
IRI

DEPARTMENT OF COMMUNICATIONS

GOVERNMENT OF CANADA

SIMULATION STUDY OF USES OF A
COMPUTER NETWORK

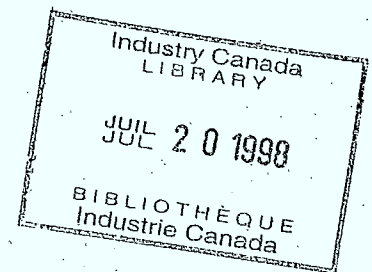
Industrial Research Institute
University of Waterloo
Waterloo Ontario

P
91
C655
G458
1971

DRAFT

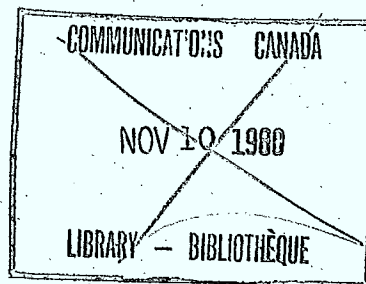
UNIVERSITY OF WATERLOO
INDUSTRIAL RESEARCH INSTITUTE

DEPARTMENT OF COMMUNICATIONS
GOVERNMENT OF CANADA



¹⁰
SIMULATION STUDY OF USES OF A

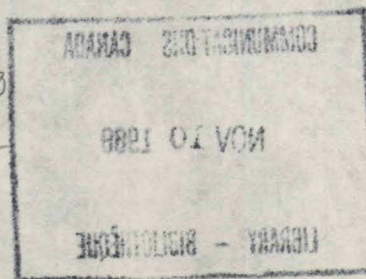
COMPUTER NETWORK & *draft*



January 26, 1971

P
91
C655
G458
1971

DN 8405650
DL 8430817



SIMULATION STUDY OF USES OF A COMPUTER NETWORKINTRODUCTION

The University of Waterloo, through its Department of Computing Science and Computing Centre, has expertise, experience, and facilities in computer/communications. We propose to exploit and extend them by addressing some of our personnel and facilities to the study of Canadian computer/communications problems. This is one of a set of coordinated research proposals; because of the timeliness and urgency of this particular research problem, this proposal is being submitted first.

BACKGROUND

Networks of computers are being constructed in various places. While there have been many statements made concerning the anticipated usefulness of such networks, the prime approach has been to construct them first, and then discover the services that they can provide. This is understandable, for the actual construction of the network strongly affects its use, and until its properties are known, no other approach is feasible.

However, there are now several networks under construction with basically different design philosophies. These philosophies do affect the user, and the constructions are sufficiently advanced to enable us to see what system parameters will result. Before building more hardware, it is now sensible to sit back and see how plausible or practical certain proposed applications are, under each design philosophy.

PROPOSED RESEARCH

We propose to perform theoretical and simulation studies of certain uses that can be made of a computer network via facilities provided by subscribing computers of the network. Such facilities include a system to SPOOL (See below) jobs to be run on computers of the network, a system to administer the sharing of jobs to be executed by computers of the network, as well as a central file system shared by the computers of the network. We intend to determine the functions of each facility and then perform a functional simulation of the facility. The object of the simulation study is to determine whether such facilities are technically feasible and economically justifiable on a computer network.

The system to SPOOL jobs to be run on computers of the network would accept job requests from various sources, order the job requests (perhaps by priority) for each processor of the network, and supply jobs to processors on demand.

The system to administer the sharing of jobs to be executed by computers of the network keeps records and makes enquiries of processors on the network to determine their loads. For each incoming job, cost, anticipated turnaround, and possibly other parameters are considered in determining which of the computers capable of executing the job should do so.

Centralized file systems are desirable as they allow great economies of scale and sharing; thus, for example, a single large disk file could provide file storage to a local network of small computers.

far cheaper than could be done individually. A major issue in organizing such a file facility is the grade of service offered: "wholesale" or "retail". If the central file system of a network were organized as a wholesale operation, it would allocate and otherwise manage files for computers of the network without knowing or caring about the organization or contents of them. Retailing involves more direct control of file format and management by the central file system so that files can be shared by processors other than the originator.

Explicitly, the simulations will consist of program modules that mimic the actions of the network switches and various subscribers, communication between the modules being limited to conventions that reflect transmissions through the network. This kind of simulation is necessary in order to examine the technical feasibility of proposed facilities. To do it we need the ability to do user-controlled multi-programming and some of the aspects of multiprocessing. The program structure providing these is called a coroutine system. Fortunately a coroutine system extension to Fortran is being developed at the University of Waterloo.

CURRENT STATUS OF THE RESEARCH

We have chosen to simulate the following networks:

- (a) The U.S. Advanced Research Projects Agency (ARPA) network designed by Bolt, Beranek, and Newman, Inc.
- (b) The so-called Davies network being implemented by the National Physical Laboratory of Great Britain.

- 4 -

(c) The experimental network being developed by A.G. Fraser at Bell Telephone Laboratories, Inc.

(d) The OCTOPUS network of Lawrence Radiation Laboratories

We are studying their properties, and are planning to collaborate with the designers and implementers of each of these networks.

The functional design of the applications to be simulated has been studied. We think we know what kind of data we are seeking from the simulation.

Coroutines can now be written and run with our system, although the translator extending Fortran by coroutine operations is still being implemented. This coroutine system has been described in a paper that has been submitted for presentation at IFIP 1971 Congress. A copy of the paper is included as an appendix.

TIME SCALE

By October 1971 the coroutine system should be running completely, the simulation of the SPOOLING system should be running, and an analysis of possible designs of the central file system should be completed.

By March 1972 the study of the SPOOLING system should be complete and simulation of some aspects of the central file system should be underway.

It is expected that useful results will be available from all phases of the project in approximately two and one-half years.

- 5 -

INVESTIGATORS

The Project Director for this research will be Professor D.D. Cowan. The Principal Investigator will be Professor W.M. Gentleman, assisted by three graduate students.

PATENT RIGHTS

It is not anticipated that any patentable idea will evolve from this investigation; however, should a patent arise it will be in the names of the Project Director and the Principal Investigator and the rights will be offered for assignment to the Department of Communications, Government of Canada, for considerations to be negotiated.

PUBLICATION RIGHTS

The Project Director, the Principal Investigator and the graduate students involved will retain the rights to publish the methods and general findings of this study.

REPORTS

Reports of the work and findings will be submitted periodically at natural division points of the work.

A summary report will be provided at the conclusion of the first year of work.

In the event that no report is provided in any three month period, a brief technical progress statement will be submitted before the expiry of such period. Such statement will indicate the work undertaken in the period and planned for the next three month period or longer.

BUDGET FOR ONE YEAR

We submit the following budget for this research for one year:

Support of three graduate students at \$3,500 per year per student	\$ 10,500.00
Professional fees	2,500.00
One secretary one day per week	1,000.00
15 hours of computer time at \$654.50/hour	9,817.50
20 cylinders of file space at \$5/cylinder/month	1,200.00
Books	100.00
Travel	1,500.00
Telephone and miscellaneous expenses	500.00
University overhead	<u>8,135.25</u>
TOTAL	<u>\$ 35,252.75</u>

Invoices will be provided quarterly: 3, 6, 9 and 12 months after authorization to proceed with the work.

AGREEMENT

It is agreed that the work should proceed in accordance with this proposal.

DEPARTMENT OF COMMUNICATIONS
GOVERNMENT OF CANADA

UNIVERSITY OF WATERLOO

Per: _____

Per: _____

J. W. Tomecko, Director
Industrial Research Institute

Dated: _____

Vera Leavoy
Assistant to Comptroller

In consideration of being retained by the University as Principal Investigator in connection with this project, I hereby agree with the University to be bound by the terms of the foregoing contract.

Dated at Waterloo, this 26th day of January, 1971

Witness to signature

D.D. Cowan

Witness to signature

W.M. Gentleman

Witness to signature

APPROVED BY

Chairman,
Department of Applied Analysis and Computer
Science

APPROVED BY

Dean of Graduate Studies

Department of Applied Analysis
and Computer Science

Research Report CSRR 2031

A PORTABLE COROUTINE SYSTEM

by

W. Morven Gentleman

A PORTABLE COROUTINE SYSTEM

by W. M. Gentleman *

University of Waterloo

Waterloo, Ontario

Canada

Abstract

Coroutines have been around at least since Conway introduced the terminology in 1963. Nevertheless, except in discrete event simulation and in some systems programs, they have rarely been used. A belief that this is because they have not been widely available in a higher level programming language has motivated the development of a portable coroutine system. This system is based on an extension of Fortran that provides a very general form of coroutines in addition to the standard program units such as subroutines and block data subprograms. The compiler, its object code, and the supplementary service routines are all essentially ANSI Fortran programs such that implementation at a new installation will be simple. This paper describes the system, tells how it is implemented, and discusses some uses of coroutines.

* This research was supported by the National Research Council of Canada.

A PORTABLE COROUTINE SYSTEM

Introduction

There is general agreement that modularity in programming is a good thing, exposing the essential structure of the program and simplifying debugging and modification. The most familiar kind of program module is the subroutine. The usual realization of subroutines has, however, several features that limit the ways they can be used.

Coroutines are another kind of program module (4, 5, 8, 9, 11). They provide greater flexibility than subroutines because they separate the three actions involved in calling a subroutine -- obtaining an executable copy of the routine, transmitting arguments, and transferring control -- and blur the distinction between CALL and RETURN. Elegant new languages such as Simula 67 (6) have been designed which provide at least a limited form of coroutine as well as, or instead of, subroutines. But implementations of such languages have not been widely available, and hence coroutines have not been widely used.

The system described here is intended to encourage the use of coroutines by making them readily available. An extension to ANSI Fortran (12) is defined that provides a very general form of coroutine in addition to the standard Fortran program units such as subroutines and block data subprograms. The main reason for using an extension to Fortran is that, as will be shown, it is possible to translate program units of the extension into program units of Fortran such that, when used in conjunction with supplementary service routines (also mostly in Fortran), the desired effects occur. Thus by writing the translator, too, in Fortran, the system can

- 2 -

be made portable -- installing it on any machine with an ANSI Fortran compiler involves little more than providing a few primitives and putting it in a library.

Two additional advantages accrue from extending Fortran. First, extending such a widely used language should alleviate the problems of programmer inertia, because rather than learning a whole new language, they need only learn the new features. Second, the translation of program units other than coroutines consists essentially of leaving the statements as they are, so the user pays little or nothing at run time for features not actually used.

System Survey

The portable coroutine system provides a very general form of coroutine, incorporating most features suggested elsewhere with others that are wholly new. The best introduction to the system is to view a running program.

A running program consists of a number of executable modules. Each module has a distinct name provided by the system, and modules "know" about each other by having special variables that contain names of modules. At any time, statements in only one module are being executed. The module that is executing can create new modules, either as duplicates of existing modules it knows about, or afresh from patterns supplied by the programmer. It can also transmit arguments to modules it knows about, such arguments persisting until subsequently changed. And it also can pass control to any other module it knows about, execution of statements in this other module beginning where it was left off the last time the module was executed.

- 3 -

Note that all that matters is which modules are known. In particular, the route by which control reached the module currently being executed is irrelevant, unlike the way that the order in the nesting list affects relations between subroutines. Note also that if a module or group of modules become unknown to the module currently being executed, and to all other modules to which control could pass, then such a module or group of modules can be expunged from the system. Note finally that nothing precludes modules from employing conventional programming facilities such as functions or subroutines, common data bases, standard I/O, etc.

Repeating the above description in more detail, the first idea is that what the programmer writes are not executable modules themselves, but rather patterns (called prototypes) from which modules can be made. Executable modules (called copies) made from the same prototype need not be identical, because a prototype can have parameters whose values affect, for example, array dimensions or initial values of variables. This is similar to the idea of macro definitions, from which (not necessarily identical) macro expansions can be produced in text.

Making a copy from a prototype is called creating a copy. Making a new copy which is identical to an existing copy in its current state (in particular so arrays have the same sizes, variables the same values, etc.) is called duplicating a copy. The language extension provides CREATE and DUPLICATE statements: these statements resemble LOGICAL IF statements in that a substatement is included to be executed if the creation or duplication fails, for instance if insufficient store is available.

- 4 -

The system assigns each copy a unique name, its copyname, and the language extension provides variables of type copyname through which copies can be referenced. For example, when a copy is created or duplicated, the name of the new copy is made available in a system copyname variable, NEWCPY. Arrays of copynames and copyname-valued functions are allowed as well as simple copyname variables. Apart from use in referencing copies, only two operations are defined for copynames: assigning a copyname to a copyname variable, and testing whether two copynames are identical. A copyname variable may not be undefined: it must refer to the null copy if no other.

The usual syntax for subroutine argument transmission would be inappropriate for coroutines, because argument transmission is not linked to transfer of control. Instead the language extension allows variables to be declared accessible, whereupon their values can be set or fetched from other copies at any time, by executing the new statements STORE and FETCH. It also allows variables to be declared dummy, whereupon other copies can at any time substitute variables for the dummy variable by execution of the new statement ASSOCIATE. Thereafter, whenever the dummy variable is apparently used, the storage locations of its current associate are actually used. The association persists until another execution of an ASSOCIATE statement establishes a new associate. Note that accessible variables correspond to the association of subroutine arguments "by assignment" or "by value", whereas dummy variables correspond to the association of subroutine arguments "by address" or "by reference" or in IBM 360 jargon "by name".

- 5 -

Variables can be both accessible and dummy, which means the value of the current associate can be set and fetched from other copies. The STORE, FETCH, and ASSOCIATE statements identify accessible or dummy variables by matching the symbolic name, an important feature because it means that the prototype of the referenced copy need not be known. Like the CREATE and DUPLICATE statements, these statements include a substatement to be executed if the main statement fails, for example if the referenced copy contains no such variable. The statements STORE and FETCH are only defined for simple variables or array elements, but ASSOCIATE may also be used with arrays. As with Fortran subroutine array arguments, this means associating the array bases, the dimensioning for the dummy array being declared with it.

Each copy has a special copyname variable "CALLER" that plays a role similar to a subroutine return address. At any time, one copy is distinguished as being the currently executing copy and statements in it are executed in the usual sequential manner. It can transfer control to any copy it knows about in two ways: if it invokes the other copy, its copyname is placed in the CALLER variable of the invoked copy, whereas if it resumes the other copy, the CALLER variable of the resumed copy is left unchanged. The copy yielding control thus has the option of whether to make its copyname available so the invoked copy knows who invoked it, or to transfer control anonymously, preserving the copyname of the previous invoker.

In either case, execution of the copy to which control is transferring proceeds from the point indicated by the resumption marker

of that copy. Initially this point is the first executable statement of the copy, but normally it is just after the statement last causing control to leave the copy. The resumption marker is in fact an integer variable GOFROM used in a super COMPUTED GO TO, so declaring it accessible allows other copies to force resumption at specified points.

If the copy being resumed or invoked is the null copy, control returns to the main program.

(Figure 1
about here)

The new statements and reserved words of the language extension are shown in Figure 1. The language extension also includes a change in interpretation and some restrictions. The change in interpretation is the natural one that data statements in prototypes should refer to create time not compile time. This has two important consequences, first that creation parameters, mentioned earlier, are allowed as values in data statements; and second that array dimensions may be defined in data statements, thus allowing arrays to be of different sizes in different copies. Most of the restrictions are consequences of the implementation; for example subroutines can be called from the main program, from coroutines, or from other subroutines, but coroutine control transfer can only occur from the main program or coroutines.

Implementation Scheme

Since Fortran is the language extended, the goal of portability makes implementation by precompiling to Fortran desirable. More, it is desirable to preserve the important Fortran feature that each program unit can be

- 7 -

separately compiled. This means coroutines must be compiled to functions or subroutines. In fact, each coroutine compiles into three subroutines: body, create, and precall.

The body subroutine consists essentially of the executable statements of the prototype, with extension statements converted to subroutine calls as outlined later.

The possibility of multiple copies of a prototype indicates that the prototype should be compiled as pure code, copies actually being storage frames for this code. The create subroutine is called to obtain an adequate block of storage for a storage frame, perform initializations, etc. Management of storage frames as blocks in a storage pool is well understood, and the only distinctions of this application are that a fixed header scheme (2) will avoid the need for backpointers when compacting the pool, and that true garbage collection (rather than a reference count scheme) is necessary because rings of pointers may be expected. The management routines are readily written in Fortran, the main interface being the allocation routine OBTAIN.

Two ways of using a Fortran subroutine as pure code are to exploit Fortran's argument transmission by address by putting an item in the calling sequence and using the corresponding entry in the storage frame as the argument, or to treat the storage in the subroutine as registers, copying in values from the storage frame in a prologue before executing the body, and copying them back afterward in an epilogue. Except for some simple variables, the first way is better, so the body subroutine has a long argument list. Since the arguments to be supplied depend only on the

- 8 -

copyname and offsets in the storage frame, a third subroutine, the precall subroutine, is produced to calculate these offsets and call the body subroutine.

A copyname variable is really a pointer to the fixed header of the storage frame of the copy. To invoke or resume this copy, the corresponding prototype must be found. This is arranged for by having the create subroutine call a machine language primitive, SAVE, with the precall subroutine as argument. This primitive records in the storage frame enough information (e.g. entry point address) that later when a second machine language primitive, CALL, is given this information, it can call the precall subroutine. (Note that these primitives are effectively providing procedure name variables).

(Figure 2
about here)

The organization of the system is thus indicated by Figure 2. Solid boxes are routines compiled from the program, dotted boxes are supplementary service routines. To invoke or resume a copy DISPATCHER locates the appropriate storage frame (and, if invoking, updates CALLER), and then calls the precall subroutine via CALL. The precall subroutine in turn then calls the body subroutine. As mentioned earlier, the body subroutine starts with a COMPUTED GO TO so execution can begin at the point indicated by the resumption marker. (Note that this might be in a DO loop, which would require the DO loop to be replaced by equivalent statements). In executing, the body may call service subroutines STORE, FETCH, and ASSOCIATE to perform these operations upon specified storage frames. Duplication is performed by calling the service subroutine DUPLICATE, and creation by calling the create routine for the specified

- 9 -

prototype. Both of these may cause blocks to move in the storage pool, so the body subroutine must return to the precall subroutine and be called again to ensure the storage correspondences are correct. Before returning for this, and of course before returning to DISPATCHER to invoke or resume another copy, the resumption marker GOFROM must be updated.

The importance of the scheme outlined above is that by using three simple machine language primitives (ASSOCIATE requires the usual primitive of finding the address of a variable) and a set of supplementary service routines totaling about 300 lines of Fortran, it is possible to implement a completely different language structure in Fortran. In this sense, Fortran is a powerfully extensible language.

Uses of Coroutines

To date, coroutines have mainly been used either for processing data streams (sequences of items of data) or for simulating several processes simultaneously.

For example, each pass of a compiler can be regarded as a stream processor whose input stream is the output stream of the preceding pass. Passes written as coroutines, rather than running to completion before the next pass is started, can be run in quasi-parallel, each yielding control whenever its input buffer is empty or its output buffer is full. In this way buffer size can be controlled, although each pass executes essentially independently.

Another important use of coroutines is in generating streams from data structures such as files. Frequently there are many ways to interpret the request "next item from structure", and for each interpretation there

- 10 -

may well be several generators positioned at various places in the structure. Writing each generator as a coroutine is a useful way to allow each the buffer memory, etc., that it needs, and generators corresponding to the same interpretation of next item will usually be so similar that copies of a single prototype are attractive.

In processing streams, a feature of considerable value (1) is that no coroutine really knows, in any hard and fast sense, just who his neighbour is. The fact that copynames are variables, combined with symbolic matching to identify accessible or dummy variables, means that modules can be bound and unbound dynamically. For example, a module can be temporarily inserted between two others to monitor intermediate results, or various different versions of a module can be experimentally connected in place.

Sometimes streams merge or split. In such cases, the processing module may want to request that not just one but a list of other modules be invoked. This is easily handled by having one coroutine that is a scheduler, accepting such requests and deciding which copy to execute next.

Discrete event simulation consists of simulating a process that consists of a sequence of discrete events, each of which affects the values of state variables of the system. When several processes affecting common state variables are simulated simultaneously, it is convenient to code each process as a coroutine, for execution must be quasi-simultaneous in order that events in different processes occur in proper order of time. A scheduler as mentioned above usually becomes necessary. Moreover as processes will often be similar, and the number of such similar processes may depend on intermediate results, creatable copies are necessary.

- 11 -

Recently two new uses of coroutines have become apparent. The first of these arises in a type of heuristic programming that occurs in such areas as artificial intelligence and statistical model fitting (3, 10). A set of alternatives exists, each of which if pursued would introduce a further set of alternatives, and so on. Rather than pursuing each set of choices in turn to the limit (a concept that may well be meaningless), it is far more desirable to proceed more uniformly, pruning branches of the choice tree whenever they appear comparatively unprofitable, but following up the others more or less simultaneously. If pursuing each alternative takes a separate program module, quasi-parallel execution and hence coroutines are called for. Further, many alternatives will use copies of the same prototype, and also storage recovery from the pruning occurs naturally as unprofitable branches are "forgotten".

The other new use of coroutines is to provide a user with effectively his own multiprogramming system, complete with powerful features such as dynamic binding, special schedulers, means of communicating between running programs, run time spawning of new jobs, dynamic storage allocation, common access to shared data bases, and message switching between jobs and various peripherals. Such facilities, when available at all, are usually available only for the computer system as a whole, and often only to systems programmers. There are times, however, particularly in an interactive environment, when a user might want to structure such a system to his own needs (7). The coroutine system needs nothing more than viewing in this way to provide this service, the only restriction being that it is natural break, rather than interrupt driven, multiprogramming.

Summary

It has been shown how Fortran can be extended, in a simple and natural way, to provide a very general form of coroutines; how this can be implemented, in Fortran using only a few primitives, so that the system is portable; and why, considering some of the uses of coroutines, certain features are desirable. It is hoped the discussion of how coroutines can be used will suggest new problems for which coroutines are a useful tool, a tool which the portable coroutine system makes available.

References

1. Balzar, R.M., "Dataless programming", 1967 Fall Joint Computer Conference, AFIPS, pp. 535-544.
2. Brown, W.S., "An operating environment for dynamic recursive computer programming systems", Comm. ACM 8, 1965, pp. 371-377.
3. Chambers, J.M., "A computer system for fitting models to data", Applied Statistics, 18, 1969, pp. 249-263.
4. Conway, M.E., "Design of a separable transition-diagram compiler", Comm. ACM 6, 1963, pp. 396-408.
5. Dahl, O.J., and Nygaard, K., "Simula - an Algol-based simulation language", Comm. ACM 9, 1966, pp. 671-678.
6. Dahl, O.J., Myhrhaug, B., and Nygaard, K., "Simula 67, common base language", Publication S-2, Norwegian Computing Centre, Oslo, Norway, 1968.
7. Green, L.E.S., "Time sharing in a traffic control program", Comm. ACM 7, 1964, pp. 678-681.

8. Knuth, D.E., The art of computer programming: Volume 1, Fundamental algorithms. Addison Wesley 1968.
9. McIlroy, M.D., "Coroutines: Semantics in search of syntax", unpublished memorandum.
10. Unger, S.H., "GIT - a heuristic program for testing pairs of directed line graphs for isomorphism", Comm. ACM 7, 1964, pp. 26-34.
11. Wegner, P., Programming Languages, information structures, and machine organization. McGraw Hill, 1968.
12. ANSI Standard Fortran, Publication X. 39 - 1966, American National Standards Institute, also "Fortran vs. Basic Fortran", Comm. ACM 7, 1964, pp. 590-625, and "Clarification of Fortran standards - initial progress", Comm. ACM 12, 1969, pp. 289-294.

Figure 1 - New Statements and Reserved Identifiers

Non-executable statements

COROUTINE name (par 1, par 2, ..., par n)
COPYNAME var 1, var 1, var 2, ..., var n
COPYNAME FUNCTION name (arg 1, arg 2, ..., arg n)
ACCESSIBLE var 1, var 2, ..., var n
DUMMY var 1, var 2, ..., var n

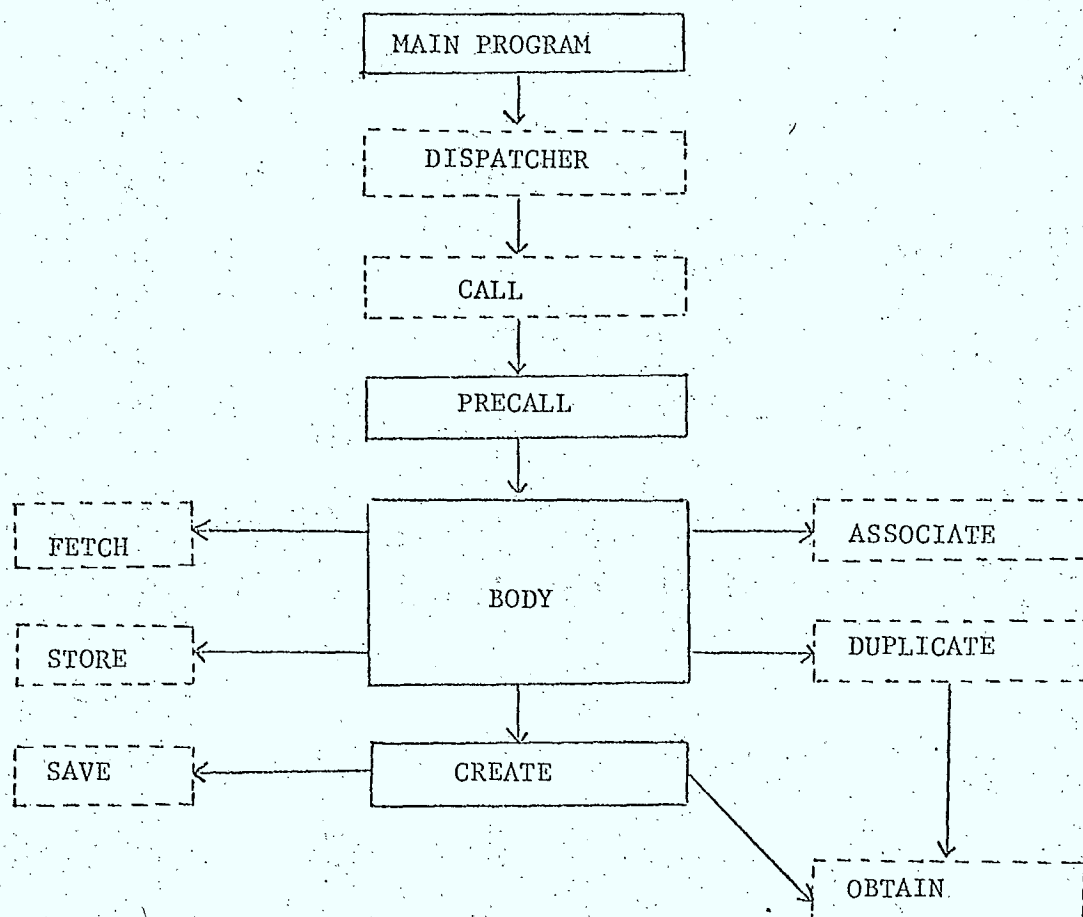
Executable statements

CREATE (name (par 1, par 2, ..., par n)) substatement
DUPLICATE (copyname) substatement
STORE (value, acc var, copyname) substatement
FETCH (variable, acc var, copyname) substatement
ASSOCIATE (variable, dum var, copyname) substatement
INVOKE copyname
RESUME copyname

Reserved identifiers

NEWCPY
CALLER
MYNAME
GOFROM

Figure 2. Relation between Compiled Routines and Service Routines
on the Nesting List



CACC / CCA



92446

SIMULATION STUDY OF USES OF A
COMPUTER NETWORK : DRAFT

P
91
C655
G458
1971

DATE DUE

[illegible]

