McROSS—A multi-computer programming system*

by ROBERT H. THOMAS and D. AUSTIN HENDERSON

Bolt, Beranek and Newman, Inc.
Cambridge, Massachusetts

INTRODUCTION

This paper describes an experimental “distributed” programming system which makes it possible to create multi-computer programs and to run them on computers connected by the ARPA computer network (ARPANET).1 The programming system, which is called McROSS (for Multi-Computer Route Oriented Simulation System), is an extension of a single-computer simulation system for modelling air traffic situations2 developed by Bolt, Beranek and Newman, Inc. (BBN) as a tool for air traffic control research. The McROSS system provides two basic capabilities. One is the ability to program air traffic simulations composed of a number of “parts” which run in geographically separated computers, the distributed parts forming the nodes of a “simulator network.” The second is the ability of such a simulator network to permit programs running at arbitrary sites in the ARPANET to “attach” to particular nodes in it for the purpose of remotely monitoring or controlling the node’s operation.

The McROSS distributed programming system is unique in several ways:

(a) Use of McROSS generates inter-computer traffic in which a group of programs are engaged in substantive conversation. There is relatively little previous experience with such inter-computer, program-to-program conversations.

(b) The component nodes of a simulator network are not bound to particular ARPANET sites until simulation “run time.” Thus on different runs the same distributed program can be distributed in different ways over the ARPANET. For example, in one run all the nodes of a simulator network might be run at BBN and on the next some might be run at BBN, others at RAND and still others at the University of Utah. This mode of using the ARPANET is significantly different from the normal one in which programs are bound to particular network sites at program composition time. (The only constraint on the binding of nodes to ARPANET sites is the requirement that each node run on a PDP-10 under the TENEX operating system.)

(c) The responsibilities of a node in a simulator network can be conveniently partitioned into sub-tasks which can be performed more or less independently of one another. The McROSS implementation mirrors this partitioning. Functions performed at the nodes are realized by groups of loosely connected, concurrently evolving processes.

The distributed simulation system represents an initial step in an on-going research program which is investigating techniques to make it easy to create, run and debug computations involving the coordinated behavior of many computers. The McROSS system is intended to serve both as an experimental vehicle for studying problems related to distributed computation and as a tool for air traffic control research. Its two goals are well matched. A satisfactory solution to the nation’s air traffic problems is likely to include a network of airborne and ground based computers working together on a single distributed computation: the scheduling and control of aircraft maneuvers. Thus, the air traffic control problem is a rich source of interesting problems in partitioned computation which can be used to measure the usefulness of the distributed computational techniques developed.

This paper is a report on one phase of a continuing research program. The intent is to describe interesting aspects of an experimental distributed programming

---

*This work was supported by the Advanced Projects Research Agency of the Department of Defense under Contract No. DAHC-71-C-0088.
system and to share our experience with others considering the construction of such systems. The paper does no more than hint at how the McROSS system can be used in air traffic control research.

The next section provides background useful for subsequent discussion of the distributed programming system. After than, the McROSS system is described, first in terms of the facilities it provides for creating distributed simulations, and then in terms of interesting aspects of its implementation. The paper closes with a discussion of some of the issues brought into focus by the experience of implementing and using a distributed programming system.

COMPONENTS OF THE McROSS SYSTEM

The main components of the distributed programming system are:

1. The ARPA computer network;
2. The TENEX operating system for the PDP-10; and
3. A simulation programming system known as ROSS (for Route Oriented Simulation System).

These components became operational within 6-10 months of one another, at which time it became feasible to implement the distributed programming system being described.

The ARPA

The ARPA computer network\(^1\)\(^2\)\(^3\)\(^4\) is a set of autonomous computer systems (hosts) which are interconnected to permit resource sharing between any pair of them. The goal of the ARPANET is for each host to make its resources accessible from other hosts in the net thereby permitting persons or programs residing at one network site to use data and programs that reside and run at any other. Each host interfaces with the network through an Interface Message Processor (IMP)\(^5\) a small dedicated general-purpose computer. The IMPs, which reside at the various ARPANET sites, are connected by 50 kilobit common carrier lines and are programmed to implement a store and forward communication network. In passing from host A to host B a message passes from host A to IMP A, through the IMP communication network from IMP A to IMP B (in general passing through a number of intermediate IMPs) and finally from IMP B to host B.

At each host there is a Network Control Program (NCP) whose function is to provide an interface between the network and processes within the host. The NCPs use the IMP store and forward network to provide processes with a connection switching network. Thus, they enable processes in different hosts to establish connections with one another and to exchange information without directly concerning themselves with details of network implementation such as the way in which hosts are connected to and communicate with IMPs.

Information can flow in only one direction on an ARPANET connection. Thus, before two processes are able to engage in a dialogue they must establish two such connections between them. A connection is completely specified by its two ends which are called sockets. A network socket is itself uniquely specified by a host identifier and a socket identifier. The purpose of the socket identifier is to specify a process or group of processes within a host and a socket relative to that process or process group. Thus, it is useful to think of a socket as having a three component "socket name" of the form H.P.N. H is the "host" component which identifies an ARPANET host, P is the "process" component which identifies a process group within H and N is the "process-local" component which identifies a socket relative to P. In the sequel

\[H_1, P_1, N_1 \rightarrow H_2, P_2, N_2\]

is used to denote a connection between sockets \(H_1, P_1, N_1\) and \(H_2, P_2, N_2\) where \(\rightarrow\) indicates the direction of information flow over the connection.

For a connection to be established between two processes each must request that it be established. There are two common ways in which connections are established. In the first, the processes play symmetric roles. Each specifies, as part of a "request for connection" (RFC), the socket name at its end of the connection, the socket name at the remote end of the connection and the direction of information flow. If the two RFCs "match" the connection is established. This connection procedure requires a priori agreement upon the sockets to be used for the connection. The second common connection procedure is used in situations in which one process wishes to provide some sort of service to other processes. The "serving" process establishes a listening socket within its host which is receptive to RFCs from any other process in the network and then "listens" for connection attempts. The serving process uses facilities provided by its NCP to detect the occurrence of a connection attempt and to determine its source. When such an attempt is made the serving process can choose to accept or reject it. This connection procedure requires that only one socket name, that of the serving process's listening socket, be known.
a priori. In the remainder of this paper

\[ \text{[H.P.N→JL} \]

and

\[ \text{[H.P.N←JL} \]

are used to denote connections established in a listening state.

The **TENEX operating system**

TENEX\(^4\) is a time sharing system for the DEC PDP-10 processor augmented with paging hardware developed at BBN. For purposes of this paper it is useful to describe TENEX in terms of the virtual processor it implements for each logged-in user (i.e., user time sharing job).

The instruction repertoire of the TENEX virtual processor includes the PDP-10 instruction set with the exception of the direct I/O instructions. In addition, it includes instructions which provide access to virtual processor capabilities implemented by the combination of the TENEX software and hardware.

The TENEX virtual processor permits a user job to create a tree-structured hierarchy of processes. Such processes have independent memory spaces and computational power. At different stages in its lifetime a single user job may include different numbers of processes in various states of activity. Several mechanisms for interprocess communication are provided by the combination of the TENEX software and hardware.

The TENEX virtual processor permits a user job to create a tree-structured hierarchy of processes. Such processes have independent memory spaces and computational power. At different stages in its lifetime a single user job may include different numbers of processes in various states of activity. Several mechanisms for interprocess communication are provided by the combination of the TENEX software and hardware.

A memory space is provided by the virtual processor which is independent of the system configuration of core memory. Each process has a memory space of 256k words which is divided into 512 pages each of 512 words. A process can specify read, write and execute protection for pages in its memory space as it sees fit.

The virtual machine includes a file system which provides a mechanism for storing information on and retrieving it from external devices attached to TENEX. Processes refer to files using symbolic names, part of which identifies the particular device on which the file resides. The instruction set of the virtual machine includes operations for data transfer to and from files which a process can execute without explicitly arranging for buffering.

The NCP resident in TENEX makes the ARPANET appear to TENEX processes as an I/O device. The name of a “file” corresponding to a network connection includes the names of both the local socket and the remote socket which define the connection. A process requests TENEX to establish a connection by attempting to open an appropriately named “network” file. The open attempt succeeds and the connection is established if and when another process issues a matching RFC. TENEX processes transmit and receive information over network connections by executing the normal file data transfer instructions.

ROSS

ROSS is a programming system for modeling air traffic situations.\(^2\) It includes facilities for creating and running simulation experiments involving air traffic in an experimenter-defined airspace. The system is currently being used in a number of air traffic control research projects.

To create a simulation experiment the experimenter uses ROSS language forms to write a “program” which defines the geography of an airspace, the wind profile for the airspace, aircraft flight characteristics and a set of “route procedures.” A route procedure consists of a sequence of one or more “instructions” for monitoring or controlling the progress of an aircraft through the simulated airspace. Execution of a route procedure is the ROSS counterpart of pilot and/or controller actions. The “flight” of a simulated aircraft through the airspace is accomplished by the execution of a series of route procedures. ROSS includes “primitive” route procedures to “create” aircraft, “inject” them into and remove them from the simulated airspace as well as ones which cause aircraft to accelerate, turn, climb and descend.

By compiling his program the experimenter creates a simulator for traffic in the airspace he has defined. To perform a simulation experiment the experimenter runs his program. The simulated airspace remains empty until the program is supplied with input which injects aircraft into the airspace and controls their flight through it. Each input line the simulator receives is passed to an internal parsing mechanism which issues a call to an appropriate experimenter-defined or primitive route procedure. The program can accept input from an on-line terminal, a predefined “scenario” file, or both. Input lines from the scenario file are identical to ones from the on-line keyboard with the exception that scenario file input lines include a time field which specifies the (simulated) time at which the program is to accept them. A user can manually “vector” aircraft through the airspace by supplying input at the on-line keyboard.

A ROSS simulator can drive displays of the airspace...
and, in addition, can generate output which can be written into files, typed on the on-line keyboard or sent back into the simulator input parser.

**THE McROSS PROGRAMMING SYSTEM**

The McROSS system provides the ability to define simulation experiments involving a number of airspaces or “simulation centers” which can be interconnected to form a simulator network. Adjacent centers in the simulator network are connected to one another by way of the ARPANET. The components of a simulator network may run as user jobs distributed among different TENEXs or as different user jobs on the same TENEX. (As of January 1972 the ARPANET included five TENEX hosts.)

Computational responsibility for performing a multi-computer McROSS simulation is truly distributed. For example, as an aircraft flies from one airspace into an adjacent one the responsibility for simulating its dynamics shifts from one computer to another.

**Goals**

The McROSS system was implemented to achieve the following goals:

1. Autonomy of parts:
   Individual components of a McROSS network should be able to operate independently of one another (to the extent that each is independent of the others for traffic). Furthermore, no center should be able to cause another to malfunction. Autonomy of parts enables a multi-computer simulation to run when only some of its components are operational. Failure of a component center in a multi-center simulation results in degradation of the total simulation rather than forced termination of it. A beneficial side effect of autonomy is that individual centers can be partially debugged without running the entire simulator network.

2. Deferral of process/processor binding:
   The binding of centers in a McROSS network to host computers in the ARPANET should be deferred until run time. This goal can be stated in more general terms. The program for a distributed computation defines a logical configuration made up of abstract relations between the computation’s parts. A given execution of the program is accomplished by a particular physical configuration of computers. The two configurations, logical and physical, are by necessity related. However, the programmer should have the option of specifying them separately. By deferring process/processor binding the configurations can be separately specified. As a result the programmer is free while composing his program to concentrate on the logical structure he wants his program to define without concerning himself with details of the physical structure on which it is to be run.

3. Capability for dynamic reconfiguration:
   In the course of a simulation it should be possible for adjacent centers to dynamically break and reestablish connections with one another. Furthermore, it should be possible for process/processor binding to be changed during a simulation. That is, it should be possible to change the physical location of a center from one ARPANET host to another. The ability to dynamically reconfigure makes it possible to remove an improperly operating center from the simulator network and replace it with another at the same ARPANET host or at a different one.

4. Decentralization of control:
   McROSS is to be used as a tool for investigating distributed computation. Among the subjects to be studied are methods for controlling such computations. In particular, various techniques for distributing control responsibilities among the parts of a computation are to be experimentally investigated. It is important, therefore, that operation of the McROSS system not require a central control mechanism for coordinating simulator networks. Stated somewhat differently, the only components required for a McROSS simulation should be simulation centers defined by McROSS programs. The realization of this goal, which makes experimentation with distributed control possible, should not preclude experimentation with centralized control.

5. Remote monitoring capability:
   A McROSS simulator network should provide ports through which its components are accessible to any ARPANET host. An appropriately programmed process running at any ARPANET host should be able to “attach!” to a component of a simulator network to monitor and control its operation. A remote monitoring process should be able to:
   
   a. obtain sufficient information to display traffic in the airspace it is monitoring.
   b. serve as the on-line keyboard for the center it is monitoring;
McROSS as seen by the user

A McROSS simulator network is defined by a program composed of a "network geometry" sub-program and sub-programs corresponding to each of the centers in the network.

The network geometry sub-program defines the logical geometry for a simulator network. Conceptually, a network is composed of nodes and arcs. Nodes in a McROSS network are simulation centers and arcs are duplex connections between centers. Figure 1 shows a four node simulator network which could be used to simulate air traffic between Boston and New York. The following geometry sub-program defines that network:

```
netbegin
neteen BOSTRM, BOSCEN, NYCE, NYTRM
netcon BOSTRM, BOSCEN
netcon BOSCEN, NYCE
netcon NYCE, NYTRM
netend
```

The `netcon` statement declares that the network contains four nodes (Boston terminal control, Boston en route control, New York en route control and New York terminal control). The `netcon` statements declare the three arcs; `netbegin` and `netend` serve to bracket the geometry declarations.

In general, the sub-program for each center has four parts:

- a route procedure module
- a local geography module
- a wind profile module
- an aircraft characteristics module.

In addition to defining procedures followed by aircraft as they fly through a center's airspace, the route procedure module includes routines specifying how the center interacts with its neighbors. Information exchange between adjacent centers is accomplished by sending messages across the connection between them. A center handles messages from neighboring centers by submitting them to its input parsing mechanism. Such messages are treated identically to input from its on-line console and scenario file.

The ability of adjacent centers to interact depends upon the state of the connection between them as each sees it from its end of the connection. A center may consider a connection to be in one of three states:

1. uninitialized:
   the "physical location" of the neighbor is unknown;
2. closed:
   the "physical location" of the neighbor is known but the connection is not capable of carrying messages (either because it has not yet been established or because it has been broken);
3. open:
   the connection may be used for information exchange with the neighbor.

In the current implementation of McROSS the "physical location" of a neighbor includes both the ARPANET host the center is running on (e.g., BBN) and the identification of the user it is running under (e.g., Jones).

McROSS provides the operations `init`, `conn`, `deconn` and `abort` for changing the state of a connection. The effect these operations have on the end of a connection is illustrated by the state transition diagram of Figure 2.

Consider the geometry for the Boston-New York simulation. Execution of `conn BOSTRM`
Figure 2—Transition diagram for the state of the end of a connection showing the effect of the operations `init`, `conn`, `dsconn` and `abort` within the BOSCEN initiates an attempt to open a connection with BOSTRM. The connection attempt succeeds if a matching `conn` is executed by the BOSTRM center. The effect of executing `dsconn` NYCEN within the BOSCEN center simulator is to break the connection between the NYCEN and BOSCEN centers by forcing both ends of it into the closed state; `abort` works in an analogous manner. Execution of the `init` operation results in a center-user dialogue in which the human user is asked by the center program to specify the physical location of the neighbor.

Two language primitives are provided for sending messages from one center to another. One takes a single operand, a message, which it sends to every neighbor whose connection is in the open state. The other takes two operands, a message and the name of a neighboring center. If the connection to the center is open, the message is sent to the center; otherwise the operation has no effect. Because they are submitted to the input parsing mechanism care must be taken that messages sent to a neighbor are formatted correctly.

McROSS includes operations which can be used by a center to obtain information about the simulator network and its immediate neighbors. For example, there is a primitive which produces a list of all nodes in the network (i.e., all centers declared by the `netcen` declaration); another one produces a list of all neighboring centers for which connections are in the open state. In addition, a center can examine the state of the connection to each of its neighbors individually.

The local geography module defines the airspace of a center by specifying names and locations (x-y coordinates) of important geographic features such as navigational aids, obstructions and airports. In addition it includes a declarative statement which names the simulator center. For example, the geography module for the BOSTRM center would include the declaration `atcen BOSTRM`.

This declaration has the effect of binding the identifier THISIS to the name BOSTRM. Thus in the BOSTRM center THISIS is bound to BOSTRM while in the NYTRM center it is bound to NYTRM. Route procedures which access the simulator center name can be written in terms of THISIS to enable their use at any center in a simulator network.

A properly authorized process at any ARPANET host can attach to a center in a McROSS simulator network and request output from it and direct input to it thereby monitoring and controlling its operation. McROSS center programs are prepared to offer two kinds of service to remote monitors:

1. broadcast service:
   Centers continuously broadcast certain information to monitors attached to them. Presently centers broadcast flight parameters of each aircraft in their air space (speed, heading, altitude, x-position, y-position, acceleration, aircraft id) and the (simulated) time. A remote monitor can use broadcast information to drive a display of traffic in a center’s airspace or it can record it for later analysis.

2. demand service:
   Each center is prepared to respond to certain specific requests from monitors. In the current implementation a monitor can request that a center:
   a. transmit its map of the airspace (which can be used as background for displaying the center’s air traffic);
   b. stop the continuous broadcast;
   c. resume the continuous broadcast;
   d. treat the monitor as its on-line keyboard by directing keyboard output to the monitor and accepting input from it;
   e. cease treating the monitor as its on-line keyboard;
   f. break its connection with the monitor.

The monitoring facility has proven useful both for debugging and for demonstration purposes. One difficulty a user faces in debugging a multi-center simulation is determining what is happening at centers suspected to be malfunctioning. A monitor, constructed appropriately, can serve as a “graphical probe” and be
used to watch the operation of first one suspect center and then another. For example, we have used such a monitor to follow the trajectory of an aircraft as it passes through several centers.

By enabling processes at arbitrary ARPANET sites to observe and control McROSS simulations, the monitoring facility provides a mechanism for using hardware and software features which are unique to various network installations. By using monitors which play an active role in his simulations a McROSS user can experiment with different ways of partitioning computational and control responsibilities in air traffic situations. He could, for example, experiment with a monitor built to provide a weather advisory service for simulator centers. Such a monitor would presumably have access to an on-line weather data base. (A weather data base to be accessible through the ARPANET is currently being designed.16) To perform its service for a center the monitor would attach to the center, requesting that the center broadcast aircraft flight parameters to it and accept input lines from it. It would then "watch" the airspace of the center and send instructions to it, as necessary, to vector aircraft around severe weather.

Unless he chooses to do so, the simulation programmer need not concern himself with remote monitoring beyond specifying at simulation run time which centers in his network are to be receptive to remote monitors. Monitors themselves are not part of the McROSS system. McROSS merely provides a mechanism for remote monitoring. No effort has been made to provide linguistic features within the McROSS system to make it easy to write programs to do monitoring.

THE McROSS IMPLEMENTATION

Some interesting aspects of the McROSS implementation are discussed in this section. The section focuses on strategy rather than detail. The result is a simplified but nonetheless accurate sketch of the implementation.

The ROSS implementation

Implementation of McROSS was accomplished by extending the existing ROSS simulation system. A ROSS simulation consists of initialization, simulation and termination phases. The simulation phase is implemented as a loop. Each pass through the loop represents a "tick" of the clock which maintains simulation time. On each tick the simulator:

1. parses and interprets input directed to it from its scenario file and on-line console since the last tick;
2. interprets the route procedure each aircraft is currently following;
3. advances each aircraft along its trajectory taking into account the aircraft's speed, acceleration and heading and the wind profile of the airspace;
4. directs output generated since the last tick to the appropriate devices;
5. performs actions necessary to maintain the local display of its airspace;
6. increments the simulated time.

A ROSS simulation can be run either in a "real time" mode in which simulated time is locked to real time, or in a "hyper fast" mode in which the simulation proceeds as fast as it can.

Process/processor binding and center-center connections

Connections between pairs of simulator network centers are duplex. Each center-center connection is implemented by two ARPANET connections. More specifically, an open connection between centers A and B is realized by the two ARPANET connections (see Figure 3):

H_A · P_A · S_AB → H_B · P_B · R BA
H_A · P_A · R_AB → H_B · P_B · S BA

To establish such a center-center connection two pairs of matching RFCs must be issued by centers A and B. To issue the correct RFCs A must know H_B, P_B, R_BA and S BA; similarly, B must know H_A, P_A, R_AB and S_AB.
The host (H) and process (P) components of the socket names for a center-center connection cannot be determined until run-time because process/processor binding is deferred until then. However, the process-local components (R and S) of the socket names can be pre-assigned and, in fact, the effect of the declarations in the network geometry sub-program for a particular McROSS network is to do exactly that.

The process local components for the four socket names corresponding to a center-center connection are always the same whereas the host and process components may change from run to run or even within the same run if either neighbor is involved in a dynamic reconfiguration.

When a center's end of a center-center connection is in the uninitialized state the host and process components of the socket names corresponding to the remote end are unknown to it. To move its end of a connection from the uninitialized state the center engages in a dialogue with the user requesting from him the physical location of the neighbor. After successfully completing the dialogue the center has sufficient information to issue the two RFCs required of it to establish the connection.

Center-monitor connections

The connection between a center C and an attached remote monitor M is realized by two ARPANET connections. One of them

\[ H_c, P_c, S_{CM} \rightarrow H_M, P_M, R_{MC} \]

is a "broadcast" connection used for continuously broadcasting information to M. The other

\[ [H_c, P_c, R]_L \]

is a "request" connection maintained by C in a listening state for M to use to make requests of C. Each monitor attached to C has its own broadcast connection but may share a request connection with other monitors.

To make a request of C, M connects to the request socket, transmits its request over it and then closes the request connection, freeing it for use by other monitors. The action taken by C of course depends upon the request. If the request were for the display map, C would transmit the map data over the broadcast connection to M.

Before the RFCs required to establish a center-monitor connection between C and M can be issued C must know H_M, P_M and R_M and M must know H_C, P_C, S_{CM} and R. To obtain the required information, M and C engage in a connection protocol which is similar to the "initial connection protocol" developed by the ARPANET network working group.\(^6\)

Each center willing to service monitors maintains as an "attach" socket a send socket in a listening state (see Figure 4.a). The attach socket for C could be denoted

\[ [H_c, P_c, A]_L \]

The process local component (A) of the name for attach sockets is the same for all centers and is "well-advertised." Therefore, if M knows the physical location of C it can issue an RFC for C's attach socket. The effect of such an RFC is to establish the connection (Figure 4.b)

\[ H_C, P_C, A \rightarrow H_M, P_M, R_M \]

Upon detecting an RFC for its attach socket, C notes H_M and P_M and transmits S_{CM} and R over the connection. Next both C and M break the connection and

![Figure 4—Schematic of connection protocol exchange between center C and monitor M](From the collection of the Computer History Museum (www.computerhistory.org))
issue the RFCs necessary to establish the broadcast connection (Figure 4.e)

\[ H_C \cdot P_C \cdot S_{CM} \rightarrow H_M \cdot P_M \cdot R_{MC} \]

where \( R_{MC} = f(R_M) \), a pre-agreed upon function of \( R_M \). C, if it has not done so previously for another monitor, sets up the listening connection

\[ [H_C \cdot P_C \cdot R \leftarrow J_L \]

and finally, reestablishes its attach connection so that other monitors can attach to it (Figure 4.d). Currently the process-local components for ARPANET socket names are numbers and \( f \) is taken to be

\[ f(x) = x + 2. \]

If \( R_M \) rather than \( f(R_M) \) were used for \( R_{MC} \) a race condition would result when \( M \) and \( C \) attempt to establish the broadcast connection. The race involves the order in which the connection with socket \( H_M \cdot P_M \cdot R_M \) would be closed and reopened by \( C \) and \( M \). In particular, the current TENEX NCP is such that an attempt by \( C \) to open the network file corresponding to its end of the (broadcast) connection

\[ H_C \cdot P_C \cdot S_{CM} \rightarrow H_M \cdot P_M \cdot R_M \]

before \( M \) closes the network file corresponding to its end of the connection

\[ H_C \cdot P_C \cdot A \rightarrow H_M \cdot P_M \cdot R_M \]

would fail. Use of \( f(R_M) \) for \( R_{MC} \) avoids the race. A similar race condition, discovered in an early version of the ARPANET initial connection protocol, is avoided in the current protocol by the same technique.

Process structure at McROSS centers

A McROSS center is realized by a collection of cooperating, asynchronously evolving, sequential processes. The collection corresponds to a partitioning of the center's responsibilities into more or less independent sub-tasks. It includes:

1. a process (SIM) to perform theROSS functions;
2. a process (CONN) for each center-center connection to establish and maintain the connection; and
3. a monitor server process (MONSER) to service remote monitors.

The CONN process at center A corresponding to the center-center connection to center B is responsible for establishing ARPANET send and receive connections with B. After it establishes the center-center connection

![Figure 5-The process hierarchy which implements a McROSS simulation center with \( n \) neighbors](image)

with B the CONN process maintains the connection. When messages from B arrive it passes them on to the SIM process for parsing.

The job of the MONSER process at a center is twofold: to engage in an initial connection protocol exchange with monitors attempting to attach to the center and to respond to requests made by attached monitors.

The processes at a center exist in a hierarchy (see Figure 5). The hierarchical structure is less the result of any one process being more important than any other than it is a consequence of the TENEX constraint that process groups be hierarchically arranged. During initialization the SIM process creates the MONSER and CONN processes. Thereafter, the processes evolve more or less independently.

The process structure at each center helps achieve autonomy of parts. The CONN processes and the MONSER process serve to protect the SIM process by insulating it from direct interaction with other centers and remote monitors.

Protocols

The current implementation of the TENEX NCP is such that if a center were to unilaterally break a connection with a neighbor (by closing the two corresponding ARPANET connections) it could leave processes in the neighboring center in an irrecoverable state. For example, a process in the neighbor sending information across the connection at the time it is broken would "feel" the close as if it had executed an illegal instruction. To prevent such situations McROSS centers en-
gage in a protocol exchange prior to breaking connections.

The center-center protocol is relatively simple. To perform an abort or dsconn operation a center sends its neighbor a "request for abort" or "request for disconnect" message and waits until it receives an acknowledgment before actually breaking the connection. Existence of the center-center protocol has two major implications. The first is that a center-center connection has more states than the three noted earlier. The additional states are transient ones which the connection passes through as the center and its neighbor advance through protocol exchanges initiated when one attempts to change the state of the connection. The transient states are invisible to the McROSS user. Immediately after a dsconn (or abort) is initiated the SIM process treats subsequent operations involving the connection as if the connection were already in the closed (or uninitialized) state. The second implication is that center-center connections carry "control" messages used in center-center protocol exchange in addition to "ordinary" messages intended for the receiver's parsing mechanism. The CONN process for each connection must be prepared to recognize and respond appropriately to both kinds of messages.

McROSS centers expect remote monitors to observe a center-monitor protocol. In addition to the connection and request procedures described earlier, the center-monitor protocol includes a disconnection procedure much like the one used in the center-center protocol.

**Interprocess communication**

To perform their tasks the processes at a simulation center must interact occasionally. For example, the arrival of a message from a neighbor requires interaction between the SIM and CONN processes. The CONN process receives the message and passes it onto the SIM process for parsing and interpretation.

One way the processes interact is through shared memory. For example, the SIM and CONN processes have read and write access to a shared "connection table." There is an entry in the table for each center-center connection which includes the state of the connection, a semaphore and other information relevant to the connection. Use of the table is coordinated by strict adherence to a convention which requires every "critical" access (every write access and certain read accesses) to an entry to be bracketed by P (lock) and V (unlock) operations on the entry's semaphore.

The situation arising at a center when a neighbor attempts to break connection with the center is a typical one which requires interprocess communication. The CONN process corresponding to the center-center connection receives a "request for disconnect" message from the neighbor. Center-center protocol requires the CONN process acknowledge the request so that the neighbor can break the connection. The purpose of the protocol exchange is to warn the center that the connection is about to be broken and that it should no longer attempt to use it. Therefore before it acknowledges the neighbor's request it is important that the CONN process communicate this information to the other processes at its center. The CONN process does this by the following sequence of actions:

\[
P(\text{CONNECTION SEMAPHORE});
\]

\[
\text{set connection-state to "not open"};
\]

\[
V(\text{CONNECTION SEMAPHORE});
\]

\[
\text{acknowledge neighbor's request}
\]

As long as processes at the center perform the following sequence of actions to send over center-center connections there is no danger of sending after the connection has been closed:

\[
P(\text{CONNECTION SEMAPHORE});
\]

\[
\text{if connection-state = open then send}
\]

\[
\text{else abort the send;}
\]

\[
V(\text{CONNECTION SEMAPHORE})
\]

Stated somewhat differently, sending over a center-center connection should be regarded as an operation which involves a "critical" read access of the corresponding connection table entry.

In addition to memory sharing, direct process control is used as a mechanism for interprocess communication in the McROSS system. Because of its superior position in the process hierarchy the SIM process can exert direct control over the other processes. A few situations occur in which it does so. For example, when a center-center connection has been closed or aborted (via dsconn or abort) the SIM process forces the corresponding CONN process to halt. If and when an attempt is initiated to reestablish the connection (via conn) SIM restarts it.

**SOME OPEN QUESTIONS**

This section briefly discusses questions representative of ones which have arisen in the course of using the McROSS system. The questions have been resolved to the extent that useful simulations can be performed using McROSS. However, none has been resolved in a
totally satisfactory manner. The intent of this section is to leave the reader with an appreciation for the issues raised by these questions; a thorough discussion of them is well beyond the scope of this paper.

**Synchronization**

Simulated time is important in the operation of the McROSS system. In particular, whenever an interaction between adjacent centers occurs it is important that the clocks kept by the centers show approximately the same time. Time synchronization is a specific example of the general problem of control in distributed computation. It is compounded by the fact that centers can start up and shut down independently of one another. A centralized approach to synchronization has been used with success in McROSS simulations. In it, one center acts as a synchronizer for an entire simulator network. When a center starts up it connects to the synchronizer and receives a synchronization message from it. Thereafter, to stay in synch with other centers in the network, the center makes use of the real time clock in the computer it runs on. A distributed approach to synchronization which does not require a synchronizing center is under consideration.

**Locally unknown names**

Names that are well defined within a simulator network as a whole are not necessarily defined at every node in the network. How should references to such names occurring within centers in which they are not defined be handled? For a specific example in which such a reference is reasonable, reconsider the four node network for simulating Boston-New York traffic. A user controlling the simulator for Boston Terminal who is manually vectoring an aircraft leaving Logan airport might reasonably issue the clearance

```
fly (V205, PAWLING)
```

which specifies that the aircraft is to follow Victor Airways #205 to Pawling. Assume that V205 is defined within the geography modules for BOSTRM, BOSCEN and NYCEN and the PAWLING is defined within NYCEN but not within BOSTRM or BOSCEN. The BOSTRM center can't fly the aircraft to Pawling because Pawling is not defined within its airspace. Ideally it should fly the aircraft along V205 to the boundary of the BOSCEN airspace and then hand it off to the BOSCEN simulator. Certainly it should be able to do better than report an error and abort the route procedure. Techniques for handling references to locally unknown names in certain limited contexts are being investigated. However, the general problem of handling such references is an open question.

**Program residence**

Where should the program (route procedures) required to fly an aircraft through several simulator centers reside? Should the program be associated with the aircraft and passed with it from center to center or should parts of the program be distributed among the relevant centers? The approach currently used in McROSS simulations is to distribute the program and pass only the aircraft and its flight parameters from center to center.

**Interruption and resumption of route procedures**

Aircraft frequently interrupt their flight plans temporarily in order to avoid severe weather or heavy traffic. The simulation analogy to a flight plan is the route procedure. How should a center handle interruption and subsequent resumption of route procedures? Interrupting the execution of a route procedure in order to follow another one is not difficult. The difficulty arises in determining how to appropriately resume the interrupted procedure. In general, the point of interruption is inappropriate as a resumption point. A considerable amount of (simulated) time can elapse between interruption and resumption during which the flight parameters (position, speed, altitude and heading) of the aircraft can change significantly. Therefore, the usual programming technique of saving the "state" when an interrupt occurs and restoring it after the interrupt has been handled is inadequate. The interruption/resumption problem is made more complex by the possibility that between interruption and resumption the aircraft may fly out of the airspace of one center and into the airspace of another. The current McROSS implementation is not prepared to handle interruption and subsequent resumption of route procedures.

**Error handling techniques for distributed systems**

The question of how to handle error situations in a distributed computational system is a challenging one. In McROSS considerable attention has been given to making nodes in a simulator network autonomous. The strategy for handling errors is to try to achieve a "local" error recovery whereby a node attempts to preserve its autonomy. As a result, while the actions it
takes are locally optimal in the sense that its continued operation is insured, they may be sub-optimal in the more global context of the entire simulation network.

Errors occurring in inter-node messages are simply handled in the current McROSS implementation. Recall from an earlier section that inter-node messages are submitted to the parsing mechanism of the destination node. When a node receives a message which is syntactically incorrect or semantically meaningless (to it) from a neighbor, it reports the error on its on-line keyboard, sends a message to the neighbor causing the error to be reported on the neighbor's on-line keyboard, and ignores the message. This procedure is locally satisfactory in the sense that it guarantees that messages which are not well-formed cannot cause the node to malfunction. However, if the incorrect message from the neighbor is one critical to the outcome of the simulation, this procedure is not globally acceptable. Ideally, upon detecting an error in a message, the node should engage in a dialogue with its neighbor in an attempt to resolve the error. The difficulty in implementing this strategy is that it is frequently unclear what should be done to resolve an error. Often the cause has been masked by the time the error is detected.

While the simple techniques used in McROSS for error handling have proven adequate, it is clear that more effective techniques can be developed.

CONCLUDING REMARKS

The results of the work reported in this paper are applicable in two areas.

One area is research in air traffic control. Researchers can use the McROSS system to conduct simulation studies preliminary to the design of an automated multi-component air traffic control system. For example, McROSS could be used to evaluate various ways of partitioning responsibility among components of such a system. Or, it could be used to compare different strategies for automated scheduling and control of aircraft. Because it exhibits autonomy of parts and the ability to dynamically reconfigure, it could be used to experimentally study the behavior of failure handling techniques for a multi-center system under various air traffic loads.

The other area is the design and implementation of distributed computation systems. The results in this area are incomplete and tentative consisting primarily of insights and attitudes developed from the experience of building and using a distributed computation system. These are summarized by reviewing the goals for the McROSS system in terms of problems they posed and techniques useful in realizing them.

Of the five goals, autonomy of parts and deferral of process/processor binding were the most significant in terms of effort required to achieve them and influence on the appearance of the system to users. Given their realization, the other three goals (the ability to dynamically reconfigure a simulator network, decentralization of control and ports for monitoring) were relatively easy to achieve.

The strategy of implementing parts of the distributed computation by process groups rather than solitary processes contributed significantly to achieving autonomy of parts. The multi-process implementation made it possible to dedicate certain processes at a node to maintaining an interface with other nodes and to dedicate other processes to performing functions crucial to the node's existence. In addition to insulating vital internal node functions from the actions of other nodes, the functional modularity resulting from multi-process nodes had the effect of reducing the complexity of the implementation: each individual process being considerably less complex than an entire node. The multi-process capability which the TENEX operating system supports for each user job was invaluable in carrying out the multi-process strategy. It is unfortunate that few operating systems allow users to create and maintain process structures.

Equally useful in realizing autonomy was the establishment of and strict adherence to protocols for part-part interactions. A center can expect monitors and adjacent centers which are functioning properly to observe protocol and can therefore interpret a breach of protocol as a warning that the offender may be malfunctioning. A consequence of the multi-process implementation of nodes is that inter-process communication occurs within McROSS at two levels: inter-node and intra-node. Use of a protocol for intra-node interactions helps insure that internal node data bases always accurately reflect the condition of the node's interface with other nodes. A useful implementation rule was to reject any technique whose success depended upon the order in which events in different centers or in different processes within the same center occur.

The major problem in implementing deferred process/processor binding was providing a way for parts of the computation to determine the location of logically adjacent parts at run time. The solution used in the current McROSS implementation, which requires run time interaction with the user, is not totally satisfactory. A more satisfactory alternative might be for each part to engage in a network-wide search for logically adjacent parts.

We expect to see a trend toward distributed multi-computer systems in the future. By its existence McROSS demonstrates that the construction of such systems is currently feasible.
ACKNOWLEDGMENT

The McROSS system grew from an idea proposed by Dr. W. R. Sutherland of BBN; we gratefully acknowledge his guidance and enthusiasm.

The realization of the current McROSS implementation is the result of building upon the work of others, most significantly the designers and implementers of the ARPA computer network and of the TENEX operating system.

REFERENCES

1 L G ROBERTS B D WESSLER
Computer network development to achieve resource sharing
Proceedings of AFIPS SJCC 1970

2 W R SUTHERLAND T H MYER E L THOMAS
D A HENDERSON
A route oriented simulation system for air traffic control studies
Proceedings of the Fifth Conference on Applications of Simulation December 8-10 1971

3 F E HEART R E KAHN S M ORNSTEIN
W R CROWTHER D C WALDEN
The interface message processor for the ARPA network
Proceeding of AFIPS SJCC 1970

4 S CARR S CROCKER V CERF
HOST-HOST protocol in the ARPA computer network
Proceedings of AFIPS SJCC 1970

5 D G BOBROW J D BURCHFIELD D L MURPHY
R S TOMLINSON
TENEX, A paged time sharing system for the PDP-10
Paper presented at the Third ACM Symposium on Operating System Principles October 18-20 1971
published in Communications of the ACM March 1972

6 ARPA network current network protocols
August 1971 Available from the Network Information Center as NIC #7104 at Stanford Research Institute
Menlo Park California 94025

7 W NAYLOR J WONG C KLINE J POSTEL
Regarding proffered official ICP
Unpublished memo available from Network Information Center NIC #6728, 1971

8 A SHOSHANI
A solution to the race condition in the ICP
Unpublished memo available from the Network Information Center NIC #6772 1971

9 E W DIJKSTRA
Cooperating sequential processes
Appears in Programming Languages edited by F Genuys
EWD123 Dept of Mathematics Technological University
Eindhoven The Netherlands 1965

10 D STERN
Weather data
Unpublished memo available from Network Information Center NIC #7892 1971