declarations and an 'announce' operation.

C.2.1.1 Message Transmission Operations

Message transmission is the primary mechanism by which executing processes communicate with their other objects in their environment (their overseeing network specification and processes). The basic message transmission operations of ALSTEN are 'send' and 'receive'. Both operations are specified "in-line", as are the 'read' and 'write' operations of Pascal (and in contrast to the 'interrupt handling' receive of Mininet [Live80]).

The send operation of ALSTEN is best classified as a 'buffering' operation with partial 'blocking'. When a process executes a send operation, its (logical) execution is blocked until all events caused by the message transmission are handled. Handling a message transmisson event may involve an alteration of the logical network which oversees the execution of the process sending the message and delivery of the message to any number of 'receiving' processes. Message 'delivery' does not require that the 'receiving' process perform a receive operation, but does block execution of the sending process until the message value has been copied into the 'IPC space' of the 'receiving' process. The address space of all ALSTEN processes is partitioned into an 'IPC space' and a 'manipulation space'. The 'IPC space' consists of queues of messages which have been delivered to the process but have not been 'received'. The 'manipulation space' of a process contains the values of variables which are local to the process.

To receive a message, a process must wait until an acceptable message is available in its 'IPC space'. When it has been completed, the receive operation of ALSTEN has the effect of transferring the message received from the 'IPC space' of the receiving process to its 'manipulation space'.

Executions of the send and receive operations of ALSTEN are specified by send and receive statements. The syntax of these statements is shown in Figure 1. These statements are introduced into the grammar of Pascal [Jens74] as new variations of the 'simple statement'.

```
<send stmt> ::=
     send [<expr>] to <bound port denoter>
<receive stmt> ::=  <simple receive> ¦ <conditional receive>
<simple receive> ::=
     receive [<variable>] from <free port denoter>
<conditional receive> ::=
     when {<receive part>} [<otherwise part>] end
<receive part> ::=  <simple receive> [do <stmt>]
<otherwise part> ::=  otherwise <stmt>
```

Figure 1. Send and Receive Statements in ALSTEN

The 'send stmt' causes the value of the expression to be transmitted through the output port identified by the 'bound port denoter'. The 'simple receive' causes the 'variable' to be assigned the value of the next message to be received from the port identified by the 'free port denoter'. If any of the simple receives in a 'conditional receive' can succeed immediately, one is chosen arbitrarily and the statement following the corresponding do is executed. Otherwise, when there is no 'otherwise part', the execution of the process is blocked until one of the receive statements can succeed. If none of the receive statements succeed immediately and there is an 'otherwise part', the statement following the otherwise is executed. This control structure presents a restricted form of the Ada select [DoD80].

C.2.1.2 Ports for Message Transmission

To emphasize independence of process descriptions, message transmission operations are issued to locally declared 'ports'. The ports of a process description are visible to network specifications that create instances of processes which execute the process description. Simple ports are declared with a direction ('in' or 'out') and an associated message type. The associated message type defines how messages transmitted through a port are to be interpreted. A message type can be a 'signal' (only control information is transmitted) or any data type which does not contain pointer or file components.

The notion of 'server' processes has had a significant impact on the design of the message transmission features of PRONET. Server processes are characterized by two properties: first, a server process must respond to requests from an unknown number of 'user' processes and, second, it must ensure that each response is directed toward the process that generated the corresponding request. When using server processes and user processes in different programs, it may be necessary to impose 'intermediary' processes on one or more of the communication paths between a server and a user. An *intermediary may mediate between differing message formats* or communication protocols.

The ALSTEN features related to the description of server processes are 'port groups', 'port sets' and 'port tags'. Port groups provide a means for collecting a number of simple ports into a single bundle. A 'bidirectional port' would be a port group containing two simple ports, one input and one output, each with an associated message type. Port sets, on the other hand, are used to denote collections of identical ports—either simple ports or port groups. Port sets provide server processes with a mechanism for communicating with an unknown number of user processes. Each element in a port set is assumed to be associated with a unique user--if the port set is a collection of port groups, the simple ports in each port group may be connected directly to the user or to intermediaries. In order that a server may restrict its communications to a particular user, we introduce port tag variables. Port tag variables are declared to range over the members of a single port set. The value of a port tag variable can be set in a receive statement and may be used in subsequent send and receive statements.

The syntax for declaring ports and port tag variables in ALSTEN is shown in Figure 2. Port declarations appear in the header of a process description and hence, the definition of any 'msg type' must appear outside of the process description (unless it is a standard type; e.g., integer, real, signal, etc.). This is necessary as these definitions must be shared by other processes (that either send or receive messages of type 'msg type') and any network specification that oversees the operations of processes executing the process description. The nonterminal <port tag type> is introduced as a new 'type' in the syntax presented in [Jens74].

```
<port decl> ::=  <simple port decl> | <port group decl>
<simple port decl> ::=
      port [set] <port id> <direction> <msg type>
<port id> ::=  <id>
<direction> ::=  in | out
<msg type> ::=  <type id>
<port group decl> ::=
      port [set] <port id> '(' <subport list> ')'
<subport list> ::=  <subport decl> {';' <subport decl>}
<subport decl> ::=  <subport id> <direction> <msg type>
<subport id> ::=  <id>
<port tag type> ::=  tag of <port id>
```

Figure 2. Port and Port Tag Declarations in ALSTEN

Figure 3 presents the ALSTEN syntax for denoting port instances in send and receive (and announce) operations. A 'bound port denoter' whose 'simple port denoter' identifies a 'port set' must contain a 'use tag part' to identify the specific instance of the port set being denoted. Recalling the syntax of the send operation presented in Figure 1, the message type of the port denoted in a send operation must be "name equivalence" compatible with the type of the expression being transmitted (if the message type is 'signal' no expression can be present). A similar restriction holds for receive operations.

```
<bound port denoter> ::=  <simple port denoter>
    | <simple port denoter> <use tag part>
<simple port denoter> ::=  <port id>
    | <port id> '.' <subport id>
<use tag part> ::=  use <port tag variable>
<port tag variable> ::=  <variable>
<free port denoter> ::=  <bound port denoter>
    | <simple port denoter> <set tag part>
<set tag part> ::=  set <port tag variable>
```

Figure 3. Denoting Ports in ALSTEN

The use of port sets, port groups and port tag variables is illustrated in Figure 4 which presents the description of a simple server process. This process implements a shared sequence of numbers. In line 2 a port set, 'user', is declared. The elements of this port set are instances of a port group containing an input port 'req' and an output port 'rsp'. Lines 10 and 11 illustrate the setting and subsequent use of the port tag variable 'user_tag' (declared on line 4). In line 10, the value of 'user_tag' is <u>set</u> to indicate which instance of the port set 'user' the request is received from. The value of 'user_tag' is <u>used</u> in line 11 to direct the response to the element in the set 'user' from which the request was received.

```
1   process script shared_sequence
2      port set user (req in signal; rsp out integer);
3      var
4         user_tag : tag of user;
5         sequence_val : integer;
6      begin
7         sequence_val := 0;
8         while true do
9            begin
10              receive from user.req set user_tag;
11              send sequence_val to user.rsp use user_tag;
12              sequence_val := sequence_val + 1
13           end (* while *)
14     end (* shared sequence *)
```

Figure 4. A Simple Server Process

C.2.1.3 Process-Defined Events

As has been discussed, the execution of a send operation causes a message transmission event to be announced in the network specification which oversees the operation of the process which executes the send operation. Thus, the transmission of a message may lead to a reconfiguration of the communication environment used by the sending process. This is particularly useful in providing a mechanism for dynamic reconfiguration of logical communication networks while maintaining a high degree of independence in process descriptions. However, a more flexible interface between processes and their overseeing network specifications which allows processes to indicate significant changes in their state or possible errors in communications is often useful. In ALSTEN, this interface is provided by event declarations and the <u>announce</u> statement.

Process descriptions may declare event names and subsequently 'announce' these events during their execution. The activities to be performed (if any) when an executing process announces an event are described in the network specification which oversees the execution of the process. Hence, this mechanism provides a flexible interface between the process-level and the network-level while maintaining the separation of these levels.

Event declarations have the form:

```
<event decl> ::=
   event <event name> [about <port id>]
```

Event declarations appear in the header of a process description and follow the port declarations of the process description. The event name is an identifier which can be used in subsequent announce operations. The optional 'about part' allows the process to associate an event with a set of ports. This is useful in indicating erroneous communication (either protocol or consistency) on a specific port.

The announce operation of ALSTEN is introduced as a statement (a 'simple statement' in the grammar of Pascal [Jens74]):

```
<announce stmt> ::= announce <event name>
        [about <bound port denoter>]
```

The 'event name' must be the name of a declared event. Further, if this event has been declared with an associated port set, the about clause must be present and must denote an instance of the associated port set.

An example of process-defined events is presented in Figures 5 and 6. Figure 6 presents the script for instances of 'mailbox' processes. The types used in the mailbox process script are shown in Figure 5. In this case, the event 'mailbox_empty'--declared in line 5 of Figure 6 and announced in line 24--is used to indicate a significant change in the internal state of the process.

```
type
   letter = array [1..120] of char;
   user_rsp_kinds = (empty, mail_item);
   user_rsp = record
     case kind : user_rsp_kinds of
        empty : ();
        mail_item : (let : letter)
   end; (* user_rsp *)
```

Figure 5. Mailbox Process Script Type Definitions

```
 1  process script mailbox
 2     port input in letter;
 3     port output out letter;
 4     port control in signal;
 5     event mailbox_empty;
 6     var
 7        next_rsp : user_rsp;
 8        done : boolean;
 9     begin
10        repeat
11           receive from control;
12           next_rsp.kind := mail_item;
13           done := false;
14           repeat
15              when
16                 receive next_rsp.let from input do
17                    send next_rsp to output;
18                 otherwise
19                    done := true
20              end (* when *)
21           until done
22           next_rsp.kind := empty
23           send next_rsp to output;
24           announce mailbox_empty
25        until false
26     end (* mailbox script *)
```

Figure 6. The Mailbox Process Script

This process script implements a simple mailbox which acts as a repository for 'letters'. Responses from the mailbox will be of type 'user_rsp' which is defined in lines 3-8 of Figure 5. Upon reception of a signal on its 'control' port (line 11 of Figure 6), the mailbox forwards the letters on its 'input' port to its 'output' port (lines 16 and 17). When there are no letters remaining on the mailbox input port, the process sends an 'empty' message to its output port and announces the event 'mailbox_empty' (lines 22-24). The process then cycles to wait for the next request to deliver its 'contents' (line 11).

The structure of this mailbox is natural considering the semantics of the ALSTEN send operation. Because senders do not wait until their messages are received, there is no need for the mailbox to receive messages as they are sent. Hence, the mailbox does not maintain an internal representation of its contents but rather, relies on the run-time support environment to maintain collections of letters. A simple mail system that uses this mailbox process script and illustrates the handling of process-defined events will be presented in the next section.

C.2.2 The Features of NETSLA

The features of NETSLA are aimed at specifying the initial configuration and subsequent modifications of a communication environment. The overriding principle followed in the design of these features is that of "centralized expression--decentralized execution" [Live80]. Centralized expression is important in presenting the abstraction to be supported by network specifications. All of the inter-process relationships that describe a communication environment appear in a single network specification. However, this communication environment is not maintained in a centralized fashion. Processes maintain their communication environment indirectly. When they execute send or announce operations, processes perform the activities specified by their overseeing network specifications; however, the nature of these activities are unknown to the process.

C.2.2.1 An Overview of Network Specifications

The syntax for specifying a network is shown in Figure 7. Like the header of an ALSTEN process script, a network header can contain port and event declarations. Network specifications that do declare ports and/or events will be used as "composite processes" in higher-level network specifications.

```
<network specification> ::=  <network header>
         {<process class specification>}
         {<event handling clause>}
         [<initialization clause>]
      end <identifier>
<network header> ::=  network <net id> ';'
      {<port decl>} {<event decl>}
<process class specification> ::=
      process class <process id>
         [<process attributes>]
         {<port decl>}
         {<event decl>}
      end <process id>
<process attributes> ::=  attributes
         <field list> end attributes
```

Figure 7. Network Specifications in NETSLA

The process class specifications contained in a network specification capture those portions of a process description that are visible in a network specification--its name, port declarations and event declarations--and a 'process attributes' part. The name, port declarations and event declarations stated in process class specification are a reiteration of the process script or network specification header which is used to implement instances of the process class. Process attributes are used to identify the characteristics associated with instances (processes) in a process class. The implementation of a process class may be a network specification (in which case instances of the process class are actually "composite processes") or a process script written in ALSTEN. In either case, this implementation is not contained in the network specification. Process implementations are compiled separately and compatibility between specification and implementation is checked in a pre-linkage phase. The remaining portions of a network specification, the event handling clauses and the optional initialization clause, describe the instantiation and subsequent modifications of the logical communication network which is maintained by the network specification.

When a logical network is instantiated, its initialization clause is elaborated. This initialization clause is used to create a collection of processes and delineate communication paths between them. A simple network specification is illustrated in Figure 8. One process class, 'proc_class' (lines 2-5), is used in this network specification. Instantiation of the logical communication network is specified in lines 6-14 and involves the creation of three processes (lines 7-9) and the establishment of communication paths between them (lines 10-13). The statement '<u>connect</u> proc1.output <u>to</u> proc3.input' (line 10) specifies that the messages sent to the output port of 'proc1' are to be transmitted to the input port of 'proc3'. A graphical representation of the logical communication network established by this network specification is shown in Figure 9.

```
1     network static_net
2        process class proc_class
3           port input in integer;
4           port output out integer;
5        end proc_class
6        initial
7           create proc1 : proc1_class;
8           create proc2 : proc1_class;
9           create proc3 : proc1_class;
10          connect proc1.output to proc3.input;
11          connect proc2.output to proc3.input;
12          connect proc3.output to proc1.input;
13          connect proc3.output to proc2.input;
14    end static_net
```

Figure 8. A Simple Network Specification

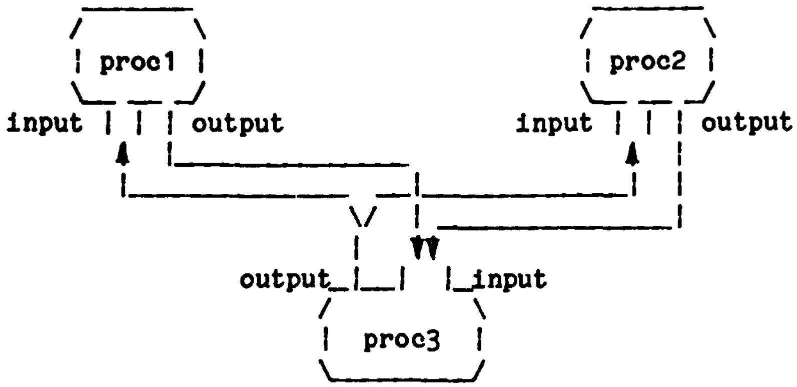


Figure 9. Graphical Representation of the Simple Network

This simple example illustrates two of the simple activities that can be performed in a network specification, creation and connection. As illustrated, creation involves the binding of a name to each process instance as it is created. In NETSLA these and other name bindings are limited to the clause in which they appear. Hence, the names 'proc1', 'proc2' and 'proc3' may be used throughout the initialization clause but would not be usable in other clauses unless they were explicitly bound to objects (process or port instances) in these clauses. Connection is shown in lines 10-13. One-to-one, many-to-one (messages are ordered by time of arrival) and one-to-many (messages are replicated) connections between ports can be specified. To be connected, ports must be compatible in both message type and direction. Message type compatibility, like type compatibility in PASCAL, is based on named equivalence of types. The definition of port direction compatibility has two components:

i) If one port is a network-level port (declared in the network header) and the other is a process-level port (declared in a process class specification), the ports must have similar directions.

ii) If both ports are process-level ports or both are network-level ports, the ports must have opposite directions.

C.2.2.2 Event Handling

The initialization clause is sufficient for the description of static networks. However, other features are needed to describe dynamically changing communication environments. In PRONET, these features are based on the notion of network events. During their execution, processes may perform operations which announce events to their overseeing network specification (using send or announce). NETSLA provides two mechanisms for handling announced events in network specifications: connections and 'event handling clauses'.

Connections are one mechanism for handling the event associated with message transmission on a port. When a connection between two ports has been established, this 'message transmission' event is handled by transferring the message from the sending port to the receiving port. The connection mechanism is distinct from the event clause mechanism in three ways: connections can be established or broken dynamically, the activities of this mechanism are defined by the language and connections can only be used to handle the 'message transmission' event.

Event handling clauses are more flexible in the types of events they can handle and the activities they can specify but are established statically and cannot be 'broken'. Event handling clauses provide a capability to specify the activities that are to be performed when a message is transmitted (if a simple connection is not sufficient), when a process defined event is announced, when an element of a network declared port set is created or when an element of a network declared port set is removed. The syntax of the event handling clauses of NETSLA is illustrated in Figure 10.

```
<event handling clause> ::=  <arrive clause>
    | <enter clause> | <leave clause> | <when clause>
<arrive clause> ::=  <arrive clause header>
      <activity list> end arrive
<arrive clause header> ::=  arrive [<id>] on
      <arrive port binding> [from <processs binding>]
<arrive port binding> ::=  [<subport id> of] <port binding>
<process binding> ::=  [<id> ':'] <process class name>
<port binding> ::=  [<id> ':'] <port set name>
<enter clause> ::=  <enter clause header>
      <activity list> end enter
<enter clause header> ::=  enter <port binding> do
<leave clause> ::=  <leave clause header>
      <activity list> end leave
<leave clause header> ::=  leave <port binding> do
<when clause> ::=  <when clause header>
      <activity list> end when
<when clause header> ::=  when <event name>
      announced by <process binding> do
<initialization clause> ::=  initial <activity list>
```

Figure 10. NETSLA Event Handling and Initialization Clauses

The bindings in the various event clause headers are used to bind names to the objects (message, process instance or port instance) involved in the event being handled. For example, *when* clauses are used to handle the announcement of process-defined events. As such, the following 'when clause header' could be used in a network specification to handle the 'mailbox_empty' event when this event is announced by a process executing the mailbox process script shown in Figure 6.

when mailbox_empty announced by box : mailbox do

In this case, the event being handled is the process-defined event 'mailbox_empty' and the name 'box' is bound to the instance of the mailbox process that announced the event. When clauses are also used to handle the standard event 'done'. Whenever an executing process terminates its activities, the standard event 'done' is announced to its overseeing network specification.

Arrive clauses are used to handle message transfer events when simple connections between ports are not sufficient. An arrive clause can be associated with the arrival of a message on a network-level 'in' port, in which case the optional 'from process binding' is not specified. An 'arrive clause' can also be associated with the arrival of a message on an 'out' port of a process instance, in which case the 'from process binding' identifies the process class of interest and can be used to bind a name to the instance which is transmitting the message. The first identifier in an 'arrive clause header' is used to bind a name to the message value being transmitted. The 'arrive port binding' in an 'arrive clause header' identifies the port set, binds a name to the port group instance through which the message is transmitted and identifies the subport being used.

When an event is announced, two possibilities exist: no 'handlers' (connections or event handling clauses) are associated with the event or at least one 'handler' is associated with the event. In the latter situation, the activities specified by each handler are performed on the event (in an arbitrary order). For example, when multiple connections are established for a port, any message transmitted through the port is replicated and delivered along each of its connections. When no handlers are associated with an event, its announcement has no effect on the communication environment being maintained by the network specification. Moreover, the object (process or overseeing network specification) that announced the event cannot determine if the event was handled. For example, when a process sends a message to a port that has no established connection or arrive clause, the message is removed from the port and the sending process cannot determine that its message has not been delivered.

C.2.2.3 Simple Activities

```
<activity list> ::=  <activity> {';' <activity>}
<activity> ::= <simple activity> | <structured activity>
<simple activity> ::=  <creation> | <termination>
     | <removal> | <connection> | <disconnection>
     | <message transmission> | <value construction>
     | <event announcement> | <attribute assignment>
<creation> ::=  create <process binding>
     | create <port binding> on <process instance>
<termination> ::=  terminate | terminate <process instance>
<removal> ::=  remove <process instance>
     | remove <port group instance>
<connection> ::=  connect <port instance> to <port instance>
<disconnection> ::=  disconnect <port instance>
     | disconnect <port instance> from <port instance>
<message transmission> ::=
       send <msg value> to <port instance>
<value construction> ::=  construct <id> ':' <type name>
       '[' <component assignment list> ']'
<attribute assignment> ::=  <attribute denoter> ':=' <value>
<event announcement> ::=  announce <event name>
       [about <port group instance>]
```

Figure 11. Simple Activities in NETSLA

NETSLA provides nine basic activities which can be used in initialization and event handling clauses: creation, termination, removal, connection, disconnection, message transmission, attribute assignment, event announcement and value construction. The syntax used in specifying these activities in NETSLA is shown in Figure 11.

The creation activity can be applied to a process class or a port set of a process instance. In the first of these variations, a new instance of the process class executing the process script or network specification associated with the process class is instantiated. The 'process binding' part of the creation activity is used to identify the process class and bind a name to the newly created instance. This form of the creation activity was illustrated in the network specification illustrated in Figure 8. The second variation of this activity creates a new port group instance in a port set on a process instance. The 'port binding' part of this variation is used to identify the port set and bind a name to the newly created port group instance. For example, a network specification containing the process class specification:

```
process class shared_sequence
   port set user (req in signal; rsp out integer);
end shared_sequence
```

could contain the activities:

```
create server : shared_sequence;
create user_port1 : user on server;
```

(Recall the 'shared_sequence' script of Figure 4.) The latter of these activities creates an element of the port set 'user' on the process instance identified by 'server'. This newly created element is bound to the name 'user_port1'.

The termination activity can be applied to a process instance or to the entire logical network being maintained by a network specification. When this activity is applied to a process instance, the activities of the process are terminated and no further messages or events will be transmitted to or received from the terminated process. When no process instance is specified in a termination activity, the logical network maintained by the network specification is terminated. This involves the termination of all process instances executing in the logical network.

The removal activity of NETSLA can be applied to a process instance or to a port group instance on a process instance. In the latter variation, no future messages will be transmitted through the port instance which has been removed. Attempts to transmit messages through a removed port will have no effect. When the removal activity is applied to a process instance, the process which has been removed may continue to execute and may generate future messages; however, no future messages will be transmitted to the identified process. (In effect, all 'in' ports on the process instance are removed.) This somewhat unusual definition of 'removal' derives from two considerations: process execution and the ALSTEN send operation. Because messages are buffered using the ALSTEN send operation, processes may have meaningful work to complete before their activity is terminated. Further, processes have an inherent termination built into their descriptions.

The connection activity involves two port instances and once performed ensures that all messages transmitted through the first port instance will be transmitted through the second until this connection is broken by a subsequent disconnection activity. This activity was illustrated in the simple network specification shown in Figure 8.

The disconnection activity applies to ports and has two variations. In the first, two (presumably connected) port instances are identified. This variation breaks a previously established connection between the identified ports. In the second variation, only one port instance is identified. This variation breaks all previously established connections involving the identified port. Once a connection between two port instances has been broken, future messages transmitted through the first port will no longer see the connection and hence, will not automatically be transmitted through the second port.

Message transmission involves the transmission of a message value through a port instance and has the same semantics as the send operation of ALSTEN.

NETSLA does not provide general variable declaration or assignment mechanisms as does Pascal (and ALSTEN). Instead, NETSLA is based on a dynamic binding of identifiers to values in event clause headers, creation activities or structured activities (discussed in the next section). For example, the value of the message being transmitted can be bound to an identifier in an 'arrive clause header'. While this is sufficient for most purposes, occasionally there arises a need to construct values of (Pascal) structured types. The value construction activity of NETSLA has been introduced to fill this need. Value construction involves the assignment of values to the components of a structured type (the type of the value being constructed is given by 'type name') and the binding of an identifier to the value constructed. The identifier can then be used in later activities to refer to the value constructed.

Periodically, the attributes of a process instance will need to be updated to reflect changes in the characteristics of the process instance. The attribute assignment activity is provided to enable the updating of the attributes of a process instance. An attribute of a process instance is denoted by a conjunction of a 'process instance' and an attribute name. The type 'value' assigned to an attribute of a process instance must be compatible with the type of the attribute.

Like process descriptions, network specifications that act as "composite processes" may need to announce events to their overseeing network specification while they are active. This capability is provided by the event announcement activity.

C.2.2.4 Structured Activities

NETSLA provides structured activities for alternation, iteration and location. The syntax for the alternation activity is presented in Figure 12. This activity is derived from the _case_ statement of Pascal and provides a mechanism for specifying alternative lists of activities to be performed on the basis of an available value.

```
<structured activity> ::=  <alternation>  | <iteration>
     | <location>
<alternation> ::=  case <value> of
     {<case list element>} [<otherwise part>] end case
<case list element> ::=
   <case label list> ':' '(' <activity list> ')'
<otherwise part> ::=  otherwise <activity list>
```

Figure 12. Alternation in NETSLA

The syntax of the location and iteration activities is presented in Figure 13. These activities provide a mechanism for selecting process and port instances in the logical network maintained by a network specification. Both activities are based on a 'selection binding' which specifies the criteria to be used in selecting groups of object (port and process) instances. The 'selection binding' is also used to bind names to the objects selected.

The iteration activity is a looping construct. The activity list specified in the iteration activity is performed for each group of objects that meet the criteria of the 'selection binding'. In each iteration of the activity list, a new group of object instances is selected and bound to the names specified in the 'selection binding'. The location activity is a simple conditional construct. The activity list specified in the location activity will be performed at most one time for one group of objects that meet the criteria of the 'selection binding'. In the case of location, if multiple groups of object instances meet the criteria, one of these groups is selected arbitrarily and the object instances in this group are bound to the specified names. In the case of iteration, the activity list specified will be applied to all groups, but the order of application is arbitrary. If no group of object instances meets the criteria of the 'selection binding', in either iteration or location, the activity list specified in the optional 'else part' will be performed.

```
<iteration> ::=  <iteration header> <activity list>
     [<else part>] end range
<iteration header> ::=  range <selection binding> do
<location> ::=  <location header> <activity list>
     [<else part>] end find
<location header> ::= find <selection binding> do
<else part> ::=  else <activity list>
<selection binding> ::=  <simple selection binding>
    | <nested selection binding>
<simple selection binding> ::=
     <port binding> [<where clause>]
    | <process binding> [<where clause>]
<nested selection binding> ::=<port binding> on
     <process binding> [<where clause>]
<where clause> ::=  where <criteria>
<criteria> ::=  <criteria factor>
    | <criteria> or <criteria factor>
<criteria factor> ::=  <criteria primary>
    | <criteria factor> and <criteria primary>
<criteria primary> ::=  not <criteria primary>
    | <connectivity criteria> | <attribute criteria>
<connectivity criteria> ::=  connected <port instance>
     [to <port instance>]
<attribute criteria> ::=
     <attribute denoter> <rel op> <value>
```

Figure 13. Iteration and Location in NETSLA

The 'selection binding' can be a 'simple selection binding' or a 'nested selection binding'. A 'simple selection binding' is used to select a single object instance (one for each iteration in the case of the iteration activity), while a 'nested selection binding' identifies a process instance and a port instance on the identified process.

The port and process bindings in the simple and nested selection bindings identify the process class and/or port set of interest. The optional 'where clause' is used to impose additional selection criteria based on connectivity or attribute values.

C.2.2.5 A Simple Mail System

This section presents the design of a simple mail system to illustrate the basic features of NETSLA. The mail system provides services that allow users to create numbered mail boxes, read the mail in a numbered mailbox and send letters to a numbered mailbox. The type definitions needed in the design of the simple mail system are presented in Figure 14.

```
type
 letter = array [1..120] of char;
 user_req_kinds = (make_box, read_mail, send_mail);
 user_req = record
    number : integer;
    case kind : user_req_kinds of
       make_box, read_mail : ();
       send_mail : (let : letter);
 end;
 user_rsp_kinds = (empty, mail_item);
 user_rsp = record
    case kind : user_rsp_kinds of
       empty : ();
       mail_item : (let : letter);
 end;
```

Figure 14. Simple Mail Systems Type Definitions

An external view of the mail system is illustrated in Figure 15 and is characterized by a set of ports, each of which has a request subport, 'req', and response subport, 'rsp'. The internal organization of the mail system is hidden, only the ports defined by the mail system are visible. In particular, users of the mail system are unable to discern whether the mail system is organized as a simple process or as a network of communicating processes.

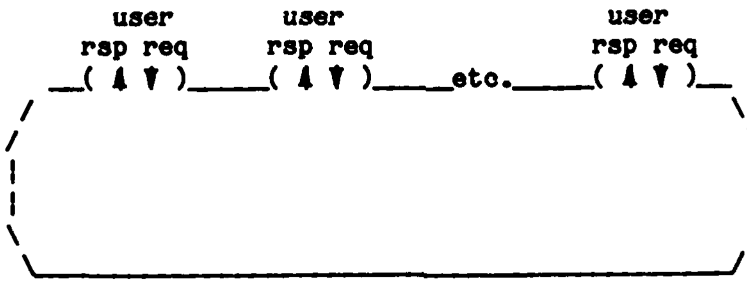


Figure 15. Graphical Represenatation of the Simple Mail System

```
1  network simple_mail;
2    port set user
        (req in user_req, rsp out user_rsp);
3    process class mailbox
4       attributes
5          number : integer
6       end attributes
7       port input in letter;
8       port output out user_rsp
9       port control in signal;
10      event mailbox_empty;
11   end mailbox

12   arrive msg : user_req on req of u : user do
13      case msg.kind of
14      make_box :
15        (find box : mailbox where
              box.number = msg.number do
16         else
17            create new_box : mailbox;
18            new_box.number := msg.number
19         end find)
20      read_mail :
21        (find box : mailbox where
              box.number = msg.number do
22            connect box.output to u.rsp;
23            send to box.control
24         else
25            construct rsp : user_rsp [kind := empty];
26            send rsp to u.rsp
27         end find)
28      send_mail :
29        (find box : mailbox where
              box.number = msg.number do
30            send msg.let to box.input
31         end find)
32   end arrive

33   when mailbox_empty announced by
          box : mailbox do
34      disconnect box.output
35   end when
36 end simple_mail
```

Figure 16. Network Specification for the Simple Mail System

A network specification which implements the simple mail system is shown in Figure 16. To match the external specification illustrated in Figure 15, this network defines a port set 'user' (line 2). The dynamic behavior of the mail system is specified in the arrive (lines 12-32) and when (lines 33-36) clauses. New mailboxes are created as requested when there are no mailboxes with the specified number (lines 15-19). Reading the contents of a numbered mailbox involves locating a mailbox instance with the correct 'number' attribute. If an acceptable mailbox instance is found, its output port is connected to the 'rsp' subport of the user port that generated the request and a signal is delivered to the 'control' port of the mailbox instance (lines 21-33). This connection is broken when the mailbox instance announces the event 'mailbox_empty' (lines 33-35). If no mailbox instance can be found, an empty response is constructed and transmitted to the response subport of the user port that generated the request (lines 25 and 26).

C.2.2.6 Event Clause Execution

To achieve decentralized execution of network specifications, the activities specified in an event handling clause will be performed--indirectly--by any process that announces the event handled by the clause. For example, any process that sends a message to the simple mail system shown in Figure 16 would perform the activities specified in the arrive clause (lines 12-32) of the network specification.

The activities specified in an event handling clause are best viewed as specifying searches and modifications of a partitioned and distributed representation of a logical communication network. This representation contains representations of all object (port and process) instances in the logical network as well as representations of the current connections between port instances in the logical network. Executions of all event handling clauses are required to be serializable.

C.3 DISSCUSSION

The language features presented reflect a concentration on inter-process relationships that describe program structure. Recall that our goals were to provide features which would support independence of processes and the description of process hierarchies, while obtaining information which would aid in the effective execution of programs. The network specifications of PRONET are, in general, more useful in support of the first goal than of the second. As PRONET has developed and the features in NETSLA have come to provide more power for describing dynamic reconfigurations, the network specifications have come to provide less useful information to an NOS. For programs which can be described by a static network, however, the features of PRONET effectively support both goals.

PRONET also includes features which provide a programmer with the ability to handle network failures. Programming for robustness in the face of such failures requires a considerable alteration of programming style, but it can be done within the framework provided by PRONET. Further discussion of these features can be found in [Macc82].

REFERENCES

[Bour78] Bourne, S. R. "The UNIX Shell," The Bell System Technical Journal Vol. 57, No. 6, (July-August 1978), 1971-1990.

[Brin78] Brinch Hansen, Per "Distributed Processes: A Concurrent Programming Concept," Communications of the ACM, Vol. 21, No. 11, (November 1978) 934-941.

[Clar80] Clark, David D. and Svobodova, Liba "Design of Distributed Systems Supporting Local Autonomy," COMPCON Spring 80, (February 1980), 729-735.

[DoD80] "Reference Manual for the Programming Language Ada--Proposed Standard Document," US Department of Defense, (July 1980).

[Ensl78] Enslow, Philip H. "What is a 'Distributed' Data Processing System?," COMPUTER, Vol. 11, No. 1, (January 1978), 13-21.

[Ensl81] Enslow, Philip H. and Saponas, Timothy G. "Distributed and Decentralized Control in Fully Distributed Processing Systems," Technical Report GIT-ICS-81/02, School of Information and Computer Science, Georgia Institute of Technology, (February 1981).

[Feld79] Feldman, Jerome A. "High Level Programming for Distributed Systems," Communications of the ACM, Vol. 22, No. 6, (June 1979), 353-368.

[Fors81] Forsdick, Harry C, et al. "Distributed Operating System Design Study: Final Report," Bolt Beranek and Newman Inc. Technical Report 4674, (May 1981).

[Hewi79] Hewitt, C. Attardi, G. and Lieberman, H. "Security and Modularity in Message Passing," Proceedings of the First International Conference on Distributed Computing Systems, (October 1979).

[Hoar78] Hoare, C.A.R. "Communicating Sequential Processes," Communications of the ACM, Vol. 21, No. 8, (August 1978), 666-677.

[Jone79] Jones, A. and Schwans, K. "TASK Forces: Distributed Software for Solving Problems of Substantial Size," Proceedings of the Fourth International Conference on Software Engineering, (September 1979), 315-330.

[Kahn77] Kahn, Gillies and MacQueen, David B. "Coroutines and Networks of Parallel Processes," Information Processing 77, (August 1977), 993-998.

[Less79] Lesser, V., Serrain, D. and Bonar, J. "PCL: A Process-Oriented Job Control Language," Proceedings of the First International Conference on Distributed Computing Systems, (October 1979), 315-329.

[Lisk79] Liskov, B. "Primitives for Distributed Computing," Proceedings of the Seventh Symposium on Operating Systems Principles, (December 1979), 33-42.

[Live80] Livesey, N.J. "Run_Time Control in a Transaction-Oriented Operating System," Ph.D. Thesis, University of Waterloo (1980).

[Macc82] Maccabe, Arthur B. "Language Features for Fully Distributed Processing Systems," Ph.D. Thesis, Georgia Institute of Technology (1982).

[Mitc79] Mitchell, James G. Maybury, William and Sweet, Richard "Mesa Language Manual -- Version 5.0," XEROX PARC CSL-79-3, (April 1979).

APPENDIX D

FAILURE HANDLING IN PRONET

Richard J. LeBlanc
Arthur B. Maccabe

D.1 INTRODUCTION

New features aiding design and description of distributed programs are central to the design of PRONET [Macc82, LeBl82]. These new capabilities are being implemented as extensions to Pascal, but since they involve only interprocess communication and interconnection of processes via message channels, they could be added to many other languages.

Among the important features of PRONET are the abstraction capabilities it provides for the specification of programs as logical networks of processes. Network specification and process description are separated in PRONET by the division of the language facilities into two sublanguages: NETSLA (Network Specification Language) and ALSTEN (an extension of Pascal for process description). These capabilities allow an encapsulated description of the connections between processes, aiding in the understanding of complex programs and providing information a distributed operating system needs for making placement and scheduling decisions.

Other programming languages/systems that support a similar separation (UNIX [Bour78], Mesa [Mitc79], Task Forces [Jone79] and PCL [Less79]) do not enforce a complete separation. In each of these languages/systems a process may specify the creation of new processes in its description. Thus, while an abstract view of the communication environment is available, neither the operating system nor a person reasoning about the program may rely on the completeness of this view. In PRONET, the conditions and activities associated with any structural modification of a communication environment (including process creation) must be stated in a network specification.

A network specification describes the initial configuration of a distributed program, in terms of processes to be created and the communication connections among them, and it describes the evolution of the network of processes in response to <u>events</u>. The event handling capabilities of network specifications are the key to providing for a centralized expression of processes interactions. Message transmissions are events which may be handled in a specification; a processes may also explicitly announce an event in order to suggest action by an event handler in its overseeing network specification. Network specifications rely on a distributed data management system to maintain information about resource availability and, hence, the activities expressed in a network specification can be performed in a decentralized fashion. The distributed data manager enforces serialization of the execution of event handlers, so network specifications need not be implemented as processes. Thus an event handler can be executed as part of the process which caused its invocation and the overall structure of a distributed program can be thought of as a tree of specifications and processes, with processes only appearing at the leaves. As a result of this structure, there is no single critical point whose failure can halt the operation of an entire program.

The failure handling features of PRONET are intended to provide a capacity for continued execution in the presence of mechanical failures and the possibility of recovering portions of a program that may have been affected by such a failure. An additional goal was that the failure handling features should only impact execution costs to the extent that they are used in a program. In order to accomplish these objectives, PRONET uses the concepts of permanent processes and stable storage. The features available support buffered communication (rather than remote procedure call) in an unreliable environment and make it possible for a programmer to ensure that the external behavior of a process is consistent with its internal state, even in the presence of failures.

D.2 DEFINITIONS OF FAILURES

The following definitions of failure, error and fault are presented by Randell, Lee and Treleaven [Rand78]:

> "When the behavior [of a system] deviates from that which is specified for it, this is called a failure. A failure is thus an event... We term an internal state of a system an erroneous state when there exist circumstances (within the specification of the use of the system) in which further processing, by the normal algorithms of the system, will lead to a failure which we do not attribute to a subsequent fault. ...The term error is used to designate that part of the state which is incorrect. ...A fault is the mechanical or algorithmic cause of an error."

Clearly, faults, and hence failures, can be encountered in any programming environment during the execution of any application.

The failure handling features of PRONET are based on a separation between algorithmic and mechanical failures and an assumed ability to detect and classify all occurrences of "failures". Considering the general definition of failure presented above, an ability to detect and classify all occurrences of failures is clearly infeasible. Hence, the failure handling features of PRONET are based on a limited view of failure. In this limited view, a mechanical failure occurs when a hardware component (processor, storage device or communication link) has failed to perform in accordance with its specified behavior. An algorithmic failure occurs when an executing process performs a primitive operation with an invalid operand (e.g., integer division with a zero-valued divisor or pointer dereference with a nil-valued pointer).

The distinction between algorithmic and mechanical failures is introduced to capture differences in the durations and causes of failures. Algorithmic failures are presumed to be permanent and a result of faulty programming. Hence, detected algorithmic failures lead directly to the termination of processes in which they occur. Mechanical failures, on the other hand, are expected to be transient and a result of a fault in the underlying programming environment (i.e., a processor crash or a communication link failure). Mechanical failures do not lead to the termination of long-lived processes but may temporarily limit their availability.

D.3 BUFFERED COMMUNICATION AND FAILURES

In a perfect programming environment, the send operation of buffered communication might be viewed as passing the responsibility for processing messages to receiving processes. In this way, processes that declare input ports accept the responsibility for correctly processing all messages that are sent to these ports. Processes that send messages can rely on the specified behavior of receiving processes to ensure that their messages are handled correctly and completely.

Clearly this interpretation of buffered communication is inappropriate when processes can encounter failures during their executions. The initial extensions developed for using CLU [Lisk77] in distributed programming environments [Lisk79] were based on buffered communication primitives. More recently [Lisk80], a remote procedure call (RPC) primitive has been adopted:

> "RPC is a very high level primitive... For some time we were hopeful that there might be an intermediate level primitive that would solve many of the user's problems, and would not be as expensive as RPC. Our experience indicates that there is no such primitive, we have looked for one and have not found it."

Much of the rational for selecting RPC which is presented in [Lisk80, Lisk82] is based on an inability to resolve the semantics of intermediate level primitives (e.g., buffered communication) with potential failures and the intended area of application of the language features being developed.

PRONET is based on an interpretation of buffered communication which is sufficiently powerful to aid programmers in their task of describing interprocess communication, yet weak enough to allow for the possibility of failure. The send operation of PRONET completes successfully when the message being sent has been correctly copied into the address space of all receiving processes and all events which are generated by the send have been handled. Further, the send operation is atomic with respect to failures--either all events associated with the message transmission are handled completely or none of these events is handled (and a failure indication is returned). Hence, after successfully completing a send operation, the sending process can assume that receiving processes will handle the message in an appropriate fashion. Receiving processes, on the other hand, accept the responsibility (in conjunction with the network specification that oversees their operation) for handling all messages that are available on their input ports.

The crucial distinction between this interpretation and the interpretation presented earlier, involves the substitution of the words "handling" and "appropriate" for the words "processing" and "correct" respectively. In some applications, under certain circumstances, appropriate handling of a message may involve ignoring the message entirely. Because of this necessarily weak interpretation of buffered communication, sender processes that need to know how their messages were (or will be) handled will need an alternative means of obtaining this information. For most applications, a simple response port will suffice. Clearly this complicates the description of such processes but processes that do not require this information will not incur additional costs.

D.4 FAILURE HANDLING

An important motivation for introducing failure handling facilities into the design of PRONET was based on the need to describe long-lived objects. PRONET does not provide an inherent distinction between long-lived and transient objects--all objects are processes. However, it is necessary to distinguish between the processes in a logical network that are capable of surviving mechanical failures and those whose activities are aborted when they are in the scope of a mechanical failure. The activity permanent can be applied to processes and provides a capacity to survive mechanical failures.

"Permanence" is an inherited property. When a "non-permanent" network applies the permanent activity to one of its processes, this activity has no immediate affect. Whenever a logical network becomes "permanent", all processes in the network to which the permanent activity has been applied will also become permanent.

If any process in a "non-permanent" network encounters a failure (algorithmic or mechanical) during its execution, the entire network fails and all processes in the network are terminated. In this way, failures encountered by processes are propagated to their overseeing network. Propagation of a failure continues until a "permanent" network (or process in the case of mechanical failures) is encountered. Failures encountered by processes executing in a "permanent" network do not directly affect other processes executing in this network.

'Permanent' processes can be explicitly terminated or removed (by their overseeing network specification) and can express their own termination but will be recovered (as described earlier) if they have not terminated and are in the scope of a mechanical failure.

Because mechanical failures can alter the internal state of any executing process, processes in the scope of a mechanical failure cannot rely on information stored in their internal state after a mechanical failure has occurred. A stable storage facility has been integrated into ALSTEN to enable the description of processes that must rely on portions of their internal state when mechanical failures are recovered. Like the facility proposed in [Lisk79], process descriptions interface to stable storage by declaring stable variables and periodically "checkpointing" the values of these variables.

When a mechanical failure is detected, all processes in the scope of the failure are halted before they can begin a new checkpoint operation. When the mechanical failure is recovered, each permanent process halted by the failure is restored using values saved by checkpointing and non-permanent processes are removed from the logical network.

The features presented thus far are useful for describing long-lived processes but do not enable receiving processes to ensure that all messages which have been successfully delivered to their IPC space are handled in an appropriate fashion when a mechanical failure occurs. An important problem is that messages will be inserted into the IPC space of a process asynchronously and hence, a process cannot use inline checkpointing operations to ensure that all messages which have been delivered will survive mechanical failures. For this reason, ALSTEN provides stable ports. Any input port declared by a process may be declared with the attribute stable. All messages which have been successfully delivered to a stable input port and not removed by the receiving process during its execution will be available after a mechanical failure. Messages are only removed from a stable input port when the process performs a checkpoint operation.

D.5 PERMANENCE AND EXTERNALLY VISIBLE BEHAVIOR

The use of checkpoints, stable variables and recovery descriptions are sufficient to describe a consistent recovery from mechanical failures, but do not enable the programmer to ensure that the recovered state is consistent with the externally visible behavior of the process. In [Lisk80] it is argued that many applications will need a capacity to incorporate 'permanence of effect' in their communications. Using buffered communication, this property would allow receiving processes to rely on the information contained in the messages they receive. Hence, ALSTEN provides a checkpointing send operation which combines both operations into a single operation and is atomic with respect to mechanical failures.

D.6 PARTITIONING FAILURES

The failure handling features described thus far are primarily aimed at handling point failures (the failure of a single process). A reasonable implementation of PRONET would be based on a partitioned and decentralized network representation. As such, mechanical failures could cause portions of the network representation to be unavailable for use. Thus portions of the network representation may not be visible during the execution of an event handling clause.

Modifications performed by an event clause execution may implicitly affect objects in inaccessible portions of the logical network representation, even though the objects explicitly modified by the event clause were available for use. Consider that port 'p1' is connected to port 'p2' and that p2 is available but that p1 is inaccessible. In this situation the activity "disconnect p2" can be performed but will affect p1, as p1 must see the disconnection.

When a mechanical failure that has caused a partitioning failure is recovered, portions of the logical network representation will need to be updated (merged) to reflect modifications performed in other partitions. In order to perform this merging of visibility partitions, redundant information must be stored in the logical network representation. In general this redundant information will be stored in the form of back-pointers which can also be used for efficient traversal of the logical network representation.

D.7 SUMMARY

The important concepts developed in PRONET are based on the separation of connectivity specifications from process descriptions. This separation allows process descriptions to be independent of one another, since they can only describe interactions with the other components of a program through messages sent to locally declared ports and by announcing events. Thus a programmer concentrates on the logical structure of a program and need not be concerned with such things as physical distribution considerations. The hierarchical structure of a PRONET program, consisting of processes and a tree of overseeing network specifications, is particularly well-suited as a description of a distributed program. Important features of PRONET allow continued execution of unaffected parts of a program in the presence of failure and recovery of failed processes through use of checkpointing and stable ports. Finally, PRONET includes an intermediate level communication approach, buffered communication, which operates meaningfully in the presence of failures. It will thus allow the exploration of the appropriateness of communication protocols other than remote procedure call for the implementation of realistic distributed programs.

An initial inplementation of PRONET on a single processor has been completed. Current plans are for a more complete implementation to be developed to run on a network of Perq workstations. As part of the Clouds operating system project [Mcke82] a real-time distributed data management system is being designed [Allc82] which should greatly simplify the implementation of PRONET and improve its performance.

REFERENCES

[Allc82] Allchin, J.E. "Integral Data Management in Distributed Real-Time Systems," Ph.D. Thesis Proposal, School of Information and Computer Science, Georgia Institute of Technology (August 1982).

[Bour78] Bourne, S. R. "The UNIX Shell," The Bell System Technical Journal Vol. 57, No. 6, (July-August 1978), 1971-1990.

[Jone79] Jones, A. and Schwans, K. "TASK Forces: Distributed Software for Solving Problems of Substantial Size," Proceedings of the Fourth International Conference on Software Engineering, (September 1979), 315-330.

[LeBl82] LeBlanc, R.J. and Maccabe, A.B., "The Design of a Programming Language Based on Connectivity Networks," to appear in Proceedings of the 3rd International Conference on Distributed Computing Systems (October 1982).

[Less79] Lesser, V., Serrain, D. and Bonar, J. "PCL: A Process-Oriented Job Control Language," Proceedings of the First International Conference on Distributed Computing Systems, (October 1979), 315-329.

[Lisk82] Liskov B. and Scheifler, R. "Gurdians and Actions: Linguistic Support for Robust Distributed Programs," ACM Symposium on Principles of Programming Languages (January 1982), 7-19.

[Lisk80] Liskov, Barbara "Linguistic Support for Distributed Programs: A Status Report," Laboratory for Computer Science, Group Memo 201, Computation Structures, Massachusetts Institute of Technology (1980).

[Lisk79] Liskov, B. "Primitives for Distributed Computing," Proceedings of the Seventh Symposium on Operating Systems Principles, (December 1979), 33-42.

[Lisk77] Liskov, Barbara, Snyder, Alan, Atkinson, Russell and Schaffert, Craig "Abstraction Mechanisms in CLU," Communications of the ACM, Vol. 20, No. 8, (August 1977), 564-576.

[Macc82] Maccabe, Arthur B., "Language Features for Fully Distributed Processing Systems," Ph.D. Thesis, Georgia Institute of Technology (1982).

[McKe82] McKendry, M.S., Allchin, J.E. and Thibault, W.C. "Clouds: A Testbed for Experimentation in Distributed Systems," Working Paper 3, Status Report, School of Information and Computer Science, Georgia Institute of Technology, (June 1982).

[Mitc79] Mitchell James G. Maybury, William and Sweet, Richard "Mesa Language Manual -- Version 5.0," XEROX PARC CSL-79-3, (April 1979).

[Rand78] Randell, B., Lee, P.A. and Treleaven, P.C. "Reliability Issues in Computing System Design," Computing Surveys, Vol. 10, No. 2, (June 1978), 123-166.

APPENDIX E

SOFTWARE FAULT TOLERANCE: OVERVIEW OF THE RECOVERY BLOCK SCHEME

Tom Wilkes

E.1 INTRODUCTION

Ever since the first computing systems were designed and built, the problem of the reliability of these systems in the face of faults and errors has been a concern of designers and researchers. However, until approximately the last decade, most work on system reliability has been focused on the area of hardware reliability, even though any non-trivial software system is more complex by several orders of magnitude than the machine on which it runs. As Randell notes ([Rand75]), a simulator for a certain machine written at the level of detail required by the hardware designers is in general many times smaller than the operating system for that machine. Since the number of possible internal states of any but the most trivial software far outnumbers the number of possible states of the hardware on which it runs, the possibility of design error in the software is correspondingly greater. Hence, the need for methods of recovery from design flaws in software is at least as pressing as that for hardware.

Also in [Rand75], Randell states:

> "If all design inadequacies could be avoided or removed this would suffice to achieve software reliability... Indeed many writers equate the terms "software reliability" and "program correctness". However, until reliable correctness proofs (relative to some correct and adequately detailed specification), which cover even implementation details, can be given for systems of a realistic size, the only alternative means of increasing software reliability is to incorporate provisions for software fault tolerance."

As Svobodova has noted ([Svob79]), distributed systems have an even greater potential for providing reliability than their non-distributed counterparts:

> "Distributed systems are often claimed to be inherently more reliable than systems based on a large central processor. That is, given that a distributed system is properly designed, it offers better reliability. First, distributed systems by their very nature provide opportunities for redundancy. Second, error propagation is restricted by physical separation of processes and resources. And finally, individual nodes in the distributed system may be less complex than a large central processor and, as

> a result, ought to have lower probability of failures. Basically, distributed systems have a potential for being more reliable than systems based on a large central processor. However, this potential needs to be exploited through proper design."

PRONET, a language for distibuted processing applications which has been under development at Georgia Tech ([Macc82]) incorporates extensive facilities for dealing with the problem of hardware failures. However, the work to date on the design of PRONET does not treat the problem of software (algorithmic) failures. Algorithmic failures present a much more difficult problem than hardware failures. Because such failures presumably result from a logical fault in the program, use of checkpointing and restarting will only result in a reproduction of the failure. (In the case where a hardware failure corrupted data and thus caused the algorithmic failure, such techniques may provide a means of recovery.) Thus some capability to execute alternative code is required, as well as some capability to undo the effects of the code which has failed. The addition of these capabilities to a distributed system will increase the complexity of programming in the system, since processes may interact in the recovery mode and during the "undo" process, as well as during their normal execution. As Shrivastava and Banatre have noted ([Shri78]),

> "...appropriate programming language tools must be provided to cope with this additional complexity in a systematic manner, otherwise resulting programs are likely to be even less reliable than versions with no redundancy."

It is in support of the design of such tools that the present survey is being undertaken.

In an excellent review art:cle ([Rand78]), Randell, Lee, and Treleaven have surveyed the issues of hardware and software reliability, and have catalogued current techniques for error recovery and fault tolerance. A repetition of their work will not be attempted here. Rather, the results of their survey will be summarized, and two important techniques for software fault tolerance -- the so-called forward and backward error recovery methods -- will be briefly contrasted. However, the bulk of the discussion will center on a particular backward error recovery scheme, the recovery block method, which is discussed in [Rand75], [Rand78], [Ande81], and many other publications which have issued from the software fault tolerance project at the University of Newcastle upon Tyne (most of which will be discussed below). Recent publications which consider the application of these recovery techniques to distributed computing systems will also be discussed.

E.2 SOME TERMINOLOGY

In [Rand78], definitions for many terms used in the discussion of software fault tolerance have been provided which have been adopted by subsequent papers from the project at Newcastle upon Tyne and also by several other authors in the field. For convenience, some of these definitions are reproduced here:

> "The **reliability** of a system is taken to be a measure of the success with which the system conforms to some authoritative specification of its behavior...
> When the behavior of a system deviates from that which is specified for it, this is called a **failure**...
> We term an internal state of a system an **erroneous state** when the state is such that there exist circumstances (within the specification of the use of the system) in which further processing, by the normal algorithms of the system, will lead to a failure which we do not attribute to a subsequent fault... The term "error" is used to designate that part of the state which is "incorrect"...
> A **fault** is the mechanical or algorithmic cause of an error, while a **potential fault** is a mechanical or algorithmic construction within a system such that (under some circumstances within the specification of the system) the construction will cause the system to assume an erroneous state..."

Note that, using the definitions of "fault" and "error" given above, the method of repair of faults and errors in a system is very different. In particular, the repair of a fault in a software component is a complex task which would be very difficult to automate, and which should be accomplished by manual means in an unharried manner. Repair of an error, on the other hand, entails the change of the erroneous state into one in which processing may continue correctly (within the specification of the system), or the restoration of a previously-existing state which satisfies these specifications. We shall see that this process is automatable. Thus, repair of an error is required for continued operation of a system, whereas the repair of the fault which caused the error is not always necessary for ensuring continued operation.

E.3 METHODS FOR SOFTWARE FAULT TOLERANCE

As has been mentioned above, [Rand78] provides a comprehensive survey of techniques for hardware and software fault tolerance. The authors consider strategies for error detection, fault treatment, damage assessment, and error recovery as comprising a classification of fault-tolerance techniques. These strategies are by no means mutually exclusive, as we shall see.

E.3.1 Error Detection

As defined in [Rand78],

> "the purpose of error detection is to enable system failures to be prevented by recognizing when they may be about to occur."

In order to fulfill this purpose in the ideal case, however, the checks which would have to be made would have to be based solely on the system specification, and independent of the actual implementation to a degree probably not realizable in practice. Also, the extent of error checking necessary would probably fall victim to performance considerations. Thus, the complete confidence afforded by the ideal case is generally not attainable, and some "very high" level of confidence is all that can be expected. However, all strategies for fault tolerance depend on error checking for their invocation.

E.3.2 Fault Treatment

Error detection seeks only to identify the symptoms of a fault, but does not try to identify the particular fault which caused the error. The identification, location, and removal of a fault is a complex job, since many errors may be caused by a particula~ fault, a particular error may be caused by several different faults, the error caused by a particular fault only occur for certain input values, etc. Thus, the automation of the task of fault removal in software is not feasible except in very simple cases. However, the treatment of faults by alternative means, such as **replacement strategies**, is more tractable; indeed, the recovery-block scheme which is discussed below is such a strategy.

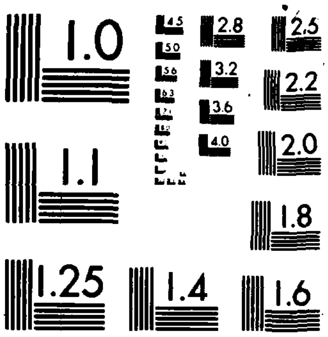
E.3.3 Damage Assessment

As noted in [Rand78],

> "Damage assessment can be based entirely in *a priori* reasoning, or can involve the system itself in activity intended to determine the extent of the damage. Each approach can involve reliance on the system structure to determine what the system might have done, and hence possibly have done wrongly. The approach can be explained, and might have been designed, by making explicit use of atomic actions."

The intent here is that atomic actions provide a "sequence of delimitations ... of amounts of possible damage corresponding to each different error detection point." Since, as the authors note, damage assessment is often necessary to attempts at error recovery, and "is usually a rather uncertain and incomplete affair", it is worthwhile to expend the effort involved in limiting the spread of damage by such means.

AD-A141 501   SOFTWARE SUPPORT FOR FULLY DISTRIBUTED/LOOSELY COUPLED   2/3
PROCESSING SYSTEMS..(U) GEORGIA INST OF TECH ATLANTA
SCHOOL OF INFORMATION AND COMPUT.. P H ENSLOW ET AL.
UNCLASSIFIED   JAN 84 GIT-ICS-82/16-VOL-2   F/G 9/2   NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

E.3.4 Error Recovery

Methods for **error recovery** are divided intc the so-callec forward and backward automatic recovery schemes. Forward recovery schemes attempt to make further use of the erroneous state. Thus, predictions about the location and consequences of software faults are necessary. Such a scheme must therefore be designed as an integral part of the system for which it is to provide fault tolerance. Also, the questions of damage assessment and fault treatment are intermingled with the question of how to continue to provide service. Despite the complexity which such a scheme adds to the system it is to serve, there are situations in which valid assumptions can be made based on knowledge of the system for which forward-recovery techniques provide simple and effective error recovery. In particular, these methods are very effective in dealing with such situations as errors caused by invalid input data. The exception-handling methods used in languages such as PL/I and Ada are examples of forward-recovery methods.

Backward-recovery schemes, on the other hand, involve restoration of what is hoped to be an error-free state, and thus require no predictions of the location or nature of faults. Rather, backward recovery is analogous to mechanical backups in hardware systems. Information about the system state previous to the fault is restored from a checkpoint, and a back-up process is started. The back-up process is necessarily not the same as the failed process, as it would presumably only fail again. In general, the back-up process (or processes) is more simple than the original process, and may provide only a primitive simulation of the functions of the original process (such as forwarding messages) in order to keep a program running.

The recovery-block scheme, an example of a backwards-recovery scheme which has been the object of detailed investigation at Newcastle upon Tyne, is described in the next section.

E.4 THE RECOVERY-BLOCK SCHEME

The recovery-block scheme described by researchers at the University of Newcastle upon Tyne ([Rand75], [Rand78], [Ande81]) is an example of a backwards-recovery method. This method is a means of providing "gracefully degrading software" ([Ande81]). The syntax for describing a recovery block is:

```
assure <acceptance test> by
   <original block>
else by
   <back-up block 1>
else by
   ...
else error;
```

where some of the "back-up blocks" may be simple retries of previous blocks. If a failure occurs in the original block, back-up blocks are tried until one completes without failure and the acceptance test is satisfied, or else an error is signalled. The back-up blocks may have to undo permanent effects made by their predecessors before doing their own work.

E.4.1 Acceptance Tests

The function of the acceptance test is to ensure that the operation performed on the system state by at least one of the alternate blocks is to the satisfaction of the invoking program. Thus, an acceptance test need not be a check on the "absolute correctness of an operation" ([Rand75]). In general, a test is based on the present and prior values of variables global to the alternate blocks and to the invoking procedure. Also, some means is provided for checking whether global variables not accessed within acceptance test have been modified, thus giving a measure of security against unforeseen side effects.

It is clear that the careful design of acceptance tests is important to the success of the recovery-block method. However, strict requirements for correctness must often yield to performance considerations, as in the following example from [Rand75]:

```
ensure sorted (S) and (sum (S) = sum (prior S))
by quickersort (S)
else by quicksort (S)
else by bubblesort (S)
else error;
```

Here, the strict requirement that the sorting algorithm yield a permutation of its input values has been relaxed to a requirement that the sum of the input values and the sum of the output values be the same.

E.4.2 The Recovery Cache

Before a back-up block may be tried, the state of global objects must be restored to that existing before the failed block began execution. This state restoration is made possible by the use of a **recovery cache**, in which the values of global variables are stored prior to their first updates in the current block. A recovery cache is essentially a differential file, and is thus less costly in space than a full checkpoint. Since recovery blocks may be nested, the cache is organized as a stack; state restoration for the current recovery block requires restoration of global variable values from the current top stack entry, and upon completion of a block, the stack entry is discarded, thus "committing" the results of the block.

Thus, the problems of state restoration and recovery for simple global variables are relatively straightforward. However, in general, the actions of interacting processes may be more complicated than simple assignment to a global variable; there may be, for instance, competition for global resources (say, peripherals) or cooperative use of resources for inter-process communication (say, a shared message buffer). As has been noted in [Shri78], for arbitrary interaction of processes, the problem of the management of recovery information and the control of processes may become extremely complex. However, they show that it is possible to break these interactions down into different classes - interference, cooperation, and competition - and to develop mechanisms to treat the recovery problems posed by these different types of interactions separately.

E.4.3 Error Recovery in Cooperating and Competing Processes

The bulk of [Shri78] is devoted to the consideration of the problem of competing resources. This problem is simpler than that of cooperating resources for the following reasons: while cooperating processes can exchange arbitrary information (for instance, via a message buffer), competing processes typically exchange only that information required to ensure proper synchronization and sharing of resources. Thus, for competing processes, the type of information exchanged is known to (and generally controlled by) the synchronization mechanism.

However, since the information exchanged by cooperating processes may be arbitrary, in general only the recipient of the information may verify it. In [Shri78], the example of a producer and a consumer connected by a bounded message buffer is considered. For verification reasons, "production" and "consumption" of a message are programmed as a **conversation**, and when the producer and consumer processes enter a conversation, they are allowed to leave it only when both pass their acceptance tests. This prevents the producer from "racing ahead" of the consumer and thus can seriously limit the amount of concurrency possible. Whether the conversation mechanism may be supplemented by some other mechanism to ameliorate this problem is currently under investigation at Newcastle upon Tyne.

E.4.4 The Domino Effect

Another problem in the application of the recovery block scheme to cooperating processes is the so-called **domino effect** ([Rand78], [Ande81]). This effect arises from attempts by the individual communicating processes to achieve backward recovery. If two processes independently establish recovery points or checkpoints, and communication between them may occur at arbitrary times, then we may have the scenario represented in the following diagram ([Ande81]):

```
Process 1:  ------[-----------[-----------[-----------[-------->
               |      |     |     |     |     |     |     |
               |      |     |     |     |     |     |     |
Process 2:  ------------[-----------[-----------[-----------[->
```

Here, the vertical lines represent occurrences of communication between the two processes, and the square brackets indicate an active recovery point to which the state of a process may be restored. If process 1 experiences a failure after its most recent recovery point, it may try to restore its state at that point. Since it has not communicated with process 2 since that point, process 2 need take no recovery action. If, however, process 2 encounters a failure after its last communication with process 1, process 2 must restore its state to its most recent recovery point, which occurred before its last communication. Thus process 1 must be restored to a point at or before process 2's recovery point, since the state of process 1 was changed by the information exchange which took place after that point. However, the most recent recovery point to which process 1 can restore occurred before this exchange, which will similarly cause another rollback in the state of process 2, etc. Thus, an uncontrolled propagation of rollbacks in process states may occur, much like a line of toppling dominos. This effect does not occur for independent, competing processes, since no such information flow occurs between them.

Thus, a basic problem in recovery is the search for a "consistent" set of recovery points, that is, for a set of checkpoints for which the domino effect does not occur. A consistent or usable set of recovery points is called a **recovery line** ([Rand78]).

As Randell has shown ([Rand75]), in order to obtain a consistent set of recovery points for a group of freely-interacting processes it is required that the pattern of interactions among the processes be known in advance. As this is a rather unrealistic requirement, we must consider two alternatives ([Verh78]): (1) prevent the interactions, as in implicit or explicit locking, or (2) synchronize the processes with respect to recovery. We shall see that it is the latter route which has been chosen by the group at Newcastle upon Tyne.

Recent work on recovery lines reported in [Ande81] has led to the following definition of a **restorable action:**

> "An atomic action is said to form a **restorable** action if: (i) on entry to the atomic action all processes establish a recovery point, (ii) these recovery points are not discarded within the atomic actions, and (iii) processes leave the atomic action simultaneously." ([Ande81])

Within a restorable action, backward recovery may be accomplished by restoring the recovery points established for each process upon entry to the action if an exception is raised by any of the processes involved in the action. This protocol can be seen to be equivalent to the conversation protocol described above.

Work on extending the recovery-block method to cooperating processes is described in [Ande81]. In particular, strategies for avoiding the domino effect are discussed. As has been mentioned above, requiring communicating processes to enter into **conversations** from which all processes involved must exit together (thus committing the results of the conversation) will avoid the domino effect at the cost of lost concurrency, much as for the requirement of two-phase locking for synchronizing processes. Indeed, the conversation mechanism is seen to fulfill the requirements of a "recoverable action" as defined above. In work by Russell ([Russ80]), certain protocols for ordering message sending and receiving have been developed for which it can be shown that uncontrolled rollback cannot occur.

E.4.5 Recoverable Monitors

While investigating the simpler problem of competing processes, however, Shrivastava and Banatre have developed language features to support recovery which may have more general interest. They introduce the idea of a **recoverable monitor**, in which access to resources is controlled by a feature called a **port**, which is similar to the class and "inner" constructs of SIMULA or Concurrent Pascal. Assuming a Concurrent Pascal-like language, the syntax of a port construct may be summarized as follows ([Shri78], [Ande81]):

```
[entry] type <name> = port (formal parameters)
      "entry is an optional feature"
begin ...local variable declarations...
   ...procedures/forward entry procedures, e.g.: ...
   forward entry procedure <name> (formal parameters);
      begin ... end;
   ...other procedures/forward entry procedures...
   [backward entry procedure <name>;
      "this procedure is optional"
      begin ... end
   s1; inner; s2
      "s1 and s2 are statements, where s1 is the prelude
       and s2 is the postlude"
end "of port definition"
```

The organization of the port construct reflects (and enforces) a resource-access protocol considered in [Shri78]. There, the types of recovery actions necessary when failure occurs at various points in the protocol are developed. In particular, the protocol requires that only the prelude and postlude of the port may acquire and release resources, respectively. Also, the backward entry feature allows specification of an "undo" block, whose purpose is to undo the effects of the execution of the forward entry blocks, which is necessary for state restoration of arbitrary global objects.

If failure occurs during the prelude of a port (s1), this means that all alternatives of the resource-acquisition block have failed, and thus the port fails. If failure occurs after acquisition and before use of a port (between s1 and inner), then to restore the state of the (abstract) port, the only action required is the release of the acquired resource; thus, the postlude (s2) must be executed. The use of a resource (inner) is considered to be an atomic call on a recovery block, and if failure is signalled, then only execution of the postlude is required. If failure is detected after the use but before the release of the resource, then the backwards procedure must be executed to undo the effects of the user procedure, and then the postlude must be executed. A failure during the postlude (s2) means that it was not possible to release the acquired resource -- an unrecoverable error.

E.4.6 Effects on Software Complexity

As has been noted above, forward-recovery methods must be designed as an integral part of the software which they are to serve. Thus, they may add significantly to the complexity of this software. In contrast, the recovery-block scheme provides a means of explicitly separating the error-detection and recovery functions from the rest of the software, and thus should add little to the conceptual complexity of a module. In addition, it is possible (and indeed desirable) that the design of any back-up blocks provided proceed independently of the primary block and of each other. This independence of the alternative blocks may produce a significant reduction in the complexity of software employing the recovery-block method as compared to software using ad hoc error detection and recovery methods ([Ande81]). Also, the requirement of acceptance tests is more rigidly enforced than the use of assertions in some systems, thus providing an enforced verification method.

E.4.7 Problems in Implementation for Distributed Systems

Several problems crop up in the implementation of backwards-recovery schemes for loosely-coupled distributed systems under decentralized control which are not apparent in implementation for non-distributed systems ([Ande81]). Cooperating processes in such systems must exchange control information in addition to exchanging data in order to coordinate the recovery process in the absence of a central coordinator. In an unsafe message-passing system, there may be significant delay between the sending and reception of these control messages, or they may become corrupted or lost. This adds greatly to the complexity of the recovery problem.

If a distributed recovery system relies on planned recovery lines, there is a need for coordination of the exits of processes from restorable actions in order to insure the existence of these recovery lines. This necessitates the existence of a central coordinator, such as that in System R (see below), which governs a two-phase commit protocol not unlike the conversation mechanism discussed above.

A system may instead search for unplanned recovery lines. Such a system is studied in the occurrence graph scheme ([Merl78]). An occurrence graph is a historical record of the dependencies between communicating processes due to the information flow between them. Such a record is kept by each process in the system. Should a process need to restore a recovery point, it must send a FAIL message to those dependent processes as given by the occurrence graph in order to maintain the consistency of the system state. Each process which receives a FAIL message must cease its normal activity and also send out FAIL messages to all of its dependents. The assumption must be made that the FAIL messages propagate faster than normal messages (and that none of them are lost). In this way a recovery line may be eventually identified. This scheme is known as the **chase protocol.** Unfortunately, recent investigations have shown that this method is highly prone to the domino effect ([Ande81]).

E.5 OTHER BACKWARDS-RECOVERY SCHEMES

Several recovery schemes which bear similarities to the recovery block scheme have been discussed in the literature (see [Ande81]). System R, an experimental data-base system ([Kohl81], [Gray81]), employs a "DO-UNDO-REDO" system for treatment of hardware failures via maintenance of an incremental log with write-ahead. A centralized "coordinator" controls a two-phase commit protocol, and independence of actions is required to avoid the domino effect. The REDO of an action is effective only for idempotent actions (i.e., those for which multiple executions are valid), and the System R scheme is thus less powerful than the alternative-block strategy of the recovery-block method. Another similar method is the **deadline mechanism** for real-time systems, where the acceptance test of the recovery-block scheme is replaced by a time-out test. Yet another fault-tolerance method is the so-called **N-version scheme,** in which the results of applying several different algorithms to the solution of a problem are compared for agreement.

E.6 UNIFIED VIEW OF PROGRAMMED AND AUTOMATIC EXCEPTION HANDLING

In a recent paper from Newcastle upon Tyne ([Cris82]), Cristian initiates the development of a formal view of the concepts underlying software fault tolerance in order to elucidate the unity between programmed exception handling and default exception handling using automatic backwards recovery. Also, his formal development demonstrates the existence of a class of design faults which cannot be treated using automatic methods such as the recovery-block method.

Cristian bases his model on a view of programs as a hierarchy of modules, assuming that this structure is the result of the application of data abstraction techniques to program development. Thus, a user would view a module M as an abstract variable of some abstract data type, that is, a set of abstract states and transitions between these states (produced by the operations exported by M). The internal structure of M (not visible to the user) is a set of state variables and procedures which operate on these variables.

The **internal state** of the module M is defined as the aggregation of the abstract states of the state variables of M. The **abstract state** of M is the result of applying an abstraction function A to the internal state of the module M. Note that this definition is recursive; the state variables of M may themselves be the abstract states of lower level modules. Presumably, however, the recursion bottoms out in the lowest level modules, where the state variables are actual data structures.

The abstract state of a module is in general a partial function defined only over some set of internal states which satisfy an invariant predicate I. The states which satisfy this predicate are said to be **consistent** with the abstraction which is supposed to be implemented by the module. However, a module may during execution pass through states which do not satisfy this invariant predicate, and thus for which the abstract state is not defined.

The **intended service** of a procedure P exported by the module M is specified by a relation **post** over pairs of initial and final states (s',s) of the state transition accomplished by the procedure. A pair of states (s',s) is said to be in **post** if the final state s is the intended outcome of invoking the procedure P in the initial state s'. The characteristic predicate associated with the relation **post** is called the **standard postcondition** of P.

The **standard domain** (SD) of a procedure P is defined as that set of initial states s' for which execution of P terminates normally in states s such that **post** (s',s) holds. If P is invoked in an initial state s' outside its standard domain SD, an **exception occurs.** Such states s' belong to the **exceptional domain** (ED) of P, that is, the set of states which do not belong to the SD of P.

To illustrate these concepts, Cristian presents the following short example. Let intended service for some procedure P exported by a module M be specified by **post** == $i = i' + j'$, where i' and j' denote the initial values of state variables i and j of M, which are of type positive integer. If the body of the procedure P is

$$i := i + j$$

and PI is the set of machine-representable positive integers, then the standard domain SD of P is i' + j' in PI, and the exceptional domain ED of P is i' + j' not in PI. Had the programmer by mistake typed "*" instead of "+" in the body of P, then the SD and ED of P would be

SD == (i' = j') and (i' = 0 or i' = 2)   ED = ~SD

A programmer, in the design of a procedure, may anticipate that the procedure may be invoked in initial states outside its standard domain, i.e. in its exceptional domain. The programmer may detect such **anticipated exception occurrences** by such means as run-time checks. However, in general such checks may be redundant, since the condition which the check is supposed to detect may be detected by the hardware before the check can be executed. Also, in general it does not make sense to continue normal execution of a program after such a condition becomes apparent. Thus some languages contain features allowing the programmer to express actions to be undertaken in place of normal execution upon an exception occurrence. Such features are termed **exception mechanisms.**

As an example of the explicit programming of handlers for exception occurrences, Cristian gives the following:

```
proc P signals OW;
begin
   i := i + j  [OV -> signal OW];
   i := i + k  [OV -> i := i - j; signal OW;
end;
```

Here, the first line of the example expresses the existence of two exit points from the procedure P: the normal exit, and another exit on occurrence of the exception OW. On the third line, if the addition causes an overflow exception (OV), a handler which merely signals the exception OW to the invoking procedure is executed. Note that, if the addition causes overflow, the assignment is not executed, and thus the initial state remains unchanged. Similarly, on line four an OV exception will cause execution of a handler which undoes the effect of the preceding line (by subtracting the value which was added there), and then signals OW. This has the effect of restoring the initial state.

The standard postcondition for this procedure may be expressed by **post** == $i = i' + j' + k'$. However, if the exception OW is signalled by P (i.e., if it uses its exceptional exit), then its **exceptional postcondition**

$$\textbf{post}\ (OW) == (i = i')\ \text{and}\ (j = j')\ \text{and}\ (k = k')$$

is satisfied. In general, if E is an exception signalled by a procedure P, then **post** (E) specifies the intended state transition when P signals E.

The procedure P given above is said to be **total**, since its behavior i specified (by means of its standard and exceptional postconditions) for a initial states, both in its standard and exceptional domains. Also, i exceptional postcondition is of the form $A(s') = A(s)$, that is, the abstr state upon exceptional exit is the same as upon invocation. A total operati for which **post** (E) has such a form is called an **atomic operation.**

Here, Cristian notes that since exception detections may signal attempts to violate invariants which are maintained by communicating processes, the notions of atomicity with respect to exceptions (**recovery atomicity**) and atomicity with respect to synchronization (**concurrency atomicity**) become interrelated.

As has been noted above, when an exception occurrence is detected, it is possible that an **inconsistent state** exists, that is, one for which the invariant I of the module M is not satisfied. Since further use of an inconsistent state can lead to unpredictable results, it is necessary to recover some consistent state. The set of state variables of the module M for which a consistent final state s may be reached by modifying the state these variables have in the inconsistent state i, and for which the final state s satisfies the relation

$$I\ (s)\ \text{and}\ \textbf{post}\ (E)\ (s',s)$$

is called a **recovery set** (RS). Further, an **inconsistency set** (IS) is a recovery set for which, for any other recovery set RS, $|IS| <= |RS|$ (where the vertical bars indicate set cardinality). Thus, IS is just the smallest of the (in general) several possible recovery sets.

When atomicity with respect to exceptions is desirable, there are some other recovery sets of interest. The **inconsistency closure** (IC) associated with the inconsistent state i is defined as the set of all state variables modified between entry into the procedure P and the detection of an exception

E during the execution of P. Note that IC is trivially a recovery set, since the final state s is identical to the initial state s' upon restoration of the initial states of all the modified variables, and

$$I\ (s) \text{ and } (A\ (s') = A\ (s))$$

is thus satisfied. A crude approximation to the IC is obtained by storing the whole set of the initial states of the state variables of M, obtaining a complete **checkpoint**.

If atomic behavior is not necessary for a procedure P, then **forward recovery** may be used, as discussed above. Then the recovery actions are (necessarily) based upon the designer's knowledge of the semantics of P. If, however, it is desirable that P behave atomically with respect to exceptions, then the use of IC sets or checkpoints to restore a consistent state is necessary. As has been noted above, this method is called **backward recovery**. As we have seen, it is possible that the IC or checkpoint may be determined automatically, yielding **automatic backward recovery**, in contrast to explicitly programmed backward recovery.

As defined above, a necessary condition for the atomicity of an operation is that the operation be total. *However, in practice* the design of total operations is difficult. Thus, in most cases the designer of an operation anticipates only some subset of the exceptional occurrences possible in that operation. The true standard and exceptional domains of the operation may therefore be other than those which the designer imagines. The portion of the exceptional domain for which the designer provides a specified exceptional exit point is called the **anticipated exceptional domain** (AED). That portion of the ED not included in the AED is called the **unanticipated exceptional domain** (UED). The operation may terminate normally when invoked in its standard domain, in a state satisfying **post** (E) when invoked its anticipated exceptional domain, and in an undetermined state when invoked in its unanticipated exceptional domain.

To illustrate these concepts, Cristian rewrites the example given above as follows, where the intended and exceptional services were specified by the relations

post $== i = i' + j'$ **post** $(OW) == (i = i')$ and $(j = j')$

and the procedure body is

```
proc P signals OW;
   i := i*j [OV -> signal OW];
```

Here, the programmer has mistakenly typed "*" instead of "+". Then the domains for this example are

```
 SD == (i' = j') and (i' = 0 or i' = 2)
 ED == ~SD
AED == ~SD and (i'*j' not in PI)
UED == ~SD and (i'*j' in PI)
```

There are several possible outcomes of the invocation of an operation in its unanticipated exceptional domain: it may never terminate (go into an infinite loop); a lower level procedure may detect (and propagate) an exception not anticipated by the designer of the operation, and for which a handler does not exist; the operation may terminate at its standard exit point in a state not satisfying its standard specification; or it may terminate at its exceptional exit point in a state not satisfying its exceptional specification.

The problem of handling unanticipated lower-level exceptions is treated in Ada by continuing the propagation of the *lower*-level exception to higher levels if no handler is present. Cristian claims that this solution is dangerous for several reasons. According to the principle of information hiding, the upper level procedure may know nothing of the lower level exception, and thus have no handler for it. Also, continued propagation violates the principle that the flow of control should return from the invoked procedure to the invoker. In effect, the flow of control is through an undeclared exit point from the procedure propagating the exception.

A simpler solution, Cristian states, is the provision of default handlers for these unnamed exceptions by the compiler. This implicitly-provided handler may be used as follows:

```
proc P signals E;
begin
   ...
end [ -> DH];
```

Here, DH denotes the default handler, and the " " before the arrow denotes any exception for which there is no handler explicitly provided.

The purposes of such default exception handlers may be the **masking** of exceptions, that is, making it appear to higher level procedures that no exception has occurred at all; the recovery of a consistent state; or the signalling of an exceptional occurrence to a higher level procedure. The CLU language is oriented towards the latter goal, in effect providing a default handler of the form

```
DH == signal FAILURE.
```

The language SESAME under development at the Universtity of Grenoble is oriented towards both recovery and signalling, providing

```
DH == reset; signal FAILURE.
```

Here, the reset primitive restores the initial state of the operation.

The recovery block mechanism, on the other hand, is oriented towards fulfilling all three goals. A recovery block, such as

```
RB == ensure post by P0
      else by P1 else FAILURE;
```

may be expressed in terms of default exception handlers as follows:

```
RB == P0' [ -> reset;
            P1' [ -> reset; signal FAILURE]]
```

where

```
Pi' == begin Pi; [ ~post -> signal FAILURE] end;
```

for i = 0, 1.

Thus default handlers are at least equivalent in power to recovery blocks. This suggests that recovery blocks are implementable under any system which provides default exception handlers and a reset primitive. Although less powerful than default exception handling, the recovery block scheme is preferable (at least at the application level) since it provides a useful abstraction of a rather messy technique.

An operation is said to be **weakly tolerant** to an exception D if D is detected and the (programmed or default) handler of D recovers a consistent state before propagating D to the invoking procedure. An operation is **strongly tolerant** to D is it can mask the occurrence of D to higher-level procedures. As may has been seen from the discussion of automatic error recovery above, these methods may be used to render the transactions of a system strongly or weakly tolerant to detected unanticipated exception occurrences.

A procedure is said to contain a **design (algorithmic) fault** if its UED is non-empty. A system strongly or weakly tolerant to failure occurrences caused by design faults is called **design fault-tolerant.**

The **commitment interval** of a transaction is defined as the time interval between the beginning and the end of transaction execution. If there is a design fault in the code implementing the operation, however, the acceptance test may not detect the consequences of the fault (since, by the definition of design fault, the acceptance test was not designed such that the part of the exceptional domain in which its effects fall was checked by the test). Thus the acceptance test will be passed, the results of the transaction committed, and recovery made impossible should the consequences of the design fault manifest themselves later. The time between the manifestation of a design fault and the detection of its consequences is called the **latency interval.**

Automatic (or programmed) backward error recovery methods are adequate if the latency intervals of all transactions are contained within the respective commitment intervals of the transactions. However, these methods cannot cope with situations where the latency intervals of transactions may stretch over several successive transaction executions.

The prevention of such situations is tied in with the problem of the adequate specification of acceptance tests such that the UED of an operation is empty, that is, so that there are no design faults (undetected exceptional occurences) in a system. This problem is a current focus of research at Newcastle upon Tyne.

E.7 DIRECTIONS IN RECENT RESEARCH

Work by the group at the University of Newcastle upon Tyne continues in the area of software fault tolerance ([Rand81]). Recent work there includes investigations into the design of reliable remote procedure call mechanisms ([Shriv82]).

Problems in the implementation of recovery blocks include the selection of checkpoint intervals and of appropriate points at which previously checkpointed information may be discarded ([Russ80]). Since the discarding of checkpoint information is equivalent to commitment to the results of the checkpointed block, this issue is of no small importance. Another problem is the design of acceptance tests for the recovery blocks, which is discussed in detail in [Ande81]. It is this problem which, as has been noted above in the discussion of [Cris82], leads to inadequacies of automatic backwards-recovery systems when design faults may manifest their presence after the commitment of the results of the affected block. The proper design of acceptance tests is a current thrust of research at Newcastle upon Tyne.

For distributed systems, the problem of coordination of the separate processes in a recoverable action may be solved by the **two-phase commit protocol** of Gray ([Ande81]). Here, a separate "coordinator" process ensures that, if any process requests backward recovery, all processes are instructed to restore to their recovery points. This is an extension of the "conversation" mechanism described above. Work has started at Newcastle upon Tyne on a search for communication protocols for recovery which can identify recovery lines without the necessity for a central coordinator, or the exchange of large amounts of control information on a (possibly unsafe) message-passing system ([Ande81]).

A possible strategy which should be considered in adding algorithmic-failure recovery mechanisms to PRONET is the notion of "overlaying" a back-up process on the address space of its failed predecessor. This scheme would have the additional advantage of allowing transparent replacement of existing permanent network processes. Old software could be replaced at an appropriate time (say, at a checkpoint) by overlaying a new version on the address space of the old software, without having to halt the entire program. A similar scheme is discussed in [Ande81], where it is suggested that older versions be retained as the back-up algorithms.

Allchin and McKendry ([Allc82]) have proposed that recent work in the database field on "semantic correctness" (as opposed to strict enforcement of correctness criteria, such as serializability) may be extended to the decentralized global operating system for a local area network which is currently under development at Georgia Tech. In their model, support for data management is constructed using abstract data types -- instances of which are "objects" -- together with nested actions. They argue that serializability is often too strong a correctness criterion for the abstract behavior of an object, and that it is sometimes necessary or desirable -- especially for efficiency considerations -- that the implementation of an object violate strict serializability. Synchronization and recovery for objects are thus user-defined, since the writer of an object has semantic knowledge of the object which would be extremely difficult, if not impossible, for the system to determine.

Similar considerations may be applied to the design of algorithmic fault tolerance features in PRONET. In particular, the use of knowledge of the writer of a recovery block about the objects on which the block is based may lead to increases in efficiency in the use of the recovery cache. Another possible line of investigation would be the application of Allchin's object-based recovery model to backwards recovery. Investigations into automatic backwards-recovery schemes thus far have been concerned with action-based recovery, that is, the recovery information has been associated with the operations rather than with the objects. Only very recently has work appeared which is concerned (even peripherally) with recovery in object-oriented languages or systems ([Cox83]).

Considerable further study of the reliability issue is required. Programming techniques must be developed to effectively utilize the failure handling features. These techniques may influence future refinements of the process description language, since they are likely to be rather complex.

REFERENCES

[Allc82] Allchin, James E., and Martin S. McKendry, "Object-Based Synchronization and Recovery," Technical Report GIT-ICS-82/15, September 1982.

[Allc83] Allchin, James E., and Martin S. McKendry, "Facilities for Supporting Atomicity in Operating Systems," Technical Report GIT-ICS-83/1, January 1983.

[Ande81] Anderson, T., and P. A. Lee, *Fault Tolerance*: *Principles and Practice*, (Englewood Cliffs, N.J.: Prentice-Hall International), 1981.

[Cox83] Cox, Brad J., "The Object Oriented Pre-Compiler", *ACM SIGPLAN Notices 18*, 1 (January 1983), 15-22.

[Cris82] Cristian, Flaviu, "Exception Handling and Software Fault Tolerance," *IEEE Trans*. *Comput*. *C-31*, 6 (June 1982), 531-540.

[Kohl81] Kohler, Walter H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Comput*. *Surveys 13*, 2 (June 1981), 149-183.

[Macc82] Maccabe, Arthur B., "Language Features for Fully Distributed Processing Systems," Technical Report GIT-ICS-82/12, September 1982.

[Merl78] Merlin, Philip M., and Brian Randell, "State Restoration in Distributed Systems," *Digest of Papers*, *8th Int*. *Symp*. *Fault-Tolerant Computing 1978*, 129-134.

[Rand75] Randell, B., "System Structure for Software Fault Tolerance," *Proc*. *Int*. *Conf*. *on Reliable Software 1975*, 437-449.

[Rand78] Randell, B., P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *ACM Comput*. *Surveys 10*, 2 (June 1978), 123-166.

[Rand79] Randell, B., "Software Fault Tolerance," *Euro-IFIP 79* (P.A. Samet, ed.), 721-724.

[Rand81] Randell, B., "Reliability and Integrity of Distributed Computing Systems," *The Coordinated Programme of Research in Distributed Computing Systems*: *Annual Report*, *Sept*. *80 - Sept*. *81*, Science Research Council, 1981, 160-165.

[Russ80] Russell, David L., "State Restoration in Systems of Communicating Processes," *IEEE Trans*. *Software Eng*. *SE-6*, 2 (March 1980), 183-194.

[Shri78] Shrivastava, Santosh Kumar, and Jean-Pierre Banatre, "Reliable Resource Allocation Between Unreliable Processes," *IEEE Trans*. *Software Eng*. *SE-4*, 3 (May 1978), 230-241.

[Shri79] Shrivastava, Santosh Kumar, "Concurrent Pascal with Backward Error Recovery," *Software - Practice and Experience 9* (Dec. 1979), 1001-1033.

[Shri81] Shrivastava, Santosh Kumar, "Structuring Distributed Systems for Recoverability and Crash Resistance," *IEEE Trans*. *Software Eng*. *SE-7*, 4 (July 1981), 436-447.

[Shri82] Shrivastava, Santosh Kumar, and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism," _IEEE Trans. Comput. C-31_, 7 (July 1982), 692-706.

[Svob79] Svobodova, Liba, "Reliability Issues in Distributed Information Processing Systems", _Digest of Papers, 9th Int. Symp. Fault-Tolerant Computing 1979_, 9-16.

[Verh78] Verhofstad, J. S. M., "Recovery Techniques for Database Systems," _ACM Comput. Surveys 10_, 2 (June 1978), 167-196.

APPENDIX F

A SURVEY OF QUEUEING NETWORK MODELS OF COMPUTING SYSTEMS

John A. Miller

F.1 INTRODUCTION

In the design and analysis of computing systems, because of their ever increasing complexity, it has become necessary to construct models of these systems. The use of mathematical or other suitably precise models, enables one to abstract the essential features of systems for detailed study of their behavior, interactions, and effects on total system functionality and performance. This process of abstraction and quantification has the advantage of enhancing the understanding of systems. For example, in attempting to understand a particular operating system, one might find the high level approach of a model more palatable than trying to ascertain the behavior of the system from the knowledge of which bits get set when. An even more important advantage of modeling is that it facilitates the use optimization in designing or improving systems.

For a model to be of use in studying a complex computing system, it must come to grips with the following complications: The demands placed upon the system are of a probabilistic nature, various activities are occuring at various places in the system, and finally these activities may be interdependent and occur concurrently. Queueing network models, first introduced by R. R. P. Jackson [Jack54], are a useful tool in dealing with these complexities. Basically, these models represent the system as a network of nodes and arcs. Each node represents a device in the system and is composed of a set of servers that are feed by a queue. Each arc represents a possible flow path for jobs or work requests. From a suitable specification of the model (a set of equations), the model can be solved to determine the performance characteristics of the system, e.g., throughput, response times, device utilizations, and queue lengths.

The purpose of this paper is to survey queueing network models for computing systems. The paper is divided into two parts. In the first part we will consider the mathematics of queueing networks. Specifically, we will consider the elementary theory, various solution techniques, and software packages used to solve queueing networks. In the second part of this paper we will consider some specific modeling studies. This will be done from an evolutionary point of view (from simple uniprocessor systems to complex distributed processing systems). A commment on notation is in order at this point - the symbols E and TT will be used to denote the summation and product operators respectively.

F.2 QUEUEING NETWORKS

F.2.1 Basic Theory

Before considering some of the more complex techniques used to solve queueing network models, let us first examine some of the elementary theory. We first consider **open Jacskson networks** [Heym82]. The solution to such networks is particularly simple since the distributions are Markovian (probabilities are dependent only on the current state of the system, not on its history or elapsed time). Specifically, the model makes the following assumptions.

1) [**Structure**] The network consists of N interconnected service centers.

2) [**Arrivals**] Exogenous (from outside) customers arrive at service center i according to a Poisson process (exponential interarrival) with rate y_i.

3) [**Routing**] After receiving service at center i, a customer leaves the network with probability r_{i0} or goes instantaneously to service center j with probability r_{ij} (where node 0 can be thought of as a special source/sink node). The routing probabilities r_{ij} form a Markov chain with transition (routing) matrix $R = (r_{ij})$.

4) [**Service Center**] Service center i consists of an infinite queue that feeds c_i identical servers. The service discipline is first-come-first-served (FCFS), and the service times are independent identically distributed exponential random variables with mean $1/u_i$.

We will want to obtain solutions to these types of queueing network models, that will specify the probability of the system being in a certain state. Here the state of the system will be a vector that specifies the number of customers at each service center, $s = (s_1,s_2,\ldots,s_N)$. From this information one can then calculate other characteristics of the system, e.g., waiting times, response times, and throughput.

First we need an expression for the total asymptotic arrivals at each queue. This is given by the **traffic equations**, where arrivals at queue i are given by

$$a_i = y_i + E_{j=1..N} a_j r_{ji} \qquad \text{for } i = 1..N.$$

This is a set of N equations in N unknowns, $a = (a_1, \ldots, a_N)$, which can be shown to have the following solution

$$a = y(I - R)^{-1}.$$

With this result in hand we are ready to find the probability that the system is in state s, P(s,t). Specifically, we are interested in the **steady state solution** where the system is in statistical equilibrium $p(s) = \lim_t P(s,t)$. Note, transient state solutions are also useful, but are generally harder to obtain.

To obtain a steady state solution, we apply the principle of conservation of flow to get a set of **flow balance equations.** These equations can be complex in general, but are not difficult for a single queue such as an M/M/c queue. An M/M/c queue has an effective service rate of

$$u_s = \begin{cases} su & \text{if } s < c \\ cu & - \end{cases}$$

where s is the number of customers in the system. The flow balance equations specify that the rate at which customers leave state s, $(y+u_s)p(s)$, must equal the rate at which customers enter state s, $yp(s-1) + u_{s+1}p(s+1)$. Hence the flow balance equations are

$$yp(0) = up(1)$$

$$(y+u_s)p(s) = yp(s-1) + u_{s+1}p(s+1) \qquad s > 0$$

which can be solved recursively to obtain after normalization

$$p(s) = \begin{cases} p(0)(y/u)^s/s! & \text{if } s < c \\ \\ p(0)(y/u)^s/c!c^{s-c} & - \end{cases}$$

$$\text{where } p(0) = [E_{s=0..c-1}(y/u)^s/s! + (y/u)^c/c!(1-y/cu)]^{-1}.$$

Getting back to the original problem, thanks to the J. R. Jackson theorem [Jack57], we can decompose the network into N $M/M/c_i$ queues. Thus the solution is formed from the product of independent component probabilities,

$$p(s) = \prod_{i=1..N} p_i(s_i)$$

$$\text{where } p_i(s_i) = \begin{cases} p_i(0)(a_i/u_i)^{s_i}/s_i! & \text{if } s_i < c_i \\ p_i(0)(a_i/u_i)^{s_i}/c_i!c_i^{s_i-c_i} & - \end{cases}$$

and $p_i(0)$ is the analogous normalization sum. Notice that the key to the tractability of this solution is the fact that a **product form** solution could be found.

Closed Jackson networks are also a useful type of model. The assumptions for closed networks are the same as those for open networks, except that there are a fixed number of customers (jobs) that circulate through the network (i.e., there are no exogenous arrivals or departures). Using the Jackson-Gordon-Newell theorem [Jack63, Gord67], we can obtain a product form solution similar to the one obtained for open networks,

$$P(s) = C\prod_{i=1..N} p_i(s_i)$$

$$\text{where } p_i(s_i) = \begin{cases} (a_i/u_i)^{s_i}/s_i! & \text{if } s_i < c_i \\ (a_i/u_i)^{s_i}/c_i!c_i^{s_i-c_i} & - \end{cases}$$

and C is the normalization constant.

These two types of queueing network models form the basis for the elementary theory of queueing networks. When their assumptions reasonably fit the real-world problem being analyzed, they provide easily obtained exact solutions. However, the real-world is usually not so cooperative, so that solution techniques to more general models will be needed.

F.2.2 Solution Techniques

When faced with complex problems, it is advantageous to have a large arsenal with which to attack the problems. Below is an overview of some of the more useful techniques used to solve queueing networks. They are presented in the rough order in which one should try to use them, i.e., if a problem yields to exact analysis use it or try the next approach, etc.

F.2.2.1 Exact Analysis

Here we consider a general solution technique that yields exact closed-form solutions. Models that have such solutions are called **BCMP networks** [Bask75], and are generalizations of Jackson networks. For a model to be a BCMP network it must satisfy the following set of assumptions.

1) **[Structure]** The network consists of N service centers and K classes of

customers.

2) [**Arrivals**] The types of arrivals determine the type of network. An open network has exogenous arrivals, a closed network does not, and a mixed network is open for some classes and closed for others. There are two basic types of exogenous processes. In the first, the arrival rate to the network is Poisson with mean dependent on the total number of customers, $y'(M(s))$, where $M(s)$ is the number of customers in the network. In this case, the exogenous arrival rate at which class k customers arrive at center i is $y_{ik} = y'q_{ik}$ where the q_{ik}'s are fixed probabilities. In the second, the arrival rate to subchain h (see below) is Poisson with mean $y'(M(s|E_h)$, in which case $y_{ik} = y'q_{ik}$ for each subchain.

3) [**Routing**] A customer of class k who completes service at center i will next require service at center j in class l with probability $r_{ik,jl}$. The routing probabilities $r_{ik,jl}$, form a Markov chain with transition (routing) matrix $R = (r_{ik,jl})$. The Markov chain is assumed to be decomposable into m subchains, where $E_1,\ldots,E_m$ denote the sets of states of these subchains (a state in this context refers the customer (center i, class k)).

4) [**Service Center**] There are four types of service centers allowed in BCMP networks. A type 1 service center consists of an infinite queue feeding c_i identical servers. The service discipline is first-come-first-served (FCFS), and all customers have the same exponential service time distribution. The service rate can be state dependent, $u(M(s_i))$ where $M(s_i)$ is the number of customers at the service center. A type 2 service center consists of an infinite queue feeding a single server. The service discipline is processor sharing (PS), and each class of customers may have a distinct service time distribution. Note PS is the limiting case of round robin (RR) where the time quantum approaches zero. A type 3 service center has no queue and c_i servers, so that at any time the center can hold at most c_i customers. Each class of customers may have a distinct service time distribtion. A type 4 service center consists of an infinite queue feeding a single server. The service displine is preemptive-resume last-come-first-served (LCFS), and each class of customers may have a distinct service time distribution. Note in LCFS an arriving job preempts the server and get service until it completes (preempted job resumes) or is itself preempted [Klei76].

In types 2, 3, and 4 the service time distributions are arbitrary, but must have rational Laplace transforms. Under this slight restriction, one is able to represent the service time distributions as a sequence of exponentially distributed stages using the method of stages [Bask75].

To solve BCMP networks, one can follow a procedure similar to the one given for Jackson networks. Here the state of the system is given by $s = (s_1,\ldots,s_N)$ where each s_i is now a vector that completely specifies the status of service center i, $s_i = (s_1,\ldots,s_{n_i})$ where n_i is the number of customers at center i and s_j is the class of the jth customer in line. The traffic equations for each subchain E_h are

$$e_{ik} = q_{ik} + E_{jl e E_h} e_{jl} r_{jl,ik} \quad \text{for } (i,k) e E_h$$

or multiplying through by y' to get a more familar form

$$a_{ik} = y_{ik} + E_{jl e E_h} a_{jl} r_{jl,ik} \quad \text{for } (i,k) e E_h.$$

These equations are a direct generalization of the ones given for Jackson networks and can be solved similarly.

We are now in a position to find the steady state solution, using what are called the (local) **independent balance equations**, which equate the rate of flow into a state by a customer entering a stage of service to the flow out of that state due to a customer leaving that stage of service. Note, if a customer is queued, his stage will correspond to the stage of service he will be in when he next gets service. Since the global balance equations are the sum of the independent balance equations, independent balance is a sufficient condition for global balance. The solutions to BCMP networks are specified by the Baskett-Chandy-Muntz-Palacios theorem. The steady state probabilities for the case of type 1 arrivals and type 1 service centers are given by

$$p(s) = Cd(s)TT_{i=1..N} p_i(s_i)$$

$$\text{where } p_i(s_i) = (1/u_i)^{n_i} TT_{j=1..n_i} e_{is_j}$$

$$d(s) = TT_{k=0..M(s)-1} y'(k)$$

and C is the normalizing constant. The rest of the cases are similar but somewhat messy products (see [Bask75] for details).

This solution is similar to the solutions obtained for Jackson networks. It is again a product form solution, implying that specific solutions can easily be computed. In fact, BCMP networks define a very general class of queueing network models that yield exact closed-form solutions. These models are flexible enough to be useful in modeling real computing systems. For example, type 1 service centers (FCFS) are good models for secondary storage I/O devices. Type 2 and 4 service centers (PS and LCFS) are good models for processors since LCFS is an efficient preemptive scheduling algorithm and PS is limiting round robin (RR). And type 3 service centers (no queueing) are good models for terminals and routing delays in computer networks. If however, we violate one of the basic assumptions we may not be able to find an exact closed-form solution.

F.2.2.2 Operational Analysis

If the purpose of the analysis is to study an existing system for say tuning or upgrading, and statistics can be gathered by monitoring the system, then operational analysis is a useful and easy to understand tool [Denn78]. It replaces the usual assumptions of stationary stochastic processes used in classical queueing theory, with simple operational (measurable) assumptions that can be verified by monitoring the running system. The basic assumptions are the following.

1) [**Measurability**] All quantities of interest are precisely measurable.

2) [**Flow Balance**] During a reasonably long observation period, the number of arrivals at each service center approximately equals the number of departures (completions) from that service center.

3) [**Homogeneity**] The routing of jobs must be independent of local queue lengths, and the mean time between service completions at a given device must not depend on the queue lengths of other devices.

To use this approach one measures certain basic quantities directly from the system, typically the following.

T = length of observation period
A = number of arrivals in time T
B = total time the system is busy in time T
C = number of completions in time T

These quantities are then used to compute other quantities called derived quantities, that will hopefully give a reasonable characterization of the average behavior of the system. Some of the more important derived quantities are the following.

y = A/T = arrival rate
X = C/T = output rate
U = B/T = system utilization
S = B/C = mean service time

Further there are operational laws and theorems that can be shown to be true when the system satisfies the basic assumptions. Examples of these are the following.

$A = C$: job flow balance
$U = yS$: utilization limit theorem

These numbers can then be used as a guide for tuning or upgrading the system, and are especially useful in identifying bottlenecks. It turns out that the equations derived from the operational approach agree with their traditional Markovian counterparts. This helps explain the robustness of stochastic queueing network models (they seem to have good accuracy even when their assumptions are in doubt).

The advantages of this approach are that it can be applied to any system that satisfies simple assumptions, calculations envolve simple formulas, and it is easy for practitioners (systems analysts) to apply (one does not need to learn queueing theory). The disadvantages are that it is only applicable to existing systems that have good monitoring capabilities, and only average behaviors are considered (part of the beauty of classical queueing theory is that it predicts non-intuitive results due to randomness).

F.2.2.3 Numerical Analysis

When the state of the system can be fully specified by the number and types of customers at the various service centers, then a steady state solutions may be obtained by solving the flow balance equations. [Note to fully specify the state of a GI/G/1 queue time must be included in the state description.] In general, these equations constitute an infinite set of linear equations. Thus we must exploit a recursiveness in these equations to obtain a closed-form solution, but this cannot always be done. However, in many cases such as closed networks, the numbers of possible states is finite, and hence the balance equations form a finite set of linear equations. We may therefore apply the techniques of linear algebra to obtain a numerical solution.

Because these equations are usually very sparse, an iterative solution technique is more efficient than the more common elimination based techniques. A simple procedure that may be used is called **Guass-Seidel iteration** [Coop81]. Suppose one has the following set of linear equations

$$Ax = b.$$

Divide each row of **A** and b by a_{ii} to get $\mathbf{B}x = d$. Letting $B = I - L - U$ where L and U are lower and upper triangular respectively, we have

$$(I - L - U)x = d \quad \text{which may be rewritten}$$
$$x = Lx + Ux + d.$$

Starting with an intial guess x^0 we may iterate using the following equation to converge to the solution,

$$x^{n+1} = Lx^{n+1} + Ux^n + d.$$

Note, in practice a more sophisticated version such as the **method of successive overrelaxation** [Coop81] is often used.

F.2.2.4 Approximate Analysis

Because real computing systems can be quite complex, the models of them need to be highly flexible. Typically, when a system is thought to be too complex to be solved by exact closed form or numerical methods, simulation is resorted to. This however need not be the case. The use of approximate solution techniques provides a way to obtain answers of reasonable accuracy, to very general queueing network models. The word reasonable is used rather loosely; one of the difficulties with approximation techniques is estimating their error bounds.

Before presenting these techniques, let us first consider some complications that make the previous techniques intractable, but have been solved by approximation techniques [Chan78].

1) [**Distributions and Disciplines**] If the arrival distribution, service distribution, and queueing discipline do not satisfy the assumptions for BCMP networks, then it is likely that an approximation technique will be needed. A good example of this is a network with **priority disciplines.**

2) [**Multiple Resource Holding**] When a customer (job) needs more then one resource simultaneously to obtain service, an approximation will be needed. An example of this is a **passive resource,** a resource that does not have a service time associated with it, but limits the population of jobs that may utilize other devices.

3) [**Blocking**] In networks where finiteness of the queues is critical, such as a packet switching network, a device (server) may be blocked, i.e., prevented from serving jobs in its queue because a queue elsewhere in the network is full and cannot accept any more jobs. Again an approximation technique will

be needed.

4) **[Scheduler]** When the delays due to waiting for a scheduler to be activated become significant, approximations will be necessary. Schedulers are a particular complication because once activated they serve many jobs in a relatively short time, so that it is hard to model the service time of a scheduler.

5) **[Parallelism]** If a system has tasks whose subtasks can be run in parallel, then approximation is again called for. An example of this is CPU:I/O overlapped processing, where the CPU and an I/O device service a job in parallel.

6) **[Routing]** If the probability that a job completing service at device i goes to device j is not a constant r_{ij}, but depends on the state of the system, then an approximation will usually be needed. An important example of a type of **dynamic routing** is load balancing (e.g., in say a pooled computer system the scheduler would send a newly arriving job to the computer with the least expected delay).

We will now look at two types of approximations that have been used successfully, decomposition and diffusion. **Decomposition approximations** solve queueing network problems by breaking the network into pieces, solving these pieces separately, and *finally aggregating* these subsolutions to obtain a solution to the whole model [Chan78]. The justification for the accuracy of this approach is first, its application to networks with product form solutions yields exact results, and second, if one partitions the network up into loosely coupled subnetworks then the approximation will likely be good since the interaction effects between the subnetworks will be minimized. The simplest decomposition approach is called the **flow equaivalent method.** Here the strategy is to partition the network into loosely coupled subnetworks, replace each subnetwork with a flow equivalent composite queue, solve each subnetwork to determine the behavior of its associated composite queue, and finally solve the new aggregate network composed of the composite queues. Note that the partitioning may need to be applied recursively to some subnetworks to achieve a tractable solution (i.e., continue breaking up the network into smaller pieces until the pieces are small enough to be solved by some other technique, ideally exact analysis).

Diffusion approximations can be used to obtain approximate solutions to queues with general arrival and service time distributions (e.g., a GI/G/1 queue) [Klei76, Chan78]. The time dependent behavior of a queue is specified by $p(t,n;n_0)$, the probability that at time t there are n customers in the queue given that there were n_0 customers at time $t = 0$. $p(t,n;n_0)$ may be found by solving a set of differential equations (one for each value of n). However, for general distributions this can't be done; hence an approximation is needed. The idea is to replace the discrete variable n by the continuous variable $x > 0$, where the correspondence between n and x is $n = [x]$.

Making the substitution of x for n and the density function $f(t,x;x_0)$ for $p(t,n;n_0)$ and taking the Taylor's expansion to second order of the differential equations, one obtains a partial differential equation, the **diffusion equation**

$$f_t(t,x;x_0) = -cf_x(t,x;x_0) + .5D^2f_{xx}(t,x;x_0) \qquad x,t > 0$$

where c and D are functions of the arrival and service distributions' means and variances [Heym82]. This equation models Brownian motion where a group of particles is released at x_0 and diffuses outward because of collisions, subject to the constraint of a reflecting boundary at $x = 0$.

To obtain a solution to this equation we will use the distribution function, $F(t,x;x_0)$ (integral wrt x of $f(t,x;x_0)$), rather than the density function. It can be shown that F also satisfies the diffusion equation

$$F_t(t,x;x_0) = -cF_x(t,x;x_0) + .5D^2F_{xx}(t,x;x_0)$$

and has the following initial and boundary conditions

$$F(0,x;x_0) = \begin{cases} 0 & \text{if } x < x_0 \\ 1 & - \end{cases}$$

$$F(t,0,x_0) = 0 \qquad t > 0.$$

The solution is given by [Heym82] where $F(t,x;x_0)$ equals

$$PHI\{(x-x_0-ct)/Dt^{.5}\} - \exp(2cx/D^2)PHI\{(-x-x_0-ct)/Dt^{.5}\}$$

where PHI is the normal distribution function. Finally, the steady state solution is found by taking the following limit

$$F(x) = \lim_t F(x,t;x_0) = 1 - \exp(2cx/D^2)$$

where $c < 0$. Under heavy traffic conditions this approximation has been shown to be good by both empirical evidence and a theorem due to Iglehart and Whitt.

In using the diffusion approximation for a network of queues, one generally assumes a product form solution and analyzes each queue independently [Chan78]. The diffusion approximation can also be useful in conjunction with the flow equivalent method when individual queues have general arrival and service distributions. Diffusion approximations are also being applied directly to special networks. For example, Foschini uses the diffusion approximation to solve routing problems for a system with parallel queues [Fosc77].

F.2.2.5 Simulation

Finally, if a model does not yield to any of the previous techniques, then one can simulate the system to get sample solutions which can be statistically analyzed to determine the characteristics of the system. However, one should not go about simulation in a haphazard manner. Such simulations can provide unreliable results. For queueing network models, **regenerative simulations** have been shown to give accurate results [Chan78, Igle78, Saue79a]. In this method confidence intervals for say mean response time are periodically estimated, and a sequencial stopping rule is applied to determine the run length (these problems are difficult for arbitrary simulations). Simulations are also useful in conjunction with analytic techniques (hybrid approach). For example, when using the decomposition (also called *hierarchical*) approach, it may be computationally prudent to obtain numerical solutions to the submodels, and then use simulation for the aggregate model. Simulation is somewhat analogous to the goto statement, it is very powerful, but should only be used in well thought out ways.

F.2.3 Queueing Network Packages

To make queueing network modeling more convenient, packages have been developed to solve these models [Saue79b]. Generally these packages take as input a specification of the queueing network (via an interactive dialogue or a special purpose language), formulate the problem mathematically, and solve the equations using the techniques described in this paper. Let us now consider some of the major packages that have been developed (note, many of these packages are available either commercially or otherwise).

The first major package to be completed was RQA by Wallace and Rosenberg in 1966. RQA solves queueing networks with finite state spaces by formulating the linear (global) balance equations, and solving them by numerical analysis. The use of this approach has two principle weaknesses. First, there is a limit to the number of states that a system can have (a few thousand states).

Second, for systems with a large number of states RQA can be intolerably slow.

In 1973, **ASQ** was completed by Keller. ASQ solves queueing networks with product form solutions using exact analysis. Later ASQ was extended to the hierarchical solution of networks, and was eventually renamed **CADS**. The chief limitation here is that the networks must have product form solutions.

Foster and McGehearty completed in 1974 a special purpose language **QAL** and implemented a simulation solution program **QSIM**. QAL provided several extensions to the networks of RQA and ASQ, such as allowing passive resources. The primary weaknesses of QAL are its lack of non-simulation solution implementations, and its lack of support for representing distinct job classes.

In 1975, Sauer completed **APPLOMB**, which solved a general class of queueing networks using regenerative simulation. During this same year **QNET4** was completed by Reiser. QNET4 sovled product form networks with multiple (local) job classes using exact analysis. In 1978, these two packages were combined to form **RESQ**, which provides a fairly comprehensive solution capability and a good user interface.

BEST/1 was completed by IBM in 1977. BEST/1 was specifically designed to solve capacity planning problems in computer systems. It solves slight variants of product form networks using exact analysis in conjunction with special approximations (the details are proprietary).

The final package we will consider is **QSOLVE**, which was completed by Levy in 1977. QSOLVE uses an approach similar to RQA in thatit uses numerical analysis. However, it is oriented toward networks similar to, but violating product form (e,g., it allows more general job classes and queueing disciplines).

F.3 MODELS

Now that we have a feel for what queueing networks are and how they are solved, we turn to the modeling process. This process which is more of an art than a science, involves a careful examination of the system (real or hypothetical) and abstracting out the essential features of the system relevent to the aspect of performance being considered. Modeling studies may focus on the total system or some subsystem such as the operating system, the database system, or the communication subsystem. In modeling general computing systems, two approaches have been successfully used, queueing networks and simulation. The disadvantage of simulations are their high cost, both computational and developmental, and their potential unreliability resulting from programming bugs and the difficulty of applying rigorous statistical analysis to their outputs [Koba78]. For all but highly detailed models, queueing networks offer a good alternative. The reason they have not been used that much is that many of the advances in solving these models has come about in the last few years, and as of yet not enough experience has been gained in their use.

An ideal scenario for the use of modeling is the following [Saue81]: In the early design phases use simple queueing network models to reject infeasible designs and guide design improvements. As the design nears finalization, it should be represented by a detailed queueing network model. At this point, if there is sufficient time and money, a detailed simulation may be helpful. It can capture some of the details ignored in the queueing network model, and if they agree it provides a partial validation of the queueing network model. Note, a nice feature of this approach is that if everything goes right, only one costly simulation will be necessary. Once the system is operational the queueing network model should be validated by comparing its performance predictions with performance measurements obtained from the running system. If there are significant disagreements, then the results can be used to correct the deficiencies, either in the model or the system. Once the model is validated, it can be used to configure other installations with greater confidence, and if changes in the system are needed, it can be used in redesign and redevelopment.

F.3.1 Some Successful Models

To see how queueing network models are used, we will look at several modeling studies. In the rest of this section we will look at some models that have been used successfully, i.e., the models were shown to be accurate and were found to be of use in designing, upgrading, and/or tuning computing systems. In the next section, we will focus on the application of queueing network models to a new area of high potential, where results just recently began coming in, namely distributed processing systems.

The first successful application of a queueing network model to a computing system was done by Scherr in 1967. He applied a **machine repairman** model to the Compatible Time-Sharing System (CTSS) at MIT [Grah78, Munt75]. This model can be thought of as a closed queueing network with two nodes, one representing the central system (memory, CPU, and I/O devices), and the other representing a collection of N terminals.

```
                  ------------
                  |          |
                  |    N     |
 -----------------|          |<-----------------
 |                | terminals|                 |
 |                |          |                 |
 |                ------------                 |
 |                                             |
 |                                             |
 |                ------------                 |
 |                |          |                 |
 --------->|||||||  system   |------------------
                  |          |
                  ------------
```

In this model N jobs circulate around the network; each job is permanently associated with a particular terminal. At the terminal node there is no queueing so that a job goes directly to its associated terminal, and remains there for the duration of its terminal service time (think time of its user) which is modeled as an exponential random variable. At the central system node the jobs queue up to obtain its services. The service time of the central system represents the sum of the program execution time and the unoverlapped swap time, and is also modeled as an exponential random variable. [Note, CTSS was an early interactive system where user programs were swapped in and out of memory, implying only one program could be in memory at a time.] Hence there are three possible states a job can be in: 1) at its terminal, corresponds to a user thinking, 2) in the central system queue waiting for service, or 3) receiving service from the central system.

Clearly this is a very simple model; surely it's too simple to be an accurate predictor of performance. Scherr compared the model's predicted mean response time with the actual response time experienced by users. Surprisingly, the model was amazingly accurate. In Scherr's words, the results were "startling" considering the simple model used to predict the performance of a "highly complex hardware-software system." This high accuracy is in part explained by the fact that the central system serves only one job at a time; hence the model agreed well with the configuration of the real system. In addition, it has been shown by theoretical results that performance measures such as average delays and throughputs are rather insensitive to service distributions [Koba78].

In 1972, this same model was applied by Lassettre and Scherr in the design of IBM's Time Sharing Option (TSO) with a single partition [Munt75]. The model's predicted mean response times were compared with measured response times that were generated form script driven workloads. When at first the model and the measurements did not agree, Lassettre and Scherr has enough confidence in their model to claim that a system error or poor scheduling policy was the cause. This indeed turned out to be the case; after the performance bug was located the model gave accurate predictions [Koba78].

The next major advance in the application of queueing network models to computing systems, came in 1971 when Moore modeled the Michigan Terminal System (MTS) as a closed Jackson network [Koba78, Munt75]. His model explicitly represented the major resources of the system, an IBM 360/67 with a dual processor, 1.5 megabytes of memory, 2 paging drums, and approximately 100 terminals. He found that the model could be simplified somewhat by treating lightly used resources as a single resource.

As specified by Jackson's results the service times for each resource were modeled as exponential random variables whose means were estimated from measurements of the system in operation. Moore observed that the exponential distribution did not fit the data he collected very well; however to predict average values (e.g., mean response time and resource utilizations) this did not have much effect on the accuracy of the predictions. He also used this measured data to estimate the transition probabilities (probability of going to resource j after completing service at resource i).

Moore measured the system over 10 to 15 minute intervals, using the data to estimate the model parameters specified above. He then compared the model predictions of mean response time and resource utilizations to those obtained from the measured data. Again good accuracy was obtained, the predicted values were typically within 10 percent of the measured values. Considering the complexity of the system, a large interactive computing system, these results are very good. In addition, Moore found that the performance was very sensitive to the load on the system, so that accurate estimation of the model parameters was essential for accurate performance predictions.

Also in 1971, Buzen used a particular type of closed Jackson network to analyze multiprogramming systems. He called models of this type, **central server models** [Koba78]. A basic central server model consists of a CPU and independent secondary storage and I/O devices.

```
                                    -------
                          ----->||||| I/O |------
                          |         -------      |
                          |                      |
     ----------------     |         -------      |
     |              |     |---->||||| I/O |-----|
     v      ------- |     |         -------      |
------->||||| CPU |-------|                      |--------
|           -------       |            :         |       |
|                         |         -------      |       |
|                         ----->||||| I/O |------        |
|                                   -------              |
----------------------------------------------------------
```

This closed network captures the basic behavior of a multiprogramming system. The number of jobs that circulate through the system corresponds to the multiprogramming level. A typical job will progress as follows: It will receive CPU service from which it will either be preempted or request I/O service, upon completion of which it will again seek CPU service. This scenario will be repeated indefinitely. Clearly a real job does not have an infinite lifetime, but if a system has a maximum multiprogramming level and is reasonably loaded, we can think of a completed job being replaced by a new job. Hence the abstract notion of an infinite job is a reasonable model. Buzen first used these models to study the throughput of batch systems [Munt75].

Later, Buzen used a central server model for a comprehensive analysis of the IBM Multiple Virtual Storage (MVS) operating system [Buze78]. The purpose of this analysis was to model the resource allocation mechanisms of MVS, so that given current or future workloads for a system, an optimal strategy for upgrading and tuning could be determined. MVS allows an installation manager to classify workloads (MVS allows batch workloads, time sharing workloads, and transaction processing workloads), and provides mechanisms by which the allocation of resources to workloads can be controlled. Additionally, within workloads there are mechanisms by which allocations to individual jobs can be controlled.

For controlling the allocation of resource amoung workloads, MVS provides two mechanisms, one which regulates access to memory, and the other which regulates access to the central processor. To control the allocation of memory, the installation manager divides it up into domains (a logical region of memory), and assigns workloads to these domains. Since each domain has a maximum allowed multiprogramming level, this domain mechanism regulates the allocation of jobs to memory according to their job class (workload classification). The allocation of the central processor is controlled by the scheduling algorithm. Scheduling amoung workloads is done using a preemptive-resume priority discipline, where jobs of higher priority preempt jobs of lower priority which are resumed upon completion of the higher priority job. Thus a workload's allocation is controlled by assigning it an appropriate priority level. [Note, within a single priority level a round-robin (or equivalent) scheduling algorithm is used.]

Given this first level of resource allocation control, MVS also provides for second level of control using mechanisms that allocate resources within workloads. Here decisions are not made on the basis of job classification, but rather by the operating system monitoring the behavior of jobs. There are two mechanisms by which this control is carried out, domain migration and exchange swapping. Domain migration is used to control the allocation of memory. The idea here is to associate several domains with each workload and to set the multiprogramming level lower for each successive domain. Then when a job has consumed to many service units (weighted sum of CPU time, I/O processing, and the memory space-time product), it is transferred to the next domain. If the job is transferred to a domain already at its target multiprogramming level, the job will be swapped out of main memory. The second mechanism, exchange swapping, is also used to control the allocation of memory. The idea here is that for all jobs a dynamic memory priority is periodically computed, and when for a given domain the priority of a job in memory falls below one waiting to be loaded into memory, an exchange swap is generated. These mechanisms keep jobs from monopolizing main memory.

The specific system that was modeled was an IBM 370/168 Model 3 with 7 megabytes of memory and a total of 46 disks, drums, and tape drives. The system was processing a variety of workloads, principlely time sharing (TSO), batch processing, transaction processing (IMS), and certain special purpose subsystems. In developing the model, Buzen first specified the job classes;

three classes were used for time sharing (short, medium, and long transactions), one was used to represent batch processing, and a fifth was used to represent various system overehead processes. Each of the five job classes was assigned a particular domain.

Next Buzen modeled the various features of MVS, especially those dealing with resource allocation. Domain migration was used for time sharing jobs. Intially, all such jobs would be in the first domain, and as time progressed some would migrate to the second and third time sharing domains. Hence to capture the steady state behavior, domain migration was modeled by assigning the appropriate fraction of jobs to the three levels. Interactive swapping (whenever an interactive job is waiting for terminal input and another interactive job is waiting to be loaded into memory, a swap is generated) was modeled by assuming that certain I/O devices are allocated to swapping and setting their mean service time to the average swapping time for that particular device. For the system that Buzen modeled both drums and slower disks were used for swapping, the drums being used until they are full. Demand paging is similarly modeled by setting the mean service time to the average paging time for the particular devices used. Note, page reads and writes were treated differently; since a job must wait for a page to be read, this activity is considered to be part of the job's demand for I/O services, whereas page writes are considered to be part of the system overhead. Central processor scheduling which used a preemptive-resume priority discipline was modeled using certain proprietary approximations (priority scheduling violates product form conditions). Finally, exchange swapping was found to be a negligible factor (not used freaquently), and was therefore not included in the model.

The complete queueing network model was formed by connecting these submodels together. The result was a central server model which was a generalization of the one shown previously. Buzen used the queueing network package BEST/1 to analyze this model. BEST/1 is specifically designed to analyze central server models of this form, and allows the following features to be modeled.

1) [Job Classes] Multiple job classes can be represented where each class has its own service time requirements at each device. Job classes can be either open or closed.

2) [Domains] Multiple domains can be represented where each domain has its own

target multiprogramming level and a separate queue. One or more job classes can be assigned to a domain.

3) **[Disciplines]** All product form queueing disciplines are allowed. In addition, the CPU queueing discipline can be preemptive-resume priority.

4) **[I/O Devices]** I/O devices each have their own service times (which include channel and controller delays).

Buzen collected data from the system using the IBM Resource Measurement Facility (RMF), to obtain estimates for the model's parameters. He then comapred the model's predictions of mean response time (broken down by job class), device utilizations, and total throughput with those measured from the running system. Typically, the models predictions were off by less than 10 percent, and in many cases the predictions were very accurate.

To help complete the picture without belaboring the point, let us briefly consider some further applications. Queueing network models have been used to study the performance of multiprocessing systems. Sauer and Chandy modeled general multiprocessing systems (tightly coupled systems such as C.mmp) to analyze the performance characteristics of such systems [Saue79a]. Specifically, they considered the effects on performance of CPU service distributions and disciplines, the level of multiprogramming, multitasking, and job priorities. In their analyses they compared the performance of a uniprocessor with unit speed, to that of a multiprocessor having N processors each with speed 1/N (for N = 2, 4, 8). Considering how cheap microprocessors are, one would think the multiprocessor would be far less expensive (compare 8 Intel 8086's to an IBM mainframe). Sauer and Chandy included in their model a performance reduction factor based on the work of Fuller (he found that the degradation in performance caused by the contension of processors for memory was less than 10 percent for actual and proposed C.mmp configurations). Basically, Sauer and Chandy found that given a sufficiently high multiprogramming level, that the multiprocessing systems could, even using a simple scheduling strategy (FCFS), obtain system throughputs close to those obtained by the uniprocessor system (in the range of 70 to 100 percent).

Finally, an interesting application of queueing network models was done by Browne, Chandy, and four other consultants, in the development of the Air Force's Advanced Logistics System (a large data management system) [Saue81]. The queueing network model was composed of four submodels: one for the CPU's (2 Cyber 70's), one for the memories both private and shared (million words), one for the database disks (100 disks), and one for the system/scratch disks and tape drives (8 disks and 24 tape drives). The model predicted that the proposed system was inadequate because of insufficient capacity in the system/scratch disk subsystem and in the CPU's. Both of these predictions were confirmed by subsequent operational experience and measurement. Amazingly, the entire modeling effort required only two months for the six consultants to complete.

F.3.2 Application to Distributed Processing Systems

Currently distributed processing systems are generating much research interest, and rightly so. They potentially provide for high system availability, reliability, and performance,and for incremental growth and configuration flexibility [Ensl78]. This flexibility provides for many degrees of freedom in the design process. Because of this, modeling of the performance of distributed processing systems becomes very important. Within the framework of the ISO Open System Interconnection Architecture several design decisions need to be made. Many ofthese decisions will have a significant impact on the performance of a distributed system [Tane81].

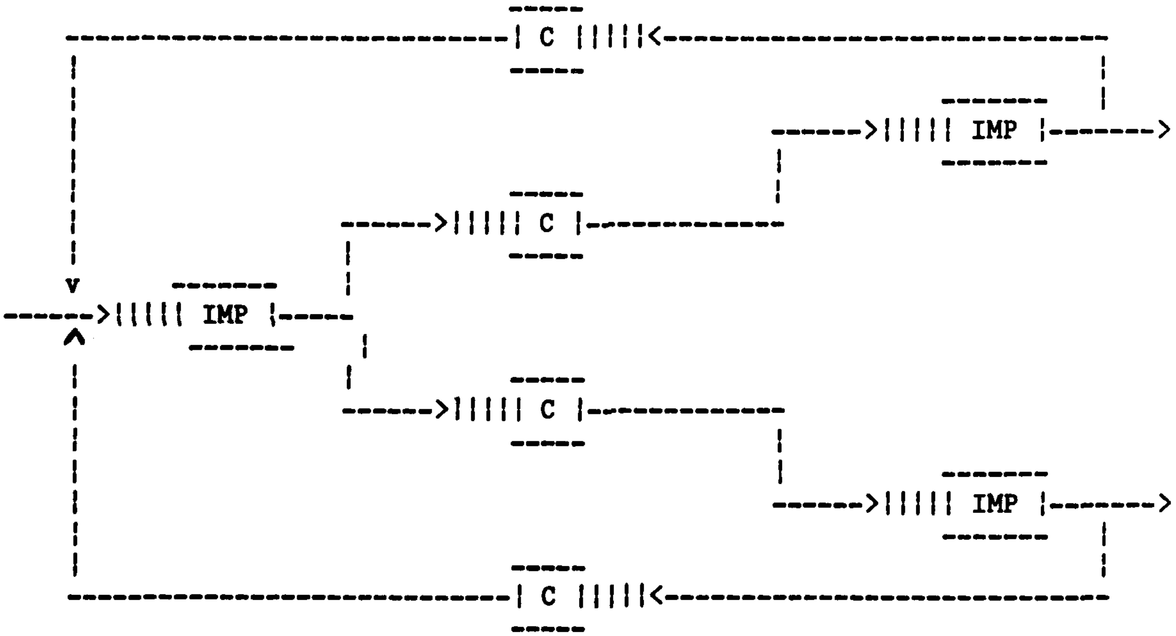
Basically, queueing *network models are used in two types* of studies of distributed processing systems [Wong78]. The first type of study is directed at the communication subnetwork, while the second is directed at the user-resource network.

Performance studies of the **communication subnetwork** are concerned with the delivery of messages. Performance measures of importance here are message end-to-end delay, message throughput, and channel utilizations. Three design areas are involved in these studies. First, the system configuration (assuming a given topology) may be modeled to answer questions such as what capacity channels to use and how many message buffers to provide. Second, the basic control algorithms of the network layer such as routing, congestion control, and access protocols can be modeled and analyzed to determine the best network control strategy. Finally, at the transport layer, flow control and virtual circuit path selection can be modeled to address end-to-end concerns.

As an example of such a study, let us consider a model of a message switched network (note: packets can be regarded as small messages) [Wong78]. The type of communication subnetwork considered in this study consists of several intermediate processors (IMP's) connected by communication channels

(C's). The intermediate processors are responsible for the usual store and forward operations of message buffering and outgoing channel selection, while the channels are responsible for transmitting messages form IMP to IMP.

The subnetwork is modeled as an open BCMP queueing network, where messages originating from user terminals and host computers move from source to destination by successively queueing for service at the two types of nodes. One type of node represents the intermediate processors, while the other represents the channels. For example, consider a piece of the queueing network consisting of three intermediate processors (one having external arrivals), and the channels connecting them.



As a first order approximation, it is assumed that the queueing delays and service times at the intermediate processors are negligible. This is done in Wong's model (based on Kleinrock's work [Klei76]) by letting the IMP nodes have no queueing and zero service time (essentially these nodes carry out instantaneous routing) [Wong78]. The service time at each of the channels in this model is given by the message length divided by the channel capacity. In addition, the messages are classified according to their source and destination IMP's. The routing of the messages at the IMP's is done on the basis of their class and can be either random or fixed. For the sake of tractability some further assumptions are needed: The external arrivals are Poisson, all message classes have the same exponential message length distribution, the queues are unbounded, the discipline is FIFO, and Kleinrock's independence assumption holds (an approximation stating that each time a message enters a node its length is redrawn from the exponential distribution).

Kleinrock solved this model for mean end-to-end delay, and plotted end-to-end delay versus throughput. He found that the delay was small until the system was operated at near full capacity (i.e., one or more channels near saturation), at which point the delay increased exponentially. Wong extended Kleinrock's solution by solving for the probability distribution of end-to-end delay for each message class (this allowed variances and percentiles to be computed). In a model validation study Kleinrock extended this model to include the processing time of the IMP's, propagation delays, and other features pertinent to the ARPA network. He found that the mean delay calculated from the model was 73 msec., while that derived from the measurement data was 93 msec. This is a discrepancy of 21.5 percent, not unreasonable considering the complexity of the ARPA network.

Many extensions to this basic model have been seen in the literature. Wong extended the model to consider the problem of buffer management using a finite buffer model [Wong78]. Wong has also modeled end-to-end flow control. Samari and Schneider extened Kleinrock's model by considering delays and service times associated with the IMP's, and by including a correction factor to account for the nonexponential nature of the interarrival time of input to the channels [Sama80]. They tested their model against a simulation model and found that they differed by less than 7 percent (note the analytic model required far less computer time). Kurinckx and Pujolle applied a similar model (where the nodes were IMP's) to study the end-to-end control through virtual circuits in a computer network built following the X.25 Recommendations [Kuri80]. They were particularly interested in determining the maximum buffer overallocation for a given probability of overflow.

Turning now to studies of the **user-resource network**, we are now concerned with the performance of higher level services. This corresponds to the application layer in the Open System Architecture. Some of the problems here are concerned with which processes to run where, and where to place data. Finding optimal (or near optimal) solutions to these problems can greatly help system performance. The implementation of these solutions would be in the system's distributed operating system and/or distributed database system.

As an example consider the problem of scheduling a set of processes on a fully distributed processing system. An ideal system level scheduler should have knowledge of the communication needs of the processes in the system and the status of the processors in the system. A particular concern would be that of efficiently scheduling distributed programs such as a distributed compiler [Mill82], so as to minimize its communication waiting delays. Currently, scheduling as complex as this has not been modeled. However, Chou and Abraham modeled system level scheduling stochastically [Chou82]. They considered the problem of scheduling a set of processes (or tasks) on a set of heterogenous processors. They presented an algorithm that optimally assigns tasks to processors, which is based on Markov decision theory.

Finally, in an attempt to determine the overall performance of a distributed processing system, Wong combined his communication subnetwork model with a model of a simple user-resource component consisting of a set of remote terminals and a single host with local terminals [Wong78]. Further assumptions for this model are: The CPU discipline is processor sharing, and the think time and CPU service time have rational Laplace transforms. Since this is a BCMP qeueing network, Wong found an exact solution for the mean response time for local and remote users. He plotted the mean response times versus the number of local users for various numbers of remote users. As expected, beyond a certain threshold the mean response time increases linearly with the number of local users.

Other than the studies of the performance of the communications subnetwork, there have been few analytic models of distributed processing systems. In particular, more work needs to be done in modeling the higher level services such as system level scheduling, and in developing integrated models that take into account both the characteristics of the communication subnetwork and the characterictics of the various resources connected to the subnetwork (e.g., processors along with their memories, terminals, secondary storage devices, and other peripherals).

REFERENCES

[Alle80] Arnold O. Allen, "Queueing Models of Computer Systems," Computer, Vol. 13, No. 4, April 1980, pp. 13-24.

[Bard78] Y. Bard, "The VM/370 Performance Predictor," Computing Surveys, Vol. 10, No. 3, Sept. 1978, pp. 333-342.

[Bask75] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," J. ACM, Vol. 22, No. 2, April 1975, pp. 248-260.

[Brue80] Steven C. Bruell and Gianfranco Balbo, Computational Algorithms for Closed Queueing Networks, Elsevier North-Holland, Inc., New York, 1980.

[Buze73] Jeffrey P. Buzen, "Computational Algorithms for Closed Queueing Networks with Exponential Servers," Comm. ACM, Vol. 16, No. 9, Sept. 1973, pp. 527-531.

[Buze78] Jeffrey P. Buzen, "A Queueing Network Model of MVS," Computing Surveys, Vol. 10, No. 3, Sept. 1978, pp. 319-331.

[Chan78] K. Mani Chandy and Charles H. Sauer, "Approximate Methods for Analyzing Queueing Network Models of Computing Systems," Computing Surveys, Vol. 10, No. 3, Sept. 1978, pp. 281-317.

[Chan82] K. Mani Chandy and Doug Neuse, "Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems," Comm. ACM, Vol. 25, No. 2, Feb. 1982, pp. 126-134.

[Chou82] Timothy C. K. Chou and Jacob A. Abraham, "Load Balancing in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-8, No.4, July 1982, pp. 401-412.

[Coop81] Robert B. Cooper, Introduction to Queueing Theory, 2nd Ed., Elsevier North-Holland, Inc., New York, 1981.

[Denn78] Peter J. Denning and Jeffrey P. Buzen, "The Operational Analysis of Queueing Network Models," Computing Surveys, Vol. 10, No. 3, Sept. 1978, pp. 226-261.

[Ensl78] Philip H. Enslow, Jr., "What is a 'Distributed' Data Processing System?" Computer, Jan. 1978, pp. 13-21.

[Fosc77] G. J. Foschini, "On Heavy Traffic Diffusion Analysis and Dynamic Routing in Packet Switching Networks," in Computer Performance, K. M. Chandy and M. Reiser (Eds.), Elsevier North-Holland, Inc., New York, 1977, pp. 419-514.

[Gord67] William J. Gordon and Gordon F. Newell, "Closed Queueing Systems with Exponential Servers," Operations Research, Vol. 15, No. 2, March-April 1967, pp. 254-265.

[Grah78] G. Scott Graham, "Queueing Network Models of Computer System Performance," Computing Surveys, Vol. 10, No. 3, Sept. 1978, pp. 219-224.

[Heym82] Daniel P Heyman and Matthew J. Sobel, Stochastic Models in

Operations Research, Vol. I: Stochastic Processes and Operating Characteristics, McGraw-Hill, Inc., New York, 1982.

[Igle78] Donald L. Iglehart, "Regenerative Simulation of Response Times in Networks of Queues," J. ACM, Vol. 25, No. 3, July 1978, pp. 449-460.

[Igle78] Donald L. Iglehart, "The Regenerative Method for Simulation Analysis," in Current Trends in Programming Methodology Vol. III: Software Modeling, K. M. Chandy and R. T. Yeh (Eds.), Prentice-Hall, Inc., Englewood Cliffs, N. J., 1978, pp. 52-71.

[Jack57] James R. Jackson, "Networks of Waiting Lines," Operations Research, Vol. 5, 1957, pp. 518-521.

[Jack63] James R. Jackson, "Jobshop-Like Queueing Systems," Management Science, Vol. 10, 1963, pp. 131-142.

[Jack54] R. R. P. Jackson, "Queueing Systems with Phase-Type Service," Operational Research Quarterly, Vol. 5, 1954, pp. 109-120.

[Jaco82] Patricia A. Jacobson and Edward D. Lazowska, "Analyzing Queueing Networks with Simultaneous Resource Possession," Comm. ACM, Vol. 25, No. 2, Feb. 1982, pp. 142-151.

[Kell79] F. P. Kelly, Reversibility and Stochastic Networks, John Wiley & Sons, Ltd., Chichester, 1979.

[Klei76] Leonard Kleinrock, Queueing Systems, Vol. II: Computer Applications, John Wiley & Sons, Inc., New York, 1976.

[Koba78] Hisashi Kobayashi, "System Design and Performance Analysis Using Analytic Models," in Current Trends in Programming Methodology Vol. III: Software Modeling, K. M. Chandy and R. T. Yeh (Eds.), Prentice-Hall, Inc. Englewood Cliffs, N. J., 1978, pp. 72-114.

[Kuri80] A. Kurinckx and G. Pujolle, "Overallocation in a Virtual Circuit Computer Network,

[Mill82] John A. Miller and Richard J. LeBlanc, "Distributed Compilation: A Case Study", Third International Conference on Distributed Computing Systems, Oct. 1982, pp. 548-553.

[Munt75] Richard R. Muntz, "Analytic Modeling of Interactive Systems," Proceeding of the IEEE, Vol. 63, No. 6, June 1975, pp. 946-953.

[Munt78] Richard R. Muntz, "Queueing Networks: A Critique of the State of the Art and Directions for the Future," Computing Surveys, Vol. 10, No. 3, Sept. 1978, pp. 353-359.

[Rose78] Clifford A. Rose, "A Measurement Procedure for Queueing Network Models of Computer Systems," Computing Surveys, Vol. 10, No. 3, Sept. 1978, pp. 262-280.

[Sama80] N. K. Samari and G. Schneider, "A Queueing Theory-Based Analytic Model of a Distributed Computer Network," IEEE Transactions on Computers, Vol. C-29, No. 11, Nov. 1980, pp. 994-1001.

[Saue79a] Charles H. Sauer and K. Mani Chandy, "The Impact of Distributions and Disciplines on Multiple Processor Systems, Comm. ACM, Vol. 22, No. 1, Jan 1979, pp. 25-34.

[Saue79b] C. H. Sauer and E. A. MacNair, "Queueing Network Software for Systems Modelling," Software -- Practice and Experience, Vol. 9, May

1979, pp. 369-380.

[Saue80] Charles H. Sauer and K. Mani Chandy, "Approximate Solution of Queueing Models," Computer, Vol. 13, No. 4, April 1980, pp. 25-32.

[Saue81] Charles H. Sauer and K. Mani Chandy, Computer Systems Performance Modeling, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1981.

[Tann81] Andrew S. Tanenbaum, Computer Networks, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1981.

[Tows78] D. Towsley, K. M. Chandy, and J. C. Browne, "Models for Parallel Processing Within Programs: Application to CPU:I/O and I/O:I/O Overlap," Comm. ACM, Vol. 21, No. 10, Oct. 1978, pp. 821-831.

[Wong78] J. W. Wong, "Queueing Network Modeling of Computer Communications Networks," Computing Surveys, Vol. 10, No. 3, Sept. 1978, pp. 343-351.

[Zaho82] John Zahorjan, Kenneth C. Sevcik, Derek L. Eager, and Bruce Galler, "Balanced Job Bound Analysis of Queueing Networks," Comm. ACM, Vol. 25, No. 2, Feb. 1982, pp. 134-141.

APPENDIX G

THE DESIGN AND EVALUATION OF A DISTRIBUTED COMPILER

John A. Miller
Richard J. LeBlanc

G.1 INTRODUCTION

The increasing availability of distributed computing systems connected by local area networks has produced interest in the application of distributed computing to software traditionally run on uniprocessors. The principal motivation for such application is to attempt to decrease the response time of programs by partitioning them into components which can be executed in parallel. This paper describes an experiment which tested the feasibility of implementing a compiler as a distributed program. It should be noted that this study is intended only as an initial examination of this problem. The results are somewhat dependent on the hardware configuration on which the study was conducted and on the nature of the task performed by a compiler. However, some generalized conclusions can be drawn from our experience.

To carry out this experiment, we constructed two versions of a compiler, a distributed version and a single-pass version. We then compared the response times of the two compilers on test programs of various sizes. It was our hope that the distributed version would show a significant improvement in response time due to its utilization of parallelism inherent to the compilation process.

The experiment was performed using the facilities of the computing laboratory of the School of Information and Computer Science at Georgia Tech. The distributed system available included a network of five Prime computers, two Prime P550's and three Prime P400's. The computers are interconnected by Ringnet [Gord79], a packet switched communication system. Ringnet is a subset of PRIMENET (PRIMENET refers to all Prime's networking products) that deals with local area communication. Ringnet is an unidirectional loop network that consists of a node controller, a coaxial transmission cable that provides an 8

*This paper has been submitted to Computer Networks.

Mbits/sec effective bit rate, and interfaces to the transmission cable at each node.

A group at the Illinois Institute of Technology have done considerable research related to this work. They first implemented a distributed compiler for the language DYNAMO on their network computer, known as TECHNEC [Huen77]. TECHNEC is a network of several LSI-11 computers that work together to execute a single job at a time. More recently they have reported work on a distributed Pascal compiler for TECHNEC [El-D79]. Their work has included considerations of automatic partitioning of object code as well as attempts to distribute the executions of a compiler. Our work is related to the latter of these efforts. This report goes beyond their publications by presenting a comparison of the performance of distributed and standard compilers.

G.2 THE COMPILER

G.2.1 The Language

The programming language used for this study was a subset of Pascal, called Jigsaw, used in compiler writing courses at Georgia Tech. This language was chosen because it is simple enough to keep the compiler development time as short as possible, yet it contains enough features to present "typical" compilation problems. The features of Jigsaw include: if and while control statements, integer, real, array, and record data types, and parameterized procedures.

G.2.2 Components of the Compiler

The process of partitioning the problem is of paramount importance in implementing a distributed program. Ideally, the component parts of a distributed program should each implement a single step of the task being performed. More importantly, the interaction between the components should be simple and infrequent (relatively speaking). Such a partitioning will result in components which can easily be connected as a pipeline. The Jigsaw compiler was partitioned into the following components (all written in Pascal): a lexical analyzer, a syntactic analyzer, and a semantic analyzer/code generator. A separate code generator would usually be needed, but the target code (for a hypothetical machine) was sufficiently similar to intermediate code (e.g., quadruples) to make a separate code generation step unnecessary. These components work together in series to decompose source statements, analyze their contents and, finally, compose the target or object code. The process is analogous to an assembly line, where the product goes through different phases on its way to completion.

G.2.2.1 Lexical Analyzer

The lexical analyzer or scanner is the first phase of the compilation process. Its function is to read lines from the source code file and break them up into their component symbols, called tokens. The tokens of a programming language are analogous to the words and punctuation marks of English text. The extraction of tokens from the input line is accomplished by using a finite state machine that does pattern matching on the characters in the line. When it finds a valid pattern it outputs a number that specifies the class to which the token belongs (e.g. identifiers make up a class), and, where additional semantic information is necessary, it outputs the token string itself. For example, when the token '123' is found, the lexical analyzer outputs the class number for integer constants and the string '123', so that the constant value may be computed later. Thus the lexical analyzer performs the mapping shown in Figure 1(a).

The lexical analysis component was constructed using a lexical analyzer generator, a relatively common compiler development tool available on our computing system. This tool consists of a table-driven lexical analyzer and a lexical table generator. To use this tool, one specifies the tokens of the language as regular expressions. The generator program reads this information and produces a table that is used by the lexical analyzer to make decisions in the pattern matching process. This standard lexical analyzer, which uses the generated table, is easily incorporated in a compiler.

G.2.2.2 Syntactic Analyzer

The second phase of the compilation process is performed by the syntactic analyzer or parser. It takes the token numbers generated by the lexical analyzer and collects them together to form phrases that are specified by grammatical rules. This grouping of tokens into phrases is accomplished by a pushdown-store machine. When it determines that a string of tokens satisfies a grammatical rule, it replaces that string with a nonterminal symbol that stands for a string of that type. In addition, for certain rules, actions must be carried out to manipulate semantic information or generate some form of code. These actions are specified by "action numbers" attached to these rules in the grammatical specification. Thus the syntactic analyzer performs the mapping illustrated by Figure 1(b).

We made use of another common tool, a LALR(1) parser generator, to construct the syntactic analysis component. It consists of a table-driven parser and a parsing table generator. To use the generator, one simply writes a specification of the grammar for the language in Backus-Naur Form (BNF). The generator will produce tables from this information which will be used by the parser to make parsing decisions.

G.2.2.3 **Semantic Analyzer**

In this implementation, the semantic analyzer constitutes the third and final phase of the compilation process. Its basic function is to implement the "semantics" (meaning) of the program. It is driven by the action numbers generated by the parser. These semantic action numbers have routines associated with them that manipulate information on a semantic stack and/or generate some form of code (which in this compiler is the target code).

Some of the semantic actions require information from the lexical analyzer. For example, one semantic action specifies that an identifier should be pushed onto the semantic stack. The identifier, in the form of a token string, is obtained directly from the lexical analyzer. Thus the semantic analyzer's task is described by the diagram in Figure 1(c).

No tools were available to automatically construct this component. Thus the semantic analyzer was entirely hand-coded. It consists of action routines to carry out semantic actions, symbol table routines to store attributes of identifiers and code generation routines to output the simple target code.

G.2.3 **The Distributed Compiler**

Having defined these basic components, we next consider the task of putting them together to form a complete compiler. Because of the high degree of modularity and the simple interfaces, an obvious way to have the components work together is to let them be separate processes that communicate with each other by sending messages. This approach was adopted to form a distributed Jigsaw compiler. The communication links for this compiler are those that are formed by fitting the previous functional mapping diagrams together. Figure 1(d) illustrates the final structure.

Notice that the components fit together in a pipeline fashion, where the input goes through successive transformations on its way to the finished product, namely the target code. Thus the only interdependency is that processes must be fed information from their predecessors. This enables the processes to be implemented efficiently as communicating distributed processes, where each process runs on a separate computer in the aforementioned local area network.

Ideally, we would like a system where each component could continue running as long as it had input to process. For instance, the lexical analyzer could run continuously and keep sending out token numbers and strings until it encountered the end of the input file. Its output would be accumulated in the message queues at the syntactic and semantic analyzers. However, since the message queues are finite, if the lexical analyzer runs faster than the other two components it will eventually be forced to wait for them, thus destroying the valuable inherent parallelism. Therefore, steps need to be taken to tune or optimize the performance of these cooperating processes.

There are three basic factors to be taken into consideration. First, the speeds of components need to be balanced. Assume that the total amount of time required by the components, is t_1, t_2, and t_3 respectively, where $T = t_1 + t_2 + t_3$ equals the total compilation time for a serial implementation. Then the maximum possible speed up factor would be

$$f_s = \frac{T}{\max \{t_1, t_2, t_3\}}$$

Clearly, the best we can do is have $t_1 = t_2 = t_3$, in which case $f_s = 3$, that is, the parallel version would potentially be 3 times faster than the serial version. Notice that this factor needs to be considered in the initial partitioning of the problem. A compromise may be required between the goal of balancing and that of conceptual separability or modularity. In the case of our compiler, we discovered that there was little conflict between these goals.

The second factor to be considered is the size of what we called the "intervals of independence". These intervals refer to the amount of time the component processes can run independently (that is, without sending or receiving messages). They are important because communication and waiting delays are avoided during these intervals. This argues for us making the intervals large. However, making the intervals too large, makes the time to fill the pipeline significant, and may result in components having to wait too long for their input. Basically, we are trying to optimize between two separate types of serialization. The first type of serialization is illustrated by the time diagram for a single-pass compiler in Figure 2(a), where dt_1, dt_2, and dt_3 are the single step processing times for the lexical, syntactic, and semantic analyzers, respectively. The second type is illustrated by a hypothetical multi-pass compiler that communicates using memory rather than secondary storage. The time diagram for it would look like the one in Figure 2(b).

The ideal design of a distributed compiler would result in overlapping execution of the three components, as diagrammed in Figure 2(c). If the intervals of independence are too small, then the behavior will approach that of a single pass compiler, where for example, the syntactic analyzer would wait for a message from the lexical analyzer, quickly do its processing, send results to the semantic analyzer (waiting if its queue is full), and finally go back to waiting for a message from the lexical analyzer. Making the intervals of independence larger provides the advantage that more processing will be done within each interval relative to the amount of time spent waiting and transmitting messages. However, making the intervals too large will result in, say, the syntactic analyzer having to wait too long to get information from the lexical analyzer before it could proceed. Clearly the optimal solution depends on the characteristics of the network, the computers, the operating system and the individual processes themselves.

A simple way to control the size of the intervals is to adjust the amount of information sent in each message. For example, the lexical analyzer sends token numbers to the syntactic analyzer and token strings to the semantic analyzer. The lexical analyzer saves these numbers and strings in internal buffers. Only when it has filled one of the buffers, does it send a message (the contents of the full buffer) to one of the other processes. This buffering mechanism enables the intervals of independence to be increased and the number of individual messages passed to be reduced.

The third and final factor to be considered is that of balancing the flow of messages between the processes. That is, for each communication link, we want the number of messages that are sent to be approximately equal. As an example, again consider the lexical analyzer. Observation shows it sends more than twice as many token numbers as it does token strings. Suppose it has just filled its buffers and sent them out.

N: token number buffer n_1 n_2 n_3 n_4 n_5 n_6

S: token string buffer s_1 s_2 s_3

The syntactic analyzer will receive the N buffer, process it and eventually send out a semantic action number buffer.

A: action number buffer a_1 a_2 a_3 a_4 a_5 a_6

Now, if the token numbers that correspond to the token strings in S were in N, then action numbers that tell what to do with the strings in S will be in A. Thus balancing will result in smooth information flow where waiting times will be small.

As an example of what can happen when the flows are not balanced, assume the message queue size equals 2 (as is the case on our system), and that the N buffer holds 8 elements and the S buffer holds 1. After about 7 tokens have been read, 3 token strings will have been sent. However, since no token numbers have been sent, the token strings will not have been received, so that the lexical analyzer will be blocked indefinitely waiting to send the third token string. Thus, as this extreme case illustrates, unbalanced message flows can even result in deadlock.

Having already taken care of the first factor, we were left with the task of choosing the buffer sizes to optimize the second and third factors. We first attacked the third factor by setting the S buffer size at 10 token strings, and testing the response times for various N and A buffer sizes. For test programs of 600 lines of code, the responses for N,A = 20 ± 2 were 1:02 minutes and the response increased slowly as N,A moved outside this range. Thus the optimal ratio of buffer sizes, N:S:A, was about 2:1:2. We then attacked the second factor by holding this ratio fixed and varying the magnitude of the buffers. For test programs of 600 lines of code, the response times and processor times are reported in Table 1.

Since our network limited messages to at most 256 bytes, we were unable to test larger buffers (20 token strings requires 240 bytes). The data in the table clearly show that increasing the size of the buffers when the buffers are small provides dramatic improvements. However, once the buffers sizes (N,S,A) reach (12,6,12), the response curve becomes rather flat and remains so through the rest of the range tested. We thus picked the minimum point of this curve as the values to set the buffer sizes for the distributed compiler, that is we let

N hold 20 token numbers (2 bytes/number),
S hold 10 token strings (12 bytes/string),
A hold 20 action numbers (2 bytes/number).

G.2.4 Single-Pass Version

A single-pass version of the compiler was also constructed to be used as a standard of comparison in evaluating the performance of the distributed compiler. It uses the exact same components as the distributed version. However, instead of having them communicate by sending messages, they communicate using procedure invocation, with the syntactic analyzer acting as the driver. When it needs a token it calls the lexical analyzer and similarly when it has determined that a semantic action needs to be performed it calls the semantic analyzer. Thus the single-pass version is implemented as a sequential process that runs on a single computer.

G.3 THE EXPERIMENT

The point of this case study was to test the feasibility of distributed compilation. As described above, distributed and single-pass versions of the same compiler were constructed, differing only in global control structures. Thus the only factor which could account for any performance differences is the method of communication between the components and the parallelism it allows. The distributed version communicates by sending messages; the single-pass version communicates by procedure invocation and parameter passing. Therefore, if we consider the total amount of processing time consumed by the compilers in compiling the same program, it seems likely that the distributed version would require a little more time, as message passing requires more overhead than procedure invocation. However, this factor will be unimportant if significant parallelism can be achieved in the distributed version, thereby substantially reducing total response time. Thus we will compare the response time of the distributed version to that of the single pass version.

For this experiment three Prime computers in our local area network were used, two Prime P550's, systems A and B, and one Prime P400, system C. These systems are compatible with respect to machine instructions and operating system, but system C is a little slower. The components of the distributed compiler were placed as follows: lexical analyzer on system A, syntactic analyzer on system B, and the semantic analyzer on system C. The single-pass compiler was run on system A.

Specifically, the following tests were performed. First the two compilers were tested under completely unloaded conditions; that is, the only other load on the system was due to the operating system. These conditions are of interest as they indicate the maximum possible benefit that can be achieved by distributing a compiler. For this case, the two compilers were run on Jigsaw test programs ranging in size from 25 to 1200 lines of code. For each program the response time and processor (cpu) times used were recorded. The results are shown in Table 1 and Figure 1. All times in the tables are in units of seconds, except for the longer response times, which are expressed as 'minutes:seconds'.

Secondly, the compilers were tested under moderately loaded conditions, where approximately five people were using each system. Although, this does not constitute a well controlled experiment, it does give an indication of the trend in the response times as the load factor is increased. The results for this case are shown in Table 2.

G.4 INTERPRETATION

The data reported in Table 1 and Figure 1 clearly indicate that distributed compilers can achieve significant improvements in response time over traditional single-pass compilers. Indeed, for programs of more than 100 lines of code the distributed compiler was 2 to 2.5 times faster than the single-pass compiler. For example, for a program of 1200 lines, the single-pass compiler took 4 1/2 minutes, while the distributed compiler took only 2 minutes. This ratio obviously would have a considerable impact when compiling even larger programs. For programs smaller than 100 lines, we see that use of the distributed version is still advantageous, although less overwhelmingly. This loss of advantage can be accounted for by the fact that the distributed version has fixed overhead involved in setting up the virtual circuits and filling the pipeline (i.e., the syntactic analyzer cannot start until the lexical analyzer has filled a buffer with token numbers).

The message buffer sizes, used to control the frequency of interactions between components of the program, turned out to be a very important performance factor. Without buffering, the best speed-up factor obtained was about 1.4 (as opposed to 2.5 with the reported buffer sizes). As buffering was introduced and the sizes increased, performance at first improved rather dramatically. The sizes used reflect a leveling off point in a graph of performance versus buffer sizes.

Let us now examine the relationship between the performance of the individual components and that of the distributed compiler. These components are roughly equal in the processing time that they consume, with the syntactic analyzer consuming the largest portion. Since tne maximum potential speed-up factor f_s is limited by the slowest component, it is very important in distributed programs to concentrate performance improvement efforts on such components. Note that the effects of improving the slowest component are much more dramatic with distributed programs. Speeding up the parser in our compiler by 15% would probably improve the performance of the distributed version by close to 15%, but it would improve the single-pass version by less than half of that factor.

To determine the amount of processing time needed to distribute the compiler, we can compare the processing time used by the single-pass version with that of the sum of the processing times of the distributed components. The difference between the sum column in Table 1 and the processor time column seems to be made up of two components: a fixed overhead of about 3 seconds, and a proportionate increase of about 5%. The fixed overhead results from the time necessary to initially set up the virtual circuits, and the proportionate increase is caused by the replacement of procedure invocation with message passing. Compared with the positive effects of parallelism, these negative effects are not significant.

Finally, we consider the data from the tests where the system was moderately loaded. Again the distributed version was faster, but the speed-up factor was much smaller, about 1.5. Thus it is apparent that the distributed compiler was more adversely affected by the load on the system than the single-pass version. This result is expected, since the speed of the distributed version depends on the smooth flow of information between the processes and loading the system increases the competition for time slices, thereby increasing the probability that a message will be sent to a process in a wait state. Hence, loading the system has the potential to increase effective message transmission time and thus slow down the individual components. A possible remedy for this problem would be to have a sophisticated distributed operating system to oversee the operation of the network. If such an operating system had knowledge of the running characteristics and the communication needs of the processes in the network, then it could possibly schedule processes so as to enhance the smooth flow of information.

G.5 CONCLUSION

The significance of this study is not merely that it demonstrates potential benefits of distributed compilation, but rather that it suggests that at least some class of programs traditionally executed sequentially can be successfully partitioned as distributed programs. We believe this class includes not only compilers, but any program which operates as a sequence of transformations on its input to produce some output. Such programs map nicely to distributed computing systems which provide a pool of assignable, general-purpose processors. In such a system, computers could be allocated to the component processes of the compiler (or other program) for the entire length of the compilation, thus achieving the ideal conditions of the unloaded tests. It should, therefore, be possible to achieve speed-up factors of the magnitude we observed.

It should be noted that it was quite easy to transform a traditionally-structured compiler into a distributed one. Using message passing as the means for communication between components requires only thoughtful design of component interfaces. No complex synchronization protocols need be devised. The message passing corresponds to simple procedure invocations in the traditional program. Again, this should hold for a broader class of programs.

We currently observe the development of systems which commonly provide conditions similar to those of the unloaded tests. With personal computers and small business systems becoming inexpensive, networks of them are proliferating. In such networks, the use of human resources rather than the use of cheap processors is optimized, so processors are, on the average, lightly loaded and thus are available for use by distributed programs. Another reason why such a system is a good candidate is that the processors are not very fast, so that use of parallelism is particularly desirable. Furthermore, the memory capacity on these systems may be limited, making a distributed program advantageous, since its component processes are naturally smaller than the entire program would be if it were monolithic.

Finally, we must consider the system dependencies of our results. The success of the distributed compiler depends on message delay times being small. Its loss of advantage as a load appeared on the system is direct evidence of this dependency. Thus our results are most applicable to high-bandwidth local area networks which can provide the necessary speed of message delivery. The introduction of the concept of buffering messages within program components as a tuning technique makes our results less dependent on the more detailed system characteristics. With proper use of buffer sizes, it is likely that our results could be matched on a variety of distributed systems connected by local-area networks.

G.6 TABLES AND FIGURES

TABLE 1

BUFFER SIZE TEST RESULTS

BUFFER SIZES			RESPONSE TIME (min:sec)	SCANNER CPU TIME (sec)	PARSER CPU TIME (sec)	SEMANTIC CPU TIME (sec)
N	S	A				
1	1	1	2:23	57	82	47
4	2	4	1:36	46	58	35
8	4	8	1:13	42	55	32
12	6	12	1:06	42	53	31
16	8	16	1:03	41	53	31
20	10	20	1:02	40	52	31
24	12	24	1:03	40	52	31
28	14	28	1:03	40	52	31
32	16	32	1:04	39	51	31
36	18	36	1:04	39	51	31
40	20	40	1:04	39	51	31

TABLE 2

TIMING DATA FOR RUNS ON UNLOADED SYSTEM

	SINGLE PASS COMPILER		DISTRIBUTED COMPILER				
PROGRAM	RESPONSE	PROCESSOR	RESPONSE	SCANNER	PARSER	SEMANTIC	TOTAL CPU
SIZE	TIME	CPU TIME	TIME	CPU TIME	CPU TIME	CPU TIME	TIME
(lines)	(min:sec)	(sec)	(min:sec)	(sec)	(sec)	(sec)	(sec)
25	7	5	5	2	3	3	8
50	13	9	8	3	5	5	13
100	25	20	13	7	10	8	25
200	47	39	22	14	19	12	45
300	1:08	58	32	20	26	17	63
400	1:31	78	42	27	35	22	84
500	1:54	96	52	34	44	26	104
600	2:17	115	1:02	40	52	31	123
700	2:39	135	1:12	47	61	36	144
800	2:59	154	1:21	53	70	40	163
900	3:19	172	1:30	60	77	45	182
1000	3:42	192	1:40	67	86	49	202
1100	4:06	212	1:50	73	94	53	220
1200	4:30	230	1:59	79	102	58	239

TABLE 3

TIMING DATA FOR RUNS ON LOADED SYSTEM

	SINGLE PASS COMPILER		DISTRIBUTED COMPILER				
PROGRAM	RESPONSE	PROCESSOR	RESPONSE	SCANNER	PARSER	SEMANTIC	TOTAL CPU
SIZE	TIME	CPU TIME	TIME	CPU TIME	CPU TIME	CPU TIME	TIME
(lines)	(min:sec)	(sec)	(min:sec)	(sec)	(sec)	(sec)	(sec)
25	11	5	10	2	5	5	12
50	17	9	16	4	6	6	16
100	35	20	28	8	12	8	28
200	1:10	40	49	14	20	14	48
300	1:40	59	1:08	20	28	19	67
400	2:27	79	1:44	28	41	24	93
500	3:04	98	2:09	35	50	30	115

FIGURE 1

COMPILER STRUCTURE

```
                            ______________
                           |              |
                           |   Lexical    |  -----> token numbers
         characters -----> |   Analyzer   |  -----> token strings
                           |______________|
```

(A) LEXICAL ANALYZER TRANSFORMATION

```
                            ______________
                           |              |
                           |   syntax     |
      token numbers -----> |   analyzer   |  -----> action numbers
                           |______________|
```

(B) SYNTAX ANALYZER TRANSFORMATION

```
                            ______________
                           |              |
     action numbers -----> |  semantic    |
      token strings -----> |  analyzer    |  -----> instructions
                           |______________|
```

(C) SEMANTIC ANALYZER TRANSFORMATION

```
              ___________          _____________           ___________
     source  | lexical   | tokens | syntactic   | actions | semantic  | target
     ------->| analyzer  |------->| analyzer    |-------->| analyzer  |------->
              -----------          -------------           -----------
                   |                                            ^
                   |              tokens strings                |
                   ----------------------------------------------
```

(D) OVERALL COMPILER STRUCTURE

FIGURE 2

TIMING DIAGRAMS

```
lexical    |---|        |---|           |---|
            dt1          dt1             dt1

syntactic      |---|        |---|    ...    |---|
                dt2          dt2             dt2

semantic           |---|        |---|           |---|
                    dt3          dt3              dt3
          ----------------------------------------------->
                                                    time
```

(A) SINGLE PASS COMPILER

```
lexical   |---|---|...|---|
           dt1 dt1     dt1

syntactic                 |---|---|...|---|
                           dt2 dt2     dt2

semantic                                  |---|---|...|---|
                                           dt3 dt3     dt3
          -------------------------------------------------->
                                                        time
```

(B) MULTI-PASS COMPILER

```
lexical       |-----|-----|...|-----|
               dt1   dt1       dt1

syntactic            |-----|-----|...|-----|
                      dt2   dt2       dt2

semantic                    |-----|-----|...|-----|
                             dt3   dt3       dt3
          ----------------------------------------------->
                                                    time
```

(C) IDEAL DISTRIBUTED COMPILER

REFERENCES

[El-D79] El-Dessouki, O., W. Huen, and M. Evans, "Towards a Partitioning Compiler for a Distributed Computing System," First International Conference on Distributed Computing Systems, October 1979, pp.296-304.

[Ensl78] Enslow, P.H., "What is a 'Distributed' Data Processing System?" Computer, January 1978, pp.13-21.

[Gord79] Gordon, R.L., W.W. Farr, and P. Levine, "Ringnet: A Packet Switched Local Network with Decentralized Control," Fourth Local Network Conference, August 1979.

[Huen77] Huen, W., O. El-Dessouki, E. Huske, and M. Evans, "A Pipelined DYNAMO Compiler," Proceedings of the 7th International Conference on Parallel Processing, August 1977, pp.57-66.

[Kieb81] Kieburtz, R.B., "A Distributed Operating System for the Stony Brook Multicomputer," Second International Conference on Distributed Computing Systems, April 1981, pp.67,79.

[Neil79] Neilson, P.A., The Primenet Guide, Prime Computer, Inc., June 1979.

[Zimm81] Zimmermann, H., J.S. Banino, A. Cariston, M. Guillement, and G. Morisset, "Basic Concepts for the Support of Distributed Systems: The Chorus Approach," Second International Conference on Distributed Computing Systems, April 1981, pp.60-66.

APPENDIX H

Architecture for a Global Operating System

M. S. McKendry, J. E. Allchin, and W. C. Thibault

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

ABSTRACT

Global operating systems are suited to distributed, local-area network environments. A decentralized global operating system can manage all resources globally, relying on functional requirements for resource allocations, rather than the relative physical locations of the resource allocation mechanism and the resource itself. Among the advantages of global operating systems are the ability to use idle resources and to control the environment as a single cohesive entity. This paper introduces an architectural approach to supporting decentralized global operating systems. The approach addresses the problem of managing distributed data by incorporating specialized data management facilities in the kernel. This data management is especially useful to the operating system itself. A capability-based access scheme provides flexible control of resources and autonomy. The approach is being utilized in the Clouds operating system project at Georgia Tech.

1. Introduction

Increasingly, the computers within an organization consist of a heterogeneous group of machines linked by high speed (yet relatively inexpensive) local area networks. Mainframes, office stations, scientific workstations, personal machines, and even real-time controllers may participate in this internetwork of machines. In many such environments, it is desirable that users view the entire decentralized resource pool as a single computing resource. Users could then be shielded from multiple user interfaces and relieved from having to decide how best to accomplish objectives using the available resources.

In effect, we want to hide the decentralization of the resources from users, so they do not have to be concerned with which particular resources are used to accomplish their objectives. Furthermore, we need to group resources for the purposes of autonomy and protection, whether the resources in question are files, machines, or logical services. For a completely general facility, support for arbitrary (possibly intersecting) groupings of resources is needed.

The concept of a global operating system embraces these requirements [Jens82] [Lamp81]. In a global operating system, all resources are managed and allocated globally. The physical locality of a resource - whether local or remote, for example - is not inherently a part of the decision process. Decisions can be made solely on the basis of cost factors and logical constraints, rather than physical locality. For example, assignment of a processor to a process might be performed on the basis of code file location, expected I/O, expected CPU utilization, and current processor and network loadings.

Transparency appears to be a key quality in the architecture of a decentralized global operating system. This has two main aspects:

Resource Access: Boundaries between machines should be transparent during access to resources if desired. This is provided by many existing inter-process communication mechanisms (e.g., [Rash81]).

Decision Apparatus: The relative physical locations of a policy apparatus and the resources it controls should be transparent to the policy apparatus. To do this efficiently may also require that data describing resources be accessible independently of its physical location.

Given a system supporting transparency in its access to resources and its decision apparatus, construction of arbitrary groupings of resources is simplified, as is allocation of resources on a global basis. These qualities are not necessarily easy to achieve, however. For example, for a decision apparatus to operate independently of the resources it controls, it may need the ability to "back-up" if a remote processor fails after

This work was supported in part under contracts from the Office of Naval Research, N00014-79-C-873 and the USAF Rome Air Development Center, F30602-81-C-0249.

*This paper is to appear in IEEE INFOCOM 83, San Diego California, April 1983.

a decision is made. Furthermore, decentralization requires that decisions be made on the basis of heuristics or probabilities, using out-of-date or inconsistent data [Jens81].

In this paper we introduce a structural approach, or architecture, for a system designed to support decentralized global operating systems. In this approach, which is still being refined by the **Clouds** project at Georgia Tech, we take the view that kernels provided to support the operating system on each machine should provide the uniformity and transparency required. Using the object model (data abstraction) as a basis, we intend to provide a sophisticated database management system within kernels, but leave the specific details of aspects such as synchronization, recovery, and atomicity to the designers of the operating system (the *client* system) that utilizes the kernels. The kernels provide mechanisms to implement these requirements without specifying policy of how the mechanisms should be used.

The paper discusses the goals (Section 2), requirements (Section 3), and architectural concepts (Section 4) for this approach. The approach is being implemented in the **Clouds** operating system, which will run as a native operating system on all participating machines. The environment assumed is a group of machines connected via an internetwork of high-speed (inexpensive) local area networks. For practical reasons, **Clouds** is being implemented initially on homogeneous machines: the Three Rivers Perq [3RCC82]. The Perq is a scientific workstation of minicomputer capacity. We are using 10 Mb/sec Ethernet technology for the local area networks.

2. Goals

The environment we are considering has been characterized as a Fully Distributed Processing System (FDPS). According to Enslow [Ensl78], an FDPS exhibits the characteristics of a multiplicity of resources, physical distribution, unity of control, network (location) transparency, and component autonomy. The primary goal of the architecture is to support a reliable, unified computing environment so that these characteristics can be fully realized. In this sense, the architecture could form the basis of a distributed timesharing system. While such constraints as "one user per workstation" might hold at various times (making some decisions trivial), the system can take responsibility for all selection and assignment of resources to users. Note that an architecture can form a basis for many different systems, not just the traditional "general purpose" systems. Systems supported might include distributed process control, for example.

Two secondary goals are apparent. Firstly, as in conventional operating systems, the architecture should facilitate high resource utilization within performance constraints such as response time or total cost. Secondly, it should help users to access or create services that are common to conventional systems and services that are peculiar to distributed systems. Many of the requirements of application programs for these services are shared by the operating system itself, which attempts to provide reliable service despite the possibility of machine and network failures.

Our final goal is to provide **tunable autonomy**-- dynamically configurable domains of resource control. A tunably autonomous system could provide a variety of resource allocation schemes, varying from highly autonomous systems, to the equivalent of tightly-coupled multiprocessing, where decisions affecting one machine can be made by any other machine. Tunable autonomy facilitates construction of logical resource groupings at multiple levels. For example office, department, division, company, and inter-company levels might be established, with differing autonomy and sharing constraints at each level.

3. Requirements

Two issues, data management and resource management, stand out as fundamental to the Clouds architecture. Only the **mechanism** for resource management has to be provided by the architecture, but requirements for effective and efficient resource **policy** must be given consideration.

3.1 Data Management

Data management is a ubiquitous problem in computer programs. The problem is particularly severe in distributed systems. Conventional operating systems contain a plethora of structures representing system state. A global operating system must do the same, and must also deal with additional issues including increased concurrency, partial configurations, and failures and associated recovery. Each node must be able to access both local and remote data. Considerable research has been expended studying general problems encountered in managing distributed data, but little attention has been paid to problems peculiar to decentralized real-time systems. As a consequence, special solutions tend to be reinvented for each application system (e.g., [Birr81]) and for each part of the operating system.

Conventional database research usually assumes an application environment in which data consistency (defined through serializability) is of prime importance. However, serializability is applicable mainly when independent activities compete for resources--it is not suited to cooperation between processes, such as is achieved through message-based interprocess communication. Thus it is necessary to deal with an orthogonal structure of atomic actions which can be the units of recovery and concurrency. Further, one of the attractions of serializability is its simplicity in the absence of semantic information concerning data accesses. This simplicity is obtained at the cost of concurrency though, and in an operating system considerable semantic information is available concerning both the accesses and the operations on the data stored. This information can be exploited to improve concurrency, and thus availability and performance.

Due to the scope of distributed data management requirements, data management takes a prominent place in our architectural approach. Distributed data management can be made quite sophisticated, supporting failure atomicity, consistency conditions including (but not limited to) serializability, creation and location of data objects, synchronization of access to data, replication of data, and invocation of operations on data. It can also provide a basis for system synchronization [Allc83].

3.2 Resource Allocation

Consideration of resource allocation requirements is critical to the success of a global operating system architecture. Ideally, an operating system should take complete responsibility for the allocation of resources to a user, if the user so desires. The architecture must provide facilities to support this allocation control.

Consider the example depicted in Figure 1. A user directly connected to machine A, wishes to run a program p, which requires serial access to file b currently on machine B, and random access to file c on machine C. The operating system must determine which machine should provide the computational resources necessary to run p, and whether any files should be relocated beforehand. Factors involved include user-specified. constraints (e.g., 'fast', or 'cheap'), optimization of particular resources (e.g., device channels, processor time, network bandwidth), current loads, and interactions with other programs. The program p may run on machine A, but unlike many conventional distributed operating systems (e.g., [Rede80]), this is determined heuristically at the time of request; it is not a default. Thus, we are advocating a more "intelligent" approach to resource allocation.

3.2.1 Tunable Autonomy

The term tunable autonomy characterizes the ability to construct arbitrary logical groupings of resources for the purposes of management. A single group of resources, or resource pool might contain one or many resources, may intersect or contain other resource pools, and can extend across arbitrary machines. Once a resource pool is defined, decisions concerning the resources within it can be made by any processor (or program) granted control over that pool, regardless of physical location. Within this framework, allowing each physical machine to be autonomous is a constraining case, but not the only possible case.

3.3.2 The Decision Process

While a decentralized global operating system should have considerable freedom to assign resources, its ability to assign them effectively will be limited by out-of-date and incomplete information. Machine states change rapidly, so a perfectly consistent description of the state of an entire system cannot be achieved without paying a high performance penalty. As a result, reaching a decision involves a more heuristic or probabilistic approach than in a conventional, single processor system. More historic data to assist prediction can help, though; a system could keep extensive statistics of past activity. For example, if file c in Figure 1 is known to be small, and program p typically makes extensive random access to c, the operating system might decide to either relocate c to machine A, or to run p on machine C.

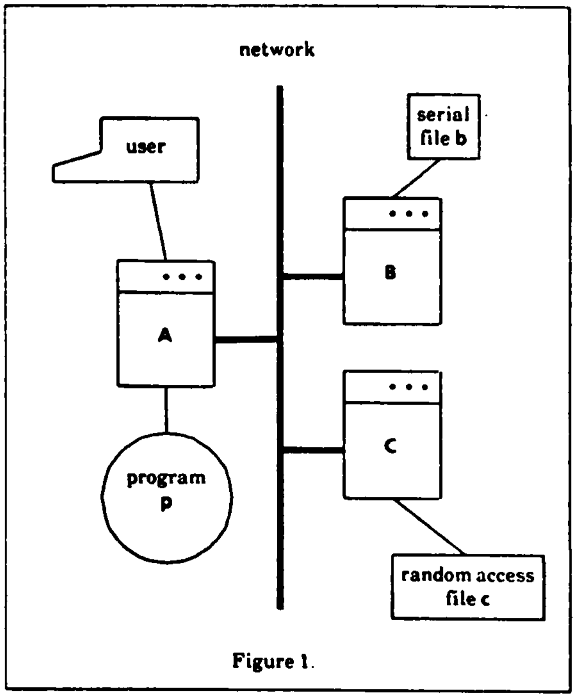


Figure 1.

A second concern of the decision process pertains to decisions that have been made, but cannot be implemented. To a limited extent, this can be avoided by preclaiming resources, but the problems of failures cannot be avoided. If a machine fails, it may be impossible to implement a resource policy decision. In this circumstance, the policy apparatus must try again, basing the next attempted decision on more recent information.

4. Architectural Directions

In many operating systems for distributed environments, a kernel provides primitive operations for inter-process communication (IPC) via messages (e.g., [Rash81]). We intend to take a radically different approach, however, adopting the dual structure [Laue79]. The architecture supports processes, (passive) objects, and invocation of operations on objects. Actions (groupings of invocations) provide a basis for reliability. The primitive facilities provided

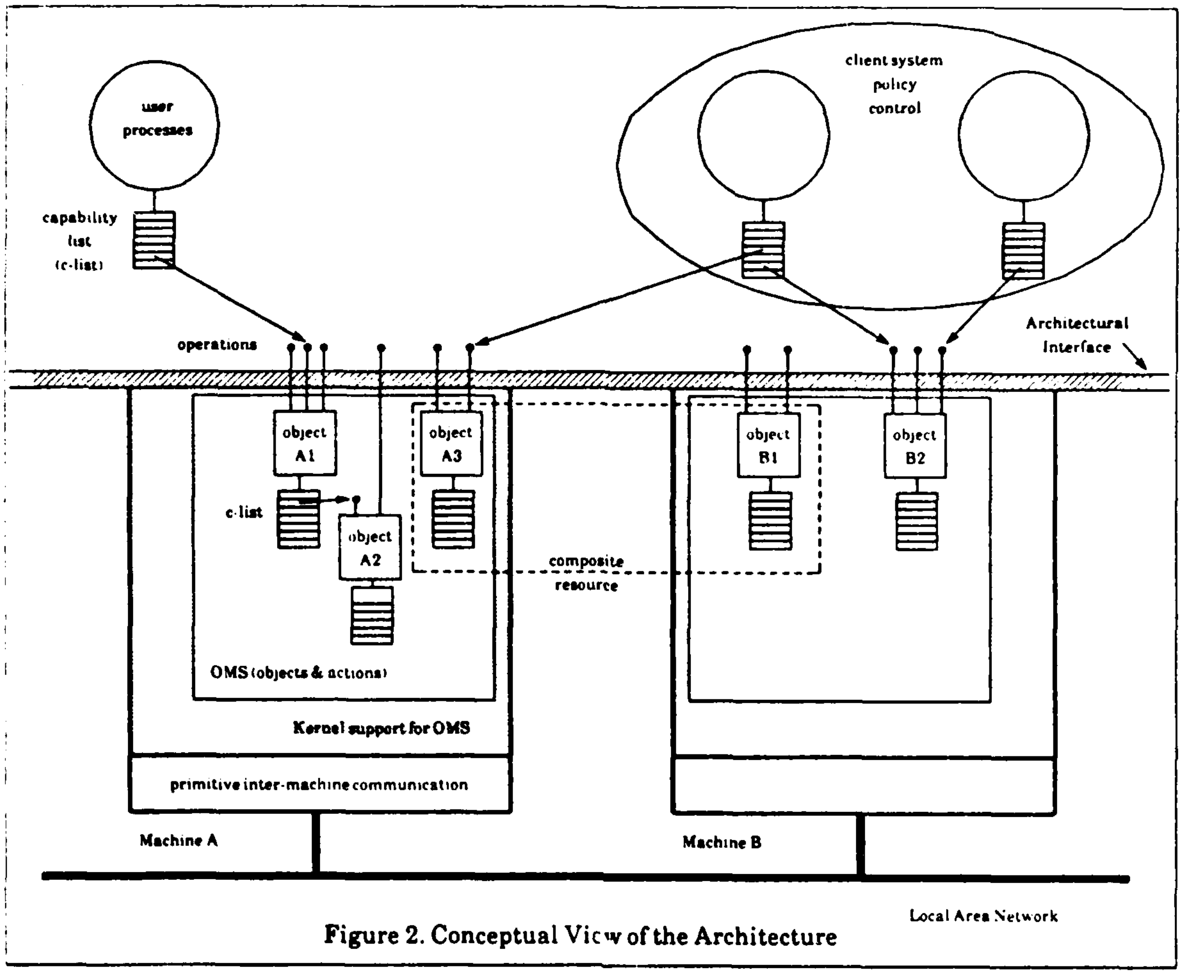


Figure 2. Conceptual View of the Architecture

by the architecture use a variety of remote procedure call semantics which vary in dimensions such as reliability and asynchrony [Spec82]. A section of the kernel, called the **object management system** (OMS), implements calls on objects. Access from processes to objects is via **capabilities**, which are protected system names managed by the kernel. Capabilities can be passed as parameters when operations are invoked; they can also be returned in a fashion similar to function values.

The kernel will provide the lower levels of a functional hierarchy. At the lowest level is the hardware, which we consider to include access to a local area network. At the level above the hardware is the primitive inter-machine communication used by the individual kernels to communicate with one another and to maintain the object management system. Data and process management mechanisms then complete the kernel and the architecture. Thus, the combination of services provided by individual kernels implements the architecture for the complete system. Above the kernels are client levels to provide policy for the architecture. Finally, user processes implement applications. A pictorial representation of the architecture is shown in Figure 2.

Kernels run processes and maintain objects. However, because this is at the instigation of higher level software, conventional message-based inter-process communication is considered to be part of the client system, as is the resource allocation policy apparatus. These characteristics are, in fact, highly desirable features, because they allow client-specifiable inter-process communication, and permit a high degree of flexibility in resource policy. A method of using objects to implement interprocess communication via messages is shown in Figure 3.

4.1 Data Management

The object management system consists of two primary components: objects and actions; both are user definable [Allc82] [Allc83]. Objects are passive entities (modified abstract data types) which are accessible only through interface procedures that define operations on the objects. Actions are ordered collections of operations on objects which require certain properties (e.g., failure atomicity) to hold throughout the life of the action. Both object recovery and synchronization between actions are controllable by the object itself (i.e., programmed within the object).

Thus, for example, weaker forms of consistency are allowed, depending on the semantics of an object. Actions can be carried out by single processes or cooperating processes.

An extended Pascal language allows object classes to be defined, and the object management system supports objects at runtime. Once created via object classes, object instances are controlled through requests to the kernel using OMS primitives, such as *create object*, *destroy object*, *create action*, *destroy action*, *commit action*, and *invoke operation*. Object classes are exported, so object variables can be typed automatically in a manner similar to Pascal pointers. Thus, the object management system can be viewed as a globally distributed heap containing long-lived objects. For transparency, all actions communicate only with the OMS at the node where the process implementing the action resides (not shown in the conceptual view of Figure 2). The OMSs, in turn, communicate using specialized protocols for inter-machine communication. They cannot use the general IPC facility, because they form its basis.

A primary goal of the OMS is support for network transparency, wherever desired. This transparency is provided by making operation invocation uniform, regardless of whether the subject of the operation is on the same machine as the client invoking the operation. The operating system is thus free to distribute processes and objects without their knowledge (unless specifically directed not to do so). Thus, languages for distributed computing, such as PRONET [Macc82] or Argus *[Lisk82] can be well supported.*

4.2 Resource Management

Because this paper is concerned primarily with describing an architectural approach, we will not discuss resource management in detail. The basic approach is to use the capability-passing mechanisms provide by the object management system to construct capability managers [Kieb78]. Due to the transparency implemented by the architecture, capability managers function independently of their location in the system. Since all invocations of operations are via capabilities and possession of a capability is taken as permission to invoke an operation, there is no structural association of particular machines with particular decisions. Any object that possesses a capability to implement policy decisions can implement those decisions. For example, each machine might contain a process-management object whose function is to instantiate and destroy processes on that machine. Any object that possesses capabilities for the operations of this process-management object can then create and destroy processes on that particular machine. The capability-based access scheme makes it unneccessary to have a "special state" for resource managers--any object can become a resource manager, thus making it possible to construct arbitrary pools of resources independently of machine boundaries. Of course, a choice of appropriate capability-passing primitives is critical to the success of this approach [Snyd81].

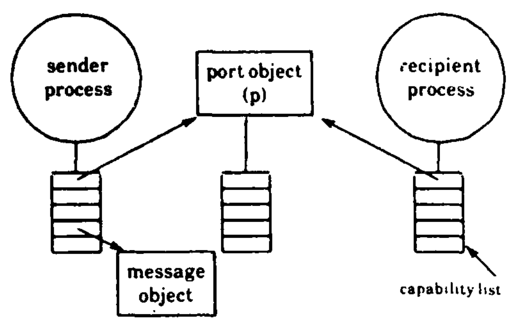


Figure 3. Implementation of Ports

5. Summary

This paper has introduced an architectural approach for decentralized global operating systems in an environment of machines connected via an internet of high-speed local area networks. A global operating system manages all resources globally, without making distinctions between local and remote resources. One characteristic desirable of such systems is **tunable autonomy**: the ability to construct arbitrary logical groupings of resources for the purpose of management. Such groupings are independent of machine boundaries.

A major motivating factor in the design of the architecture is the need for reliable data management in the low levels of the system. This can be achieved efficiently by making some constraints, such as serializability, optional according to particular needs. Requirements for global resource management also motivate the operating system architecture.

The architecture described provides processes and objects; invocation of operations on objects is performed through capabilities. Objects are maintained by the object management system, a component of the kernel. Despite its integration into a low level of the operating system, the object management system is quite sophisticated, providing an action environment in which actions may initiate, commit, and abort, with appropriate effect on object states. To assist performance and reliability, the object management system supports variable recovery, synchronization, replication, and consistency conditions including (but not limited to) serializability.

From the base architecture, message based inter-process communication and a resource policy apparatus can be constructed. The capability-based invocation of operations on objects makes it possible to construct resource managers as capability managers. Arbitrary objects can manage capabilities, depending on the capability-passing primitives of the object management system to provide the neccessary access control. Thus, resource pools can be constructed dynamically, and can exist independently of machine boundaries.

6. Acknowledgements

The authors would like to acknowledge contributions to Clouds work by B. Maccabe, R. LeBlanc, P. Enslow, N. Griffeth, M. Merritt, N. J. Livesey, R. Mays, and other participants in the Georgia Tech FDPS program.

7. References

[3RCC82] Three Rivers Computer Corp., Perq System Software Reference Manual, Pittsburgh, Pa., May 82.

[Allc82] Allchin, J.E., M.S. McKendry, "Object-Based Synchronization and Recovery,".Georgia Institute of Technology GIT-ICS-82/18, September 82.

[Allc83] Allchin, J.E., "Synchronization and Recovery in Distributed Systems," Georgia Institute of Technology, Ph.D. Thesis, in preparation.

[Birr81] Birrell A., R. Levin, R. Needham, and M. Schroeder, "Grapevine: An Exercise in Distributed Computing," Proc. 8th Symp. on Operating Systems Principles, ACM, December 81.

[Ensl78] Enslow, P. H., Jr., "What is a 'Distributed' Processing System?" IEEE Computer, pp. 13-21, January 78.

[Jens81] Jensen, E.D., "Distributed Control," In Lampson, B., Ed.,Distributed Systems - Architecture and Implementation, Berlin: Springer-Verlag, 81.

[Jens82] Jensen, E.D., "Decentralized Executive Control of Computers," Proceedings of the 3rd Int. Conf. on Distributed Computing Systems, Miami, 82.

[Kieb78] Kieburtz, R.B., and A. Silberschatz, "Capability Managers," IEEE Trans. on Software Engineering, Vol. SE-4, No. 6, pp. 467-477, November 78.

[Lamp81] Lampson, B.W., M. Paul, and H.J. Siegert, Eds., Distributed Systems - Architecture and Implementation, Berlin: Springer-Verlag, 81.

[Laue79] Lauer, H.C., and R.M. Needham, "On the Duality of Operating System Structures," In Lanciaux, D., Ed., Operating Systems: Theory and Practice, Amsterdam: North-Holland, 79.

[Lisk82] Liskov, B. and A. Scheiffer, "Guardians and Actions: Linguistic Support for Robust Distributed Programs," Symposium on Principles of Programming Languages, ACM, January 82.

[Macc82] Maccabe, A.B., and R.J. LeBlanc, "The Design of a Programming Language Based on Communication Networks," Proceedings of the 3rd Int. Conf. on Distributed Computing Systems, Miami, 82.

[Rash81] Rashid, R.F., and G.G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," Proc. 8th Symp. on Operating System Principles, ACM, December 81.

[Rede80] Redell, D. D., Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell, "Pilot: An Operating System for a Personal Computer," CACM, Vol. 23, No. 2, pp. 81-92, February 1980.

[Synd81] Snyder, L., "Formal Models of Capability-Based Protection Systems", IEEE Trans. on Computers, vol. C-30 No. 3., pp. 172-181, March 81.

[Spec82] Spector, A.Z., "Performing Remote Operations Efficiently on a Local Computer Network," CACM, Vol. 25, No. 4, pp. 246-260, April 82.

APPENDIX I

ALGORITHMS FOR MAINTAINING REPLICATED DATA USING WEAK CORRECTNESS CONDITIONS

James E. Allchin

Abstract

A suite of decentralized algorithms for maintaining distributed replicated data is presented. The algorithms do not necessarily achieve serial consistency, but they are adequate for many simple data storage problems in operating systems and realtime systems. Applications which appear well-suited to the suite include mail systems, naming servers, appointment calendars, certain types of file dictionaries, operating system load tables (*e.g.*, routing), and device state in distributed process control systems. The algorithms are robust and are intuitively easy to understand. The algorithms assume an unreliable network and tolerate node failures, network partitions, lost, duplicate, and out-of-order messages. Both goals for replicating data--high availability and rapid response time--are met by the algorithms. The basic algorithms use resolution tables to state the outcome of conflicts between concurrent actions. Each algorithm is oriented toward different application requirements and provides a different degree of message traffic overhead and availability. The efficiency of the algorithms depends on the acceptability of weak correctness conditions in the applications. The desired correctness condition is formally stated and the basic algorithm in the suite is proven correct

1. Introduction

The correctness condition usually applied to data storage systems states that the result of any set of transactions executed should be the same as some serial execution of that set of transactions. This *serializability*-based correctness condition assumes only that transactions execute correctly if run serially. Distributed systems containing replicated data require relatively complicated algorithms to achieve serial consistency and still obtain acceptable performance. These algorithms restrict concurrency in order to achieve consistency. In certain applications, however, serializability as a correctness condition is not required because the results for some class of non-serializable executions of transaction steps are correct. See, for example, [Lamp76], [Kung80], and [Garc80]. In addition, there are applications where even though the transactions *desire* to see a serial view, they will *accept* some class of non-serial views and consider these views correct as well. These applications will accept non-serial views in return for certain advantages not possible if strict serial consistency is enforced. Performance [Jens82, McKe83], availability [Fisc82], and simplicity [Oppe81] have been cited as encouragements to weaken correctness. Thus, there is an interesting class of application areas for which trading serial consistency for high availability, increased performance or algorithm simplicity is warranted.

There are many approaches for supporting copies of replicated data [Bern81]. Most of these maintain serial consistency. However, maintaining serial consistency across network partitions (caused by assumed failure of the communication system) defeats data availability, since at least one copy cannot be used and in the worse case, only one copy can be used. If weak consistency can be tolerated, then it is possible to overcome this problem. However, resynchronization of the data copies must still be addressed during node restart or network merge following a partition. Contending with these issues in an unreliable environment complicates the solutions still more. Algorithms which handle all of these issues tend to be complex, using a variety of expensive handshaking protocols. Establishing that these algorithms are correct, under all the possible failure conditions, is generally quite difficult.

In this paper, we present a suite of decentralized algorithms to maintain distributed replicated data with weak consistency. Algorithms from the suite can be customized to balance particular tradeoffs required in different application systems. The algorithms assume an unreliable network and tolerate lost, duplicate, and out-of-order messages, node failures and network partitions. Both goals for replication--high availability and rapid response time--are met by the algorithms. The basic structure of the algorithms depends on resolution tables to state the outcome of conflicts between concurrent actions. Each algorithm is oriented toward different application requirements and provides a different degree of message traffic overhead and data availability. The efficiency of the algorithms depends on the acceptability of weak correctness conditions in application systems.

Work which is similar to ours is discussed in [Fisc82], [John75], [Oppe81], and [McQu78]. Most of our problem formulation is based on Fisher and Michael [Fisc82]. Unlike their approach, though, we do not require each node to transmit the entire node's view of the database whenever communication occurs; only the changes to the view are sent, and regardless of the number of deletes and duration of failure, no node need maintain an unbounded list of changes relative to the database size (assuming the database is itself bounded). We believe that in particular cases (*e.g.*, small sized databases) passing the entire database is appropriate while in many other applications, this requirement is not acceptable.

Our work is particularly interesting because we use resolution tables which allow easy visualization of the conflict resolution strategy and we provide a formal proof of one algorithm from the suite (with other proofs following in a straightforward manner from the framework presented). Further, it is our belief that the suite of algorithms address a wide range of important problems in a clean and efficient manner. Allchin [Allc83] contains additional information on the desirability of supporting both serializable and non-serializable synchronization facilities in decentralized systems (in particular operating systems). Specific programmer-oriented tools for controlling atomic action synchronization and recovery are also presented.

2. Environment and Application Domains

The general environment assumed by the algorithms involve some collection of nodes arbitrarily connected via some communication system. The communication network and nodes are considered unreliable; that is, both may fail either partially or totally. Messages if delivered, however, must arrive ungarbled. That is, message corruption must be detectable. In addition, it is assumed that the nodes and the network do not manufacture messages which violate the protocol of the algorithm.

Each node contains a *view* of the entire database which may or may not be current depending on the state of communication activity. It is later proven that with sufficient reliability and assuming changes to the data cease, then all views will converge to contain the same data. That is, the views are mutually consistent [Thom79].

Each view consists of a set of *elements*, item names and associated values. Clients manipulate views by specifically referencing (via names) particular elements in the views. There are four operations which manipulate a node's view. These eventually alter the other remote views (if the changes are not superceded before the other nodes learn of the first change). The four basic operations are

Insert (x,y)	adds an element with name x and value y
Update (x,y)	replaces the value of the element with name x with the value y
Delete (x)	removes the element with name x
List (set of names)	returns an ordered pair of element names and values for all elements requested which exist in the local view at the time of the operation

Fischer and Michael [Fisc82] referred to a very similar environment and operation structure as a distributed dictionary problem. In fact, the main difference is that we include an *Update* operation This is an important change, not simply a trival extension. This is true because we also require basically the same two restrictions:

$\mathbf{R_1}$. Neither *Update(x,y)* nor *Delete(x)* can be performed at node i unless the element x is in the local view at node i.

$\mathbf{R_2}$. All item names used in *Insert* operations must be unique.

The second restriction explains why the inclusion of the *Update* operation alters the problem's structure. An *Update* is thus a primitive operation which can not be formed from *Insert* and *Delete* operations. These restrictions are required by the algorithms and are quite reasonable in the application domains discussed below. $\mathbf{R_1}$ is quite intuitive since operations by definition must name elements from the local view. $\mathbf{R_2}$ provides the assurance that once an item name has been deleted, it can not be reinserted. This avoids a conflict which would require some additional means to order the *Insert* and *Delete* operations (conflict resolution). Throughout the remainder of this paper a *change* refers to either an *Insert*, *Update*, or *Delete* operation.

There are two additional operations: *Transmit* and *Receive*. *Transmit* is used by a node to send information concerning its view to other nodes. *Receive* is automatically invoked when a message containing information from some other node is received. No information flows between [illegible] views unless *Transmit* operations are issued by a client. The frequency of *Transmit* operations dictate how current a particular view is for some node. It is presumed that clients will issue *Transmit* operations often enough so that views will converge acceptably often for the applications.

All six operations must be non-interfering when manipulating the local view. Regardless of the method used, atomicity with respect to concurrent activity among the operations is assumed. Because the maintenance of the view should be relatively inexpensive for many applications, mutual exclusion may suffice.

The operations and associated restrictions presented above appear to be sufficient for several application areas. Applications which include problems related to maintaining some form of replicated dictionary mesh well with our approach. For example, some distributed applications which include this type of problem are mail systems, naming servers, file directories, appointment calendars and operating system load data maintenance (e.g., routing tables). In addition, applications like process control systems which alter data values rapidly, but do not require serial consistency can also be supported. These applications tend to be self correcting in nature and do not necessarily require serial consistency.

3. General Suite Structure

We divide databases into two types: independent and dependent. *Independent* databases permit elements to be changed by any node in the network. Thus once a data item name has been created, it can be manipulated by any node in the network. *Dependent* databases permit elements to be changed only by the node which created the data item name. That is, changes depend on which node was the item's creator.

We also consider two levels of fault-tolerance: propagation and no propagation. *Propagation* implies every node must guarantee all other nodes receive a change, even when a node is not directly responsible for the change. Thus, even if the node which makes some change fails (or outward communication from the node fails) the change can still propagate through the network, depending on the state of the other nodes and remaining communication system. This approach implies that data availability is more significant than message traffic overhead and local storage space. The information regarding the change is stored at each node until that node is sure every other node has seen the result of that change. *No propagation* implies that only the node responsible for performing a change must ensure that every other node has received the change.

The message distribution procedure is not specified in the suite, since the network topology and availability requirements dictate how messages are actually distributed throughout the network. Messages could be broadcast, multicast, or simply sent to some *next* node.

The structure of the suite consists of a base algorithm and resolution tables to specify algorithm actions when changes occur locally or are received from a remote node. There is a different resolution table for each combination of database type and fault-tolerance discussed above. The base algorithm need not be changed. In the following we present an overview discussion of the basic data structures and base algorithm. Then in Section 3.2 we discuss certain aspects of the algorithm in detail. Finally, in Section 3.3 the base algorithm is presented together with the first resolution table.

3.1 Algorithm Overview

The basic algorithm is assumed to be replicated at all nodes. Three basic data structures are used in the algorithm; each node has a separate set of these variables:

V_i the database view for node *i*.

SS_i a list at node *i* of changes which may not have been seen by the other nodes. (SS represents a synchronization set.)

t_i a timestamp array which details how current node *i*'s knowledge is of every other node.

Both SS and t are transmitted to some collection of remote nodes upon a client's *Transmit* request. V is not sent between nodes except during a cold start of a node; see Section 5.6.

When a change occurs at some node *i*, the change is reflected in the SS_i and the database view V_i. The change is marked with the current value of a node-relative *Clock*. Synchronization sets contain at most one entry for every item name changed. A particular change may be superceded at any time, either before leaving the originating node or at some later intermediate destination. Since a change may be removed from the SS before all nodes have seen that change, another method is used to permit a node to determine when the changes have been processed by remote nodes. The timestamp array t is used for this purpose. This array is indexed by node number. The value of each entry represents a node-relative *Clock* number. For example, if $t_i[5] = 3$, then this means that node *i* has seen the *result* of all changes from node 5 through time 3 (relative to node 5). We use the term result here because changes can be superceded in the synchronization set at any time. Thus a node may never see certain changes, it could see some newer change.

In the propagation approach a node *i* maintains an SS entry for every change entry applied to the database locally until node *i* is sure every other node has received the change. When a SS arrives it is merged with the local SS. Entries may be added or deleted to both V and SS according to the resolution table. Removal of a change entry from a node's SS can occur in one of two ways:

case 1: A node can be passed a SS containing a change entry which has been seen by all nodes except for the receiving node.

case 2: A node can receive a SS which does not contain the change item and the received t array shows that the sending node has seen a result from the node where the change originated at least through the time when the change occurred. Because the sending node definitely has seen the change and does not have the change entry on the SS, we know (by induction) that case 1 must have occurred at some node in the past. Thus the change entry at the receiving node can be deleted.

In the no propagation approach, the node performing the change is the only node which maintains the change entry on the SS. The other nodes perform the change, but do not change their SS. An originating node *i* can determine when an entry has been seen by all nodes when it receives t arrays from all remote nodes which reflect a time for node *i* greater than the time when the change was performed.

After node *i* receives and processes some remote SS_j, each entry in the local time array t_i is set to the maximum of t_i and the remote time array received (t_j). In essence, node *i* now has a view representing both nodes *i* and *j* through the times given in the new timestamp array.

3.2 Details Concerning the Base Algorithm and Resolution Tables

In this algorithm, the *Clock* is assumed to provide real time. However, as discussed later, it is possible to consider the *Clock* function as simply a monotonic strictly increasing function. Assuming that the *Clock* function reflects time is particularly attractive since failures do not require special corrective action to ensure the monotonicity property. This *Clock* property is stated below:

C_1. $Clock_{q+1} > Clock_q$ for all executions q of the *Clock* function

It is assumed for this presentation that item names satisfy restriction R_2 by using unique names generated by the *Uniquename* function. This, of course, is not required in applications in which duplicate names are impossible.

The procedure *Resolve* processes new synchronization sets against local synchronization sets. Every entry in each SS must be examined. If the other SS contains an entry with the same data item name, then the two entries must be resolved and only one entry kept. Fake entries are created if only one of the SS's contain a particular data item name change entry; see the following paragraph. The order of processing each synchronization set is unimportant, but an entry must be processed only once.

Resolution tables are used to apply two synchronization sets against each other to update the database view and create a new synchronization set which includes the most current information concerning changes. The table format has been extended for simplicity to include rows and columns to represent entries which may be present in one of the synchronization sets, but absent in the other. This permits the resolution table to be used uniformly. There are two possible reasons why a particular entry could be missing: the result of the change has already been seen and removed or the result of the change has never been seen. Each axis includes the lines **AbsentSeen** and **AbsentNotSeen** to represent these conditions. Refer to the procedure *Resolve* and the type and variable definitions on the following pages for the definition and use of *changeitem*.

AbsentSeen

- there is no entry with the specified name in the associated synchronization set (absent from SS),
- and by the associated t we know that changes have been processed through the value in the time array (seen by the node). That is:

$t[z.cn] \geq z.ct$, for some changeitem z

AbsentNotSeen

- there is no entry with the specified name in the associated synchronization set (absent from SS),
- and by the associated t we have definitively not seen the change (notseen by the node). That is:

$t[z.cn] < z.ct$, for some changeitem z

The procedure *Perform Action* used in *Resolve* is simply a dummy procedure which represents performing the actions defined in the resolution table on both SS and V. Thus,

Perform Action (x, y, SS, V)

represents using the x.op and y.op fields (of the changeitems x and y) to select the appropriate x axis and y axis array positions in the table. The actions specified at that location are then to be performed on SS and V. (Note that a dummy x or y entry is created if the item is missing from the corresponding synchronization set; the op field is set accordingly.)

In all of the resolution tables presented below *Update* conflicts are resolved in favor of the higher change time (ct); that is, *probably* the latest change wins. It is possible that the latest change may not win, though, because the different node clocks may not be physically synchronized. It is also possible for two changes to be made at exactly the same time. This conflict is resolved by using a total ordering on the node numbers. This algorithm does require that the *Clock* functions of the individual

The basic algorithm is assumed to be replicated at all nodes. Three basic data structures are used in the algorithm; each node has a separate set of these variables:

V_i the database view for node *i*.
SS_i a list at node *i* of changes which may not have been seen by the other nodes. (SS represents a synchronization set.)
t_i a timestamp array which details how current node *i*'s knowledge is of every other node.

Both SS and t are transmitted to some collection of remote nodes upon a client's *Transmit* request. V is not sent between nodes except during a cold start of a node; see Section 5.6.

When a change occurs at some node *i*, the change is reflected in the SS_i and the database view V_i. The change is marked with the current value of a node-relative *Clock*. Synchronization sets contain at most one entry for every item name changed. A particular change may be superceded at any time, either before leaving the originating node or at some later intermediate destination. Since a change may be removed from the SS before all nodes have seen that change, another method is used to permit a node to determine when the changes have been processed by remote nodes. The timestamp array t is used for this purpose. This array is indexed by node number. The value of each entry represents a node-relative *Clock* number. For example, if $t_i[5] = 3$, then this means that node *i* has seen the *result* of all changes from node 5 through time 3 (relative to node 5). We use the term result here because changes can be superceded in the synchronization set at any time. Thus a node may never see certain changes, it could see some newer change.

In the propagation approach a node *i* maintains an SS entry for every change entry applied to the database locally until node *i* is sure every other node has received the change. When a SS arrives it is merged with the local SS. Entries may be added or deleted to both V and SS according to the resolution table. Removal of a change entry from a node's SS can occur in one of two ways:

case 1: A node can be passed a SS containing a change entry which has been seen by all nodes except for the receiving node.

case 2: A node can receive a SS which does not contain the change item and the received t array shows that the sending node has seen a result from the node where the change originated at least through the time when the change occurred. Because the sending node definitely has seen the change and does not have the change entry on the SS, we know (by induction) that case 1 must have occurred at some node in the past. Thus the change entry at the receiving node can be deleted.

In the no propagation approach, the node performing the change is the only node which maintains the change entry on the SS. The other nodes perform the change, but do not change their SS. An originating node *i* can determine when an entry has been seen by all nodes when it receives t arrays from all remote nodes which reflect a time for node *i* greater than the time when the change was performed.

After node *i* receives and processes some remote SS_j, each entry in the local time array t_i is set to the maximum of t_i and the remote time array received (t_j). In essence, node *i* now has a view representing both nodes *i* and *j* through the times given in the new timestamp array.

3.2 Details Concerning the Base Algorithm and Resolution Tables

In this algorithm, the *Clock* is assumed to provide real time. However, as discussed later, it is possible to consider the *Clock* function as simply a monotonic strictly increasing function. Assuming that the *Clock* function reflects time is particularly attractive since failures do not require special corrective action to ensure the monotonicity property. This *Clock* property is stated below:

C_1. $Clock_{q+1} > Clock_q$ for all executions q of the *Clock* function

It is assumed for this presentation that item names satisfy restriction R_2 by using unique names generated by the *Uniquename* function. This, of course, is not required in applications in which duplicate names are impossible.

The procedure *Resolve* processes new synchronization sets against local synchronization sets. Every entry in each SS must be examined. If the other SS contains an entry with the same data item name, then the two entries must be resolved and only one entry kept. Fake entries are created if only one of the SS's contain a particular data item name change entry; see the following paragraph. The order of processing each synchronization set is unimportant, but an entry must be processed only once.

Resolution tables are used to apply two synchronization sets against each other to update the database view and create a new synchronization set which includes the most current information concerning changes. The table format has been extended for simplicity to include rows and columns to represent entries which may be present in one of the synchronization sets, but absent in the other. This permits the resolution table to be used uniformly There are two possible reasons why a particular entry could be missing: the result of the change has already been seen and removed or the result of the change has never been seen. Each axis includes the lines **AbsentSeen** and **AbsentNotSeen** to represent these conditions. Refer to the procedure *Resolve* and the type and variable definitions on the following pages for the definition and use of *changeitem*.

AbsentSeen

- there is no entry with the specified name in the associated synchronization set (absent from SS),
- and by the associated t we know that changes have been processed through the value in the time array (seen by the node). That is:

  $t[z.cn] \geq z.ct$, for some changeitem z

AbsentNotSeen

- there is no entry with the specified name in the associated synchronization set (absent from SS),
- and by the associated t we have definitively not seen the change (notseen by the node). That is:

  $t[z.cn] < z.ct$, for some changeitem z

The procedure *Perform Action* used in *Resolve* is simply a dummy procedure which represents performing the actions defined in the resolution table on both SS and V. Thus,

Perform Action (x, y, SS, V)

represents using the x.op and y.op fields (of the changeitems x and y) to select the appropriate X axis and y axis array positions in the table. The actions specified at that location are then to be performed on SS and V. (Note that a dummy x or y entry is created if the item is missing from the corresponding synchronization set; the op field is set accordingly.)

In all of the resolution tables presented below *Update* conflicts are resolved in favor of the higher change time (ct); that is, *probably* the latest change wins. It is possible that the latest change may not win, though, because the different node clocks may not be physically synchronized. It is also possible for two changes to be made at exactly the same time. This conflict is resolved by using a total ordering on the node numbers. This algorithm does require that the *Clock* functions of the individual nodes be logically synchronized [Lamp78]. This is accomplished within the *Receive* procedure. Section 5 discusses alternative methods for conflict resolution.

3.3 The Base Algorithm and the First Resolution Table

The base algorithm is presented on the following pages. The notation for the algorithm is based on a derivative of Pascal. Some additional notation is used to avoid trivial details. The first resolution table is presented in Figure 3-1. This table represents the independent / propagation type of system structure. The other resolution tables are presented in Section 4.

There are a variety of simple modifications possible for the base algorithm separate from the resolution table. These algorithm modifications are discussed in Section 5. Because the modifications are so simple we consider all the possible derived algorithms to be part of the suite.

Types and Variables

Primitive Types:

```
string
integer
ts                  {the type returned from the Clock function}
value               {the type for the items in the database}
```

Defined Types:

```
node        =   1..MaxNodes

itemname    =   record
                    itemstring     :  string;
                    creator        :  node;
                    creationtime   :  ts
                end;

item        =   record
                    itemn          :  itemname;
                    val            :  value
                end;

changeitem  =   record
                    citem          :  item,
                    op             :  (Insert, Update, Delete, AbsentSeen,
                                       AbsentNot Seen);
                    cn             :  node,
                    ct             :  ts;
                    knownby        :  set of node
                end;

tsarray     =   array[node] of ts;

changeset   =   set of changeitem;

message     =   record
                    from           :  node;
                    remoteT        :  tsarray;
                    remoteSS       :  changeset
                end;
```

Global Variables (for each node):

```
V          :    set of item;    {this node's view of the database}

SS         :    changeset;      {the synchronizing set}

t          :    tsarray;        {Time array; e.g., t[5] = 3 ⇒ this node has seen
                                 the result of all changes from node 5 through
                                 time 3 (relative to node 5)}

allnodes   :    set of node;    {the current list of all nodes which can view the
                                 database}

i          :    node;           {the node number of this node}
```

The Basic Algorithm

```
function Uniquename (xstring : string): itemname

begin
          Uniquename : = <xstring, i, Clock>
end;


function Insert (xname: itemname; vin : value): (ok, alreadyexists)

var
          x       :   item;
          time    :   ts;
          r       :   changeset;
          local   :   boolean;

begin
          if xname $\in$ V.itemn then Insert : = alreadyexists;
          time : = Clock;
          x : = <xname, vin>
          t(i) : = time;
          r : = {<x, Insert, i, time, {i}>}
          local : = true;
          Resolve (local, SS, t, r, t, V)
          Insert : = ok
end;


function Update (xname: itemname; vin : value): (ok, nonexistent)

var
          x       :   item;
          time    :   ts;
          r       :   changeset;
          local   :   boolean;

begin
          if xname $\notin$ V.itemn then Update : = nonexistent;
          time : = Clock;
          x : = <xname, vin>
          t(i) : = time;
          r : = {<x, Update, i, time, {i}>}
          local : = true;
          Resolve (local, SS, t, r, t, V)
          Update : = ok
end;
```

```
function Delete (xname: itemname): (ok, nonexistent)

var
        x       :   item;
        time    :   ts;
        junk    :   value;
        r       :   changeset;
        local   :   boolean.

begin
        if xname $\notin$ V.itemn then Delete := nonexistent;
        time := Clock;
        x := <xname, junk>
        t(i) := time;
        r := {<x, Delete, i, time, {i}>}
        local := true;
        Resolve (local, SS, t, r, t, V)
        Delete := ok
end;
```

```
function List (wanted: set of itemname): set of item

begin
        return from V the wanted items, if present
end;
```

```
procedure Transmit

begin
        Save(t, SS, V);             {save in permanent storage all changes since the last save.
                                     This can be done by an incremental log.}
        Send(<i, t, SS>)            {Send the view information (represented by SS) in a message
                                     to some set of other nodes}
end;
```

```
procedure Receive (m: message)

var
        local   :   boolean;

begin
        local := false;
        Resolve(local, SS, t, m.remoteSS, m.remoteT, V);
        if Clock $\leq$ max {m.remoteT[j] | $1 \leq j \leq$ MaxNodes} then
                Clock := max {m.remoteT[j] | $1 \leq j \leq$ MaxNodes } + 1
end;
```

```
procedure Resolve (     local     : boolean;
                    var old SS  : changeset;     var   oldt    : tsarray;
                    var newSS   : changeset;           newt    : tsarray;
                    var V       : set of item)

var
        tempoldSS      :   changeset;
        matchitems     :   changeset;
        x              :   changeitem;
        y              :   changeitem;

begin
        tempoldSS := oldSS;     {note: In practice oldSS does not need to be copied.  It is shown this
                                 way for clarity}

        if not local then begin
            for each x ∈ tempoldSS do
            begin
                matchitems := {z ∈ newSS | x.citem.itemn = z.citem.itemn}
                newSS := newSS - matchitems
                if matchitems = ∅ then begin      {make a dummy changeitem entry}
                    if newt[x.cn] ≥ x.ct then y.op := AbsentSeen
                    else y.op := AbsentNotSeen
                end
                else let y ∈ matchitems;     {note: | matchitems | = 1

                Perform Action (x, y, oldSS, V)
            end
        end;


        for each y ∈ newSS do
        begin
            matchitems := {z ∈ tempoldSS | y.citem.itemn = z.citem.itemn}
            if matchitems = ∅ then begin          {make a dummy changeitem entry}
                if oldt[y.cn] ≥ y.ct then x.op := AbsentSeen
                else x.op := AbsentNotSeen
            end
            else let x ∈ matchitems;              {note: | matchitems | = 1}

            Perform Action (x, y, oldSS, V)
        end;


        oldt[j] := max {oldt[j], newt[j] | 1 ≤ j ≤ MaxNodes}
end;
```

y \ x	Insert	Update	Delete	AbsentSeen	AbsentNotSeen

Insert	SS:M(x,y)
	V:nc
		SS:nc
		V:nc
			SS:nc
			V:nc
				SS:nc
				V:nc
					SS:A(y)
					V:A(y)

Update	SS:R(x,y)
	V:R(y)
		SS:$*_1$
		V:$*_2$
			SS:nc
			V:nc
				SS:nc
				V:nc
					SS:A(y)
					V:A(y) if absent
					R(y) if present

Delete	SS:R(x,y)
	V:D(x)
		SS:R(x,y)
		V:D(x)
			SS:M(x,y)
			V:nc
				SS:nc
				V:nc
					SS:A(y)
					V:D(y) if present

Absent Seen	SS:D(x) V:nc	SS:D(x) V:nc	SS:D(x) V:nc	SS:- V:-	SS: V:-
Absent Not Seen	SS:nc V:nc	SS:nc V:nc	SS:nc V:nc	SS:- V:-	SS:- V:

nc: no change

$*_1$: **if** (x.cn = y.cn) **and** (x.ct = y.ct) **then** M(x,y)
else if (x.ct < y.ct) **or** (x.ct = y.ct **and** x.cn < y.cn) **then** R(x, y)

$*_2$: R(y) if x replaced in $*_1$

SS:	A(j)	≡ Add j to oldSS union local node number into knownby of j if knownby = allnodes, then D(j)
	R(j,k)	≡ Replace j on oldSS with k union local node number into knownby of j if knownby = allnodes, then D(j)
	D(j)	≡ Delete j on oldSS
	M(j,k)	≡ Merge knownby sets into j if knownby = allnodes, then D(j)
V:	A(j)	≡ Add j.citem to V
	R(j)	≡ Replace the item with the same name in V with j.citem
	D(j)	≡ Delete item with the name j.citem.itemn from V

Figure 3-1 Resolution Table for Propagation / Independent

4. Other Resolution Tables

The resolution table presented with the base algorithm (Figure 3-1) assumes that every node should propagate a change and that items can be changed anywhere throughout the network (*i.e.*, the database is independent). There are a variety of other assumptions which can be supported by simply altering the resolution table provided with the base algorithm.

First, we consider the situation where the database is dependent. Recall that changes can occur to a data item *only* at the node which originally created the item in this type of database. In addition, we assume that other nodes are not responsible for ensuring the changes are seen by every other node; the node making the change is responsible for verifying this (*viz.*, no propagation). This problem is somewhat trivial, but nevertheless quite common. Consider operating system load tables which specify the current load information for the node. This is then used by other nodes in some decentralized load distribution procedure. Clearly, only one node will be changing the load information and if the changing node fails there is little reason for concern over change propagation. Figure 4-1 contains the resolution table for this problem. Note that there are no *Update* conflicts in this example, so logical *Clock* synchronization is actually not needed. If the database was dependent, but propagation was desired, then the resolution table of Figure 3-1 would be used.

The second alternative resolution table we consider represents another common problem: even though changes can occur at any node (independent), propagation of this information by every node in the network is not required (no propagation). This may be reasonable in environments such as high speed contention or ring based local area networks in which the nodes appear fully connected. Thus the originating node for a change is responsible for ensuring that every other node learns of the change. Figure 4-2 contains the *resolution* table which specifies the actions to be taken in this environment.

The following table summarizes the requirements satisfied by the different resolution tables

requirements	dependent	independent
propagation	Figure 3-1	Figure 3-1
no propagation	Figure 4-1	Figure 4-2

x y	Insert	Update	Delete	AbsentSeen	AbsentNotSeen
Insert	SS:- V:-	SS:- V:-	SS:- V:-	SS:nc V:nc	SS:A(y) if y.cn = i V:A(y)

Update	SS:R(x,y)
	V:R(y)
		SS:R(x,y)
		V:R(y)
			SS:-
			V:-
				SS:nc
				V:nc
					SS:A(y) if y.cn = i
					V:A(y) if absent
					R(y) if present

Delete	SS:R(x,y)
	V:D(x)
		SS:R(x,y)
		V:D(x)
			SS:-
			V:-
				SS:nc
				V:nc
					SS:A(y) if y.cn = i
					V:D(y) if present

Absent Seen	SS:K(x) V:nc	SS:K(x) V:nc	SS:K(x) V:nc	SS:- V:-	SS:- V:-
Absent Not Seen	SS:nc V:nc	SS:nc V:nc	SS:nc V:nc	SS:- V:-	SS:- V:-

nc: no change

SS:	A(j)	≡ Add j to oldSS union local node number into knownby of j if knownby = allnodes, then D(j)
	R(j,k)	≡ Replace j on oldSS with k union local node number into knownby of j if knownby = allnodes, then D(j)
	D(j)	≡ Delete j on oldSS
	K(j)	≡ Union remote node number into knownby of j if knownby = allnodes, then D(j)
V:	A(j)	≡ Add j.citem to V
	R(j)	≡ Replace the item with the same name in V with j.citem
	D(j)	≡ Delete item with the name j.citem.itemn from V

Figure 4-1 Resolution Table for No Propagation / Dependent

y \ x	Insert	Update	Delete	AbsentSeen	AbsentNotSeen
Insert	SS:- V:-	SS.nc V:nc	SS:nc V:nc	SS:nc V:nc	SS:A(y) if y.cn = i V:A(y)

Update	SS:S(x,y)
	V:R(y)
		SS:*$_1$
		V:*$_2$
			SS:nc
			V:nc
				SS:nc
				V:nc
					SS:A(y) if y.cn = i
					V:A(y) if absent
					R(y) if present

Delete	SS:S(x,y)
	V:D(x)
		SS:S(x,y)
		V:D(x)
			SS:nc
			V:nc
				SS:nc
				V:nc
					SS:A(y) if y.cn = i
					V:D(y) if present

Absent Seen	SS:K(x) V:nc	SS:K(x) V:nc	SS:K(x) V:nc	SS:- V:-	SS:- V:-
Absent Not Seen	SS:nc V:nc	SS:nc V:nc	SS:nc V:nc	SS:- V:-	SS:- V:-

nc: no change

*$_1$: if (x.cn = y.cn) and (x.ct = y.ct) then R(x,y)
else if (x.ct < y.ct) or (x.ct = y.ct and x.cn < y.cn) then D(x)

*$_2$: R(y) if x replaced in *$_1$

SS:	A(j)	≡ Add j to oldSS union local node number into knownby of j if knownby = allnodes, then D(j)
	R(j,k)	≡ Replace j on oldSS with k union local node number into knownby of j if knownby = allnodes, then D(j)
	D(j)	≡ Delete j on oldSS
	S(j,k)	≡ if j.cn = k.cn and j.ct = k.ct then R(j,k) else D(j)
	K(j)	≡ Union remote node number into knownby of j if knownby = allnodes, then D(j)
V:	A(j)	≡ Add j.citem to V
	R(j)	≡ Replace the item with the same name in V with j.citem
	D(j)	≡ Delete item with the name j.citem.itemn from V

Figure 4-2 Resolution Table for No Propagation / Independent

5. Variations

The following sections discuss extensions of the base algorithm and resolution table. The particular variations presented adapt the basic scheme to accomodate a variety of different application requirements.

5.1 Sending Individual Changes Immediately

When the synchronization set is sent from a node, all changes which may not have been seen by some other node are sent. Because this set may only be sent occasionally by some applications, it is desirable to consider the possibility of sending a change immediately (without the remaining members on the synchronization set). Whether this is appropriate depends on several factors. If changes are rapid, a substantial load on the network could result. This is possible because changes are overwritten in the synchronization set as soon as they are detected. If changes are sent immediately, then some changes could be sent which would not have been in the base algorithm. There are, however, a variety of applications which could benefit by the rapid distribution of a change. The synchronization set would be transmitted as a backup precaution to ensure that all changes are eventually acknowledged.

The same resolution approach can be used to solve this problem. However, care must be taken because each change sent is independent from the preceding one. The receiving node can not determine whether all preceding changes have been seen or not. That is, receiving a particular change from some node *j* *does not* imply reception of all previous changes from node *j*. Therefore, when the change is received the new change should be resolved against any changes of the same name in the local SS, but the local SS entries should not be resolved against absent entries in the incoming SS. This is exactly what is required when performing changing locally at a node and thus the *local* variable is set to **true**.

Below are the code fragments to accomplish sending changes immediately. Another message type is defined which is sent for every *Insert, Update,* or *Delete* performed. We will assume that the two message types can be distinguished.

- In *Insert, Update* and *Delete* immediately before returning *ok*:

```
        Set r to have a knownby list of ∅
        Save (t, SS, V);                       {Incremental save}
        $Send_2$(< i, r >);                    {Special send of single change}
```

- Add a new *Receive* operation:

```
    procedure $Receive_2$ (m: record from : node;  remoteSS : changeset; end)

    var
        local    :    boolean;

    begin
        local := true;                         {pretend it's local}
        Resolve (local, SS, t, m.remoteSS, t, V)
    end;
```

5.2 Specifying Conflict Strategies for Ordering Update Operations

All of the resolution tables presented thus far have considered only retaining the *most recent* change. As mentioned previously, this is not always achieved because the clocks may not be physically synchronized. (However, in environments such as local area networks, the clocks are usually very close.) Even if the the change which would be retained was the most recent, this may be inappropriate for some applications. That is, sometimes it may be desirable to choose the earlier change rather than the later one. For example, if two clients conflict when changing an item in a reservation database, it is the earlier change which should probably win.

Using just older and newer as the only conflict resolution strategies is still overly restrictive. There are several other functions which could be used to resolve conflicts. The functions *Maximum* and *Minimum*, for example, appear quite well suited for conflict resolution for some application data items in reservation and similar systems. Any (commutative) function which totally orders the data values will suffice. Note that in the base resolution table, node numbers were totally ordered and used to break *Update* conflicts which tied on their change times. This was used because each node has a separate execution agent and thus could not create the needed total order.

A trivial extension to address different conflict resolution strategies for each type of data item is to include with each item (when *Inserted*) the type of conflict resolution strategy which should be performed on *Update* conflicts.

5.3 Functional Operations

Update operations replace the value of a data item in the view. This prompts the *Update / Update* conflicts which must be resolved through some type of total ordering on the changes. There are a variety of operations which do not have this inherent conflict problem. For example, the commutative operations of *Increment* and *Decrement* can not conflict since the result would be the same regardless of the order executed. Thus, items in the database could be marked as being manipulated only through some specified set of functional operations and avoid all conflicts. The changes to the resolution table would be quite simple. One new column and one new row must be added for *functional* operations. Instead of replacing entries on the synchronization set, functional changes must add new entries. As the entries are verified to have been seen by all nodes, the entries are deleted as before. It is assumed that data items which use functional operations can not be manipulated through the *Update* operation. If a *Delete* operation is performed, then all functional entries on the synchronization set should be removed. Thus, an *Insert* can be performed followed by any number of *functional* operations and finally followed by a *Delete* operation. The modifications to the resolution table are straightforward and are not shown here.

5.4 Atomic Changes

The atomic operations (which change the database) presented thus far are the primitives *Insert*, *Update*, and *Delete*. If it was desired to combine these operations into a larger transaction, then the transaction would not maintain the same properties as the smaller operations. Since each change to a view receives a *Clock* timestamp, it is not possible to ensure that multiple changes will be treated uniformly with respect to conflicts. What may be desired in certain cases is that multiple changes either all win, or all lose in a conflict. One alternative is to assign the same *Clock* time to every change in the transaction. This guarantees that if two transactions containing only *Update* operations manipulate the same items, then the transactions can be serially ordered.

5.5 Limiting the Size of Synchronization Sets

Changes remain on the synchronization sets of all nodes responsible for information propagation until all nodes have acknowledged the change. During node and network failure the sets could become quite large. This is the cost for not passing the entire view around the network. If data items are not deleted, then the size of each synchronization set is bounded by the size of the view. There could be a change for every data item in the view, but since changes are overwritten if an entry already exists, the set size does not change regardless of failure duration. If, however, *Delete* operations occur, then the simplistic scheme presented thus far would allow the synchronization set to become unbounded. There appear to be two straightforward solutions to this problem. Each of these is discussed below.

First, the SS could be limited to contain only *n* members with each node *i owning* O_i members. The nodes could be assigned different amounts, provided that each SS has sufficient space for all the entries. That is,

$$n = \sum_{i=1}^{MaxNodes} O_i$$

If a local client makes a request of the system and its allocation on the SS is depleted, then no *Inserts* or *Deletes* should be accepted. *Updates* can be accepted only if the item is already in the node's SS. This allows all remote node synchronization sets to be accepted. This is of course a pessimistic strategy: the entire system could stop accepting *Inserts* and *Deletes*, if a single node fails. However, in the case of a simple node failure, it is relatively simple to eliminate the failed node from *Allnodes* and demand that the failed node reinitialize when it restarts (see Section 5.6). It is much more complicated if the network communication system has failed and the network is partitioned. The second alternative could be used in cases where this solution is unacceptable.

The second solution involves replacing the *Delete* entries on the synchronization set with a *DeleteRange* entry. Two *Delete* entries (related to the same node responsible for some change) can be combined if the view contains no intervening view items created by the same node. This is true even if the node which is creating the *DeleteRange* entry did not delete all intervening items in the view. When a *DeleteRange* entry is received, it can be expanded to match all items in the range. The test for intervening is made on the *ctime* field. For example,the following entry would delete all items created by node number 5 from time 1 through time 8.

DeleteRange (creator = 5; ctime = 1..8; cn = 4)

It can be proved that this is sufficient to bound (within a proportionality constant) the SS size to the size of the view.

5.6 Online Inclusion / Removal of Nodes

Even though the suite supports "automatic" reintegration of nodes supporting the database in most cases, throughout the life of the database certain nodes may fail beyond automatic database repair (*e.g.*, disk crash). In addition, nodes may be added or removed from supporting the database. These situations require a means for a node to resynchronize with the current members of *Allnodes*. We will refer to this situation as *cold starting*.

Since *Allnodes* can be changing over the life of the database, there is no reason not to place *Allnodes* directly into the database. The value then propagates naturally throughout the network when a change is issued. Removing a machine from the participating group of database nodes is straightforward. However, adding a new member requires an agreement procedure which is quite similar to that of two phase commitment [Dole82, Gray78]. A sketch of the procedures is given below.

- Removing a Node *i* :

```
$Allnodes := Allnodes - \{i\}$
Update Allnodes
```

- Cold Starting a Node *i* :

```
$V := \emptyset$; $t[k] := 0\ \forall k$; $SS := \emptyset$; $Allnodes := \{i\}$; $Stillbooting := true$; $Talkingnodes := \emptyset$;

▶Send request for boot service to everyone; pass node number $i$
All receiving nodes with $Stillbooting = false$ should send node $i$ their node numbers
For all nodes which respond (before timeout limit), add their node numbers to Talkingnodes
While ($Talkingnodes \neq \emptyset$) and (Stillbooting) do
begin
        Pick some node, say $j$, from Talkingnodes    {picked as desired; e.g., closest}
        $Talkingnodes := Talkingnodes - \{j\}$
        Send request for V, SS, and t to node $j$
        If this message is received by $j$, then $j$ must add $i$ to Allnodes before replying
        If $i$ receives $j$'s view information (before timeout limit) then
        begin
              create a new SS by making an Insert entry for every item in the database
              Merge the returned remoteSS into SS (through standard resolution)
              $t[k] := remoteT[k], \forall k \neq i$
              $Stillbooting := false$
        end
end

if $Stillbooting = true$ then occasionally request boot service (by repeating above from ▶)
```

- Cold Starting the First Node *i* :

```
$V := \emptyset$; $t[k] := 0\ \forall k$; $SS := \emptyset$; $Allnodes := \{i\}$; $Stillbooting := false$;
```

If there are no nodes which respond, then the node is free to continue, however it must occasionally attempt to communicate with other nodes which may be supporting the database. Note that this procedure allows nodes to join the current group of view communicating nodes. It does not support two nodes which are both *Stillbooting* to share information. This is allowed only after each has joined the primary group.

6. A Formal Model of the Base Problem

In this section we provide a formal framework for considering the algorithms presented in his paper. In addition, the correctness condition for the base algorithm and resolution table is given.

Let **W** be the domain of values. Let **D** be the domain of element names. Each view of the database then is a subset of **D** × **W**.

Let $BasicOps = \{Insert(x,y), Update(x,y), Delete(x) \mid x \in \mathbf{D}, y \in \mathbf{W}\}$. Let $OtherOps = \{List(Q) \mid Q \subseteq \mathbf{D}\} \cup \{Transmit(m), Receive(m) \mid m \text{ is a message}\}$. And finally, let $AllOps = BasicOps \cup OtherOps$.

Fix some particular execution of the system. Each instance of an operation from *AllOps* corresponds to an event. Let *E* be the set of all events occuring in the particular fixed execution.

Let *op* : $E \to AllOps$ be the operation associated with each event, *where* : $E \to node$ be the node at which some event occurred, and *when* : $E \to ts$ be when the event happened relative to the *Clock* at the node where the event occurred. That is, $when(e) = Clock_{where(e)}$.

Define → to be a relation on $E \times E$, "happened before", such that

O$_1$. if $e_1, e_2 \in E$, $where(e_1) = where(e_2)$, and $op(e_1)$ is performed before $op(e_2)$, then $e_1 \to e_2$.

O$_2$. if $e_1, e_2 \in E$, $op(e_1) = Transmit(m_1)$ and $op(e_2) = Receive(m_1)$, then $e_1 \to e_2$.

O$_3$. if $e_1, e_2, e_3 \in E$, $e_1 \to e_2$ and $e_2 \to e_3$, then $e_1 \to e_3$.

O$_4$. if $e \in E$, then $e \to e$.

We can now define the correctness conditions for the base algorithm and resolution table. Recall that that approach supports any node making changes to the database and each node is responsible for ensuring every other node has seen some change. Let *view* : $E \to 2^{(\mathbf{D} \times \mathbf{W})}$ be defined as follows: $(x,y) \in view(e')$ iff there exists $\dot{e} \in E$ such that

V$_1$. $\dot{e} \to e'$ and $op(\dot{e}) \in \{Insert(x,y), Update(x,y)\}$

V$_2$. $(\forall e)\,[Removed(e,\dot{e}) \Rightarrow \neg\,(e \to e')]$

$$\begin{array}{lll} \text{where } Removed(e,\dot{e}) & \equiv & [[op(e) = Delete(x)] \text{ or} \\ & & [op(e) = Update(x,y') \text{ and } op(\dot{e}) = Insert(x,y) \text{ and } y \neq y'] \text{ or} \\ & & [op(e) = Update(x,y') \text{ and } op(\dot{e}) = Update(x,y) \\ & & \qquad \text{and } Earlier(\dot{e},e) \text{ and } y \neq y']] \end{array}$$

$$\begin{array}{lll} \text{and} \quad Earlier(e_1,e_2) & \equiv & [[when(e_1) < when(e_2)] \text{ or} \\ & & [when(e_1) = when(e_2) \text{ and } where(e_1) < where(e_2)]] \end{array}$$

7. Proof of Correctness of the Base Algorithm and Resolution Table (3-1)

To prove the base algorithm and resolution table correct we must show that for all *e'*, *view(e')* = $V_{e'}$. That is, the formal view and the database must contain the same set of *(x,y)* after every event *e'*. The following additional notation is required for the proofs.

V_e (and ${}_eV$) ::= V at *where(e)* immediately after (respectively before) completing event *e*

t_e (and ${}_et$) ::= t at *where(e)* immediately after (respectively before) completing event *e*

SS_e (and ${}_eSS$) ::= SS at *where(e)* immediately after (respectively before) completing event *e*

Because SS's contain representatons of events, we will refer to SS's as if they actually contain events. Of course, only events from *BasicOps* have such representations. Thus, $e \in$ SS implies that $op(e) \in$ *BasicOps*. It is obvious from the program code that the program variables *cn* and *ct* for some *changeitem* contain the values of the functions *where* and *when* for the event associated with the particular *changeitem*. For notational convenience, we will therefore consider *where* and *when* to be stored with each event which is in a SS.

We will not consider any operation to be an event which is rejected because of an error. Thus, R_1 and R_2 are assumed to hold.

In the proofs, minimality is referenced. Event *e* is *minimal* to event *e'* with respect to some condition *B* iff $e \rightarrow e'$ with condition *B* holding after event *e* and there does not exist an event *d* such that $d \rightarrow e \rightarrow e'$ and condition *B* holds after event *d*. Thus minimality corresponds to the concept of earliest.

Lemma 1

if $e \rightarrow e'$, then

(a) $t_e(i) \leq t_{e'}(i)$
(b) ${}_{e}t(i) \leq {}_{e'}t(i)$ if $where(e) = where(e')$
(c) $t_e(i) \leq {}_{e'}t(i)$ if $where(e) = where(e')$.

Proof

By inspection and the *Clock* property C_1.∇

Lemma 2

if $(x,y) \in V_{e'}$, then there exists $\hat{e} \in E$ such that $\hat{e}$ is the event which placed (x,y) into $V_{e'}$, $op(\hat{e}) \in \{Insert(x,y), Update(x,y)\}$, and $\hat{e} \rightarrow e'$.

Proof

By inspection of the resolution table (in particular axis y: *Insert*, *Update*; axis x: all) and induction on → with initially $V = \emptyset$, it is clear that there may be several events which precede e' and which could have placed (x,y) into $V_{e'}$. Obviously, only one of these events actually placed (x,y) into $V_{e'}$. Let $\hat{e}$ be that particular event.∇

Lemma 3

if $op(e) \in BasicOps$, $op(e') \in BasicOps$, $e \rightarrow e'$, and $e \neq e'$, then $when(e) < when(e')$.

Proof

(1) if $where(e) = where(e')$, then by C_1: $when(e) < when(e')$
(2) if $where(e) \neq where(e')$, then it must be the case that
$e \rightarrow e'' \rightarrow e''' \rightarrow e'$ with $where(e) = where(e'')$ and $where(e''') = where(e')$ and $op(e'') = Transmit(m_1)$, and $op(e''') = Receive(m_2)$
(3) Thus by the code in *Receive*, we ensure that $when(e) < when(e')$.∇

Lemma 4

if $op(\hat{e}) \in \{Insert(x,y), Update(x,y)\}$ then $(x,y) \in V_{\hat{e}}$.

Proof

(1) if $op(\hat{e}) = Insert(x,y)$, then by the y axis *Insert* row in Table 3-1, R_1, and R_2, we conclude $(x,y) \in V_{\hat{e}}$ (only AbsentNotSeen is possible on the x axis).
(2) if $op(\hat{e}) = Update(x,y)$, then using R_1: if $(x,y) \notin V_{\hat{e}}$ then there must exist $e \in E$ such that $Earlier(\hat{e},e)$, $e \rightarrow \hat{e}$, and $op(e) = Update(x,y')$ *where* $y' \neq y$
By lemma 3. $when(e) < when(\hat{e})$
By definition then $Earlier(e,\hat{e})$ and thus $(x,y) \in V_{\hat{e}}$
(3) ∴By (1) and (2). $(x,y) \in V_{\hat{e}}$.∇

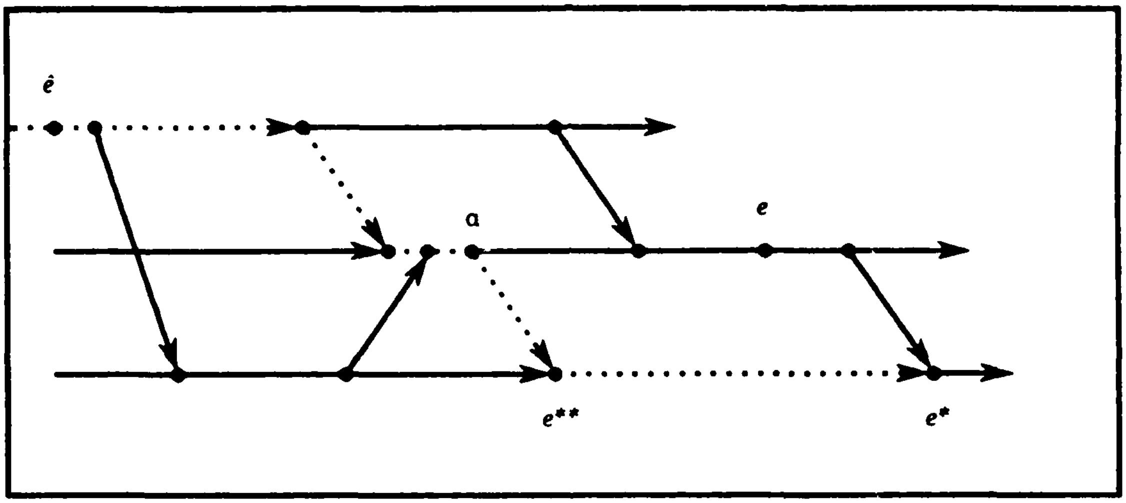
Lemma 5

Let $\hat{e} \in E$ such that $\hat{e} \rightarrow e^*$, $op(\hat{e}) \in \{Insert(x,y), Update(x,y)\}$. In addition, assume $\hat{e}$ is the event which made $(x,y) \in {}_{e^*}V$.

Then if there exists $e^{**} \in E$ such that $\hat{e} \rightarrow e^{**} \rightarrow e^*$, $where(e^{**}) = where(e^*)$, $\hat{e} \notin SS_{e^{**}}$, and e^{**} is minimal, then $Earlier(\hat{e},e)$ for all $e \in E$ such that $t_{e^{**}}(where(e)) < when(e)$ and $op(e) \in BasicOps$.

Proof

The following diagram illustrates the lemma. Chosen on the diagram are representative events for $\hat{e}$, e^*, e^{**}, and e.



Assume given

(1) $\hat{e} \notin SS_{e^{**}}$ means that *during* event e^{**} either

case (a): *Knownby* for the event $\hat{e}$ in the SS = *Allnodes*
Clearly, by the table for any node j there is a path of events from $where(\hat{e})$ to $where(e^{**})$ which includes node j.

case (b): $RemoteT(where(\hat{e})) \geq when(\hat{e})$ and $\hat{e} \notin RemoteSS$ (*i.e.*, y.op = AbsentSeen)
Again by the table and case (a), for any node j there is a path of events from $where(\hat{e})$ to $where(e^{**})$ which includes node j.

(2) Let e be such that $t_{e^{**}}(where(e)) < when(e)$. By (1) then there must exist an event α such that $\hat{e} \rightarrow \alpha \rightarrow e$, $\hat{e} \neq e$, $\alpha \rightarrow e^{**}$, and $where(\alpha) = where(e)$

(3) By lemma 3. $when(\hat{e}) < when(e)$ and thus $Earlier(\hat{e},e)$.∇

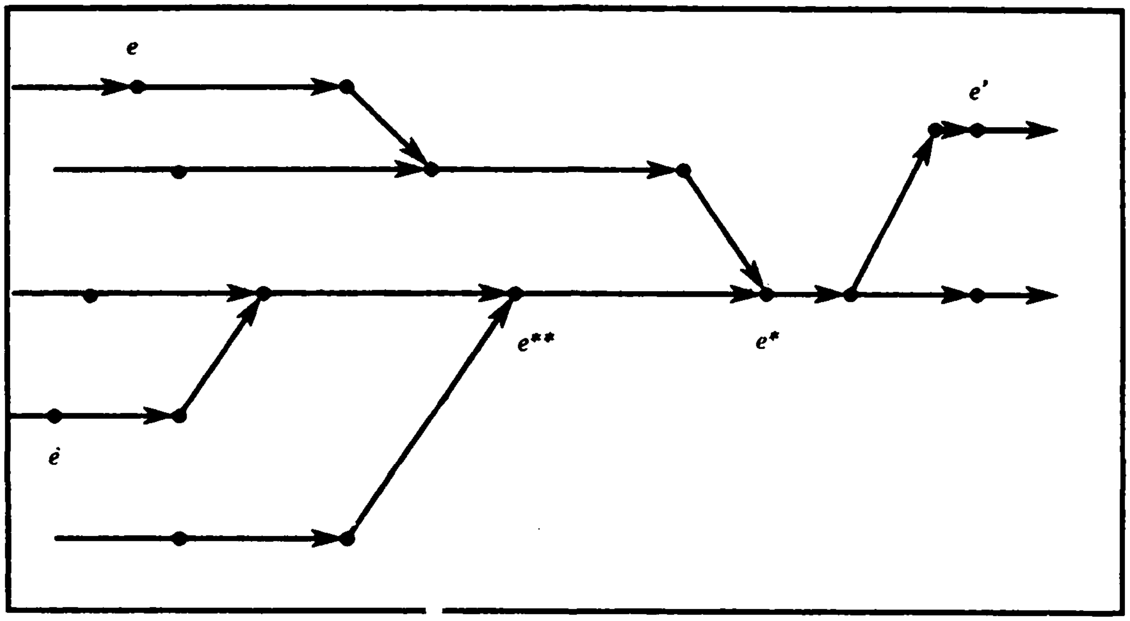
Now the correctness of the algorithm is shown.

Theorem

$(\forall e' \in E)[view(e') = V_{e'}]$

Proof

Again we use a diagram to help illustrate the proof. Chosen on the diagram are representative events for $\dot{e}$, e^*, e^{**}, e and e'.



$\subseteq$ Assume $(x,y) \in view(e')$

(1) By $\mathbf{V_1}$ and $\mathbf{V_2}$. there exists $\dot{e} \in E$ such that

(a) $\dot{e} \rightarrow e', op(\dot{e}) \in \{Insert(x,y), Update(x,y)\}$ and

(b) $(\forall e)[Removed(e,\dot{e}) \Rightarrow \neg (e \rightarrow e')]$

(2) Thus by (1b) there can not exist an $e \in$ E such that $Removed(e,\dot{e})$ and $e \rightarrow e'$

(3) By lemma 4. $(x,y) \in V_{\dot{e}}$

(4) Let us assume $(x,y) \notin V_{e'}$

(5) Let $e^* \in E$ be such that $\dot{e} \rightarrow e^* \rightarrow e'$, $(x,y) \in {}_{e^*}V$, $(x,y) \notin V_{e^*}$, with $\dot{e}$ the event which made $(x,y) \in {}_{e^*}V$, and minimal

(6) If $(x,y') \notin V_{e^*}$,for any $y' \in$ **W**, then by the table there must exist $e \in E$ such that $op(e) = Delete(x)$ and $e \rightarrow e^*$. Thus $Removed(e,\dot{e})$, a contradiction

(7) If $(x,y') \in V_{e^*}$,for some $y' \in$ **W**, $(y' \neq y)$ then by lemma 2 and $\mathbf{R_2}$ there must exist $e \in$ E such that $op(e) = Update(x,y')$ and $e \rightarrow e^*$

(8) Either $op(\dot{e}) = Insert(x,y)$ or $op(\dot{e}) = Update(x,y)$

if $op(e) = Update(x,y')$ and $op(\dot{e}) = Insert(x,y)$ then $Removed(e,\dot{e})$, a contradiction

if $op(e) = Update(x,y')$ and $op(\dot{e}) = Update(x,y)$ then by the table ([e] axis y: *Update*; [$\dot{e}$] axis x: *Update*, *AbsentSeen*, *AbsentNotSeen*)

Update/Update: $(x,y') \in V_{e^*}$ only if $Earlier(\hat{e},e)$, $\therefore$ $Removed(e,\hat{e})$, a contradiction

Update/AbsentSeen: by definition, $\hat{e} \notin {}_{e^*}SS$, and $when(e) \leq {}_{e^*}t(where(e))$. This is impossible by the minimality of e^*

Update/AbsentNotSeen: by definition, $\hat{e} \notin {}_{e^*}SS$, and $when(e) > {}_{e^*}t(where(e))$. Therefore there must exist $e^{**} \in E$ such that it is minimal and $\hat{e} \rightarrow e^{**} \rightarrow e^* \rightarrow e'$, with $where(e^{**}) = where(e^*)$, and $\hat{e} \notin SS_{e^{**}}$.
By lemma 1. ${}_{e^*}t(where(e)) \geq t_{e^{**}}(where(e))$. Thus $when(e) > t_{e^{**}}(where(e))$. Finally by lemma 5. $Earlier(\hat{e},e)$ and $\therefore$ $Removed(e,\hat{e})$, a contradiction

(9) Thus $(x,y) \in V_{e'}$.

$\supseteq$ Assume $(x,y) \in V_{e'}$

(1) By lemma 2. there exists $\hat{e} \in E$ such that $\hat{e} \rightarrow e'$,$op(\hat{e}) \in \{Insert(x,y), Update(x,y)\}$ and is responsible for placing (x,y) into $V_{e'}$

(2) Assume there exists an $e \in E$ such that $Removed(e,\hat{e})$ and $e \rightarrow e'$

(3) Let $e^* \in E$ be minimal such that $e \rightarrow e^* \rightarrow e'$ and $\hat{e} \rightarrow e^*$

(4) By inspection of the resolution table we know that once an event's item is removed from V that same event's item can not be returned to the database. Thus by (1), (2), and (3). $(x,y) \in V_{e^*}$

(5) Clearly, $op(e) \in \{Update(x,y'), Delete(x)\}$
if $op(e) = Delete(x)$ then $(x,y) \notin V_{e^*}$, a contradiction.
if $op(e) = Update(x,y')$ and $op(\hat{e}) = Insert(x,y)$ then $(x,y') \in V_{e^*}$, so $(x,y) \notin V_{e^*}$, a contradiction.
if $op(e) = Update(x,y')$ and $op(\dot{e}) = Update(x,y)$ and $Earlier(\hat{e},e)$ then by the table clearly $(x,y') \in V_{e^*}$, so $(x,y) \notin V_{e^*}$, a contradiction.

(6) By (5) then there can not exist an $e \in E$ such that $Removed(e,\hat{e})$ and $e \rightarrow e'$. Thus, $(\forall e)$ $[Removed(e,\hat{e}) \Rightarrow \neg\ (e \rightarrow e')]$

(7) $\therefore$ By (1) and (6). V_1 and V_2 hold for e'. Thus $(x,y) \in view(e')$.∇

Now the fact that the views are mutually consistent is shown.

Corollary

If sufficient correct communication between nodes occurs, and changes to the data cease (no events from the *BasicOps*), then all database views will converge to contain the same data.

Proof

Consider the theorem and a sequence of events which are taken only from *OtherOps*. If *Transmit* and *Receive* operations are occasionally performed on every node and a communication path exists between every node, then the corollary follows.∇

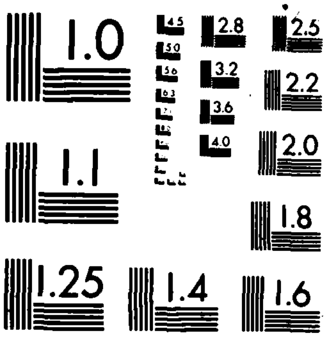
8. Summary

This paper has presented a suite of decentralized algorithms for maintaining distributed replicated data of the type which is usually found in directories or dictionaries. The algorithms are robust and are intuitively easy to understand. Although they do not attempt to guarantee serial consistency, they are adequate for many simple data storage problems. The algorithms require little support from the communication system (basically only that if a message is delivered, it is ungarbled). Applications which may benefit from the type of algorithms presented include mail systems, naming servers, appointment calendars, certain types of file dictionaries, operating system load data maintenance and distributed process control systems. The main approach taken to accomplish the goals of the algorithms (availability, performance, and simplicity) involves custom-tailoring the algorithms to the special requirements of client applications. This tailoring is simplified by using resolution tables which specify the resolution strategy for action conflicts. The correctness condition for one of the algorithms was defined and the algorithm was proved to be correct.

Acknowledgements

This work was performed while the author was a member of the Clouds decentralized global operating system group, Martin McKendry leader. Martin was invaluable in providing both technical and emotional support. Thanks to Nancy Griffeth for being an excellent opponent in the game of "trying to find a hole." Eric Allender, Richard LeBlanc, Mike Merritt, and Jerry Spinrad also provided valuable criticism. Eric was especially helpful in reviewing this work.

AD-A141 501 SOFTWARE SUPPORT FOR FULLY DISTRIBUTED/LOOSELY COUPLED 3/3
PROCESSING SYSTEMS..(U) GEORGIA INST OF TECH ATLANTA
SCHOOL OF INFORMATION AND COMPUT.. P H ENSLOW ET AL.
UNCLASSIFIED JAN 84 GIT-ICS-82/16-VOL-2 F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

References

[Allc83] Allchin, J.E., "Synchronization and Recovery in Distributed Systems," Ph.D. Thesis, Georgia Institute of Technology, in preparation.

[Bern81] Bernstein, P. and N. Goodman, "Concurrency Control in Distributed Database Systems," *Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.

[Dole82] Dolev, D. and R. Strong, "Distributed Commit With Bounded Waiting," *Proceeding of the 2nd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems*, 1982.

[Fisc82] Fischer, M.J and A. Michael, "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," *SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1982.

[Garc80] Garcia-Molina, H. and G. Wiederhold, "Read-Only Transactions in a Distributed Database," *ACM Transactions on Database Systems*, Vol. 7, No. 2, June 1982, pp. 209-234.

[Gray78] Gray J., "Notes on Database Operating Systems," *Lecture Notes in Computer Science*, R. Bayer *et. al.*, ed., Springer-Verlag, 1978, pp. 393-481.

[Jens82] Jensen, D., "Decentralized Executive Control of Computers," *3rd International Conference on Distributed Computing Systems*, Miami Florida, October 1982, pp. 31-36.

[John75] Johnson, PR. and R.H. Thomas, "The Maintenance of Duplicate Data Bases," Network Information Center Document #31507, Bolt Beranek and Newman, Inc., January 1975.

[Kung80] Kung, H.T. and C.H. Papadimitriou, "An Optimality Theory of Concurrency Control for Databases," Technical Report MIT/LCS/TM-185, Laboratory for Computer Science, Massachusetts Institute of Technology, November 1980.

[Lamp76] Lamport, L., "Towards a Theory of Correctness for Multi-user Data Base Systems," Massachusetts Computer Associates, Report No. CA-7610-0712, October 1976.

[Lamp78] Lamport , L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July 1978, pp. 558-565.

[McKe83] McKendry, M., J. Allchin, and W. Thibault, "Architecture for a Global Operating System," to appear in *IEEE INFOCOM 83*, San Diego California, April 1983.

[McQu78] McQuillan, J.M., "Routing Algorithms for Computer Networks -- A Survey," in *A Practical View of Computer Communications Protocols, IEEE Computer Society*, 1978, pp. 86-91.

[Oppe81] Oppen, D. and Y. Dalal, "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," Xerox Office Products Division, OPD-T8103, October, 1981.

[Thom79] Thomas, R., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, Vol. 4, No. 2, June 1979, pp. 198-200.

APPENDIX J

FACILITIES FOR SUPPORTING ATOMICITY IN OPERATING SYSTEMS

James E. Allchin
Martin S. McKendry

Abstract

One of the problems fundamental to operating systems is maintaining the atomicity of a sequence of operations despite concurrent activity or system/client failures. *Atomic actions* have been used for this purpose in database systems and recently in programming languages. This paper introduces support for atomicity in the kernel of an operating system. This support is not limited to managing just one type of data (*e.g.*, files) and could be used to ensure that any action (or task) be accomplished atomically on a set of user definable objects. The atomicity framework presented uses processes, actions, and objects. Requirements for atomicity are discussed and system primitives are defined which include the ability to create and terminate nested actions, control concurrency between actions, and recover from action aborts. The facilities presented provide system designers and programmers with the ability to control consistency requirements using whatever semantic knowledge is available The atomicity thus attained is called *semantic atomicity*. Unlike other work, we do not tightly bind processes to actions, thus allowing the facilities presented to be applicable to a wide class of systems (including applications where actions are supported by cooperating processes). One particular approach for integration of the facilities is discussed related to the Clouds decentralized global operating system. The desirability for semantic atomicity is illustrated through a file directory system example. Use of the facilities to address the problem of actions supported by cooperating processes is also illustrated through an example.

1. Introduction

Much of the recent work concerning reliability and data integrity in systems has focused on *atomic actions* (atomic transactions) [Gray78, Davi78, Eswa76]. We will refer to atomic actions simply as actions throughout this paper. Actions represent *tasks* which must be accomplished indivisibly. As such they form the basic units of both recovery and concurrency control and can be characterized by two properties:

- *failure atomicity*: either all results of an action are applied to the objects referenced by the action or none are applied
- *concurrency atomicity*: the effect of executing actions concurrently must be the same as if each action executed indivisibly (*i.e.*, atomically). Thus, an action's steps can be interleaved with other actions' steps so long as the result appears the same as if the actions were run serially. That is, the execution sequence is correct if it is *serializable* [Eswa76].

Actions can terminate either abnormally (by *aborting*) or normally (by *committing*). Actions which are used within other actions for failure containment reasons are called *nested actions* [Davi73, Reed78, Moss81, Lync83]. Nested actions appear atomic to the surrounding action or scope. That is, both of the atomicity properties above apply, but become relative to the current nesting scope. Thus nested actions fail independently of each other and the surrounding action, but commitment depends on the surrounding action to commit. During execution an *action activation tree* is naturally formed Nodes in the tree are actions and edges represent nesting relationships When a nested action is created, it becomes a *child* of the surrounding or *parent* action. All the immediate children of a parent action are *siblings*. *Ancestors* of some action *x* represent the set of actions which completely define the scope of *x*: these include the action *x* and all actions on the path to the root action (including the root action). *Descendant* actions are similarly defined.

An action which is not nested is called a *permanent* action because if the action completes normally, changes by the action are permanently applied. Permanent actions are root nodes in the action activation tree. Changes made by a nested action are considered temporary until the permanent root action commits. If an action (or nested action) aborts, then all descendants of the action are aborted (maintaining failure atomicity). Unless specifically qualified, the term action will denote both permanent and nested actions.

Action support is relatively commonplace in distributed data storage systems using several different implementation approaches (*e.g.*, [Svob81, Lamp81]). However, most other application areas tend to use a variety of specialized *ad hoc* techniques to attain the atomicity properties of actions when they are required. One apparent reason why *ad hoc* approaches are used is that object types are usually defined *a priori* by the action facility. Different data granule sizes may be used, but facilities do not exist which allow arbitrary objects to be defined and operated upon by actions. Many simply use disk pages or files. For example, suppose that it is desired to operate upon a specialized queue, a set, a file, a tree-based file directory and a storage allocation module atomically. In any system which rigidly structures objects, this becomes either impossible or exceeding expensive because these general objects must be mapped onto the supported objects (*e.g.*, disk pages). Thus an *extensible* (programming) environment for managing actions is desired. Other research addressing extensible environments include [Lome77, Lisk82, Reed82].

Extensible schemes which have been proposed have used atomic actions to structure processes (*e.g.*, [Lome77, Lisk82]). This approach is a very convenient structuring methodology, but it can not address certain system problems. In particular, communicating processes [Dijk68] are incongruent with this structure, even though the processes may be cooperating to perform some action. (Consider a producer/consumer example with unbounded message stream where both producer and consumer are actions.) Processes, performing actions, in this structuring approach can communicate only after one of the actions has committed. This, however, is clearly impossible if the processes must communicate to complete the actions. Unlike these prior extensible schemes we address action structure and processes independently and do not bind actions tightly to particular processes. The above mentioned problems are then avoided.

Although atomic actions address certain problem areas well, there are environments where the atomicity properties stated above are either too strong or inappropriate. It is well known that serializability is too restrictive for certain applications [Lamp76, Garc82]. In some sense a more general form of atomicity is involved in these applications. This is usually directly related to having more semantic information available [Kung79, Papa79]. However, trading serializability for performance has been noted as well [Fisc82, Jens82, McKe83]. Different levels of consistency have been discussed by Gray [Gray75], but these levels are oriented toward a simple data framework. As such, the consistency degrees suggested are not sufficient to capture the lower levels of consistency available in a general setting. In this light, a general atomicity support system should permit different degrees of atomicity to capture the necessary correctness conditions, without being overly restrictive.

We are investigating atomicity mechanisms which can be embedded in operating systems and hardware to allow applications, as well as certain portions of the operating system, to benefit from the common facilities. We believe that integration of extensible facilities for achieving atomicity into an operating system is quite novel. Even though atomicity support for data storage (in particular files) has been suggested, our work involves a much more radical integration such that arbitrary aspects of an operating system or application can be structured using actions. We specifically desire not to limit what task an action may perform. For example, we do not want the only objects which can be supported to be storage pages. While this is acceptable for integration of action facilities for storage systems like files or databases, we instead desire a general programming environment for actions where the properties of actions can be defined over any part of the system. Thus, the objects that an action may manipulate may be programmable. Each object referenced could further use nested actions when manipulating other objects. The atomicity which we desire in this environment we will call *semantic atomicity*, as opposed to the absolute atomicity of the conventional approach. That is, the meaning of atomicity depends on precisely what the action is attempting to do [Allc82]. The concept of semantic atomicity encompasses the notion of absolute atomicity as stated above.

One uniform structuring approach for systems uses data abstraction and the object model [Jone79]. Within this paper, we will structure the world accordingly. We consider this choice to be neither universally good nor bad; and the basic concepts presented for providing atomicity facilities are not limited to this particular view. Message-based systems may approach certain aspects of the atomicity problem differently (*e.g.*, assigning processes to actions), but the fundamental aspects of the atomicity facilities presented appear adequate. Thus our contribution spans both message-based and procedure-based systems.

This paper describes the general system architecture we propose for managing atomicity, the synchronization and recovery facilities we provide, and how these mechanisms might be incorporated into an actual system. As an example environment, we use the Clouds [McKe83] decentralized global operating system currently under construction for a local area network of Three Rivers Perq computers. We believe that atomicity is particularly important for distributed systems because of the independent failure modes of the nodes. Semantic atomicity is also important because of the desire in distributed operating systems to sacrifice consistency for performance [Jens82, McKe83]

Section 2 details some of the requirements which must be addressed by any atomicity facilities incorporated into an operating system kernel. Our system model and the general atomicity primitives we propose are presented in Section 3. Section 4 discusses how these primitives might be incorporated into an object-based system. Section 5 and 6 contain examples (5 illustrates synchronization and recovery in a directory object including operations which implement semantic atomicity and 6 illustrates an action performed by cooperating processes communicating through messages). Substantial additional information is available in [Allc83].

2. Atomicity Requirements

Compared to database systems, operating systems contain entities with more complex semantics. While automatic support for atomicity is highly desirable, it may be more efficient in many cases to provide the systems' constructors with the tools necessary to build atomic actions. This seems reasonable for operating systems and system applications because the writers are usually quite knowledgeable about the semantics of the system and can probably provide (cheaper) atomicity using these tools. From these tools, automatic action support could be constructed for specific application areas (*e.g.*, database systems, object repositories, action-based languages, *etc.*). Thus the approach of providing synchronization and recovery tools appears promising.

The tools approach has a tacit assumption concerning the reasons for recovery. *Errors*, unexpected conditions (such as software modules failing to meet their specification), can not be handled with this approach. We, however, are much more concerned with *failures* (expected, although undesirable, conditions--for example, node and network failures, access rights violations, or process faults such as division by zero). Thus, unlike recovery blocks and conversations [Rand78, Russ80, Shri78], failures must be anticipated.

As discussed in Section 1, it is important to address semantic atomicity. Consider a file directory. Most clients of the directory do not care when a listing is made if they see transient (uncommitted) changes. Forcing operations of this type to be atomic will result in low levels of concurrency on the directory. Of course, a file-backup client of the directory may insist on seeing a serial view. Thus, what is acceptable depends on the semantics of use. In many cases it is possible in operating systems to know *a priori* these requirements and thus (if the facilities were available) take advantage of these semantics.

It is also important not to exclude cooperating processes from the atomicity support. In fact, it appears desirable in operating systems to *not* automatically assign actions to processes. Instead a more dynamic scheme is required which will allow one process to support many actions or several cooperating processes to support one action (*e.g.*, the client/server model with cooperating servers fits this paradigm).

In general there are five areas of support necessary for atomicity. First, there must be some method for the users to create and terminate actions. Second, there should be synchronization facilities (in addition to process synchronization) which can be used by processes to maintain the atomicity requirement between actions (concurrency atomicity). Locks and timestamps [Kohl81] are typical synchronization tools used in database systems for this purpose. Third, there must be recovery facilities which permit flexible management of data necessary to recover the action to a consistent state (failure atomicity). Logs [Gray78] containing either before or after images (or both) have typically been used for this purpose in database systems. In nested action environments automatic propagation of synchronization and recovery information to the parent action is also desirable. Fourth, because the support for atomicity is not performed completely automatically, there must be facilities which permit user defined processing on the transition of an action to another state (*e.g.*, performing recovery on the *operation* → *abort* transition). Finally, there must be process agreement facilities which allow the processes performing an action to reach a consensus, despite failures, concerning action state transitions (particularly the *operation* → *commit* transition).

3. System Primitives for Supporting Atomicity

3.1 System Model

Physically we view the environment as composed of *nodes* and an interconnection *network*. The nodes communicate through messages sent through the network. The nodes contain two different types of memory: *volatile* and *permanent*. Nodes may crash (fail) erasing the contents of volatile memory, but without disturbing permanent memory. When a crash occurs we assume that all processing stops and that random messages and random changes to permanent memory do not occur. The network is also unreliable and can lose, duplicate, or re-order transmitted messages. Messages if delivered, however, must arrive ungarbled. That is, message corruption must be detectable.

There are three logical entities in the system: processes, objects, and actions. *Processes* are active agents which execute at a single node. Processes may be directly created and terminated only through the kernel at that node. Node failures can indirectly terminate a process. *Actions* are units of concurrency and recovery. Actions may span node boundaries and may be concurrently performed at several nodes. The node where an action is created is considered the *coordinator node* for the action. Actions, via processes, manipulate *objects*. An object may be considered to be an instance of a generalized abstract data type (even though not necessarily implemented this way) which can only be operated upon through well-defined operations. During an action, objects referenced must not be moved from the node where the action first referenced the objects. If an object is moved between nodes, no action may operate upon the object during the migration.

Processes, actions, and objects are identified through *processids*, *actionids* and *objectids*. Maintenance of process identification is assumed external to the action support environment, however it is assumed that the identification is unique within the node where the process is created. Action identification must be unique across all nodes. Action identifiers are provided by the action support primitives discussed in Section 3.2. Object identification need only be unique within the node where the object is located for the action support facilities. Even though system-wide uniqueness is not required by the action facilities specifically, it may be necessary for other aspects of particular systems (*e.g.*, if objects can be globally addressed). One kernel primitive is provided to generate unique objectids; this could be changed appropriately to achieve the uniqueness required.

Normally processes do not recover from node failures. However, we require a special kernel primitive which allows processes to be automatically restarted at some location following a node crash. That is, on restarting the system after a failure a *checkpointed* process will resume at some user definable location with certain variables re-initialized to checkpointed values. This is necessary to guarantee correct processing of the action state transitions.

3.2 Action Creation, Use, and Termination

Action creation is performed through the following kernel function:

function *create action* (actiontype : (permanent, nested), parent : actionid) : actionid

The actionid is an index into a kernel-protected action identification table. This table, one local to each node, contains information concerning the state of the known actions, which processes are performing the action, and which objects have been affected by the action. Processes are free to store the actionid as desired (or even pass it). In a production implementation, capabilities which associate processes to actions would probably be necessary. The actiontype parameter specifies whether the action should be created as a permanent action (no nesting) or relative to the specified parent action. It is possible that the parent does not exist anymore and in this case the caller receives an error on invocation.

Processes may execute on behalf of only one action at a time; binding actions to processes is performed dynamically. Assuming the action specified still exists, a process can become *linked* to the action through the following kernel call:

procedure *link* (newaction : actionid)

This dynamic assignment permits processes to manage several actions, if desired. This ability is particularly attractive for server processes managing several user actions, for example. Linking to an action x automatically unlinks any action y currently linked to the process. A *null* actionid is available to unlink the process from all actions.

It may be necessary to determine what actionid is currently linked to a process. This is useful for the synchronization discussed below.

function *tellid* actionid

Termination (commitment or abortion) of an action is performed as shown below. Both procedures can return errors if the action does not exist (*e.g.*, already aborted). If a process terminates, then all actions associated with the process (determined from the action identification table) are aborted. (Recall we are not addressing software errors.) Both of the termination procedures operate on the action currently linked to the invoking process.

If a nested action is being committed, all synchronization state and recovery logs are inherited by the action's parent (because the child has completed). If a nested action has visited remote nodes, a one-phase distributed commitment protocol is begun. If a permanent action has visited remote nodes, a two-phase commit protocol is used [Gray78]. Once all recovery information is safely stored in permanent memory, special user-definable procedures are performed to complete the commit processing (see below). The timelimit associated with the commit procedure is useful when multiple processes are cooperating on an action. If the associated processes do not request commitment within the specified duration, then instead of committing, the action is aborted. All cooperating processes must agree that the action is complete by executing the *commit* primitive before final commitment occurs. Thus, we avoid the domino effect [Rand78].

procedure *commit* (timelimit : timedurationtype)

procedure *abort*

As a process, on behalf of an action, accesses an object, a series of events occur. These events are diagrammed in Figure 1. Special client procedures can be defined for all three of the special events: BOA (beginning of action), EOA (end of action), and Abort.

Processes must inform the kernel when a new object on the current node has been referenced. In addition, the processing code for the events of BOA, EOA, and Abort must be defined for this object. If special event processing is not required for one or all of these, a special procedure name of *none* can be used. The same event procedure code can be shared by several objects if the processing required is the same. Event procedures can not use actions during their processing. Processes inform the kernel of the referenced objects through the following primitive:

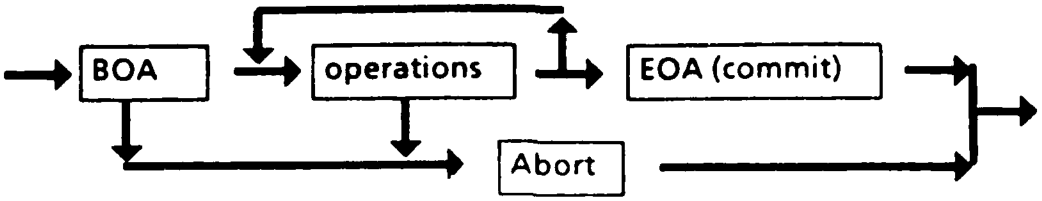


Figure 1 Action Events Related to Objects

procedure *touchobject* (object : objectid; boa, eoa, abort : **procedure**);

The first time *touchobject* is executed for an action/object pair, the operating system updates the list of objects referenced by that action. This is used to execute the associated event procedures on termination of the action. The event procedures which may be defined are

BOA beginning of action.
This procedure is performed immediately following the invocation of the *touchobject* primitive assuming that this is the first time this object has been touched by the current action.

EOA end of action (commit).
The EOA code is executed after the recovery area is safely stored in permanent storage following permanent action commitment; it is not executed on nested action commitment. This event processing procedure must be written in an idempotent manner. That is, it may be (re)executed many times due to system failures and any complete execution must be correct regardless of prior partial executions.

Abort abort action.
Once an action has been aborted by a process, the Abort code associated with all objects touched by the action is executed. The Abort event does not occur if volatile memory fails. This is explained further in Section 3.4.

Even though these are specified as procedures here, this same event scheme could be used in a message-based system. This could be accomplished by defining event messages (possibly by exceptions or emergency messages) to represent these action state changes and requiring each process to appropriately handle the events.

If a process transmits an actionid to a remote node while processing an action, the local kernel must be informed which node was accessed. This information is used for coordinating action state transitions. The following kernel call is used to inform the local kernel of the access. If the process does not actually access the remote node for some reason after executing this call, it is unimportant since the atomicity system will discover this from the remote node during action termination.

procedure *offnode* (remotenode : nodeid);

The last kernel primitive permits processes to request the state of an action. This is particularly useful when multiple processes, working cooperatively on some action, must reach agreement before deciding whether to abort or commit. This can be used in the implementation of the conversation concept [Rand78]. Section 6 illustrates this primitive in a cooperating process environment.

procedure *notify* (action : actionid; state : (active, aborted, complete, unknown))

These primitives are sufficient to manage both permanent and nested actions in an elegant manner

For example, even in a nested user action it is possible to perform a controlled violation of the current action nesting for maintaining operating system state data (*e.g.*, process queues). In addition, intrinsic to the system is the concept of cooperating processes performing an action. This is a natural extension to the use of cohorts [Gray78] in distributed database systems, where a transaction has a support process (a cohort) on each node it visits.

3.3 Action Synchronization Facilities

Processes may need to perform specialized synchronization with respect to one another if they are linked to actions. It is possible to control action synchronization via most any general process synchronization scheme, because processes have access to actionids and also have the ability to determine those conditions that constitute a conflict. However, for convenience and efficiency, we propose common action synchronization mechanisms be available in the kernel. This does not prevent coding specific synchronization as necessary to obtain additional concurrency (*e.g.*, [Lamp76]). We provide two basic action activation tree synchronization mechanisms: multi-mode locking and counting semaphores.

3.3.1 Action-based Multi-mode Locking

Locks [Eswa76] are a reasonable choice for one mechanism, because there are many concurrent data structure maintenance algorithms in operating systems which use a locking model (*e.g.*, [Kwon82]) Our approach requires a lock compatibility table to be defined before lock operations can be used. The goal is to provide a framework more general than simple read/write locking modes. The directory example presented in Section 5 illustrates why this approach is desirable.

The lock domain, mode compatibilities, and the lock protocol used are determined by the process defining them. *By associating a domain with each lock type, it is possible to solve the phantom* problem [Eswa76]. That is, entities do not have to exist at the time they are locked. Again, the directory example illustrates the significance of this. By allowing programmers to control lock protocols, coordination schemes such as non-two-phase protocols can be used [Moha82], driven by the semantics of the accessing pattern. Below are the locking operations.

```
modetype        = integer;         {system dependent}
lockidtype      = integer;         {system dependent}
instanceidtype  = integer;         {system dependent}
compatibilities = record
                      moderequesting  : modetype;
                      compatibleset   : set of modetype;
                  end;

procedure defineconflict (lockid : lockidtype; accesstable : set of compatibilities)
function setlock (lockid : lockidtype; thing : instanceidtype; m : modetype; timeout :integer)
                  : (okfirsttime, okothertimes, timeout, invalid)
function testlock (lockid : lockidtype; thing : instanceidtype; m : modetype; aid : actionid)
                  : (ok, conflict, invalid)
function releaselock (lockid : lockidtype; thing : instanceidtype; m : modetype)
                  : (ok, notset, invalid)
function releaseall (lockid : lockidtype) : (ok, invalid)
```

Suppose a process requests a lock in some mode m and is linked to action x. If only ancestor actions of x from the action tree hold incompatible lock modes to m, the lock is set. For example, in the simple shared read / exclusive write situation, only x's ancestors can hold write mode locks and still permit x

to obtain a write lock. As actions commit the ownership of the locks propagate to the immediate parent action. If a *setlock* is executed and the lock cannot be set because of mode incompatibilities, the process is suspended until the lock can be set or until a timeout occurs. Once *x* operates on some lock, *x*'s ancestors may not touch that lock until *x* terminates.

The special status indicators of *okfirstime* and *okothertimes* used in *setlock* are provided so that applications can detect when they have already locked an object in the given mode. This information is useful for determining when it is necessary to save the state of the object through the recovery facilities.

3.3.2 Action-based Counting Semaphores

Locking as presented above allows actions to avoid one another in order to achieve serializability. It is also desirable in some cases to have the ability to apply additional ordering constraints. For example, guaranteeing one sibling will execute before another appears to be a common problem (see for instance the example in Section 4.1). Our work is novel in generalizing action tree synchronization in this manner.

In a nested action environment semaphore values are managed according to the visibility that an action has depending on the action's location in the action activation tree. Thus an action may use a *V* value from an ancestor, but not from an sibling. Upon commitment the changes to the semaphore are appropriately propagated. If the action aborts, the borrowed *V* values from it's ancestors are returned. Thus this mechanism is an extension of standard counting semphores to the realm of reliable computing in a nested action environment. As with the locking mechanism presented above, once a child executes a semaphore operation on some semaphore, no ancestor may reference that semaphore until the child completes. If processes are cooperating performing some action, then because they will be using the same actionid, the action-based semaphores become equivalent to standard semaphores. Further details and the associated algorithms are included in [Allc83]. The operations are shown below:

```
semaidtype      = integer;          {system dependent}

function definesemaphore (initialvalue : integer) : semaidtype
function destroysemaphore (semaid : semaidtype) : (ok, invalid)
function actionP (semaid : semaidtype; timeout :integer) : (ok, timeout, invalid)
function actionV (semaid : semaidtype) : (ok, invalid)
```

3.3.3 Guaranteeing Progress

Specific support for deadlock and livelock is not provided by the kernel. Appropriate system structure and associated lock and semaphore protocols can prevent deadlock in many cases. However, if deadlocks can occur in the system, the responsibility for appropriate action lies with the implementor (using timeouts, *etc.*). If locking were the only mechanism used for action synchronization, then deadlock detection would be straightforward (although probably expensive). However, as discussed above, processes may perform specialized synchronization between actions without using locks. This makes the problem extremely difficult because it may not be possible to determine which actions another action may be waiting for. We will not discuss this further here.

3.4 Action Recovery Facilities

Logging appears to be a reasonable method for maintaining action recovery information. To support logging, system primitives are available to write and read records associated with action/object pairs.

During the life of an action, records may be written to the log. If the action aborts, the log is deleted after the Abort processing event is complete. If the action commits, the log is inherited by the action's parent. This involves no data movement from the recovery log, simply a notation to be made regarding which action owns the log. If the action is permanent, then the log is placed on permanent storage. Each node maintains a local log for any action known at that node and each saves their portion of the complete log during commitment. After the logs are safely stored, the event EOA occurs. The log is automatically discarded upon completion of the event EOA.

Any information desired may be placed in the log. However, because the log for an action does not become saved in permanent memory until the associated permanent action commits, recovery from an action abort cannot require the log to recover across a volatile memory failure. In general, though, we suspect that assuming actions will complete is the proper assumption for operating systems and many applications. This optimsitic viewpoint dictates that changes be made to the current version of shared entities using the log to maintain the unaltered version. This approach results in much of the overhead associated with supporting actions to be tied to abort and not commitment.

During the first write by an action to the log for some object, the log is officially created. To notify the recovery facility that the log records should be returned, a reset operation is used. If the items to be saved are memory pages, then it is possible to integrate some of the logging system with the memory management system (*e g* , by manipulating page tables). The log write, read, and reset primitives are defined below.

```
procedure writelog (object : objectid, <array of items to save, length and address>)

procedure readlog (object : objectid,
                  <array of addresses of where to place returned items>, status : (ok, endoflog))

procedure resetlog (object : objectid)
```

One possible extension of the recovery facilities involves client-controlled checkpointing of the log into a staging area of permanent memory during nested action commitment. However, this can become expensive when multiple nodes are involved forcing two-phase commitment to be used during every action commitment. However, for long running actions, this may be necessary. The general approach in this case would be to subdivide the long running action into a group of nested actions which could checkpointed upon completion. The parent action would simply guarantee that the changes would only become visible if the entire task was accomplished.

3.5 Implementation Structures

Shown below is a rough sketch (Figure 2) of the structure necessary to support the facilities discussed above

4. One Possible Application Using the Primitives

This section describes how the system primitives defined above might be incorporated into an actual operating system. The approach used in Clouds [McKe83] is to define a programming language (a Pascal derivative) which has specific support for actions and atomicity. The compiler converts the language constructs into the necessary system primitives. The language approach is convenient because it organizes the atomicity primitives into a uniform structure and removes some of the causes for errors in their use.

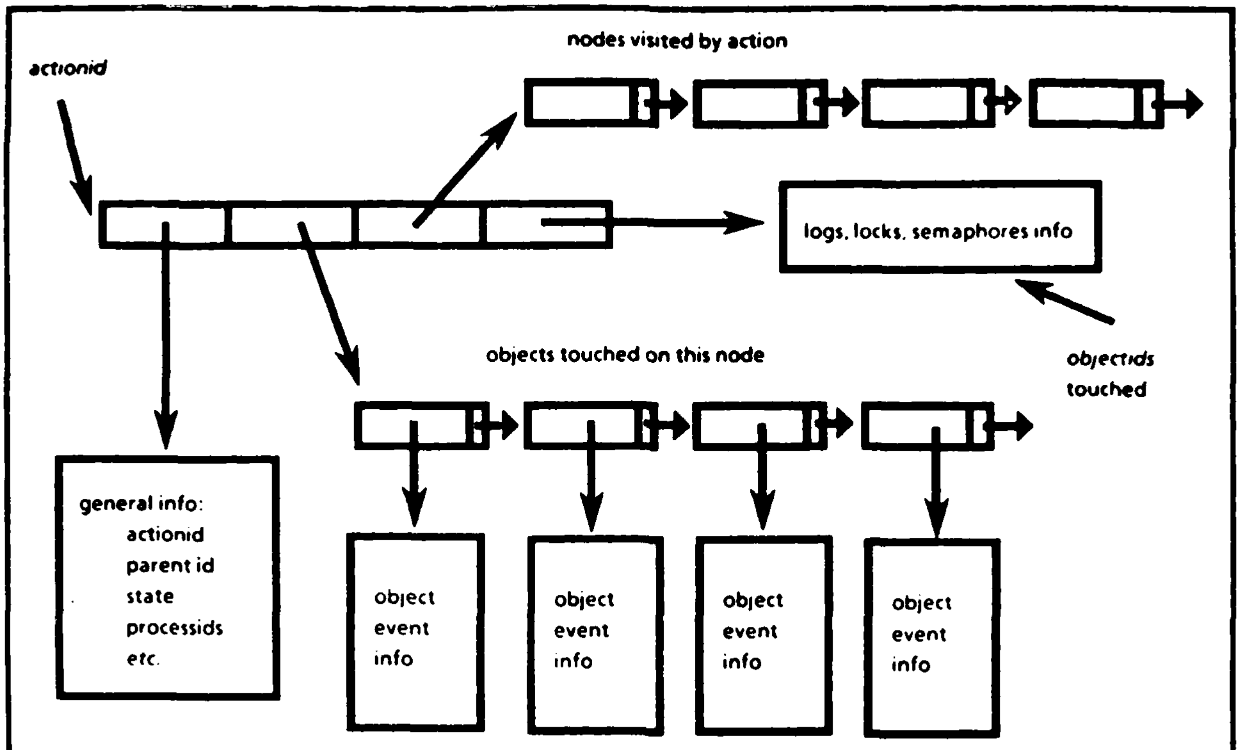


Figure 2 Conceptual Data Structures

In Clouds, an object type is a globally-named generalization of an abstract data type which can only be operated upon through well-defined operations. An object is an instance of some object type. Objects are distributable in Clouds and may reside anywhere in the network. For the purposes of this paper we will consider only objects which support actions (*viz.*, support recovery and action synchronization). Objects in this class can be considered to be composed of three basic components: the data portion (data and the operations on the data), synchronization necessary for shared access, and recovery control. In addition, some object state may be kept in permanent memory in order to survive volatile memory failures. Figure 3 illustrates a conceptual internal structure of an object.

Nested actions can be used to specify units of synchronization and recovery. Each object type operation can be denoted as an action definition (similar to [Lome77]). Action atomicity is used to transform the state of the objects referenced in the action into a new (consistent) state. Semantic atomicity is desired for all actions and it is the responsibility of each object type to ensure that appropriate abstract behavior is provided.

Object definers can control synchronization among actions by specification statements which are provided when an object type is defined. An access statement is used to specify the object operation compatibility necessary to arbitrate access between actions. These operation compatibilities are managed using generalized locking modes (possibly one for each object procedure) to ensure the specification. The locks are managed through a two-phase locking technique which ensures that serializable abstract behavior can be achieved. The locks are held until action termination. Incompatibility betwen a requesting action and an accessing action of some object causes the requesting action's process to block until some specified timeout occurs or access is allowed.

To force a certain path or order of executions of the procedures by actions, the order statement can be used. The format of this statement is similar to path expressions [Camp74] and can be directly compiled into operations on action-based semaphores. Sequencing (;), repetition (n:), and alternation

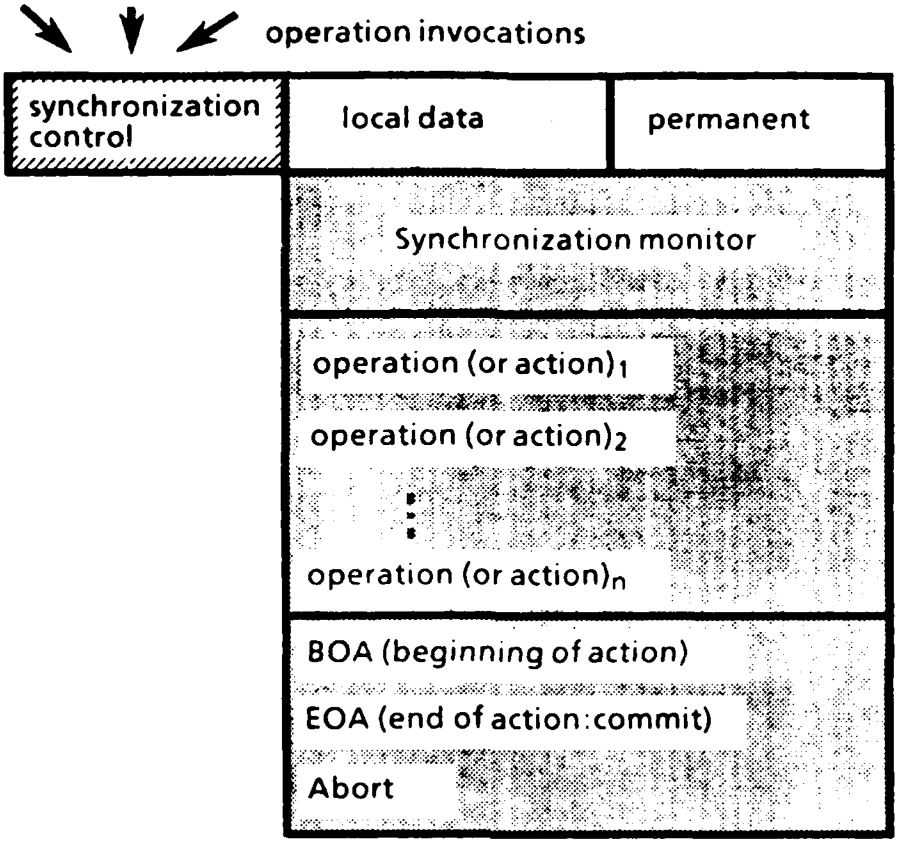


Figure 3 Data Object Structure

(,) can be specified in the **order** statement.

Object definers are additionally provided with the tools necessary to synchronize action access to local data within the object. This is accomplished by using a **sync monitor** similar to a standard synchronization monitor [Hoar74]. It can be used to control mutually exclusive access to local variables within an object. Any object procedures can be placed into the monitor. Synchronization within the **sync monitor** is possible through statements which allow events to be waited upon and signalled. Programming arbit-ary action synchronization is possible through the synchronization monitor and via lock statements which are directly compiled into lock system primitives. Thus a dual approach for action synchronization is provided: static specification when an object type is defined and dynamic programming tools to address special problems. This generality permits the tradeoffs of simplicity and performance to be adequately addressed. The synchronization facilities are discussed in more detail in Section 4.1.

Each object type contains special procedures to manage the action events of BOA, EOA, and Abort. The compiler generates some additional setup code for these procedures, but in general they behave in the same manner as discussed in Section 3.1. Each object type can also define variables which must be made permanent upon commitment of a permanent action.

In the Clouds object framework, actions can be organized naturally by objects which reference operations (which may be nested actions) on other (conceptually lower level) objects. For example, consider the object types shown in Figure 4. When a *createfile* operation is executed the *getspace* and *createentry* actions compose to form the *createfile* action.

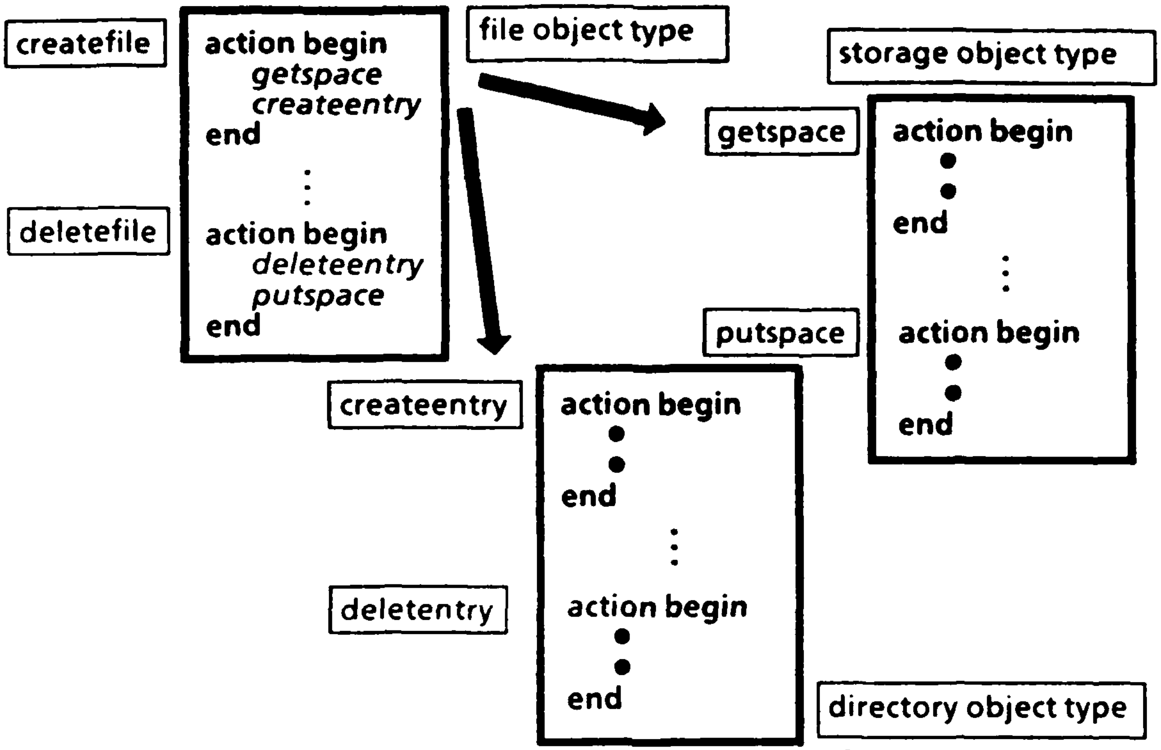


Figure 4 Natural Nesting Example

Invocation of object operations by a process is performed by procedure call.

```
<capability for object instance>.<operation>(<parameters>)[weak]
[exception
    <cause1> : <statement list>
    <cause2> : <statement list>
    others   : <statement list>
end ]
```

By default the call is reliable and is performed in a manner which ensures "once and only once" semantics (even if the target object is located on a remote node) [Spec81]. The calling process waits for completion of the call or until an exception is raised (*e.g.*, timeout). If the target object does not support actions, an error is returned if the executing process is acting on behalf of an action. If an action has been linked to the executing process, the compiler generates code to notify the kernel accordingly (through *touchobject*). This is used to return to the objects upon commit or abort of the action (refer to the EOA and Abort events discussed above). The optional keyword **weak** specifies that no value is to be returned and that if the target object is on another node in the network, then only one *send* need be performed (no waiting); the transmission is assumed unreliable. The operation therefore can be executed zero or at most one time. This option can not be used if the executing process is linked to an action. If an exception (*e.g.*, timeout) is raised and actions were created during the invocation, then these actions are aborted.

4.1 Action Synchronization

The access specification, as discussed above, is used to state the object type operation compatibilities in order to arbitrate access between actions. Incompatibility, a *conflict*, causes requesting actions to wait for the conflict to be removed (or until some specified timeout occurs). The general form of the specification follows:

<compatibility> ::= <mode requesting$_1$>: [<mode held$_1$>, ..., <mode held$_n$>]

access = (<compatibility$_1$>; ... <compatibility$_m$>)

For example, **access** = (read : [read]; write : []) represents the usual one writer / multiple reader synchronization assuming that there are two operations on the object type (read and write).

Locks can be declared and then manipulated via the system primitives discussed in Section 3.3. The format of the declaration follows:

lockvariable : **lock** (<compatibility$_1$>; ... <compatibility$_m$>) **domain** = instanceidtype

The **order** specification is used to state specific orderings among the operations which must be enforced. Since this is similar to path expressions, only an example will be given here. Consider a spool queue with the operations of *enter* and *remove*. Assume there can be only *n* entries in the queue maximum and that we desire *remove* operations to wait if no *enter* operation has been committed relative to the action invoking the *remove* operation. The specification might be given as follows:

order = n : (enter; remove)

This specification enforces that at least one *enter* is committed before a *remove* operation is allowed and that at most *n enter* operations can be performed before a *remove* is committed.

4.2. Recovery Facilities

The recovery facilities are again very similar to the corresponding system primitives. However, specifying the objectid is not required (supplied by the runtime system) and each log record is typed by a variable placed first in the log record which contains the name of the operation which performed the *writelog*. The format then is

save (<var$_1$>, <var$_2$>, ..., <var$_n$>) {corresponding to *writelog*}

restore (<logrectype>, <var$_1$>, <var$_2$>, ..., <var$_n$>) {corresponding to *readlog*}

5. A Directory Example

For convenience in this example we will use the language notations presented in Section 4. The purpose of the example, however, is not to defend particular language constructs, but rather to illustrate the use of the atomicity facilities. In following we analyze how a directory object type might be defined using the facilities presented in this paper. We show two possible approaches using different levels of sophistication to achieve different amounts of concurrency. The first approach requires only an access statement to control action interleavings. The second requires an alternative specification and minor programming, but achieves higher concurrency. A more formal treatment of this design process in presented in [Allc83].

Suppose we wish to create an action-based directory object (such as the one shown in Figure 4 with *Add* and *Delete* substituted accordingly) with the following operations:

Add(k : key; v : value) : status

```
Delete(k : key) : status
Lookup(k : key) : status, value
ListSerial(currentkey : key) : status, value, nextkey
ListApprox(currentkey : key) : status, value, nextkey
```

Let us assume that all operations other than *ListApprox* require a serially consistent view of the directory, but that *ListApprox* has semantics such that the invoking action will accept seeing (possibly) uncommitted changes. One possible access specification is given below.

```
access = ( Add        : [ListApprox];
           Delete     : [ListApprox];
           Lookup     : [Lookup, ListSerial, ListApprox];
           ListSerial : [Lookup, ListSerial, ListApprox];
           ListApprox : [Add, Delete, Lookup, ListSerial, ListApprox])
```

This specification, although correct, may not achieve an acceptable level of concurrency. Even though two actions could correctly operate on different keys in the directory, this is not allowed by the specification.

To improve concurency a different access specification could be used together with programming to specifically control directory entry sharing. In the alternative specification below we only synchronize the operation *ListSerial*; the other operations can be synchronized via the built-in lock facility.

```
access = ( Add        : [Add, Delete, Lookup, ListApprox];
           Delete     : [Add, Delete, Lookup, ListApprox];
           Lookup     : [Add, Delete, Lookup, ListSerial, ListApprox];
           ListSerial : [Lookup, ListSerial, ListApprox];
           ListApprox : [Add, Delete, Lookup, ListSerial, ListApprox])
```

We declare a lock as follows:

```
x : lock (read : [read]; change : [ ]) domain = key
```

We can then use *setlock* and *releaselock* to dynamically control action synchronization on the directory entries. Using this approach the *Add* operation might appear as follows:

```
action Add (k : key; v : value)
begin
    setlock (x, k, change, timelimit);
    ... {put entry into the directory}
    save (k, v)  {implemented through writelog}
end;
```

Note that we may lock instances which do not exist, thus avoiding the phantom problem and preventing loss of serial consistency. The choice of synchronizing *ListSerial* at the directory level instead of the key entry level avoids the overhead of acquiring and releasing a lock for each key. Instead only one lock must be accessed. Thus, tradeoffs involving granule size are possible. The Abort event code might appear as follows in the directory object.

```
entry procedure Abort;
var
    k : key;  v : value;
begin
```

```
        restore (logrectype, k, v, stat);
        while (stat $\neq$ endoflog) do
        begin
            case logrectype of
                add     : {remove entry from directory}
                delete  : {put entry back into directory}
            end;
            restore (logrectype, k, v, stat)
        end;
        {locks are automatically released by the runtime system using releaseall}
    end;
```

6. A Cooperating Process Example

There are many interactions among processes which do not necessarily involve cooperation of the processes to complete the actions the processes are performing. Consider message queues where the receiving process is not allowed to examine messages until the sender process commits the group of messages sent. A typical example is a spooling system where the spooler process does not see producing processes' output until it is committed In a sense the output is *cached* until the commit occurs. This situation appears quite common and can easily be supported by the atomicity facilities presented. The language structures associated with the Clouds system also model this paradigm.

There are however, examples where multiple processes must interact to complete an action. We now illustrate how the atomicity facilities discussed in this paper might be applied to solve this class of problems. Presented below is a sketch of two processes which cooperatively perform an action. As discussed before, commitment is possible only if both processes request commitment. If one of the processes aborts, the other one can detect this through *the notify* system primitive and can then abort the action also.

Process$_1$:

```
    create action
    Send actionid
    link to action
    repeat until done
        check action status
        if action aborted, then abort
        do work
        write log records as necessary
        if error abort
        send message
    end
    ...
    commit
```

Process$_2$:

```
    receive actionid
    link to action
    repeat until done
        check action status
        if action aborted, then abort
        receive message
        do work
        write log records as necessary
        if error abort
    end
    ...
    commit
```

Process$_1$ is responsible for initially creating the action; it then communicates the actionid to process$_2$. Both processes then link to the action and begin processing. Both processes must occasionally check the state of the action and abort if the other process has already aborted the action. Only if both reach commit will the action actually be committed.

The processes can define the objects in any manner that is convenient since the primitives primarily use the objectids as simply a manner to structure the logs. If the two processes manipulate the same object, the first one to issue *touchobject* is the process responsible for performing the action event procedures (*e.g.*, EOA). As desired, both processes, because they are performing the same action, will write to the same log. Of course, if the processes could enter a shared object concurrently, standard process synchronization must be used. Since they are performing the same action, interference can not be prevented by action synchronization.

7. Summary

This paper has explored the issues involved with integrating facilities to support atomicity into the kernel of an operating system. For generality these facilities should not bind actions and processes tightly permitting either a single process or multiple processes to perform an action. It has been suggested that a more general type of atomicity, *semantic atomicity*, is desirable for efficiency in some cases. It has been proposed that system designers and programmers be given direct control over accomplishing atomicity (both concurrency and failure). We have presented a set of requirements for supporting these kinds of tools. These requirements include the ability to create and terminate actions, to control concurrency between actions, to recover from action failures, to perform special processing on transitions in the state of actions, and to incorporate process agreement facilities which allow processes performing an action to reach a consensus concerning action state transitions. A set of kernel primitives for atomicity was presented within a framework of processes, actions, and objects. The generality for message-oriented systems was also discussed. The mechanisms appear especially important in distributed environments. A distributed operating system environment was used to demonstrate one possible approach for actual integration of the primitives. Finally two examples were presented: a directory system (using some semantic knowledge concerning the actions operating on the directory) and two processes cooperatively performing an action.

Our work has addressed a fundamental problem confronting operating systems, particularly distributed ones. It appears that a significant advance in reliability and system organization might be possible with a well engineered set of orthogonal mechanisms to address atomicity.

References

[Allc82] Allchin, J.E., and M.S. McKendry, "Object-based Synchronization and Recovery," Technical Report GIT-ICS-82/15, Georgia Institute of Technology, September 1982.

[Allc83] Allchin, J.E., "Synchronization and Recovery in Distributed Systems," Ph.D. Thesis, in preparation.

[Camp74] Campbell, R., and A. Habermann, "The Specification of Process Synchronization by Path Expressions," **Lecture Notes in Computer Science 16**, Springer-Verlag, 1974.

[Davi73] Davies, C, "Recovery Semantics for a DB/DC system," *Proceedings of the 1973 ACM National Conference,"* pp. 136-141.

[Davi78] Davies, C, "Data Processing Integrity," from **Computing System Reliability**, Cambridge University Press, 1978, pp. 288-354.

[Dijk68] Dijkstra, E.W., "Cooperating Sequential Processes," in **Programming Languages**, F Genuys, ed., Academic Press, New York, 1968.

[Eswa76] Eswaran, K., J. Gray, R. Lorie, and I. Traiger, "The Notions of Concurrency and Predicate Locks in a Database System," *Communications of the ACM*, Vol 19, No. 11, November 1976.

[Fisc82] Fischer, M., and A. Michael, "Sacrificing Serializability to Attain High Availability in an Unreliable Network," *SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1982.

[Garc82] Garcia-Molina, H., and G. Wiederhold, "Read-Only Transactions in a Distributed Database," *ACM Transactions on Database Systems*, Vol. 7, No. 2, June 1982, pp. 209-234.

[Gray75] Gray, J., *et. al.*, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," IBM Research Report RJ1654, September 1975.

[Gray78] Gray, J., "Notes on Database Operating Systems," in **Lecture Notes in Computer Science**, R. Bayer, *et. al.*, eds., Springer-Verlag, 1978, pp. 393-481.

[Hoar74] Hoare, C.A.R., "Monitors: An operating System Structuring Concept," *Communications of the ACM*, Vol 17, No. 11, November 1976, pp. 624-633.

[Jone79] Jones, A., "The Object Model: A Conceptual Tool for Structuring Software," in **Operating Systems: An Advanced Course**, R. Bayer, *et. al.*, eds., Springer-Verlag, New York, 1979, pp. 8-16.

[Jens82] Jensen, E.D., "Decentralized Executive Control of Computers," *Proceedings of the Third International Conference on Distributed Computing Systems*, Miami, 1982.

[Kohl81] Kohler, W.H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 149-183.

[Kung79] Kung, H.T., and C.H. Papadimitriou., "An Optimality Theory of Concurrency Control for Databases," *Proceedings of the 1979 SIGMOD Conference,"* Boston, MA, May, 1979.

[Kwon82] Kwong, Y. and D. Wood, "A New Method for Concurrency in B-Trees," *IEEE Transactions on Software Engineering*, Vol SE-8, No. 3, May 1982, pp. 211-222.

[Lamp76] Lamport, L., "Towards a Theory of Correctness for Multi-user Data Base Systems," Massachusetts Computer Associates, Report No. CA-7610-0712, October 1976.

[Lamp81] Lampson, B.W., M.Paul, and H.J. Siegert, **Distributed Systems - Architecture and Implementation**, Springer-Verlag, Berlin, 1981.

[Lisk82] Liskov B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust Distributed Programs," *ACM Symposium on Principles of Programming Languages*, January 1982, pp. 7-19.

[Lome77] Lomet, D.B., "Process Structuring, Synchronization, and Recovery using Atomic Actions," *Proceeding of an ACM Conference on Language Design for Reliable Software*, SIGPLAN Notices 12, No. 3, March 1977, pp. 128-137.

[Lync83] Lynch, N., "Concurrency Control for Resilient Nested Transactions," *SIGACT - SIGMOD Conference on the Principles of Database Systems*, March 1983.

[McKe83] McKendry, M.S., J.E. Allchin, and W.C. Thibault, "Architecture for a Global Operating System," to appear *IEEE INFOCOM 83*, April 1983.

[Moha82] Mohan C., D. Fussell, and A. Silberschatz, "Compatibility and Commutativity in Non-Two-Phase Locking Protocols," *SIGACT - SIGMOD Conference on the Principles of Database Systems*, Los Angeles, Ca., March 1982.

[Moss81] Moss, J., "Nested Transactions: An Approach to Reliable Distributed Computing," Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA., 1981.

[Papa79] Papadimitriou, C., "Serializability of Concurrent Updates," *Journal of the ACM*, Vol. 26, No. 4, October 1979, pp. 631-653.

[Rand78] Randell, B., P.Lee, and P. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys*, Vol. 10, No. 2, June 1978, pp. 123-165.

[Reed78] Reed D., "Naming and Synchronization in a Decentralized Computer System," Ph.D. dissertation; Tech. Rep. TR-205, M.I.T. Lab for Computer Science, September 1978.

[Russ80] Russel, D.L., "State Restoration in Systems of Communicating Processes," *IEEE Transactions on Software Engineering*, Vol SE-6, No. 2, March 1980, pp. 183-194.

[Shri78] Shrivastava, S.K., and J.P. Banatre, "Reliable Resource Allocation between Unreliable Processes," *IEEE Transactions on Software Enginerring*, Vol SE-3, No. 3, May 1978, pp. 230-241.

[Spec81] Spector, A., "Multiprocessing Architectures for Local Computer Networks," Ph.D. Thesis, Stanford University, August 1981.

[Svob81] Svobodova, L., "A Reliable Object-oriented Data Repository for a Distributed Computer,", *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 47-58.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.