REPRINTED FROM IBM SYSTEMS JOURNAL, VOL17, NO 4, 1978; © 1978, 1999

Today there is a wide range of choice for configuring the data processing facilities of an organization—centralized systems, decentralized systems, small computers, and networks of communicating computers—for distributed data processing. This paper considers the factors that relate to organizations and their data processing requirements and to the various possible data processing configurations. Price-performance ratio, organizational needs, and other factors that recommend the flexibility of distributed data processing are discussed in detail. Also discussed are possible distributed data processing architectures, choice criteria, communications, and application and operating system design principles.

Distributed data processing

by A. L. Scherr

For the past several years, there has been an increasingly strong trend in large enterprises toward the distribution and/or decentralization of data processing. There are virtually no large corporations in which the data processing capacity is focused in a single large computer system, and there are very few organizations that operate in a single geographic location. There are a variety of causes for this phenomenon. Multiple distributed data processing locations may have occurred as a result of corporate mergers, because of security considerations, for organizational reasons, because the capacity of a single location was inadequate, or for other business reasons. Whatever the cause, distributed data processing has become a fact of life. Most of the factors that have led to the present data processing situation are themselves becoming more important; and thus, it appears that the distribution of data processing will become even more common in the future.

This paper first defines "distributed data processing" and contrasts it with what is called "decentralized processing." Next, the potential motivations for going to the distributed approach are

 0018-8670/99/$5.00 © 1999 IBM

enumerated. Implementation techniques are then described. A set of parameters for describing the various approaches is developed that leads to a classification scheme for distributed processing techniques. An example is given of the addition of a new application to an existing distributed system and its effect on the overall balance of the system. Finally, there is a discussion of the general question of application design, the management of the overall system and its growth, and the required characteristics for system facilities in support of distributed data processing.

Throughout this discussion, the general theme is that the form of distributed processing used is determined primarily by the characteristics of the applications to be implemented and the particular motivations for selecting distributed data processing, rather than by abstract or theoretical considerations.

definitions

Distributed data processing is defined as the implementation of a related set of programs across two or more data processing centers or *nodes*. The programs are related in that they share or pass data between them. Each node is generally capable of performing data processing applications independently, and thus would normally have data storage and program execution facilities.

The communications required for the sharing or passing of data between programs can be accomplished either *synchronously* or *asynchronously*. Synchronous communication implies that both programs are in execution simultaneously and that messages pass in both directions, with one program waiting for the other to respond to its last message. Asynchronous communication implies that the sending program does not wait for a response and the two programs may not be in execution at the same time. Often, in the asynchronous mode of operation, the messages that comprise the communications are queued and processed in a batched mode.

The means of communication may be a shared direct-access storage device or a telecommunications link. In the asynchronous case, portable storage media, such as a tape reel or a diskette, may be used. This paper emphasizes the use of teleprocessing links, although many of the conclusions are valid for the broad spectrum of communication mechanisms.

The key element in the definition that has been given for distributed data processing is the communication that goes on between the programs. If two or more nodes are executing applications programs that do not communicate, then the overall system is an example of what is called *decentralized processing*. There can be a spectrum of degrees of communication among nodes, and a threshold exists where the level of communication reaches a point at which the factors to be discussed in this paper become important.

Motivations for distributed processing

There are a variety of reasons for the selection of distributed processing to implement a set of data processing applications. Ultimately, they all offer economic advantages of one kind or another. However, the assignment of value to at least some of the factors must be done in terms of the user's overall enterprise, not just as an optimization of data processing. The ultimate economic value to a particular establishment requires the assigning of dollar values to such factors as response time, reliability and availability, and the ability of the organization to manage data processing in the context of its business. The following are descriptions of the major motivations for selecting distributed data processing as the method for implementing a set of applications.

reliability and availability

One motivating factor in moving to distributed processing is that of better overall system availability. The two complementary approaches to achieve this are the following:

- Limiting the scope of the impact of failures by partitioning the workload among several processors.
- Reducing the impact of failures by providing redundancy.

Figure 1 Limiting the scope of failures

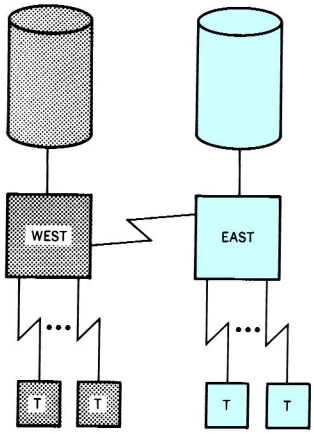


Figure 1 illustrates a simple example of the technique for limiting the scope of failures. Assume that an on-line terminal system is implemented in two computers, one servicing the West Coast and the other one the East Coast. The scope of failures is reduced in that if one of the two machines fails (gray), only half the terminal users are affected. This statement is true to the extent that the terminals on the operational machine can function without access to the data in the other machine. In the case of n machines partitioned to particular sets of terminal users, the scope of failures can typically be reduced to $1/n$th the terminal users. This is in contrast to a single centralized processor failure, wherein all terminals would be affected.

Figure 2 Fail-soft capability

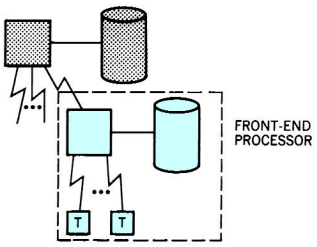
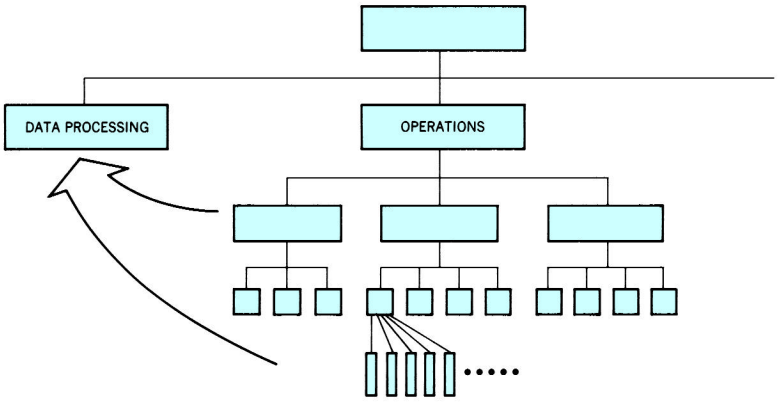


Figure 2 shows terminals connected to a front-end configuration that, in turn, uses a centralized system to perform functions that require increased capacity and/or access to a centralized data base. In the event of failure of this centralized system (gray) or the communication facilities required to gain access to it, the front-end machine could be programmed to perform essential functions, albeit at a reduced level.

An example is point-of-sale terminals in a retail store operation with the application being the processing and approval of credit requests. A large central system handles credit authorization based on individual customer records. In the event of a central system failure or communications failure, the front-end system could perform this operation by using a list of known bad risks

Figure 3 Organizational imbalance



and credit limits established for use in this particular situation. In many cases, the economic advantage of keeping the retail store operational would offset the additional costs associated with possible increased credit risks.

Figure 2 can also be configured for multiple front-end machines, wherein each machine would be assigned to a particular retail store or group of stores. The failure of a front-end machine would affect only the operation of the particular terminals connected to it. Thus, it is possible to achieve a reduction in failure scope while providing for a fail-soft capability.

It should be noted, however, that there is a possibility for a reduction in availability by the addition of redundancy simply because the aggregate failure rate of any system is increased when more hardware is added. Thus, proper use must be made of redundancy, alternate paths to critical hardware, hardware designed for high availability, etc., so that higher availability can, in fact, be achieved.

These techniques of partitioning and redundancy are applicable to different degrees in particular business organizations. Therefore, it is necessary to know the application characteristics in order to evaluate the full potential for reliability and availability advantages.

organizational needs

Distributed data processing is often a means to manage computer usage more effectively. Over the years, the size of the classic centralized data processing organization has grown significantly. This growth has occurred as data processing has become an integral part of doing business. Figure 3 suggests a business organization in which operations functions have gradually been subsumed by the data processing organization. In many cases, as

growth has occurred and as a particular business has become more automated, line management has experienced a loss of control. More and more costs derive from the data processing organization; more and more problems seem to result from computer-related factors. Thus, line managers have felt the need to play a more active role in data processing, and have sometimes successfully gained control over their own computer resources. While in some cases this has meant going to decentralized processing, more often it has resulted in distributed processing because of the inherent relationships among the line organizations.

Another element leading to distributed processing is that as a centralized data processing organization becomes large enough and critical enough it requires more structuring, using the same theories and techniques used to structure line organizations. Thus, there are examples of data processing organizations that are highly divisionalized to create the classic contention system among line organizations as well as between line and staff functions.

Coming from either direction, the result is the same—a divisionalization of the control of the computing resources among several management chains. It is often possible to match this kind of organization with a network of related computer systems in a distributed processing environment. Doing so allows for a level of independence in the management of the individual computer resources while at the same time providing for essential communication among areas and providing for appropriate degrees of centralized management control.

communications costs

Over the years, both communications and computing costs per transaction have decreased, but the author sees indications that the former costs may decrease at a slower rate than the latter. By moving computing capability nearer the end user, both decentralized and distributed processing offer the possibility of eliminating or at least reducing communications costs. Therefore, as time progresses, a move to either method should become easier and easier to justify economically. There are many applications that can justify such a move today, and the trend indicates that it is becoming easier for more and more applications.

One of the basic techniques used to accomplish a reduction in communications costs includes partitioning and isolating particular functions to local areas, so that communications to a centralized or remote system are unnecessary. For example, it might be possible to use a local computer to do inventory control, etc., in a warehouse rather than having a centralized computer for all warehouses. Depending on values associated with the application, displaced communications costs may outweigh possible increases in computer costs due to redundancy. In this case, the decentralized approach would be attractive if the value of computer-assisted

communications between warehouses or between a particular warehouse and a central location were insignificant or could be effectively handled by using other forms of communication.

Other examples of communications cost reduction techniques include the use of remote computers to concentrate the communications with a number of low-speed terminals into a single high-speed line, and the use of a local processor to perform portions of the application without requiring interaction with a centralized machine.

single-system capacity limitations

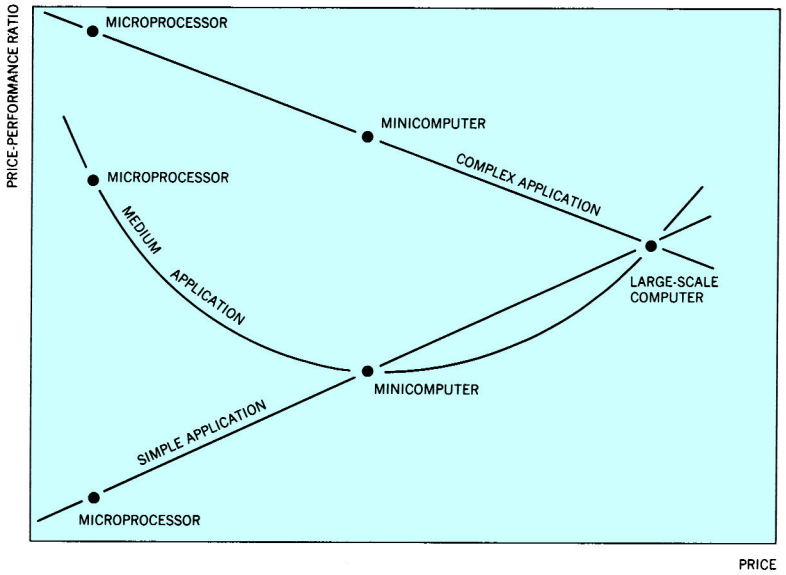
One of the more common reasons for moving toward distributed data processing, or at least decentralized processing, is the fact that a single computer system with adequate capacity is not available. Even though the performance of high-end systems has grown dramatically over the past decade, this growth is still less than the capacity required by a number of computer users. Many enterprises that use digital computers as an integral part of their business have found that the capacity of a single machine has been exceeded simply as a result of increased business volumes. For example, consider a manufacturing plant that is controlled by a data base system that receives orders, controls inventory levels, and schedules manufacturing operations and the shipping of finished goods. Here, the need for computer system capacity is directly proportional to the business volumes of the manufacturing operations. Thus, a doubling in business for the manufacturer would require a proportionate increase in capacity of the computer system.

Another limitation that arises from the use of a single system occurs as a result of the operational complexity associated with ever-increasing workloads. As applications are added over the years to a central system, the cost grows for migration to a larger capacity system and/or new operating systems software. Thus, when a new operating system is installed to support new applications, the old application programs must be at least tested on and possibly converted to the new software. Even if new software is not introduced, adding major new applications often requires the rescheduling of existing workloads and the retuning of the system. In addition, there may be disruptions due to previously unencountered bugs in the operating system that are triggered by the new application. In any case, partitioning the workload across two or more nodes can serve to reduce these problems and allow for more manageable growth.

price-performance ratio

Another motivating factor in the use of distributed processing is that by partitioning applications across several machines, it is often possible to use specialized minicomputers or microprocessors. Such machines can have price-performance advantages over a large, general-purpose system. A simple example il-

Figure 4 Price-performance ratio as a function of price for three computer architectures and application complexities



lustrates this point. Suppose that a particular application program requires only eight-bit arithmetic, and the benchmark program to measure the performance of various candidate processors is a stream of instructions doing eight-bit arithmetic. This benchmark could be executed on virtually any computer ranging from a simple microprocessor to the most powerful large system. The fact is that microprocessors now available at the corner electronics store can execute this type of program with a price-performance ratio more than a thousand times better than the central processor of a modern large-scale system.

Figure 4 shows the relationship between processor price-performance ratio and processor price for a variety of architectures and degrees of application complexity. The price-performance ratio is optimized when the application complexity matches the architecture. The variation in price-performance ratio can be as much as three orders of magnitude between a microprocessor and a large-scale computer, and as much as one or two orders of magnitude between a microprocessor and a minicomputer.

price-performance considerations

The characterization of applications as either simple or complex, with respect to processor architecture, is relatively straightforward at a detailed level. For instance, data items and instructions used to implement an application should closely match the architecture of a machine that optimizes the price-performance ratio. This machine would have little or no unused functional capability, and when the I/O and terminal devices are added to the picture, these same considerations apply. However, because the

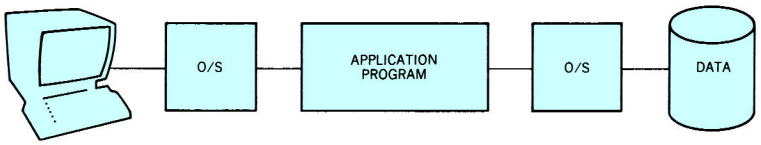
price-performance differentials for main storage, direct-access storage, and terminals are less dramatic between microprocessors and large-scale computers, the price-performance differences tend to be diluted when a total system view is taken. Nevertheless, there are applications that show significant advantages in price-performance ratio, when they are optimized against a machine architecture that is appropriate to the application. Overall configuration costs, depending upon the approach taken, can vary by factors of from two to ten. Obviously, if the capacity requirements of an application are large enough, and if there is a close match to a minicomputer or a microprocessor implementation, the price-performance advantages of selecting such an approach could lead to a substantial dollar savings. Furthermore, if a number of these simpler machines were required to achieve the needed capacity, a distributed processing network configuration would be required.

There are several additional advantages obtained by the use of smaller, simpler machines in a distributed environment. The first is that it is usually more economical to do a pilot version of an application. As an example, consider pilot implementation alternatives for an application whose ultimate implementation would require a single large-scale computer or one hundred minicomputers. These alternatives are the following: (1) the ultimate large-scale computer or a smaller version of it, or (2) a small number of minicomputers. Depending on the price of a smaller version of the large machine, the differences in pilot hardware costs can be substantial. Obviously, pilot studies could be done by using a portion of the resources of the large machine, but this is often not practical from a technical and/or operational point of view.

Another positive aspect of using small machines is the finer granularity of growth increments. Continuing the example of one hundred small machines versus one large one, if additional capacity were required, the addition of one machine to the array of small machines might satisfy the demand for additional capacity. On the other hand, to grow from the large-scale computer might require a second large machine or a transition to one having substantially more capacity (e.g., the next larger machine in the line). Thus, in this example, the growth increment for the smaller machines would be one percent; for the large machine, the increment could be as much as one hundred percent.

Finally, smaller machines in the minicomputer or microprocessor categories require, because of their own simplicity, simpler software support. Since the price-performance ratio of a simple machine is dependent on the nature of the programs it is running, the conclusion is that the operating system software must be correspondingly simple. It is inconceivable that good price-perform-

Figure 5 Idealized application flow



ance ratios can be achieved with an operating system whose complexity exceeds the capability of the hardware. This last factor leads to more tangible advantages in that simpler software allows for reduced costs of installation, problem determination, tuning, education, etc.

Techniques for distributed processing

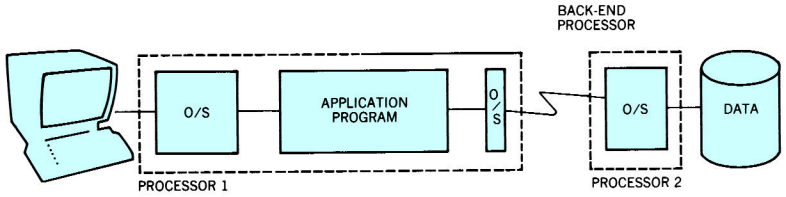
This section categorizes various approaches to distributing functions and data associated with application processing between two or more processors. The first part of the discussion is at a very general level, looking at an idealized application flow. Identified are the advantages and disadvantages of three generic ways to partition the processing across two separate systems. Then application structuring in the distributed environment is discussed in more detail, with a set of terminology established to describe each of the techniques for distribution.

Figure 5 shows an idealized high-level flow for an application. Messages flow from the terminal through elements of the operating system to the application. The application program, in turn, processes the incoming message, requests the necessary stored data records through other portions of the operating system, and then returns a response to the terminal.

As a way of discussing the various modes of splitting functions between systems, the following question is addressed. Where is the best place to split the function flow between two systems so as to optimize the price-performance ratio? It should be noted at this point that the primary factor for determining the split of the application workload across two or more nodes is often not hardware price-performance optimization. Considerations of complexity, management structure, reliability, backup, recovery, response time, etc. are often more important. Nevertheless, any system must have adequate performance, and the following discussion focuses on this factor.

Assume that two systems are connected via a communication line. Generally, in obtaining solutions to this problem, the following two conditions must hold: (1) the processors are both fully

Figure 6 Back-end processor configuration



utilized, and (2) communications between them are minimized. The added element in going from one to two systems is the additional communication overhead. Therefore, the delays and processing time associated with communication must be small in comparison to the time between communications. Thus the point chosen to break the flow must be one that separates it into relatively independent segments. In addition, there must be substantial amounts of processing done on both sides of the boundary for meaningful utilization of the capacities of both sides. These principles apply equally well to cases where more than two processors are participating.

The following are three partitioning points: (1) in the operating system between the terminal and the application; (2) in the operating system between the application and the data base; and (3) within the application itself. Partitioning within the operating system part of the flow, (1) and (2), has the general advantage that applications can be run unchanged in a distributed environment.

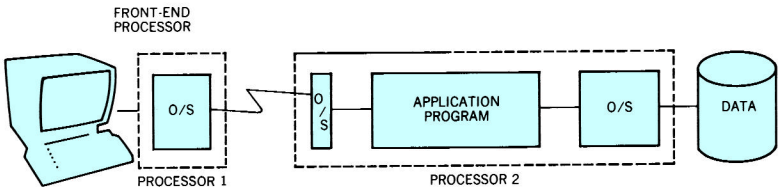
back-end processor

Splitting the application flow in the operating system portion between the application and the data creates a system whose function is almost exclusively that of managing access to the data bases. Such a system has been called a *back-end processor*. In this configuration, the content of the communications between the two processors is data access requests and data records. Figure 6 is a back-end processor view of the application flow.

Generally, the disadvantage of this approach is that there is usually a very high level of communication between the application and the data management portion of the operating system. This is due to the fact that there are typically many calls to the data base for each incoming message from the terminal. The actual number of data base calls per incoming message from the terminal could vary from a handful to several hundred. Thus, the level of communication between the two processors with this distribution approach is usually much higher than in the other two alternatives.

Another factor that tends to increase the level of communication across the intersystem link is that in transaction-oriented systems, where data base recovery is an essential capability, com-

Figure 7 Front-end processor configuration



munication must occur between the terminal side and the data side of the operating system. This additional communication is required to create an environment where recovery is possible. For instance, in the event of an application failure, the incoming message must be requeued and the partial data base changes must be *backed out*. To be able to accomplish such a restart, records must be created that contain both data base and communication status information.

Still another factor is that it is difficult to achieve a reasonable load balance between the two systems. In data base–data communication systems like IMS and CICS, the communcation and application parts of the processing often represent a comparatively small fraction of the total required capacity. Also, it is relatively difficult to separate the two sides of the operating system support. Nevertheless, this back-end processor approach to partitioning has appeal in cases where multiple nodes use the back-end processor for access to common data. In this way, contention for access between systems can be managed at a central point and the usual problems of preventing simultaneous access for update can be handled easily. The author is not aware of any practical situations where a back-end processor that is used exclusively for data access provides a price-performance advantage. Generally, the overhead of communications between the two systems dominates any advantage. As was pointed out, however, there are functional advantages.

front end processor

Splitting the flow in the operating system elements between the terminal and the application yields a partitioning that creates a *front-end processor*. In this case, the communications between the two systems take the form of incoming messages (or requests) and outgoing responses. This approach has the advantage that the communication required in this leg of the flow is relatively small compared to the back-end processor situation. Generally, the problem with this approach is to find enough work for the front-end processor so that the load on the processor running the application is significantly reduced from the single-processor case. Such a reduction occurs in the two special cases that are discussed below. Figure 7 is a front-end processor view of the application flow.

Consider first a front-end processor that is managing a significant network of terminals and, perhaps, routing messages to multiple systems that are running applications. In this situation, the processor load could become substantial. Other reasons for preferring this kind of arrangement include the ability to separate network management from systems that are running applications and the controlling data. The second case that provides for a significant front-end processor load is the formatting of data for display terminals and the handling of the interactions that are required to create input for an application program. Such an interaction could include the checking of input variables for the correct range and syntax, etc. There are cases where the work needed to create complex input messages is adequate to load a front-end processor and substantially reduce the amount of interprocessor communication by localizing the simple, high-frequency interactions with the user. In addition, the complexity level of this type of processing is well matched to a small, simple system.

application structure

Both the back-end and front-end processor approaches described thus far have the advantage that they do not disturb the structure of the application. Their disadvantage, of course, is that they do not provide a general solution to the problem of distributing a workload across a variety of nodes. What remains, then, is a need to consider the structure of the applications with respect to where they can be split so as to minimize communication and allow for the balancing of processing among several nodes in a network. Generally, it is necessary to identify points in the flow of the applications programming and their associated data where the level of communication is low. The placement of both data and programs in the two nodes must also be examined from this point of view.

The next sections of this paper describe two generic ways that applications can be split across two or more systems in a distributed environment. These two techniques can be used in combination in much the same way that series and parallel connections of electrical components can be used to implement any electrical circuit. The distinction to be drawn has to do with whether two systems relate as peers or in a master-slave combination. Peer distribution is termed *horizontal distribution*, and master-slave distribution is termed *vertical distribution*.

horizontal distribution

Horizontal distribution is typically characterized by the fact that each of the nodes performs essentially the same functions. Usually the differences between the functions are in the instances of data stored on each of the nodes and the particular set of terminals to which they are connected.

Communications between the systems take the form of program-to-program requests and responses in both directions. These

communications are minimized because of relationships that exist among the terminals connected to a particular system and the data stored there. As an example, consider two nodes used for retail credit authorization. Assume that each node is assigned to cover a particular geographical area. Terminals for each area are connected to their corresponding node, along with the stored data that reflect the account status of the card holders for that area. The level of communications between the two nodes would be a function of probability that a card holder from one geographical area is shopping in a retail establishment in the other node's geographical area.

The state of California is an example in which the geography is ideal for this type of split because the major population centers, San Francisco and Los Angeles, are separated by a large zone with relatively little population. Therefore, it would be relatively easy to balance the load between two nodes with the probability of communications between them being quite low. In contrast, in a single-center geographical region such as the one that is dominated by New York City, this approach would not be practical.

vertical distribution

In a vertical arrangement, the terminals or other sources of work are placed on one node, usually the smaller one, with another node being placed upstream. In such a configuration, data for the relatively simple and/or more common functions would be maintained on the nodes that are connected to the terminals. Further, the application would be designed so that the bulk of the transactions that originate in these terminals would be handled by the local system. Only in the event of a more complex request, or one that requires data stored on the upstream node, is a request sent to the remote node. Thus the form of the communications in a vertically distributed system is that of requests flowing in one direction and responses in the other.

An example of vertical distribution is an application wherein the inventory of a local warehouse is maintained in a machine at that site. If, during order entry, a particular item is not in stock locally, an inquiry might be sent automatically either to the machine that is servicing another warehouse or to a central location to determine whether the item is available elsewhere.

The discussion so far has focused on the form of the communication between two systems and their request-response relationship. That is, the forms of communications can be as follows:

- *Front-end processor:* Terminal input message–output message.
- *Back-end processor:* Data access request–data record.
- *Vertical and horizontal distribution:* Application-to-application communications.

The next sections describe additional choices regarding distribution techniques. These choices are primarily centered around the choices for the placement of data in the overall system, data protection, and the frequency of data updating.

Interactive versus batch communication

The particular form chosen for the communications necessary to synchronize data copies across two or more nodes is a function of the following two goals: (1) minimizing reprocessing at each of the nodes that contain data copies; and (2) minimizing the level of communication overhead. There are also two choices of communication mode: (1) communicate immediately as changes occur, or (2) communicate changes all together in a batch, periodically. The batch approach has the potential for the greatest efficiency, but it does not provide data currency. The choice must be made on the basis of the need to have access to up-to-the-second information.

Given that access to constantly current data is a requirement, there are two further alternatives. Multiple copies of the data can be kept in synchronization by immediate communication among nodes; or a single copy of the data, kept in one node, can be shared by all users. In the latter case, requests that require the use of these data flow to this node. The single-copy alternative can be more efficient than the multiple synchronized-copy approach if there is a high level of record updates relative to read-only access and if the access time to the data for remote applications is fast enough for acceptable response times.

Thus, the essential new parameter when considering data placement, update, and synchronization is the timing of the request-response communication. A way to characterize this timing is to consider whether a response is returned to the requester. In the case of immediate communication, the requesting program waits for the result or confirmation. In the batch case, there is no synchronization with the requesting program. In the following discussion, these cases are termed *interactive* and *batch*, respectively.

Summary of techniques

Table 1 summarizes the important combinations of parameters for the distributed processing configurations that have been described. The two parameters used describe the nature of communications to nodes in the network and are the following:

1. The form of communications used:
   - Batch (asynchronous)
   - Interactive (synchronous)
2. The request-response content of the communications:
   - Input messages–output messages
   - Data requests–data records
   - Application–defined messages (i.e., any communication not in the pure forms of the first two types)

Table 1 Summary of distributed processing configurations discussed in this paper

Form of communication	*Content of communication*	*Examples*
Batch (asynchronous)	Input messages	Remote Job Entry (input and output)
		Conversational Remote Job Entry
	Output messages	Transaction tape to be processed against data base
	Data requests	Changed-record tape to update data base copy
	Data records	Requests to send a copy of a data set
	Application defined	Partially preprocessed bulk input data from data entry application
		Summary statistics (daily, hourly) for remote operation forwarded to central system
Interactive (synchronous)	Input messages	Message routing (front-end processor to multiple destinations)
		Input message checking and assembly
	Output messages	Processing output variables from application programs using locally stored formats and fixed data
	Data requests	Application requests (GET/PUT) sent to remote node for data access
	Data records	Communications required to maintain synchronization of two identical data bases in two nodes
	Application defined	Multinode transactions, e.g., inventory searches in multiple locations; update of customer balance in one node, teller cash drawer balance in another drawer (check-cashing application)

Although the terminology of "front-end processor" and "back-end processor" implies a static assignment of function to particular processors, this is not necessarily the case. Two or more of the forms of communications can occur between any two processors in the system as different aspects of the application programs are activated. Therefore, the alternatives shown in Table 1 can be considered on a communication-by-communication basis rather than as a static choice. The major considerations that lead to preferring the assignment of a static role to a node in the network are the availability of data and operating system functions in that node, both of which are static choices in themselves.

Transaction-based data recovery

Another consideration regarding data placement is the ability of the node that contains the data to provide protection in the event of a failure. For the purposes of this discussion, an atomic unit of work called a *transaction* is used. This means that the data base is never allowed to reflect anything other than successfully completed transactions. Thus, if a transaction program fails during its execution, any data changes up to that point are removed. Further, no data changes are available to other programs until the successful completion of the transaction. To implement this concept, audit trails and checkpoint records must be recorded, and care must be used in synchronizing and sequencing all the events that occur at the end of a transaction. Although this definition is consistent with the notion of a transaction usually held by a user at a terminal, it should be noted that a transaction may require actions in several nodes, with the work in each node tied together by a complex protocol that gives the effect just defined.

In any node that has transaction-based data recovery, all incoming requests for processing, regardless of their form, must be considered as transactions. In situations wherein transactions are being executed in two or more nodes, the data bases in all the participating nodes simultaneously reflect the successful completion of the transaction. If there is a failure in any of the nodes during the process, changes in all nodes are removed. The provision of this multinode synchronization capability increases the level of communication overhead between participating nodes. In this paper, the term *transaction mode* is used to describe the type of data base recovery just discussed.

By considering whether the requester and the responder in a communication are operating in transaction environments, other important distinctions can be made. In the trivial case, neither the requester nor the responder is operating in a transaction mode. The second case is where the requester is not in a transaction mode, whereas the responder is. In this latter case, the request triggers the start of the transaction and the final response signals the end. Intermediate communication may occur. An example of such intermediate communication is when the requesting program is providing an interface to the terminal user and doing input verification and output formatting, i.e., functions that are not usually related to any data base. However, the node in transaction mode may have to participate in input verification when access to the data base is required or the input is inconsistent with the data base content.

The third case is one in which both the requester and responder are operating in a transaction mode. Here, it must be presumed that protected data are being simultaneously updated in two

nodes, and, if the communication between nodes is synchronous, a multinode transaction is being performed. That is, all data base changes in all nodes are synchronized, and an error in any node causes any change in all nodes to be removed.

In the final case, the requester is operating in a transaction mode and the responder is not. This combination is reasonable only when the communication is asynchronous and the requested function is not modifying protected data.

Application growth

This section describes some of the complications that are introduced when new applications are added to a distributed system. Generally, new applications create new uses for and new relationships among existing data. As a result, the system's overall design may have to be changed to remain viable. An example is presented to illustrate such a situation.

The example consists of a three-stage application growth scenario in which the third stage establishes a relationship between the first two. The first stage is a horizontally distributed system to support on-line bank tellers. The second stage involves the use of a separate system to do credit card authorization. The third stage is the connection of the preceding two systems to accomplish electronic funds transfer.

Figure 8 On-line bank teller application

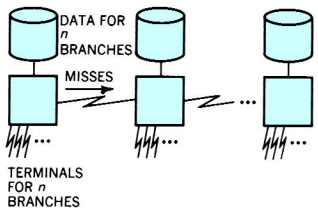


Figure 8 shows the configuration for the on-line bank teller application. The approach selected for this implementation is a horizontal distribution, where the terminals for branches are connected to a particular machine that holds the data associated with the accounts in those branches.

Generally, branch banking can be managed so that branches are roughly the same size and have roughly the same traffic. Therefore, all terminals can be made to have very similar traffic patterns. This homogeneity, plus the fact that people tend to bank where they have an account or in a branch that is nearby, allows the communication overhead to be low enough for the system to be viable. The probability that a transaction requires access to data in another node is a key parameter in the design. This probability is referred to later as the *miss ratio*. One of the techniques used to minimize the miss ratio is to place the data and terminals for a set of branches in adjacent geographical areas in the same machine.

The second application to be implemented is the authorization of credit. Assume that the same banking establishment requires a system with terminals in retail establishments to perform the

credit authorization for its bank credit card. This application has little homogeneity. There is a wide variance in traffic among terminals, and there is a skewed distribution of terminals on a geographical basis. Since these characteristics do not lend themselves to a distributed implementation, assume that this application is placed on a single node system.

The third stage of growth of the example on-line banking system is electronic funds transfer. When customers make purchases in a retail store using the second application system, the charges are reflected against the customer's checking account balance in the first application system. This new application obviously will place a significant additional load on the combined system.

Probably the most attractive solution to the problem of handling this additional load would be to add more nodes to the first application system and redistribute the terminals and data so that each node would handle fewer branches than before. In this way, additional capacity would be made available to handle the new application. Because each node has fewer branches, however, the miss ratio would increase, with a corresponding increase in communication overhead. This increased overhead would add to the load of the system, and still more capacity might be required. If more nodes were added to solve this problem, the resulting increase in communication overhead might totally offset the increased capacity. In this way, the situation might degenerate to the point where no number of nodes would be able to handle the required load.

Another simple solution to the problem would be to use machines that have greater processing capacity in each node. This alternative is sometimes not attractive because of the economics of replacing existing equipment.

If neither faster equipment nor the simple extension of the distribution techniques already used yields viable configurations, the only other approach would be to redesign the system. Generally, this would involve redistribution of data and programs between nodes, including the node for credit authorization along with the addition of equipment to augment capacity. In many application growth situations, this approach is the only one that can be used.

The lesson to be learned from this example is simply that growth of applications must be carefully planned and managed. New applications typically create new relationships among existing data and application programs. As a result, the balance of the system design, which is based upon having particular values for certain interaction probabilities, may be upset. Thus the system may have to be rebalanced. As was just discussed, this rebalancing could be done by increasing the number of nodes in a straight-

forward way, by increasing the power of each node, or by selectively redistributing data and/or particular application programs. In order to maintain maximum flexibility and keep every possible alternative open, it is desirable to design the applications initially from the point of view that redistribution may be necessary as a result of unforeseen and unforeseeable requirements placed on this system by the implementation of new applications.

None of these problems is new. They have all been present in applications in centralized processing environments. Only the emphasis is different. The overall goal is to obtain the benefits of distributed processing while preserving the management control and flexibility of the centralized approach. In particular, the following are needed:

- Centralized design of the system data base and control over its content, level of usage by particular elements of the application programming, synchronization, recovery, and distribution.
- The structuring of the application program itself into distributable pieces and the definition of the unit of distribution. The transaction, as defined in this paper, can be used as an example of such a unit. Other definitions could be used that would more precisely fit the needs of the application itself.
- Once a distributed application is up and running, it is necessary that the level of the programs, the data bases, and the operating systems themselves be centrally controlled. This usually requires a central library control and distribution package that is operating in a central node of the system.

Neither are the tools needed to implement this centralized control new. Such tools include a data description directory that could be used to record the content of data records and where-used information, plus overall performance measurement facilities. The measurements could be used to relate transaction frequency, data usage, and network utilization. This information is essential in the process of identifying performance bottlenecks and predicting effects of new applications and/or redistribution of data and programs.

Most importantly, perhaps, there has to be a level of compatibility between the nodes of the system to allow for both data and program redistribution to support growth. Compatibility must exist, even between dissimilar nodes, for the following items: source programs, communication protocols, transaction definitions, portable media, and data structures.

Conclusions

This paper has discussed some theoretical aspects of systems design as well as the pragmatic motivations for seeking distributed

processing solutions to application implementations. Theoretical considerations do not usually dictate where or how to distribute. They can only assist in maximizing the efficiency of decisions based upon such criteria as system reliability and availability, data security, organization needs, and response time. If, for example, a machine is being placed near a manufacturing line to provide insulation from central system failures in the area of work scheduling, there are very few degrees of freedom with respect to the placement of certain data and where certain programs must execute.

The overall importance of the factors that determine the form of processing (distributed, centralized, or decentralized) to be selected begin with such motivations as have been described here. Considerations proceed to the structural requirements of the application data and programs, and continue to the various choices described in the techniques sections of this paper. Distributed processing offers an unprecedented level of flexibility in the design of application systems. Because flexibility is inevitably a two-edged sword, however, it is more necessary than ever before to proceed with understanding and with deliberate, manageable plans. It is for this reason more than any other that the cornerstone of any effective distributed processing system must be the ability to implement a high degree of centralized control.

ACKNOWLEDGMENTS

The author gratefully acknowledges the contributions to this paper made by his colleagues in the IBM Corporation, especially Harold Lorin of the IBM Systems Research Institute and David C. Larkin. Robert L. Patrick, an independent consultant, also made many helpful suggestions.

GENERAL REFERENCES

H. Lorin, "Notes for a course in distributed processing," obtainable from the author at the IBM Systems Research Institute, 219 East 42nd Street, New York, NY 10017.

G. M. Booth, "Distributed information systems," *1976 National Computer Conference, AFIPS Conference Proceedings* **45**, 789–794 (1976).

S. E. Scrupski, "Distributed processing grows as its hardware and software develop," *Electronics* **49**, No. 11, 91–97 (May 27, 1976).

P. A. Alsberg, "Data distribution strategies," *Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks* (Lawrence Berkeley Laboratory, University of California, Berkeley, CA 94720), May 1976.

T. Riddle and D. Zatyko, "Dough on the go: Inside an EFT system," *Computer Decisions* **8**, 26–33 (March 1976).

G. S. Champine, "Six approaches to distributed data bases," *Datamation* **23**, No. 5, 69–72 (May 1977).

Allan L. Scherr

System Communications Division, Kingston, New York

As a member of the MIT Project MAC, Dr. Scherr did research on time-sharing system performance and user characteristics. After joining IBM in 1965, he participated in the definition of the System/370 architecture, and in 1968 he became the design manager for the TSO Project. In 1971, Dr. Scherr was given overall project responsibility for the first MVS release (OS/VS2 Release 2), and he subsequently became the systems manager for both OS/360 and OS/VS2. In 1977, he was assigned to manage the design of a new programming system in support of distributed processing, and he is now the manager of distributed systems programming. Dr. Scherr received the Ph.D. degree in electrical engineering and computer science from the Massachusetts Institute of Technology in 1965.