The structure of System/88, a fault-tolerant computer

by E. S. Harrison
E. J. Schmitt

In recent years, there has been a growing requirement for continuous processing capability approaching 24 hours per day, 7 days per week. Industries such as finance, transportation, securities, and telecommunications have continuous-availability requirements that can approach downtimes of not more than three minutes per year. This paper describes configurations of the Stratus/32 continuous processing computer system that are marketed as the IBM System/88 through an agreement with Stratus Computer, Inc. The system achieves its fault tolerance via hardware duplexing coupled with a distributed operating system that allows system resources to be distributed over many separate computers while maintaining a single systems image to the end user. This single systems image may also be extended across a network of multiple systems. The way in which software makes this distribution possible and the way in which system resources are named to allow transparent distribution across the system are described in the paper. Also described are the transaction processing services that are part of the operating system and allow transaction programs to be written to operate effectively over the distributed system, by means of a requester-server structured approach.

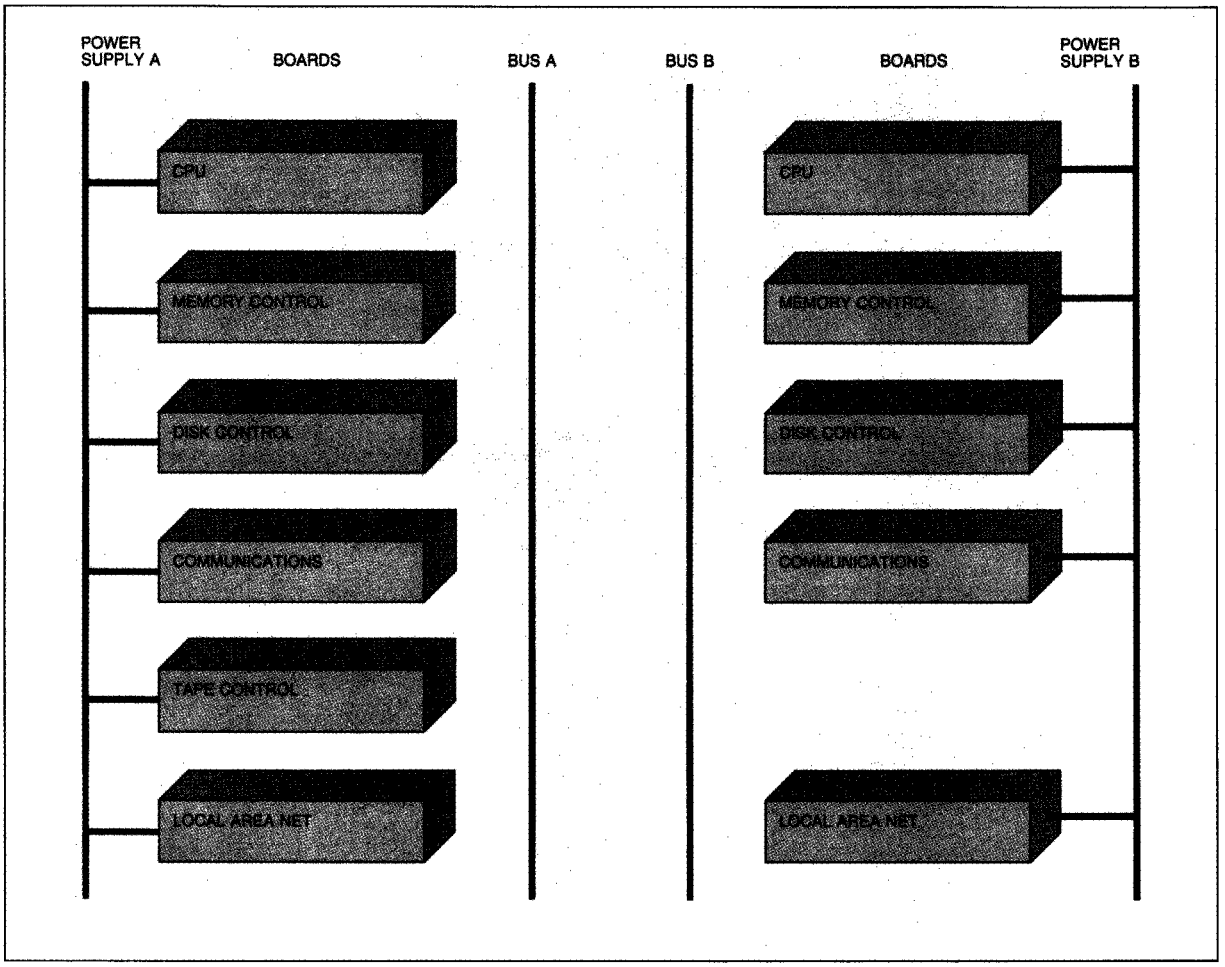
Discussed in this paper is the IBM System/88, which is a fault-tolerant computer system based on the concepts, design, and architecture of the Stratus/32, manufactured by Stratus Computer, Inc. IBM markets the Stratus system, both hardware and software, under the name System/88. Also provided are value-added products, both hardware and software, such as the following:

- Device support for the 5150 PC, 5262 printer, and other selected IBM devices
- Software enhancements in support of national language requirements
- Systems Network Architecture (SNA) products

The hardware and software act together to permit the operating system to continue operation in the presence of a single hardware failure. The operating system is a multiprogramming, multiprocessing system designed for virtual storage and multiple users and which provides a transaction-oriented environment for customer applications. At the same time it provides the facilities required to develop and execute interactive transaction-oriented applications. The system is designed to operate in a nonstop mode which is achieved mainly through the use of the following duplexed hardware components:

- Main memory
- CPU complex
- Input/output controllers
- System bus
- Power supplies

Figure 1 Duplexed hardware component architecture



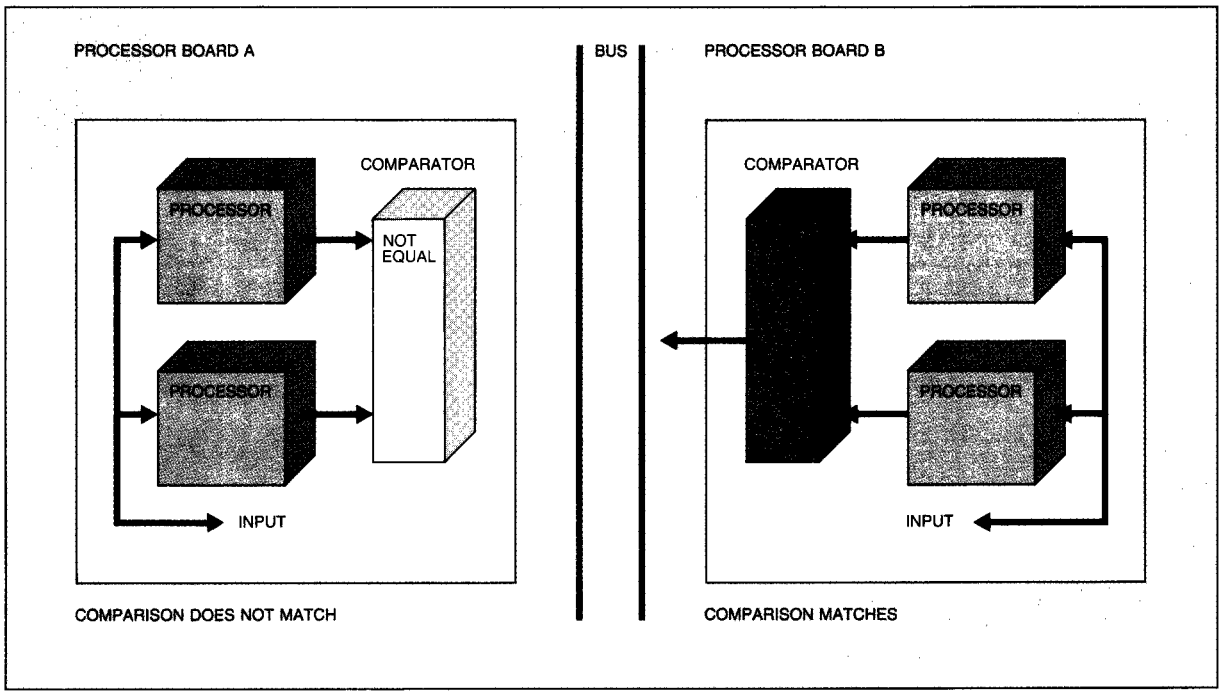
All duplexed components work on the same function at the same time. Out-of-step conditions are immediately recognized and diagnostics initiated. If the diagnostic checks indicate a permanent failure, the unit is taken out of service, and the duplexed partner continues operation. Figure 1 illustrates the duplexed hardware components. Figure 2 illustrates the comparator's functions. The not-equal comparison on Board A indicates a failure on that board. Processing continues on Board B. Diagnostics are then run on Board A, which is removed from service if the failure proves to be permanent.

Each duplexed component comprises two identical sets of customer-replaceable boards, and each board contains duplexed Motorola 68000-based processing elements and comparator circuits. Thus, there are actually four copies of each processing element, two on each duplexed board. If the comparator circuitry on any board detects a difference in the outputs of the processing elements of the board, the board is taken out of service and diagnostics are initiated. If the diagnostics detect a transient error, the board is returned to operation. A permanent error requires the board to be replaced. In any event, the duplexed partner continues operation.

System overview

Fault tolerance begins with power-up diagnostics that locate potential problems before they occur. By

Figure 2 Hardware self-checking via the comparator



combining continuous hardware checking of parallel processing operations with duplexed components, the system provides an extremely reliable operating environment. Only in very rare instances does its service become unavailable because of hardware failure.

The system is composed of multiple, logically independent microprocessors.

If a hardware failure of one of the duplexed boards occurs, this fact is reported automatically to a remote service center. A replacement board is shipped by the service center and usually arrives at the computer center within 24 hours of the reported failure. In the usual case, the system has to run with one of the boards simplexed for a maximum of 24 hours.

Once the replacement board has arrived, the user can replace the failing board by opening the cabinet, removing the failing board, and inserting the new board into the slot. The system continues to operate during this time and automatically brings the new board to the same state as its duplexed partner.

Diagnosing and repairing system failures remotely is possible through service centers located in Boca Raton, FL, and Gaithersburg, MD, in the United States and in Greenford, England, and Sydney, Australia. These service centers are part of a network that makes system service available on a worldwide basis 24 hours per day.

Hardware. The system is composed of multiple, logically independent microprocessors that provide the user with a multiprocessing shared-memory computer that may be termed a *module* or *processor.*

The family of processors have the same system structure: a high-speed central bus with all component boards attached to the bus and able to communicate with one another. The major board types are the following:

- Processors
- Memory
- Disk controllers
- Communication controllers

The latest product line of System/88 consists of Models 81, 82, 83, and 84. The processors are based on Motorola 68020 processing elements, which allow for full 32-bit addressing and 32-bit data access in addition to multiprocessing capabilities:

- Model 81 is a uniprocessor.
- Model 82 is a two-way multiprocessor.
- Model 83 is a three-way multiprocessor.
- Model 84 is a four-way multiprocessor.

The logical structuring of the processor boards is as follows: Eight processor slots, numbered one through eight, and pairs of adjacent slots [(1,2) (3,4) (5,6) (7,8)] comprise the logical processors. A duplexed Model 84 is illustrated in Figure 3. The processor boards consist of duplexed elements. If one of the processor boards fails, only that board is affected; all other processor boards remain in operation. In the case of a duplexed Model 84, only one of the four processors runs in simplex mode; the other three processors continue to run in duplex mode.

Processor configurations are also extremely flexible, in that each processor can run either in simplex mode or duplex mode.

Processor configurations are also extremely flexible, in that each processor can run either in simplex mode or duplex mode. For example, a Model 81 can be duplexed, which means that processor slots 1 and 2 are occupied. Without any additional boards, this can be configured as a Model 82 working in simplex mode by physically moving the processor board in slot 2 to slot 3. Duplex mode for Model 81 is shown in Figure 4A, and simplex mode for Model 82 is shown in Figure 4B.

It is possible to increase the processing power by adding processor boards dynamically while the system continues to operate. Model 81, for example, can be upgraded to Model 82 by inserting processor boards in slots 3 and 4 (for duplexed operation).

Figure 3 Multiprocessing support illustrated by a duplexed Model 84

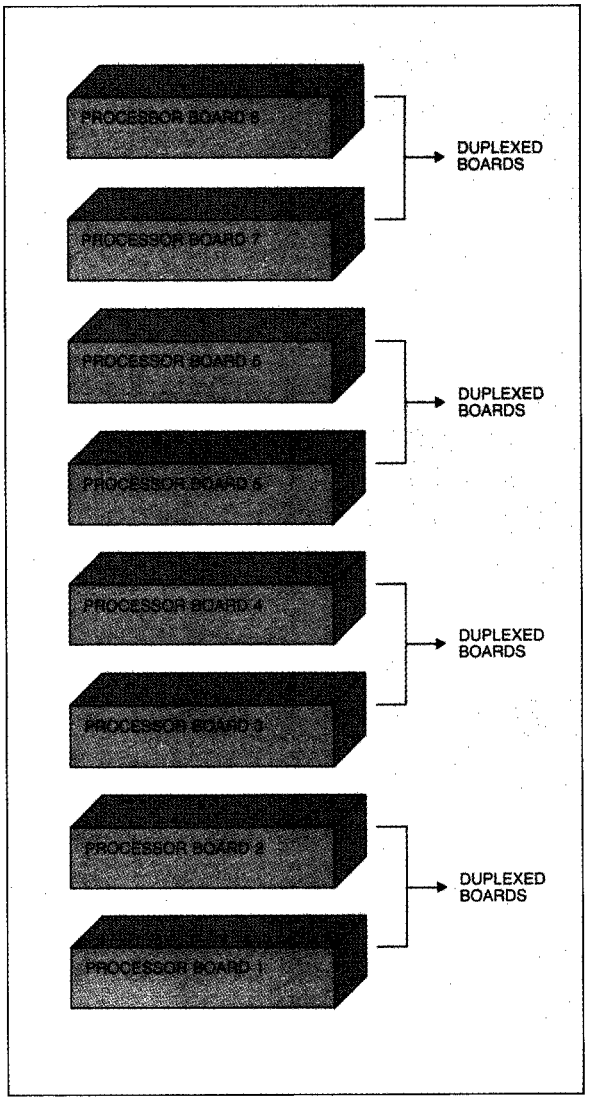
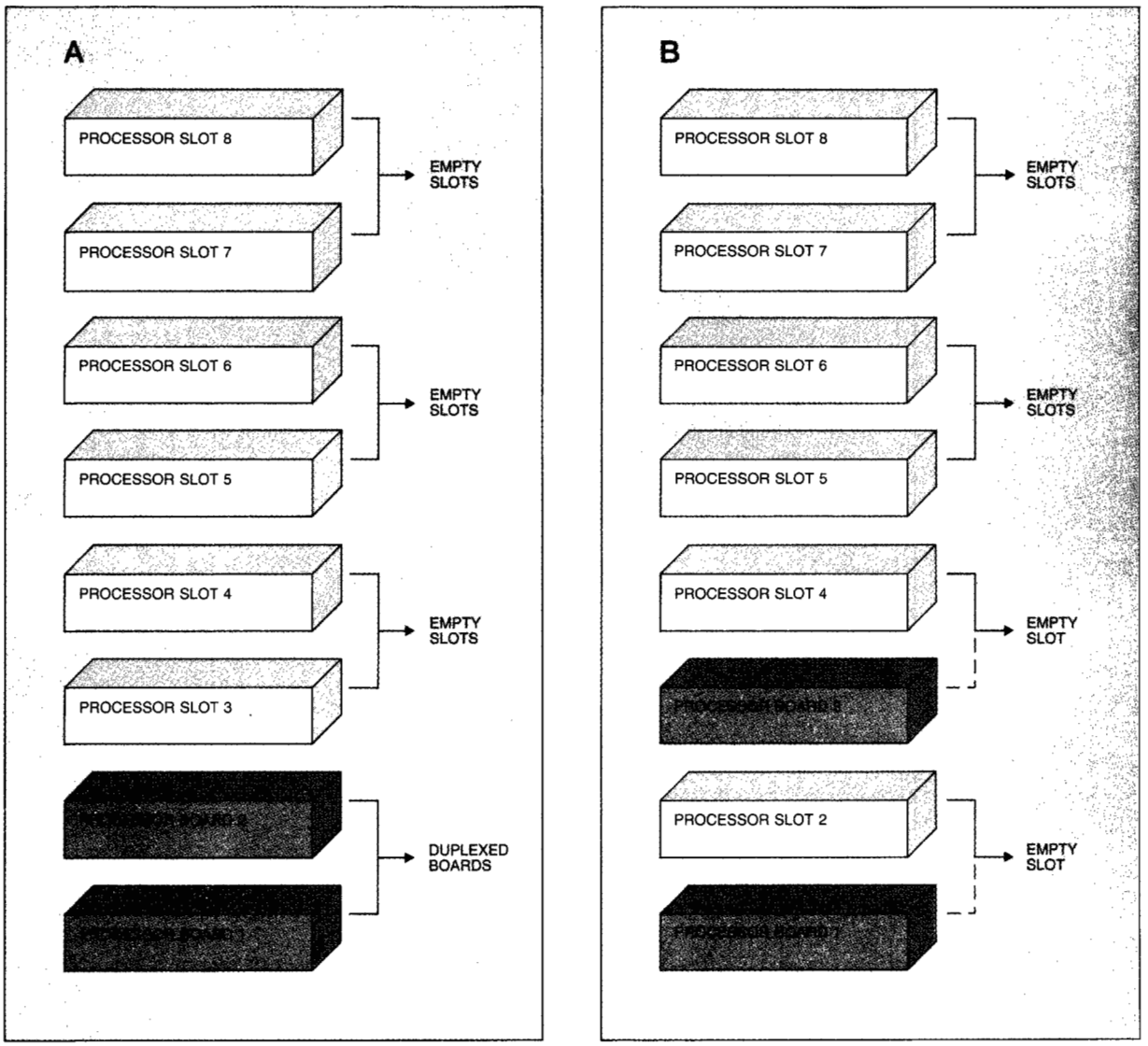


Figure 4 (A) Model 81 in duplex mode; (B) Model 82 in simplex mode



This allows dynamic vertical growth within each processing module of the system.

The system provides integrated, fault-tolerant disk operation through the following two key mechanisms:

- Duplexed self-checking disk controllers
- Mirrored disk files, where data are replicated on two separate disk files

Each individual controller uses dual logic and comparative circuitry to immediately detect and isolate incorrect operations. A duplexed controller ensures that processing continues, with no performance degradation, should a controller fail.

Users are protected against lost data due to read/write failures or head crashes through the use of mirrored (replicated) data on two physically different disks, each separately controlled by one of the duplexed disk controllers.

The mirroring is accomplished by the operating system software. Writes are executed to both disks, and reads are executed to the disk whose head is closer to the data, to improve disk I/O performance.

Essentially, there are two physically separate data paths to two separate copies of data on disk. Users include the operating system itself, which has duplicate copies of the system tables and control blocks and is, therefore, protected from head crashes and controller failure. Figure 5 illustrates the duplexed disk controllers and mirrored file operation.

The system uses a general-purpose operating system (OS) that provides services to user processes.

Operating system

The system uses a general-purpose operating system (OS) written in PL/I that provides services to user *processes.* Each user process consists of an execution point and a virtual address space. The low-order portion is reserved for OS and is mapped into the address spaces of every user process. This is referred to as the *kernel space.* Hence, all of the OS is explicitly shared by having all processes mapped to the same operating system code and data.

The user code and data are mapped into the high portion of the address space. The user code requests services from OS via a normal call to an operating system program that is usually handled entirely within the user's process without the need for a process switch.

The operating system code that implements the call is part of every user process address space, but it usually runs at a higher level of privilege.

The actual process can operate at two levels of privilege: *kernel mode* and *user mode.* While in kernel mode, the process has access to all kernel data; while in user mode, kernel data are not accessible. When a kernel call is made that requires access to kernel data, the process is forced to "trap" into the kernel by executing a specific hardware instruction. The OS code that handles the trap validates the user arguments and enters kernel mode. Upon return from the kernel program, the system goes into user mode and returns to the user's program.

Process creation and destruction. Processes are created either by the kernel or by other processes. In general, a process issues a system call, **s$start_process,** with various parameters, such as priority level and directory information. This calls an entry in the kernel to allocate a structure to contain information about the new process and to schedule the process for execution. An option is provided to allow initial execution of a user-supplied start-up command macro. This allows the user to customize any initialization (activation of terminals, connection to database) that is needed before the process runs.

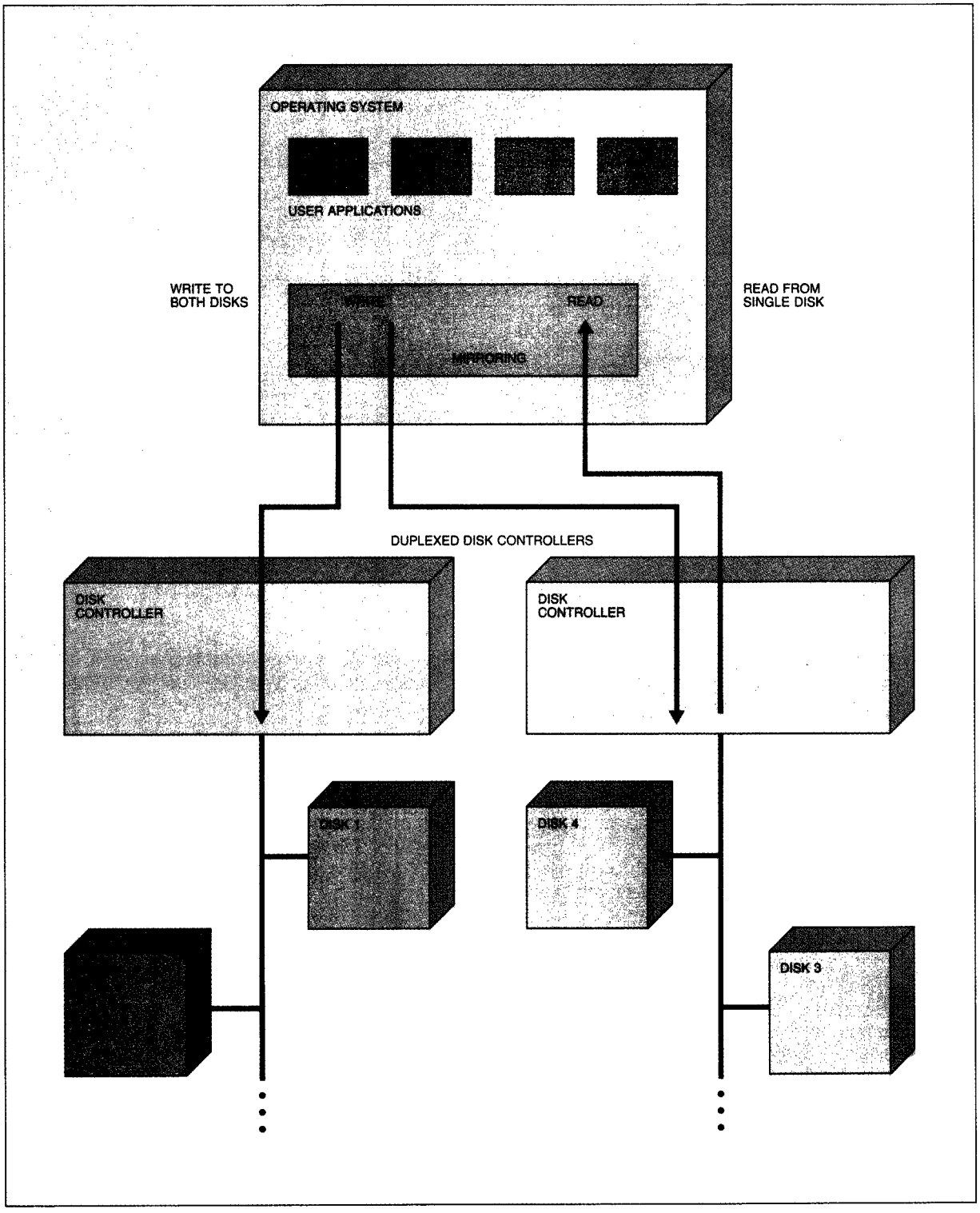
Process destruction is initiated either by the process itself (in which case a simple call to the kernel is sufficient) or by another process. In the latter case, a program interrupt is used to force the process into the destruct state. Process cleanup and resource releasing are executed prior to placing the process in the stopped state. Ultimately, the final destruction is accomplished by a special system process called the *overseer.*

Process types. The system distinguishes among the following types of processes: log-in, slave, and batch.

Log-in processes are created by the overseer for attached terminals. These processes are initialized to a pre-log-in state, and some form of work invitation message may appear on the terminal monitor. A user may physically log in from a terminal to the operating system and begin an interactive session, issuing commands, editing files, and running programs from menu-driven interfaces. When finished, the user may log out, and the terminal process reverts to the pre-log-in state.

Processes that are not log-in processes are called *slave processes.* These processes can be created by log-in processes, by system processes (e.g., the overseer), or by other slave processes. Slave processes present an environment for executing application programs that are typically not associated with terminals. *Batch processes* are slave processes created by the

Figure 5 Duplexed disk controllers with mirrored file operation



overseer and run under control of a typical operating system queued batch facility.

All processes in the OS present the exact same user environment to the programs that run in the process. This means that a program written to run from an interactive log-in terminal can also be run in batch (background) mode, as long as any required input is prepared ahead of time and stored where the batch process can access it. Similarly, any job typically run as a batch job can be run in a log-in process with no special action necessary.

Distributed operating system

A system may be a single *module*, and contains the necessary hardware and software elements to operate as a single, stand-alone system. The following hardware elements are part of each module:

- CPU processing boards
- Main memory
- Power supplies with battery backup
- Disks, tapes, and communications I/O adapter cards
- Terminals
- Controller boards for disk, tape, communications, local-area network (LAN)
- Cables and wires

The software consists of the following elements:

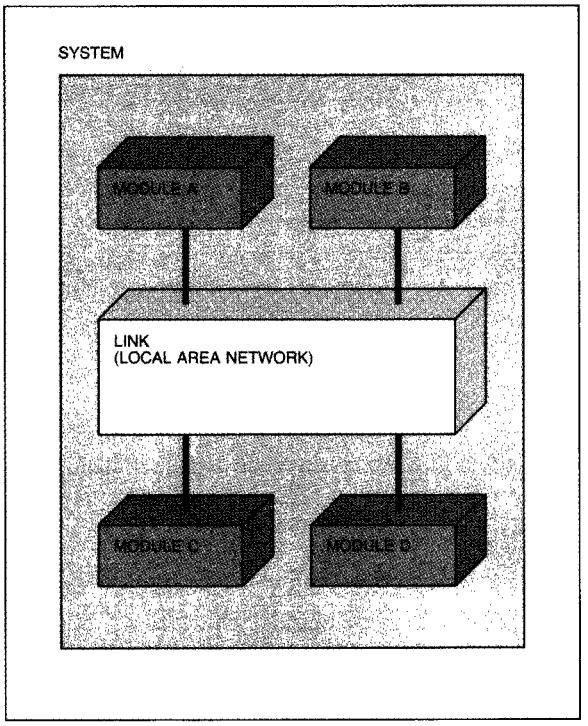
- Operating system and user applications that run in the main CPU
- System programs running in the controllers
- System and user programs running in the line adapters

Two or more modules can be interconnected; modules located at the same site (i.e., the same or neighboring buildings) can be connected through a high-speed local-area network. This connectivity is provided by the LINK facility, where the modules are all members of a single system and can access and share resources with one another.

The maximum distance the modules can be physically separated over the LINK is three miles. If modules must be separated farther than that, they may be connected by means of an X.25-based network of systems. The product that provides this support is called NETWORK.

NETWORK allows systems to be connected over X.25 links or packet-switched data networks (PSDNs). In this case, the system modules are controlled via separate operating systems, and the modules are parts of separate systems; still, it is possible to access resources directly by their names. In this sense, there exists over a network of systems the concept of a single-system image. This means that resources may be addressed directly, and the operating systems support the distribution aspects involved by means of system requesters and servers. (Distribution is discussed in more detail later in this paper.)

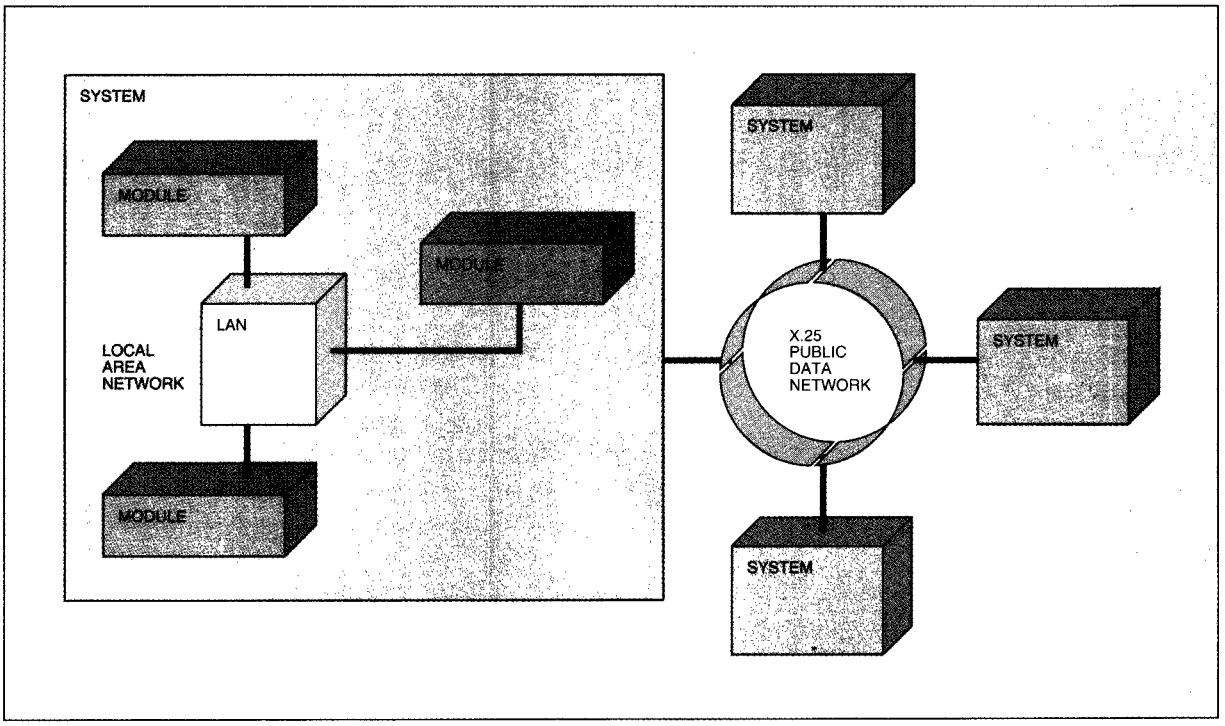
Figure 6 A system of multiple processing modules



The concept of a *system* is, therefore, that of a single module or a group of modules connected via the local-area network. All modules in a system behave as an integrated unit with respect to resource sharing and administration. Figure 6 illustrates the concept of a system of processing modules, and Figure 7 illustrates NETWORK in the form of a network of systems.

LINK details. The LINK interconnects multiple independent modules via a high-speed local-area ring network. Interconnections may be duplexed to ensure continued operation in the event of a single

Figure 7 A network of multiple systems



LINK controller board failure. This also allows the user to double the bandwidth between adjacent modules in the system. Each LINK runs at 1.4 megabytes (MB) per second, and, when duplexed, multileaved throughputs of 2.8 MB per second are possible.

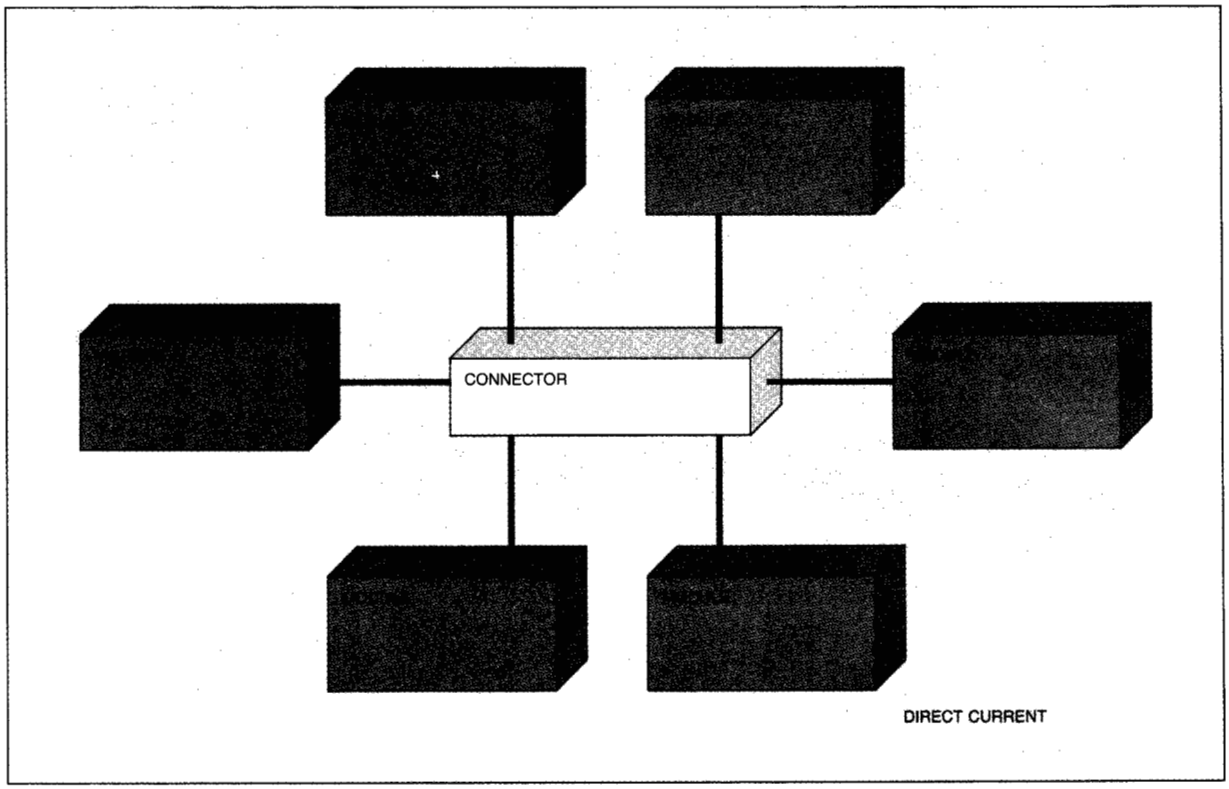
Different models of the system may be mixed on LINK, while maintaining a single system to the user. This allows great flexibility when one is considering "growing the system" horizontally. (Horizontal growth is discussed later in this paper.)

The LINK controller uses dual logic and comparative circuitry to control LINK operation and interface to the main module bus. Modules are connected by nonpowered, passive devices called *LINK connectors* that provide for the connection of up to six modules. The connector contains a relay for each controller, and in the event a module on the LINK is disconnected, the relay provides an automatic bypass, thus ensuring continuity between the remaining modules. This is extremely important in a system that provides continuous availability, because it would be impractical for the malfunctioning of a single system module to bring down the whole system.

Each module passes a direct current down the center wire of the coaxial cable that energizes the relays in the connecter. This puts the module into the ring and illuminates a diode within the module frame. If the module develops a malfunction, the current is cut off and the relays de-energize, thus bypassing the module and extinguishing the diode. This is a great improvement over designs in which one failed module in a normal ring network disables the entire network. This is not the case with LINK. Figure 8 illustrates the function of the link connectors.

To connect more than six modules, a *LINK extender* can be used to connect two or more link connectors. LINK extenders can also be used to increase intermodule distances in increments of 1500 feet, up to a maximum of three miles. The overall limit of the configuration allows up to 32 modules on a single link, and the maximum cable length achievable in a system is ten miles.

Figure 8 Link connectors



Ring network protocols are normally either Carrier Sense Multiple Access with Collision Detection (CSMA/CD) or token-passing protocols. The CSMA/CD protocols allow nodes to transmit at any time. If two nodes start transmitting at the same time, causing a collision, they both stop sending. After waiting a variable amount of time, they retransmit. In a token network, collisions are avoided by allowing transmission by the token holder only. This avoids collisions and thus allows high ring utilization, but lowers the throughput at low ring utilization, because each node must wait for possession of the token before transmitting.

The approach used in the LINK product is a combination of the above two methods. In one case, the LINK uses an implied token in that each module assumes it has the right to transmit as long as it is not currently transmitting. This means that collisions may occasionally occur. Otherwise, when a collision is detected, the transmitting module stops and delays for a variable length of time before retransmitting. This delay is calculated by module number, and the lowest-numbered module gains control of the LINK in these cases.

The LINK controllers constantly transmit a stream of one-bits to the next module in the ring when in an idle condition. This acts as an I-am-here indication and allows link controllers to recognize a failing module.

Each transmission on the link is marked by a preceding zero bit to break a stream of one-bits and to define the start of the data packet. Each packet consists of a 16-byte header structured as follows:

- Source
- Destination
- Control information
- Data—up to 4 kilobytes
- Two-byte cyclic redundancy check (CRC) character
- Single-byte reply field (added by the destination module)

Each module receives and immediately retransmits messages not destined for it, which involves a retransmission delay of two bits at each module. However, there is no main-CPU software or overhead involved, because the transmission is executed entirely by the link controller. Destination modules append a response byte before retransmitting.

When a module receives a packet originated by it, that packet has gone completely around the ring, and therefore is removed. As the message is removed, the CRC and response bytes are checked. If an error is detected, the packet is retransmitted.

Packets being transmitted around the ring also provide for collision detection. When a packet is transmitted, it should be the first one received by the transmitting module. If it is not first, a collision is assumed to have occurred, and the delay and retransmit procedure is executed.

NETWORK details. NETWORK provides for the interconnection of multiple systems via routes based on the CCITT X.25 standard that may traverse packet-switched data networks (PSDNs) or point-to-point X.25 lines. A modem is required for each line and provides the standard RS-232C interface. This interface is connected to a synchronous line adapter, which in turn is connected to a communications controller board in a module.

NETWORK supports the network interconnection configurations illustrated in Figure 9. A connection between two systems may be achieved in one or more of the following ways:

A. Single direct line: One system assumes the role of data circuit terminating equipment (DCE), and the other assumes the role of data terminal equipment (DTE).
B. Multiple direct lines, providing increased reliability: Within each system, all lines can be connected to the same or different modules.
C. Indirect routing through intermediate systems: Each system is connected to one or more other systems, and the NETWORK provides the best route selection between any two of the systems.
D. Indirect routing through a packet-switched data network between two systems.
E. Indirect routing through a hybrid network: This may be a combination of direct and public networks, using the interconnections in Figures 9A through 9D.

Best-path routing in NETWORK. In order to establish intersystem communications using NETWORK, the system chooses a route through the various nodes in the network. This is accomplished via a network routing table that is maintained in each module (of each system) to indicate all possible routes from the current system to each system in the network. These routes are sorted in order of increasing end-to-end delay time. The best routes available to every system are kept current, and, if a route becomes inoperative, the next best path route is automatically chosen. This can be as simple as having two routes through two different links between adjacent systems. This function, termed *best-path routing*, is illustrated in Figure 10. The support is accomplished by the operating system in a manner that is nondisruptive to the end user (either the application or the terminal user). The end user does not have to carry out any recovery procedures and does not have to be aware that a link in the network has failed. The system handles this rerouting dynamically on behalf of the user. In Figure 10, paths 1 and 2 are two alternate paths between systems. Path 2 is selected as the faster path. In case of failure in path 2, path 1 is selected.

The best routes available to every system are kept current.

A special system process, called the "network watchdog" process, executes on every module of a system. For both LINK and NETWORK communications, this process periodically checks to see whether other modules have either lost or regained contact with the current module. As part of this procedure, the network watchdog cleans up any resource attachments that have become inoperative due to communication failures. It also monitors for failed or reinstated network routes and automatically causes the updating of the network routing tables.

Figure 9 System interconnections using NETWORK

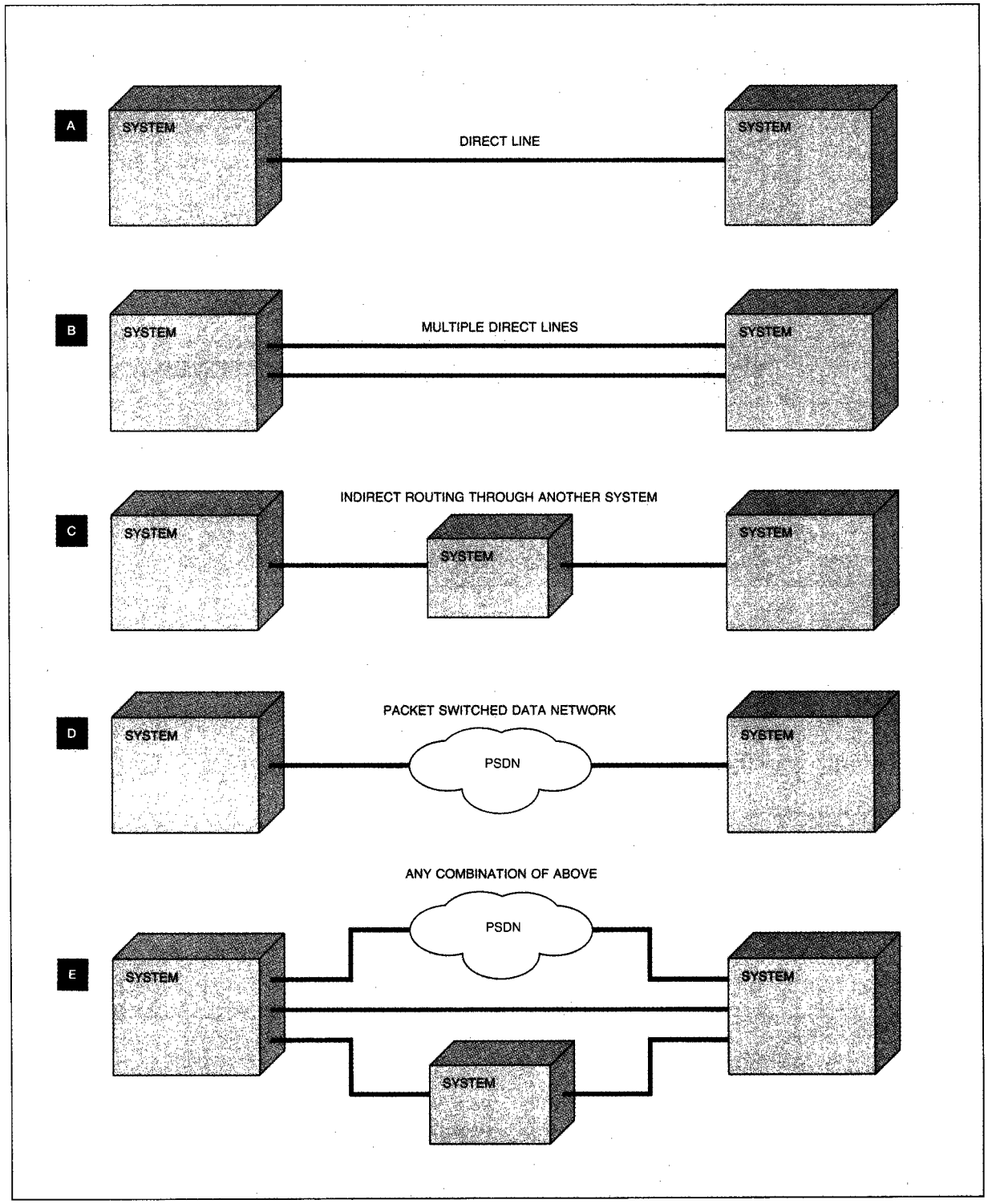
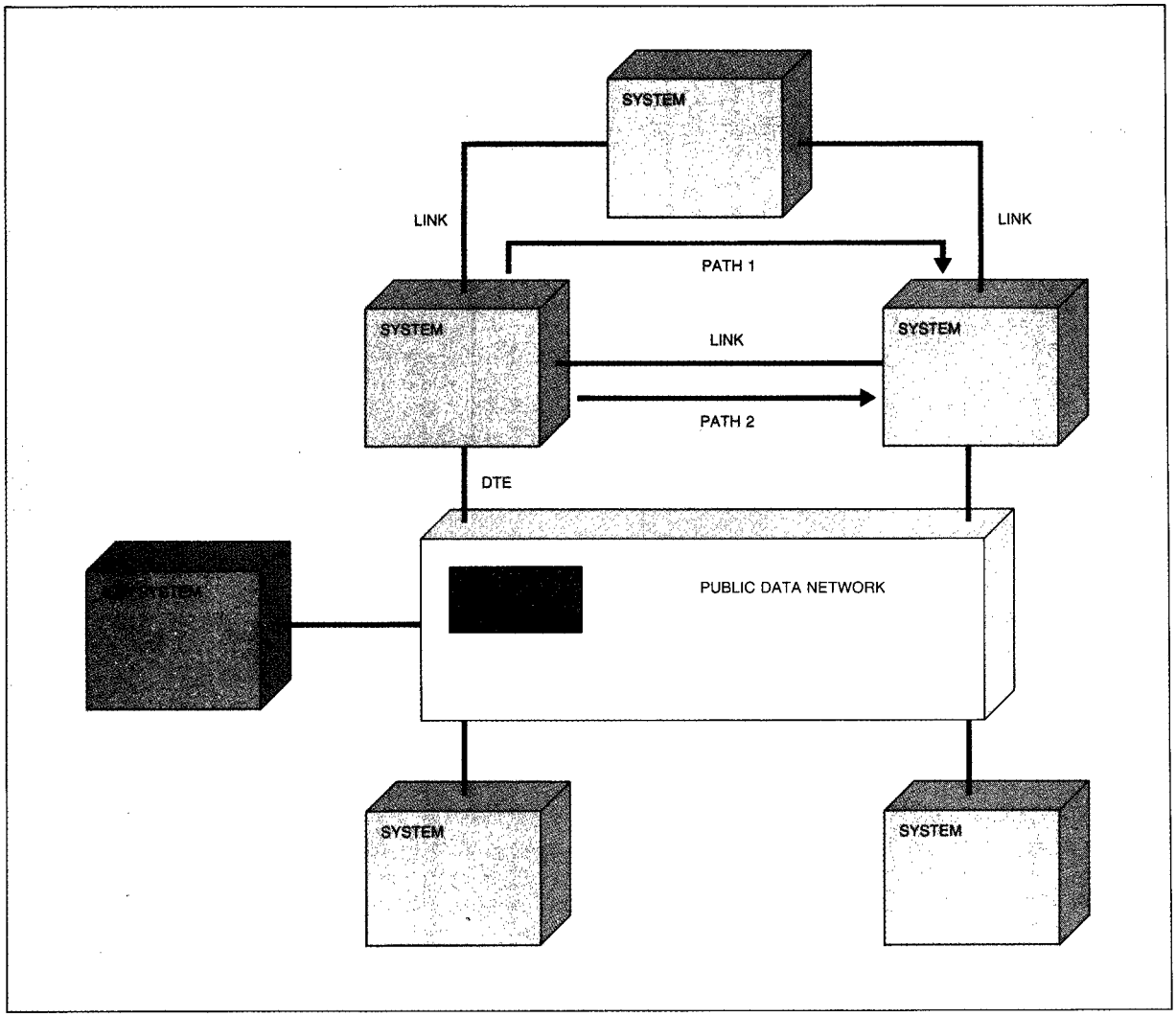


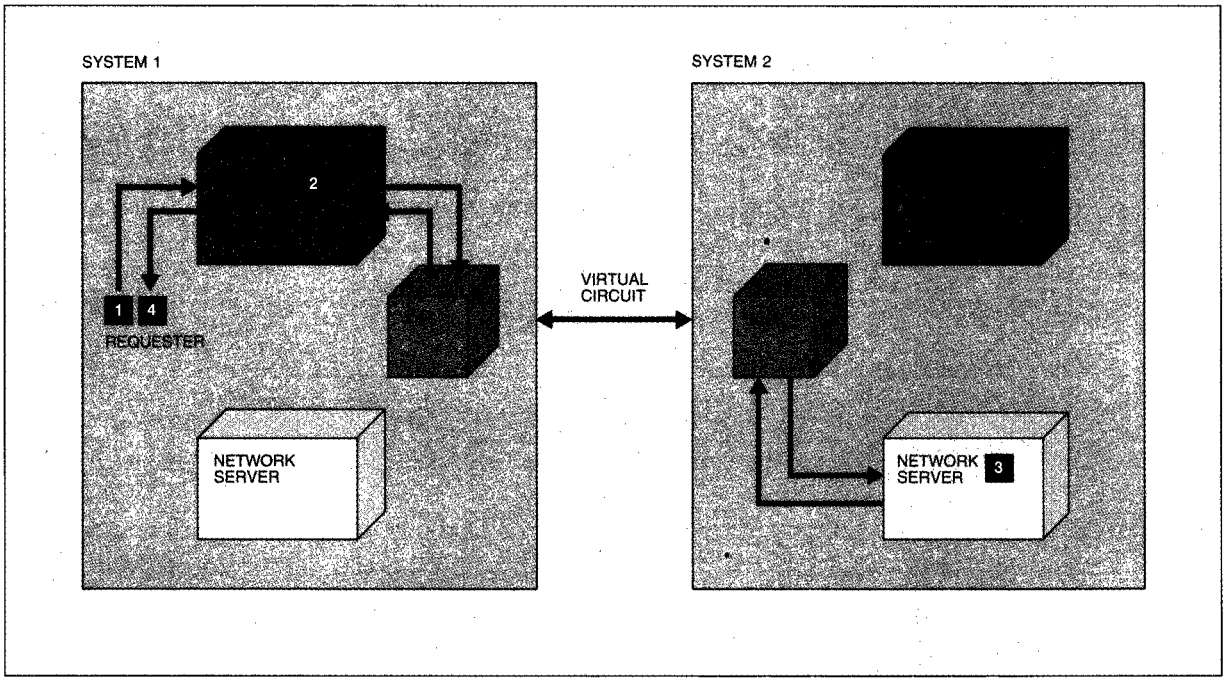
Figure 10 Best-path routing



Requesters and servers. The use of the LINK and NETWORK facilities is completely transparent to the end user. When the application references a remote resource, whether it be in another module in the same system or in another module of a different system, the system automatically allocates and accesses that resource on behalf of the end user. It does this via a series of message exchanges between a *requester process* and a *server process* in the two modules. The requester process is, in general, a user process that needs service from a module other than the one on which it is currently executing. To obtain this service, the requester process sends a request message to the server process in the appropriate module. The server process is a system process that is logically part of the operating system. The server process performs the request and returns a reply to the requester. This sequence is referred to as a network transaction.

Intrasystem servers. Each module provides one or more server processes to serve requesters in other modules of the same system. These servers exchange messages with requesters via the LINK attachment. Because all modules within a single system contain identical tables describing the total configuration,

Figure 11 Path of a NETWORK transaction



each module is aware of the resources of the other modules in the system. If an end user wants to access a record in a file, the system has a record of the disk/module in which the file resides, via the system directory and the file's *pathname* (explained later). The system routes the request to the particular server on that module. The server executes the request and transmits the record back to the requester. Multiple server processes can be assigned to the same module, thus allowing two or more network transactions to be processed simultaneously. Requests are assigned to idle servers as they arrive, and if all servers are busy the request is queued.

Intersystem servers. Network transactions across systems are accomplished by a pair of system-provided processes called a *network client*, running in the requesting system, and a *network server*, running in the server system. Communication between the two processes is achieved by means of an X.25 virtual circuit. The virtual circuit is a bidirectional communication path between two processes that is designed to permit access to the CCITT X.25 packet layer (level 3). The packet layer is implemented within the system via a special process called an *X.25 gateway* process. This process executes in the module (referred to as a *gateway module*) within the system that contains the actual communication line connection to the external network. Figure 11 illustrates the relationship between the network client and the network server. NETWORK is in operation when a user process requests services or resources that are in another system; it is transparent to the user. The following sequence of steps describes a NETWORK transaction. Note that a NETWORK transaction can also include LINK transactions. Figure 11 illustrates the following example:

1. The requester user process needs service from a remote system. NETWORK sends a request message to the network client process within its system. If the requester and network client processes are located in different modules, the request is transmitted to the gateway module via the LINK.
2. The network client forwards the request message over the virtual circuit to the corresponding network server process in the remote system.
3. If the request can be executed in the network server module, e.g., if the file exists in this module, it is executed; otherwise, the request is forwarded via LINK to a link server in the target module of the server system.

4. The reply message is eventually returned to the requester, using the reverse path.

Both intrasystem and intersystem server performance are controlled by assigning scheduling priorities and by the use of multiple server processes. Network transactions vary widely in size.

At LINK speeds, these differences are not noticeable. However, at NETWORK speeds a single large transaction can monopolize a server for a relatively long time, during which time other smaller transactions must wait. The use of multiple client-server pairs between two systems reduces this effect, because it is unlikely that several large transactions will occur simultaneously.

Naming conventions. All resources within a system are uniquely identified via a resource name. Resources that can be named include the following:

- Boards
- Processing modules
- Devices: terminals, printers, and communication channels
- Disks
- Tape units
- File directories
- File data sets

An overall naming convention is established to allow users to reference certain resources within a system. The convention identifies the resource name as a set of qualified names called *path names.* The qualifiers of the path name are as follows:

- System name, for example, **%Sys1**, where % is the delimiter and **Sys1** is the actual system name

The second qualifier may consist of the name of any one of the following resources:

- Module
- Disk
- Device: terminal, printer, and communication channel
- Tape

The fully qualified path name uniquely identifies the resource within the system. Because the full *resource name* contains the *system name*, this name is also unique within a network of systems connected via the NETWORK product.

Examples of resource names are the following:

- **%sys1#m4** identifies processing module **m4** in system **sys1**; the # and % signs are the delimiters and are user-definable.
- **%sys1#d02** identifies disk name **d02** in system **sys1**.
- **%sys1#term5** identifies terminal name **term5** in system **sys1**.

Directories and files are identified via the disk on which they reside. Consider the following example: **%sys1#d02>dir2>file1**. This identification defines **file1** contained in directory **dir2** residing on disk **d02** contained in system **sys1**. The delimiter is **>**. All qualified names are user-definable.

Directory structure. The directory structure consists of a hierarchical structure of objects, such as the following:

- Subdirectories
- Files
- Links

Links are pointers to other objects, which are in other directories. The directory structure is illustrated in Figure 12.

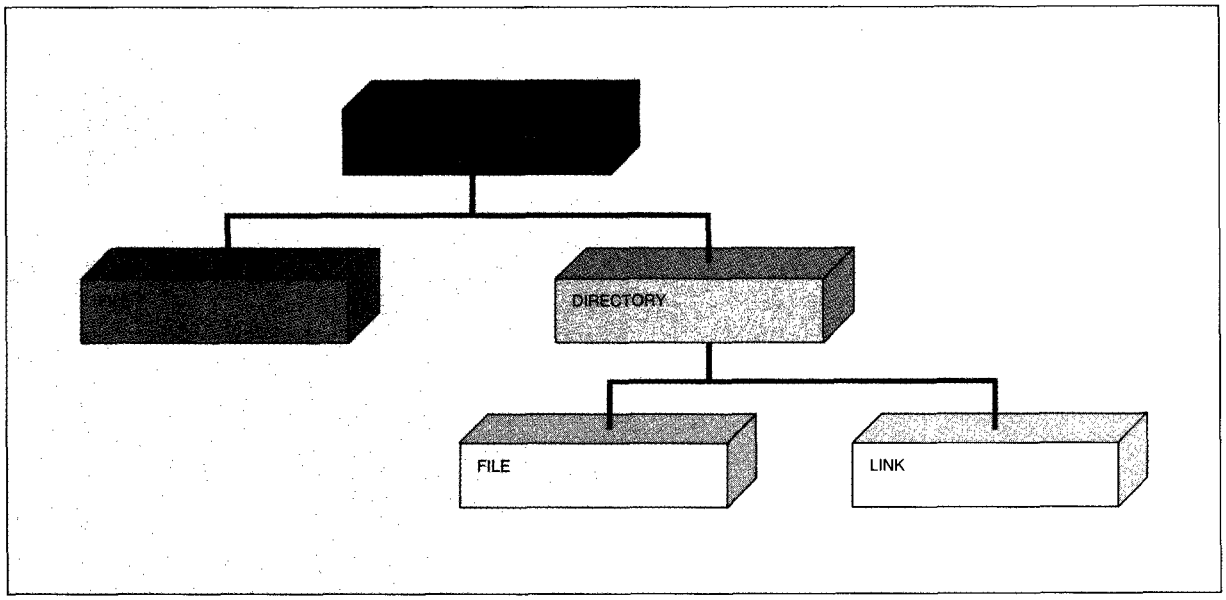
The root of a directory defines a logical disk, and all objects on the directory must reside in a single logical disk. A logical disk can be composed of more than one physical disk, and each logical disk has a *packmaster directory* on it that uniquely identifies it to the system. The packmaster may be considered to be the root of the directory.

One packmaster on each module is designated the *master disk*, and this is used to keep the *system* and *group directories* for that module. System directories contain the following libraries and files:

- System files
- Command libraries
- Object libraries
- Include libraries
- Tool libraries

Group directories are used to help categorize users. For example, each department could have its own group directory, and each group directory could contain *person directories*, which are directories associated with each user. Group directories are illus-

Figure 12 Directory structure



HARRISON - FIGURE 12

trated in Figure 13. Users of the system are known by user names, which are of the form **person·group**.

Note that the accessing of resources in the system is location-transparent, even though the path names of the resources contain identification of the system and module at which the resource is located. One may conclude that if the resource is moved to a different module or system, the name must be changed. This is not the case, however. If the resource is to be moved, a LINK may be inserted in the directory slot for the resource to indicate its new location. The search then continues at the new directory location, and the resource is eventually located by the system. Thus, it is possible to move resources between modules of a system without changing the resource names.

Starting a multimodule system. It is not necessary to power-on every module in a system manually. All the modules in a single system on the LINK can be powered-on and started up from any one individual module. This is accomplished via a special setting of the key switch on the console of a module.

The key switch on the console of a processing module can be set to the "System Master" position. When this is set and the "Power On" switch is pressed, this module (as well as all other modules on the same system) begins the power-on sequence. After power-on, each module begins its own start-up sequence. Thus, one person can start the entire system from a single module.

At automatic start-up, the code in the programmable read-only memory (PROM) on the CPU board runs diagnostics and self-tests. After these have run successfully, the module finds the master disk, and code in the PROM starts up the master disk and reads a utility program that loads the operating system from the default boot partition of the master disk. The operating system initializes all configured devices and disks and then creates a process that executes the commands in a special file called MODULE_START_UP. In general, this file contains commands that create other system processes for software program products, such as NETWORK, X.25, and SNA.

System configuration concepts. The overall physical configuration of a system is defined to the system by a set of special files called *table files* that can be created and modified only by a specific privileged user of the system, who is known as the *system administrator.* These files are always stored on the master disk of each processing module in a directory called **system>configuration**.

Figure 13 Example of a group directory

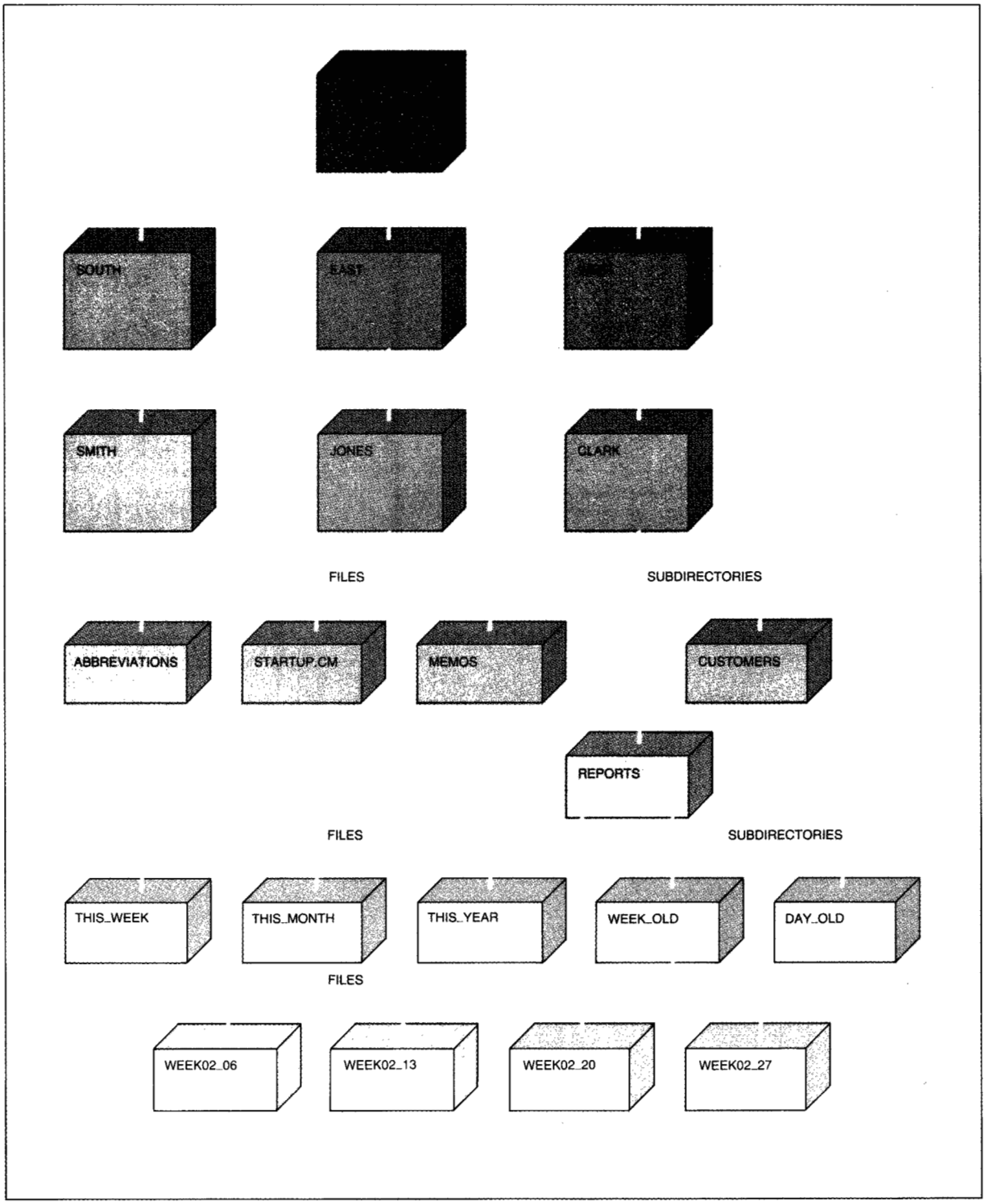


Table files exist for the following components:

- Systems
- Modules
- Boards
- Disks
- Devices
- Gateways
- Nodes
- User registration and access control

The system administrator manipulates these files via a corresponding table input file **.tin**, which is essentially a sequence of record descriptions that specify the contents of the records in the table file. To change the contents of a table file, the system administrator modifies the contents of the corresponding **.tin** file and then executes a special system command, called **create_table**, using the **.tin** file as input.

All modules within the system on the LINK each maintain identical copies of all table files in their respective system directories on their master disk. The system administrator manages this via a system command **broadcast_file**, which sends an updated table file from the executing processing module to all the other modules in the system. Thus, with one command, the table is updated in the master disk system directory of all modules. This reduces the burden placed on the system administrator and allows definition updates from a single module of the system.

There are several privileged commands that tell the operating system on a particular module to immediately recognize new components added to the table file. These are called the **configure_components** commands. Thus new components, such as terminal devices, disks, and modules, can be dynamically added to the system and allocated to users without requiring an IPL procedure. This is of immense importance for a system in which services must be continuously available to users.

This concept promotes horizontal growth of the system by allowing nondisruptive growth of processing modules and increased processing power and function within the system. The additional module(s) can be utilized for such purposes as the following:

- Off-loading the current modules that may be at full capacity
- Increasing the overall system performance
- Dedicating the module as an exclusive file server, transaction processor, or communication server

Administrator procedure: Adding a module. Assume that we are to add module **m3** on the LINK that already contains two existing modules **m1** and **m2**. The system administrator actions to accomplish this are as follows:

1. Physically connect **m3** to the local-area network LINK.
2. From a terminal at either **m1** or **m2**, dynamically update the module, disk, and device **.tin** files with the new information about **m3**.
3. Issue **create_table** and **broadcast_file** commands. This sends the updated table file to the other modules of the system.
4. Issue **configure_components** commands on both **m1** and **m2** so that the new resources added on module **m3** are recognized in modules **m1** and **m2**.
5. IPL **m3** so that users on this module can access resources available in the updated three-module system.

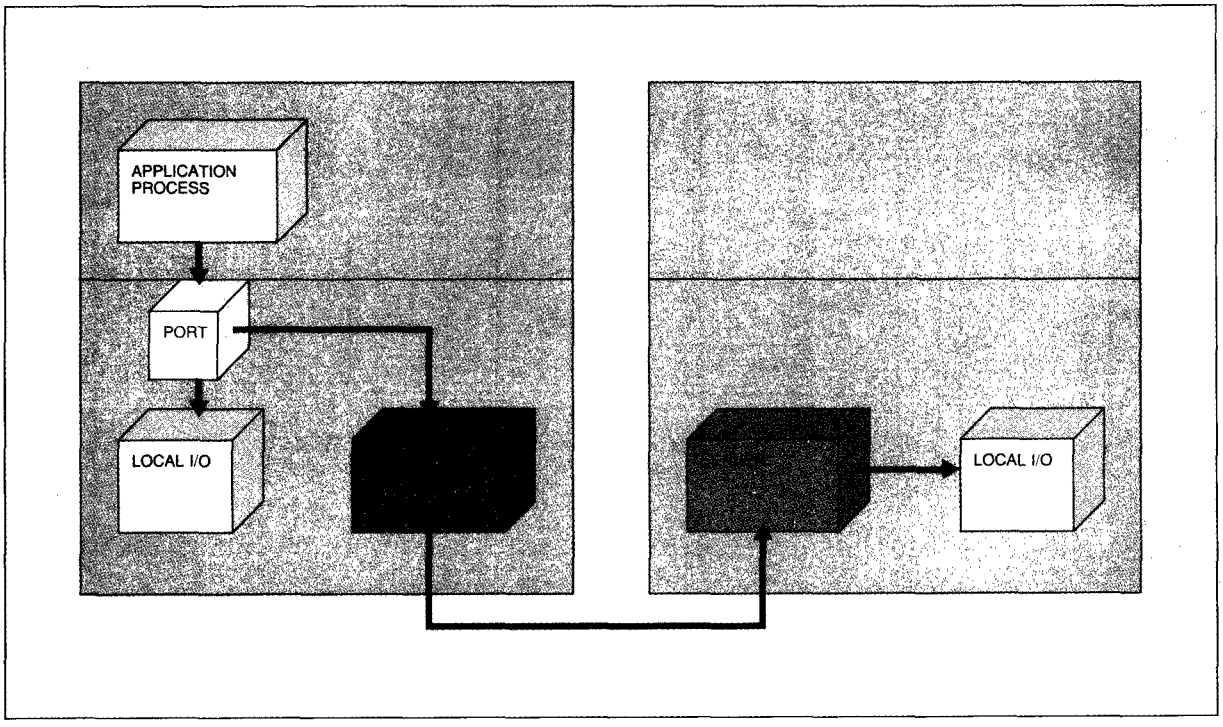
Generic I/O concepts

The operating system provides users with an I/O system that consists of a standard set of routines that apply to all devices and files. These routines are accessed directly through entry points into the operating system space or more conveniently through language I/O facilities supported by the system's high-level languages.

All I/O uses the concept of a *port*, which is a per-process logical channel through which all I/O operations are performed. The operating system maps a named file or device into a local *port* ID via an application process call to the *attach* function. This port ID is returned to the application process, and from then on the application process simply refers to this port ID for all subsequent I/O activity, such as read and write.

Within the operating system space, the port ID actually maps to a unique set of pointers, called *transfer vectors*, that invoke the unique file or device routines to execute the requested function. In addition, if the device or file is located in a remote processing module, these vectors include calls to the appropriate forwarding routines, either LINK or NETWORK, to invoke the remote servers to carry out the request. Figure 14 illustrates these concepts, which are em-

Figure 14 Generic Input / Output structures



bodied as generic I/O structures. If the *attach* request specifies a local resource, the respective transfer vectors for the local I/O handler are set up for the port ID. Similarly, for a remote resource, the remote router transfer vectors are associated with the port ID. Thus, requests may be routed to the appropriate server for execution in the target processing module.

Transaction processing capabilities

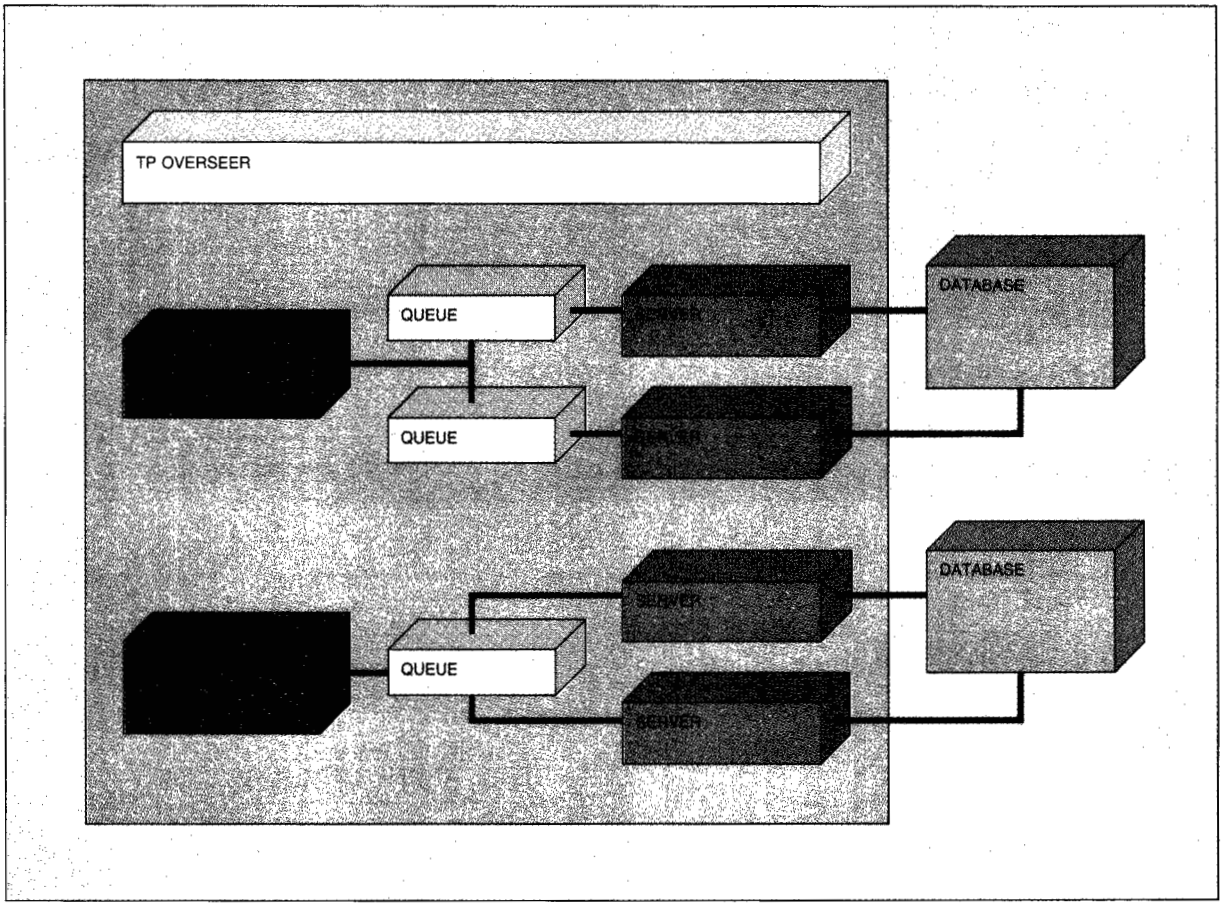
Up to this point, we have discussed fault tolerance and the distributed nature of the operating system over several processing modules and have seen that in each module different levels of multiprocessing are available. In this section, we discuss some basic capabilities of the transaction processing system. These facilities are important because a major use of the system is in on-line transaction processing for which continuous availability of services is required.

The system contains an integrated transaction processing facility that allows programmers to write programs in any one of the following six programming languages:

- PL/I
- FORTRAN
- COBOL
- Pascal
- BASIC
- C

The system has no distinct transaction manager in that terminal users do not have to log in to the transaction system to use functions provided by it. However, in each module there exists a transaction processing (TP) overseer that takes note of changes made by a protected transaction to the file system. Any transaction may be protected; file changes made by the transaction are either all made or all backed out to maintain file consistency and integrity. Transactions can be protected by calling a **start transaction** system routine; protection can be ended by calling a **commit** system routine. Transaction protection is not mandatory. When the transaction does not need protection, these statements need not be issued in the transaction, so that nonprotected transactions are not penalized with any performance overhead.

Figure 15 Transaction requester server model



All system facilities are available to the transaction. This gives a more integrated software system because all functions are contained within the operating system and used by the transactions. No duplication of function exists between the operating system and the transaction system that can lead to problems in maintenance and software development.

Because a single transaction may have work executed in multiple modules or even systems, **commit** support is provided by a two-phase protocol that allows the files at the different processing modules to be either all updated or all backed out.

Discussed earlier in this paper was the system requester/server model for providing distribution between modules (and systems). The user also has this structure available when writing transaction programs. Basically, a requesting process is connected to one terminal (or to several terminals, if multitasking is used). The requesting process receives the input (transaction requests) from the terminal operator.

The requesting process then requests service from one or more servers that are located in the same or different modules of the system to cause the transaction function to be carried out. Typically, these server processes are located at the database site. For example, a transaction may entail a simple database lookup. Once this function has been executed, a reply may be sent back to the requesting process, which then sends a reply back to the terminal operator. The requester/server structure is very general, and servers can either be single-threaded or multi-

threaded, depending on the demand made on the service being provided. The server configurations are illustrated in Figure 15.

Transaction protection. For transaction protection, a TP overseer must be active in each module of the system. Also required are the following two files: the Transaction Work Area (TWA) file and the Transaction Log File (TLF).

For the user to initiate protection within his application program, a **s$start_transaction** instruction is issued. To end this protection, either of two instructions, a **s$commit** or a **s$abort**, is issued. The only changes protected are those to transaction files.

When the **s$start_transaction** command is issued, an internal transaction ID is assigned by the system. This allows the system to keep track of the changes made to transaction files during the execution of the transaction. The transaction ID is not seen by the application. Therefore, only one protected transaction can be active at any one time for a particular application.

As changes are made to files throughout the system (possibly in different modules) by the transaction, the TP overseer in each module keeps a table of the changes made to the transaction files for each protected transaction. These changes are kept in memory. All file records changed by the transaction are locked automatically by the system so that data file consistency is maintained for other system users. For each server request, the system appends to the request the transaction ID that allows the TP overseer to build the appropriate tables. The changes to the file system can occur on any module of the system in a manner transparent to the user.

The following occurs when a **s$commit** command is issued. The system, on receiving the **s$commit** in the module where the initiating transaction is executing, communicates with each TP overseer involved for this particular transaction. Each TP overseer then sends the tables kept in memory to the TLF on disk. When the transmission is completed, a response is sent to the initiating module. Phase 1 of the **commit** has been completed when responses have been received from all TP overseers involved in the transaction. Should some crash occur in the system after this point, the transaction system at re-IPL time can continue applying the changes, because all changes are on disk, along with the identification and status of the transaction.

Phase 2 of the internal **commit** support works as follows. The initiating TP overseer sends a request to each participating TP overseer to begin writing out the records that have been changed. This is essentially a two-step process.

In the first step, all of the actual changed records are created in the TLF file. Information regarding what is in the TLF at any stage is kept in the TWA file. Both the TWA and TLF are then written to disk.

At this point, the afterimages of the transaction changes are stored on the TLF. The next step is to apply these changes to the actual physical records.

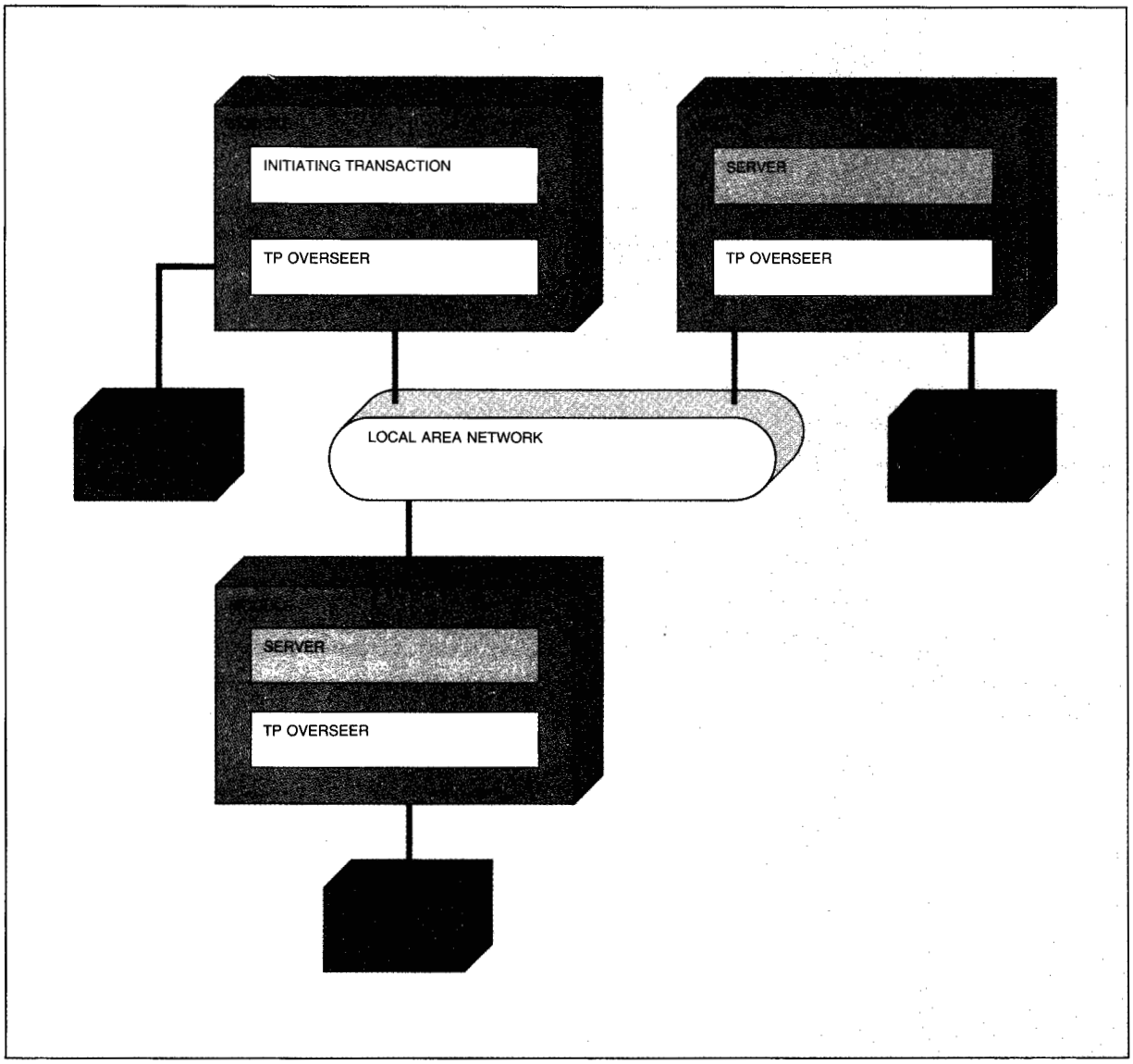
As these are being applied, the TWA is updated to indicate which of the records have been written out to the actual physical locations of the records, so that if a crash does occur the writing out of the records can continue after re-IPL.

Once all records have been written out to disk for the transaction, all locks held by the system for this transaction are released. The file system now reflects all changes made by the now-committed transaction. This flow is illustrated in Figure 16, where the following steps are taken: (1) a protected transaction is started that involves three processing modules; (2) servers make changes to the file system in each of the three modules; (3) an initiating transaction commits changes made or not made.

System protection against deadly embrace. The system continually checks for a "deadly-embrace" situation. That is, user A has locked record *a* and wants record *b*, while user B has locked record *b* and wants record *a*. If such a situation arises, the system backs out one of the transactions on the basis of user parameter settings. On **s$start_transaction**, a parameter is defined that allows the user to indicate how he wants the system to handle a deadly-embrace situation if it should arise. The user is given the following two choices of deadly-embrace avoidance parameters:

- Priority of process: Abort the lower-level process that issues the request and causes a deadly embrace.

Figure 16 Transaction processing



- Length of waiting time: Abort the transaction that has been waiting for the lock (that creates a deadly embrace) the shortest time.

Systems Network Architecture (SNA) implementation

The system offers communications capabilities that allow a wide attachment of asynchronous and bisynchronous (BSC) terminals. In addition, it is possible to connect across a BSC link to a System/370 host, thereby allowing applications to communicate with System/370 applications. BSC terminals can also connect downstream to the System/88, and a user passthrough application allows terminal users to communicate with the System/370 host.

However, no significant SNA capability existed in the system as originally announced. Many users required that their system be incorporated fully into SNA

Figure 17 System Network Architecture capability

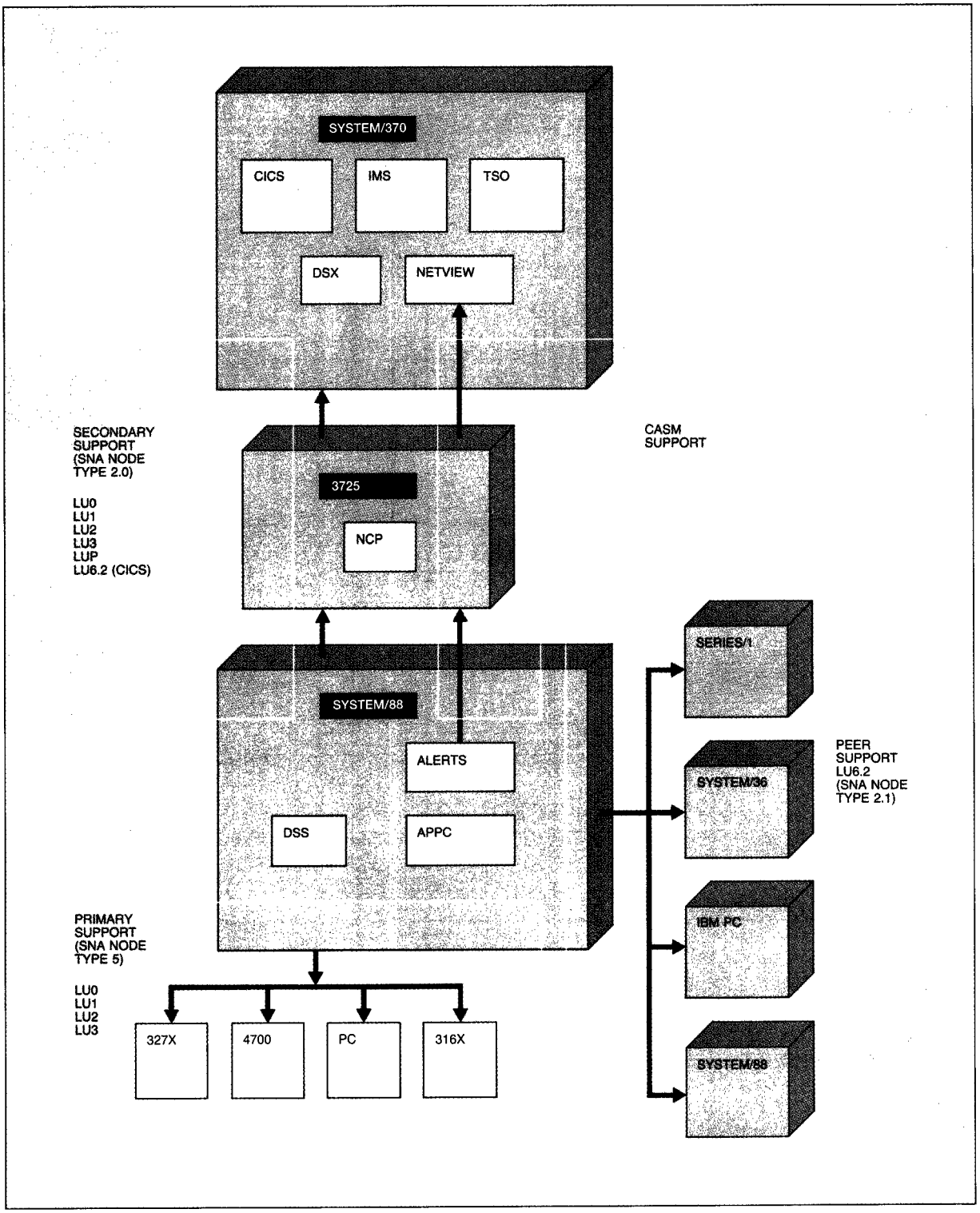
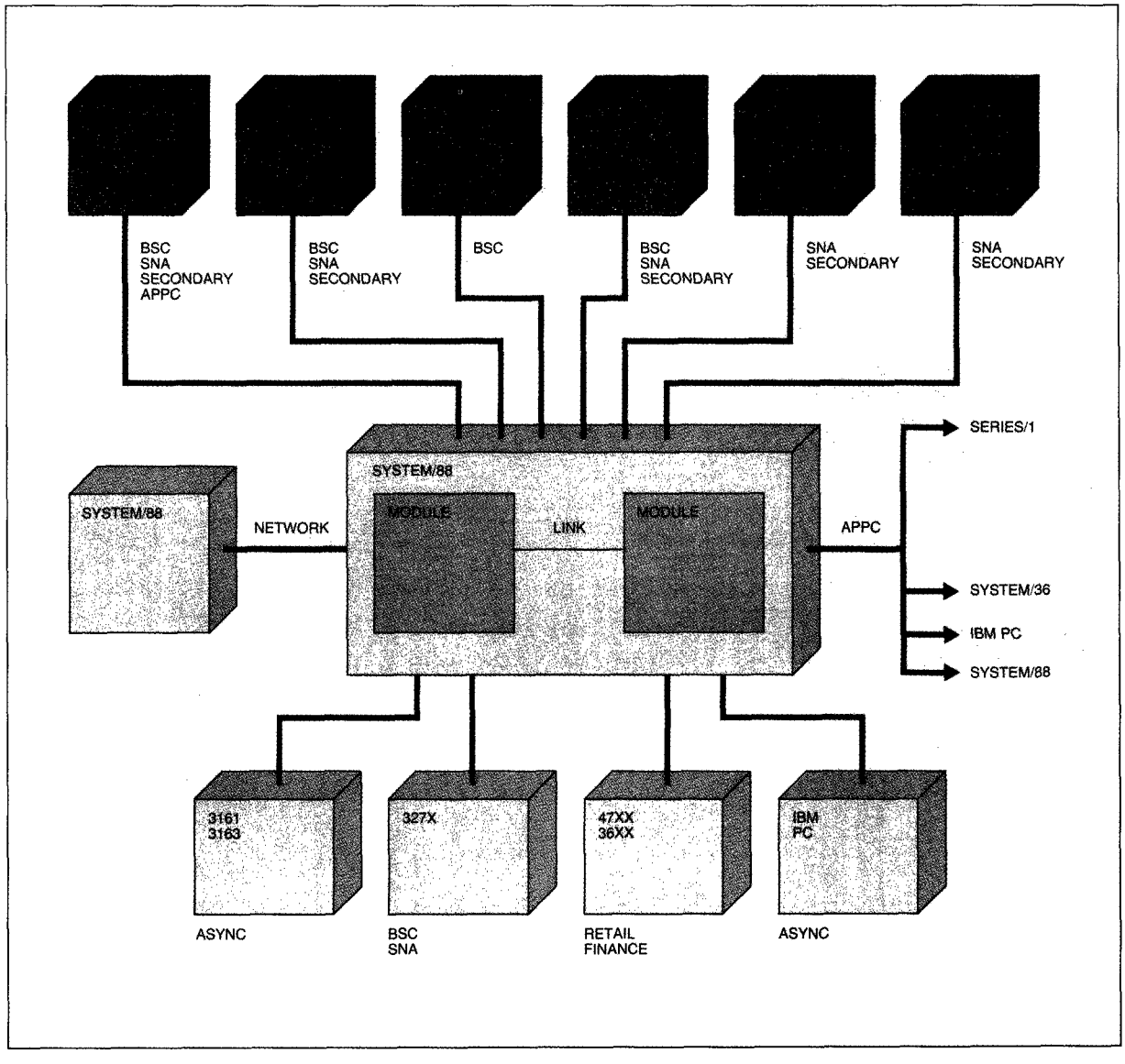


Figure 18 System communications capability



networks and have an affinity for System/370 subsystems and other SNA products. This connectivity is particularly important for continuous-availability applications within an SNA network, because the system provides a fault-tolerant layer where routing to System/370 hosts (and other systems) can be achieved. In this fault-tolerant layer, limited stand-in processing is possible in the case of host-availability outages owing to network or system failures. In essence, the system can provide a fault-tolerant view of the network to a user.

The interesting challenge in adding the SNA function is that of function placement within the distributed operating system. It is important to take advantage of the distribution characteristics already existing in the system and not to introduce artificial constraints in which the SNA function can execute within a

system configuration. These considerations give the application programmer the ability to run the application on any processing module independently of the locations of the internal SNA function and Synchronous Data Link Control (SDLC) links in the system.

The SNA function has been modularized into the following major components:

- SNA network administration provides system administration capabilities to define SNA resources to the system via a menu-driven interface.
- Configuration manager provides a common configuration management for the SNA products.
- Control point provides a common control point for the SNA functions.
- Logical unit services provide user session management.
- Path control provides network layer routing function.
- SDLC link control manages the flow of data across a physical link.

The SNA function has been designed so that distribution among the foregoing SNA components is possible. Each of these major components can execute in separate processes by utilizing the interprocess capabilities and can execute in any processing module of the system. Similarly, transaction programs using the SNA support can run in the same or another processing module from the logical unit (LU) that supports it. The SDLC link support can execute in yet another processing module.

By making the SNA components completely distributable, the user can effectively place the SNA function throughout the system as best fits requirements. The SNA function consists of the following major functional areas:

- Primary support is principally for downstream-attached cluster controllers, including support for LU types 0, 1, 2, and 3.
- Secondary support is upstream of an SNA host. The system appears as a cluster controller (Physical Unit Type 2.0) to the host and supports LU types 0, 1, 2, and 3.
- Advanced Program-to-Program Communication provides support for LU 6.2. This is a means of providing advanced program-to-program support to allow effective peer connection to such systems as System/36 and System/38.
- Communications and System Management (CASM) consists of two major components: (1) *Alert Generation* causes alerts to be generated by the system and sent to Netview in the SNA host. Alerts generated by downstream-attached terminals are also passed through to the SNA host. (2) *Distributed Systems Services* (DSS) interfaces with *Distributed Systems Executive* (DSX) in the SNA host. This allows files and program fixes to be controlled at a central System/370 database and to be distributed to System/88s in the SNA network.

The SNA capability is illustrated in Figure 17. The total communications connectivity has been greatly enhanced by the addition of the SNA function and is illustrated in Figure 18.

Concluding remarks

The System/88 architecture is a design of Stratus Computer, Inc. This system combines duplexed hardware with distributed operating system software to provide a high-availability, fault-tolerant computing system. Fault tolerance is built into the hardware so that no special programming is required by the application programmer and little is required by the operating system. Fault tolerance is accomplished with no system or application software overhead and with no performance degradation.

All hardware operations are continuously checked at every machine cycle, and component failures are located when and where they occur. Failing components are automatically taken out of service, and the duplexed partner continues processing without interruption. Service calls are placed automatically to a support center, and replacement components are sent directly to the customer site, usually within 24 hours. The user can replace boards dynamically.

The system is designed to grow easily, as customer needs grow, by offering both horizontal and vertical growth. Multiple systems can be interconnected locally or through communication networks. Additional users, terminals, printers, and DASD can be added dynamically. With proper planning, this can be accomplished without interrupting service, while maintaining a single-system image to end users.

The operating system is an advanced virtual-memory system that supports multiprogramming, multiprocessing, multiple processors (processors connected horizontally), and networking.

User-oriented software facilities provide a comprehensive interface between the operator, application program, and operating system. A rich set of higher-level languages and sophisticated development and debugging tools are also provided. A powerful transaction processing facility includes all the tools and structures required to develop transaction processing applications that demand fast response time in a high-volume, on-line environment.

This system provides an integrated solution for enterprises that require continuous processing capabilities that approach 24 hours per day, 7 days per week.

Acknowledgment

The authors thank Dave Van Voorhis for his rigorous review of this paper.

General references

IBM System/88: Connectivity, GG66-0239, IBM Corporation; available through IBM branch offices.

IBM System/88: Introduction to Operating System, SC34-0664, IBM Corporation; available through IBM branch offices.

IBM System/88: Operating Systems Reference, SC34-0665, IBM Corporation; available through IBM branch offices.

IBM System/88: Cobol Transaction Processing Services and Reference, SC34-0674, IBM Corporation; available through IBM branch offices.

Edward S. Harrison *IBM Communication Products Division, 1000 NW 51st Street, Boca Raton, Florida 33432.* Dr. Harrison joined IBM in Hursley, England, in 1970. He participated in the design and development of DOS/VTAM until 1974. In 1975, he was a member of the SNA architecture and design team that was responsible for VTAM ACF (SNA 3). From 1976 to 1982, he was lead communications designer for the IBM 8100/DPPX, for which work he received an IBM Division Award. Since 1982, he has been working on midrange, general-purpose, and fault-tolerant computer systems. Dr. Harrison, Senior Technical Staff Member, is currently lead architect on the System/88 program. He graduated in 1966 from King's College, London University, with a B.S. honors degree in electrical engineering. In 1971, he was awarded a Ph.D. degree in computing science by the University of Newcastle upon Tyne, England.

Edwin J. Schmitt *IBM Communication Products Division, 1000 NW 51st Street, Boca Raton, Florida 33432.* Mr. Schmitt joined IBM in 1962 in Poughkeepsie, New York, as a diagnostic engineer for System/360 Models 65 and 75. He is a senior programmer in the fault-tolerant systems programming development group, where he is engaged in design and support for System/88. Mr. Schmitt received his B.S. in electrical engineering from Manhattan College in 1962. In 1966 he joined the Kingston Programming Center, where he worked on numerous OS/360–370 supervisor and recovery management projects. From 1972 to 1975, he was the design manager for the Systems Network Architecture implementation for VTAM. Mr. Schmitt was team leader for the 8100/DPPX communications development from 1976 to 1983. Since 1983, he has been involved in the architecture and design of communications within minicomputers and fault-tolerant computer systems.

Reprint Order No. G321-5299.