Design considerations for distributed applications

by J. J. Rofrano, Jr.

Probably the hardest part about developing a distributed application is determining where to start. There are multiple hardware and software platforms to understand, network traffic implications, and numerous tools and technologies to consider. One question, however, transcends the importance of what platform to pick or what tool to use: that is, how do you design it? This paper represents the results of two years of work with customers regarding this question. The paper explores some of the implications of working in a distributed environment, reviews some rules for data and function placement, and introduces a methodology for distributed application design.

Two popular terms, *cooperative processing* and *client/server,* are used to describe distributed applications. These two terms, however, often mean various things to different people. In the context of Systems Application Architecture* (SAA*), an SAA application is one that runs on an SAA platform, conforms to Common User Access* (CUA*), uses the Common Programming Interface (CPI), and is designed for cooperative processing.[1] Most people understand hardware and software platforms and what a common programming interface is. However, the expression "designed for cooperative processing" brings to mind a broad definition of an application that can be split between a programmable workstation (PWS) and a host. Likewise, the term client/server has been often broadly defined as an application whose "client" part makes requests from a "server" part that resides somewhere in a local area network (LAN) environment.

A distributed application requires more logic than just the communications between a workstation and a back-end server or host. Different programming paradigms must be blended and interfaced. One paradigm views the PWS as an extension of the host world; the other views the host as an extension of the PWS world. We examine the two approaches in greater detail but, regardless of one's perspective, the graphical user interface found on the present programmable workstations has an impact on the design of the application.

The mix of skills needed to implement distributed applications may have an impact on the organization of current software development departments. Most end-user's jobs were probably defined by existing computer applications, whereas new applications should expand the existing scope. In fact, if the application does not change the work habits of people, then the technology is not being exploited. Many things are to be considered in designing the physical connection between the portions of the distributed application, such as scalability, configurability, and performance.

This paper reviews some of the issues involved in understanding the distributed processing environment and designing applications to take ad-

vantage of it. Allan L. Scherr introduced a matrix approach to determining where to place application function and data in a network.[2] This paper is based on his work and introduces a tool called the object/action matrix, which is built on Scherr's initial matrix design. We expand on the rules for data and function placement, and review a methodology for distributed application design using the object/action matrix.

SAA is the example environment, but since this methodology is independent of the application development tools that are used, it will apply to the Advanced Interactive Executive* (AIX*) and other environments.

Distributed application models

Since there has been widespread usage of the terms cooperative processing, distributed processing, and client/server computing, we first define the terms as they are used throughout this paper.

Distributed processing is the distribution of function or resources across two or more interconnected processors. These processors can be any combination of mainframe, midrange, or programmable workstation. The distribution may be transparent or overt. Distributed processing is a generic term that includes cooperative processing and client/server computing.

Cooperative processing is a term that mainframe users created to talk about an application whose functions are divided between a mainframe processor and a programmable workstation (PWS). It is a "host-centric" view of the world in which the PWS adds value to the mainframe by providing a better human interface and perhaps some additional processing. Cooperative processing is a form of distributed processing.

Client/server computing is a term that personal computer (PC) users created to talk about an application whose functions are divided between a programmable workstation and a LAN server. It is a "workstation-centric" view of the world in which the LAN server adds value to the workstation by carrying out work on its behalf. Some users extend the definition to include *enterprise servers*, which are traditional mainframes and minicomputers that have now taken on a new role. The network operating system provides several transparent services, making this environment very attractive to programmers and end users alike.

From a software design perspective, the *client* is the one making the request, while the *server* is the one who does the actual work to satisfy the request. Most people think of client/server in hardware terms, with the PWS always being the client. One look at X Windows** confirms that this view is flawed. The server is the place the work actually gets done. For a database request, the server is the machine that applies the request to the actual data. For a display request in an X Windows environment, the server is the PWS that executes the display request on behalf of the host process that made it.

For both of these forms of distributed processing, the work is shared between two or more processors. This may be the result of a conscious decision by the application developer to execute function on a particular platform, or it may be a transparent system service provided by the underlying network or operating system. The objective is to extend the domains of applications across local area networks (LANs) and wide area networks (WANs), and expand the role of both the traditional host and the programmable workstation by exploiting the unique capabilities of each.

The relationship between distributed parts of an application can be either peer-to-peer, call/return, or event-driven. The three communication models that implement these relationships are: the conversational model, the remote procedure call (RPC) model, and the message and queueing model.

The *conversational* model is one model for distributed processing where the two application halves agree on who has the right to send and who will receive data based on established protocols (peer-to-peer). The initiating application usually starts with the right to send data. When the initiating application has completed sending and agrees to receive data, the roles are reversed. This role reversal continues until processing is complete and the conversation is terminated. This model is implemented on some systems through the Advanced Program-to-Program Communications (APPC) interface.

Remote procedure call is a call/return model where application functions interact in a requester/server relationship. The requesting program makes a request of the server program to provide some service. The server program then carries out the task and completes the process, usually by returning some results. Since services can be local or remote, this model introduces an element of transparency in which the application can be unaware of where the actual service is performed. This model is implemented by the Open Software Foundation (OSF) Distributed Computing Environment (DCE) remote procedure call application programming interface (API), often referred to as DCE/RPC. The Transmission Control Protocol/Internet Protocol (TCP/IP) also has an RPC mechanism.

Message and queueing (MQ) is an event-driven model for writing distributed applications. The communication between functions is performed by placing a message event on a queue, which is routed to the called function's queue. The message is then taken off the queue and processed. It is, by default, an asynchronous model but if a result is needed, the function called can return a message, thus simulating a synchronous call. E-Mail is one example of message and queueing. Because of this, some people refer to messaging as "datagrams."

Messaging uses a connectionless paradigm. There is no knowledge of the underlying transport or topology. The message is sent to the first queue along with the destination address and all interqueue routing is transparent to the calling application. In fact, the function being placed on the queue may not even be running at the time the message is sent. This provides for off-line or batch processing of messages.

Messaging is widely used in computer-integrated manufacturing (CIM), where it is appropriate for alerts and the asynchronous starting and stopping of jobs and machinery. The added advantage of being connectionless allows the shop floor to be reconfigured during run time. The application is unaware of the change. The IBM Distributed Automation Edition (DAE) is one implementation of this model and is generic enough to be applicable for application developers outside of CIM.

Two more terms may require clarification: programmable workstation and host.

Programmable workstation (PWS) refers to a Personal System/2*- (PS/2*-) class workstation running Operating System/2* (OS/2*), AIX, or PC-DOS. We assume that the PWS has a graphical user interface such as Presentation Manager*, AIXwindows*, or Microsoft Windows**. The graphical user interface assists in the distributed goal of exploiting each processor for what it does best. A traditional time-sharing system simply cannot afford to dedicate the resources necessary to manipulate the end-user interface like a PWS. The PWS gives all applications an easy-to-use common "look and feel" independent of the back-end processor.

The PWS can physically be connected to other computers via a local area network, or be attached to a mini- or mainframe via a wide area network.

In this paper, the back-end processor is referred to as a *host*. What makes it a host is its ability to execute function on an application's behalf and to share resources with other workstations. This paper considers mainframes, minicomputers, and LAN servers all to be hosts when they fit the above description.

Whether trying to exploit the capabilities of the PWS from host-based applications or trying to access host data and resources from PWS-based applications, the objective is the same: to integrate the role of the PWS into the *enterprise information system.*

Most applications will operate in multiple modes. They do this by sometimes becoming a client to the OS/2 LAN services to perform a function, while at other times they may need to use a conversational model like APPC to get at back-end functions on the server. Even servers may become clients of other servers in the process of servicing a request.

All distribution is performed by some form of program-to-program communications. The question is, does the developer write that communications function into the application logic, or provide some means of making it transparent to the application by either utilizing network/system services or by writing the developer's own requestor/server interface?

The distributed view

The role of the PWS has often been referred to as the window to the enterprise. All applications will be transparently accessed from a common workstation platform whether they are local, remote, or distributed. The enterprise network will become less apparent to the end user. Distributed processing expands the role of the PWS to that of an active participant in the enterprise. Rather than just being the window into the user's applications, it becomes active in the execution of the user's applications as well (see Figure 1). The end user sees a seamless application with transparent access to function and data.

Advantages of distributed processing. One of the primary reasons for implementing distributed processing is to utilize each processor for its unique capabilities. The resulting application should provide a better result than could have been achieved with either the PWS or mainframe technology alone.[3] We are already familiar with mainframe capabilities. A better understanding of PWS capabilities is needed. Some mainframe end users may already be familiar with PWS capabilities, but does the "glass house" programming department understand it? The user interface of a distributed application needs to remain highly interactive while the back-end host or server is also busy processing data. The user does not necessarily have to wait for host processing.

Multitasking. Some tasks such as a credit check or customer number search can be done in the background while the user continues to interact with the workstation in the foreground. Other functions such as accessing insurance rate tables and other "table lookup" functions can be performed without ever going up to the enterprise host by storing the data on a work group LAN server or even at the workstation. This workstation processing can also be done in the background because of the multitasking capabilities of most workstation operating systems. Programs should not be written with the assumption that the end user is waiting for a transaction to complete. Users could very well be working on the next transaction or even working with another part of the application during wait times.

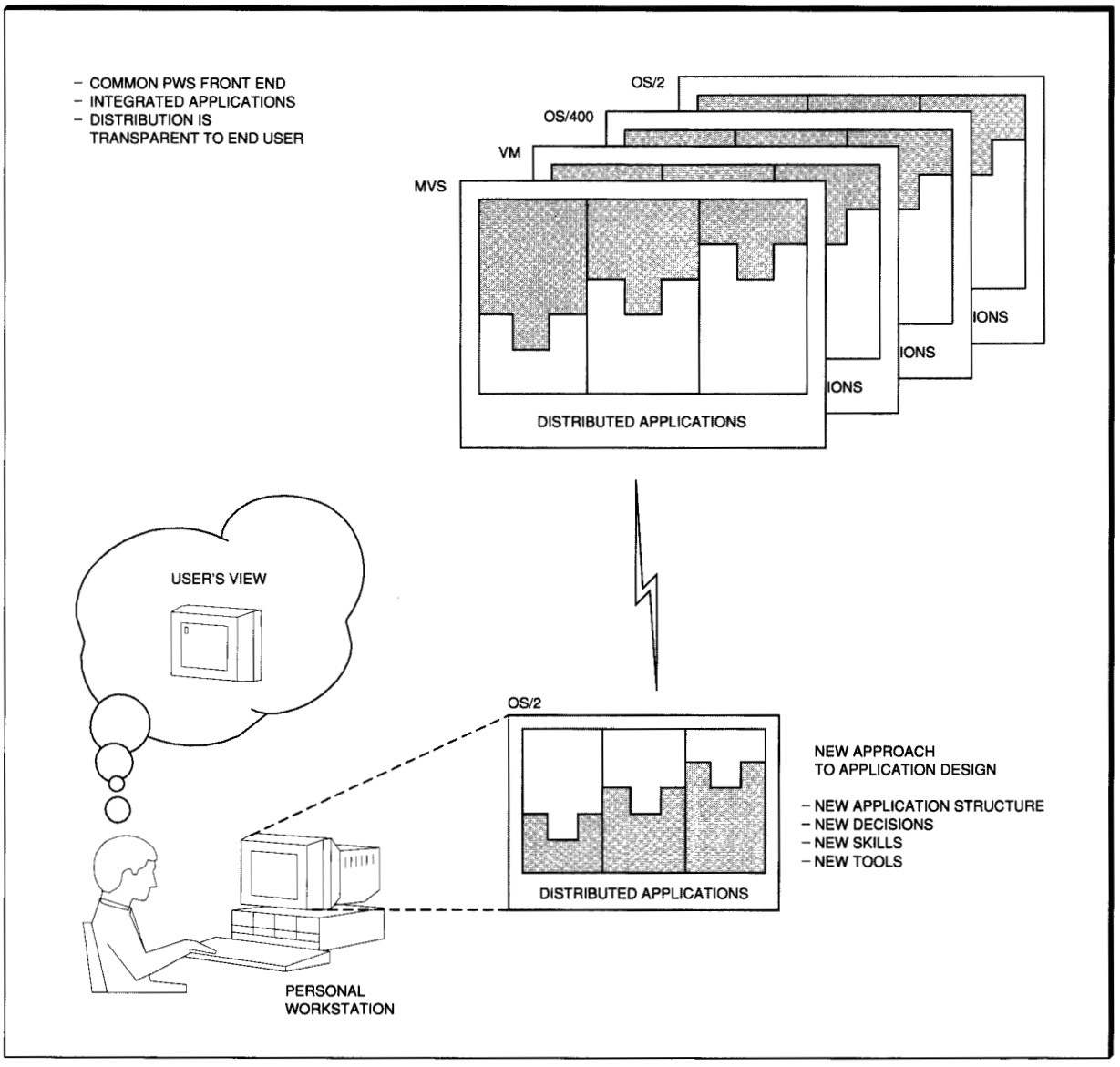
Design changes. In order to design an application with simultaneously highly interactive front and back ends, we need to change our current monolithic, hierarchical, "panel-oriented" way of designing applications and move to an overall interface design that is modular and event-driven.[4] Event-driven applications place the end user in control of application flow, not the programmer. They also give programmers the freedom they need to place function on various platforms while maintaining the user interface code at the PWS. We discuss these two application designs in more detail in the next section.

The user interface of a distributed application needs to remain highly interactive.

Productivity. Increased productivity and ease-of-use of the programmable workstation are some of the advantages distributed processing offers. Some people find it easier to interact with a graphical interface than with the traditional character-based interfaces on host systems. An advantage in some environments is the ability to invoke multiple copies of an application and run them concurrently. Direct manipulation of screen objects will be an advantage in others. For example, in a data entry application, the re-keying of information can be virtually eliminated, along with the errors re-keying introduces, by using direct manipulation at the field level to move data easily from one application to another. Greater input integrity may be a by-product of the graphical user interface on the PWS.

Work-load shift. Because we have programmable intelligence at the workstation, an advantage of distributed processing might be to off-load host processing. At the same time, the increased intelligence of the PWS can now process transactions faster than was humanly possible on a non-programmable terminal (NPT). The result may be that the host work load actually increases by exploiting the PWS multitasking environment. This will depend on application design and whether transactions are driven synchronously or asyn-

Figure 1 The distributed view



chronously. There may be a work-load shift, but not necessarily in the intended direction.

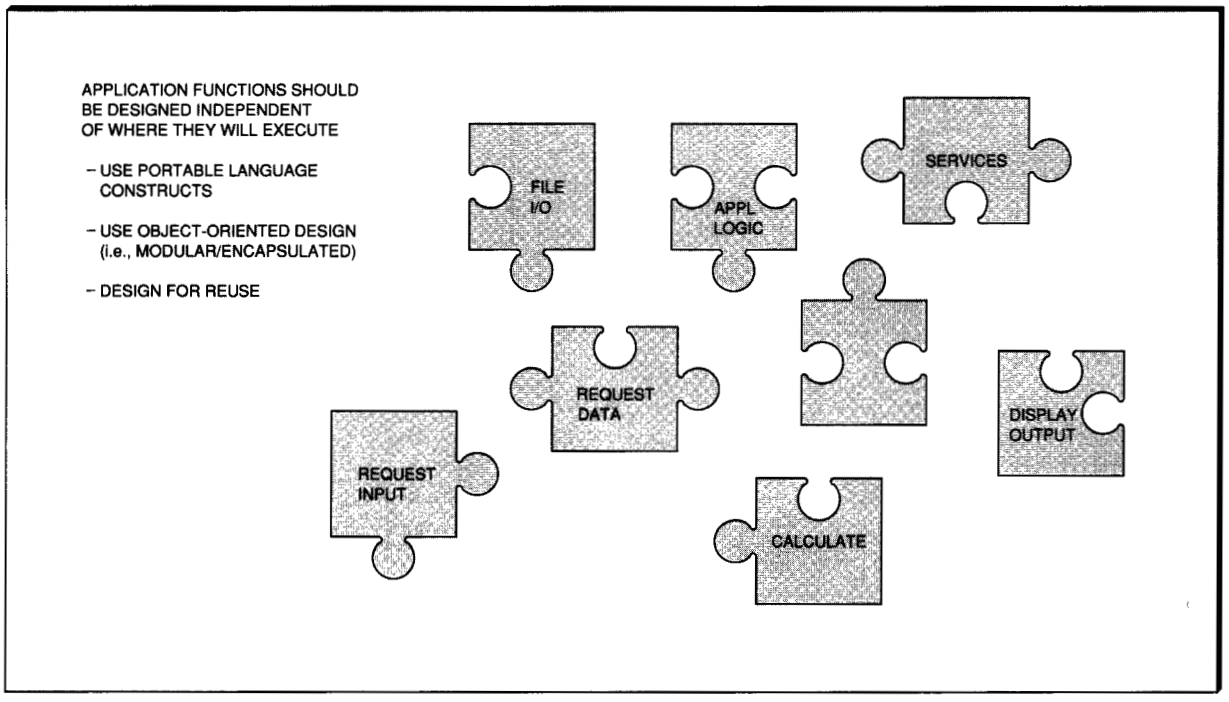
Greater connectivity options. The PWS will also provide the ability, in some instances, to continue processing if a host link fails or to have the option of working remotely without a host link and batch upload to the host later. The possibilities are limited only by our understanding of the capabilities of each platform.

The addition of a consistent graphical user interface for applications seems most powerful of all the extended possibilities in distributed processing. The increased user productivity will affect all applications.

Distributed application design

While distributed processing introduces transparent application execution to the end user, the ap-

Figure 2 Application anatomy



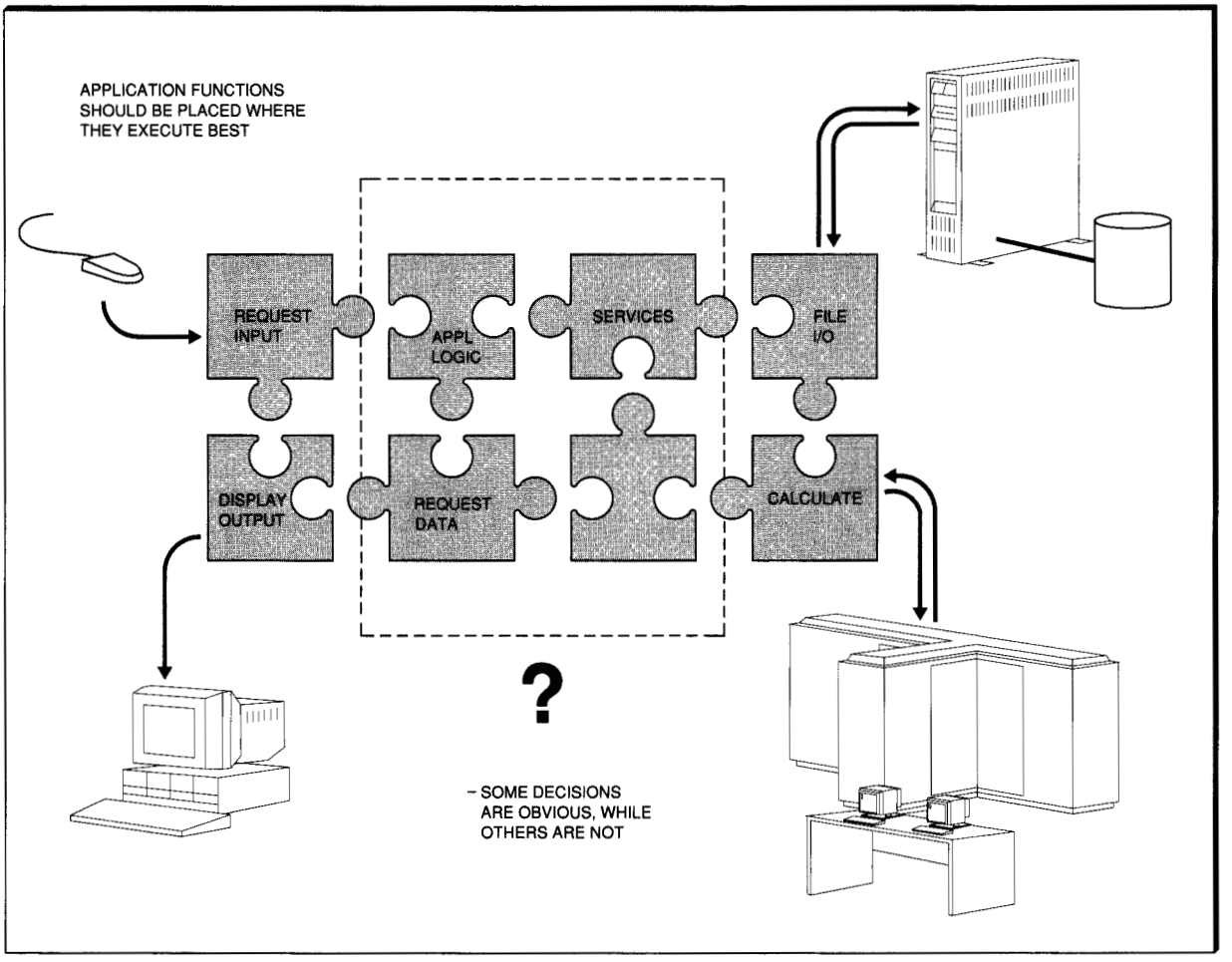
plication programmer has some conscious decisions to make, as outlined in the lower righthand corner of Figure 1. A new approach to application design is required, along with a new application structure to support placement of function and data across multiple platforms. In the past, the programmer never worried about where to place the data or end-user interface or where the code would run—it all ran in the same place. This changes in a distributed environment. The developer must acquire skills to know when and where to split application function and must decide on what platforms to run them. Tools are necessary to help make and execute these design decisions.

Ideally, the functions should be placed on the platform where they perform best or where sharing is desired. The benefits are in reuse of code by multiple applications, as well as good performance by executing code on the platform it runs best. "Best" is whatever is best for the enterprise; in some instances this means best performance, in others it means the cheapest platform. Whatever the goals are, function should be placed where it will best meet these goals. This topic is discussed in greater detail when we explore function placement.

Having function execute on multiple platforms raises the problem of how to determine the split point. The easiest solution is not to think in terms of split points, but to design the application from the start with separate functions that will execute on an unknown platform (see Figure 2). The platform will probably be determined by what the function does or what object it must manipulate. (For example, the manipulated functions include the data functions on the same platform as the data, the compute-intensive calculation function on the processor with a floating-point accelerator, or the end-user interface function on the workstation.)

The application becomes a collection of self-contained functions that, when executed in a particular sequence, perform a business process. These same functions executed in another fashion may perform a completely different process. The ability to reuse functions in multiple applications is very advantageous in today's environment of

Figure 3 Assembling the pieces



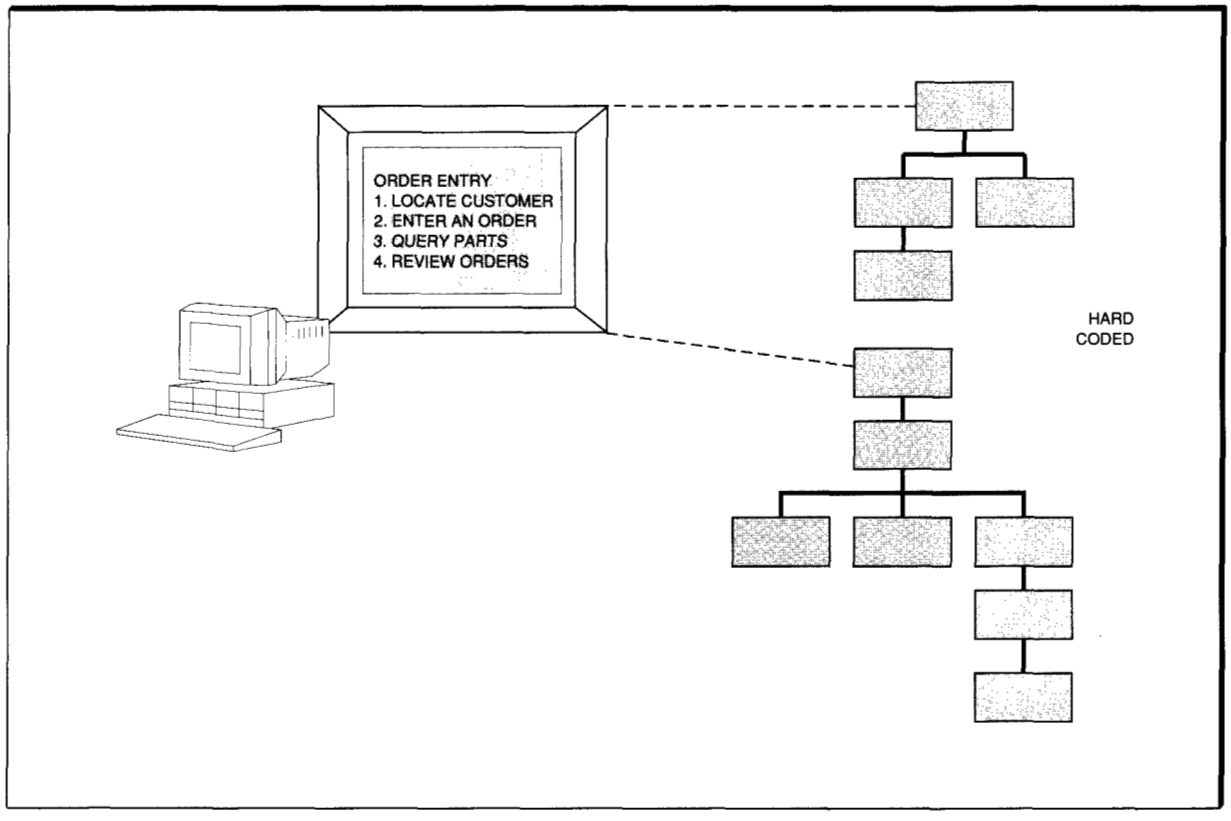
growing application backlogs. Using object-oriented design is a good way to achieve this goal. These functions could even be available as end-user computing tools. Think about the power of end-user tools like spreadsheets hooking into company functions, like calculating commission rates using the latest rate tables.

Platform portability can be attained by using common programming interfaces for both languages and services. This makes both of them portable as the objects they manipulate move. There is a requirement to have a common way to call these functions in all environments, directory services, and routers to keep track of where data and function objects reside. The absence of these enablers on a particular platform should not stop the programmer from writing the modular, portable code that will take advantage of these common services as they become available.

Applications then, are broken down into a series of requests to be carried out on some platform. These requests should be made through an interface that is common to all platforms and transparent to the application programmer.

At some point the programmer will want to assemble these functions (or code modules) into an application (see Figure 3). When it is assembled, there will be some placement decisions that will be obvious, while other placement decisions will

Figure 4 Panel-driven structure

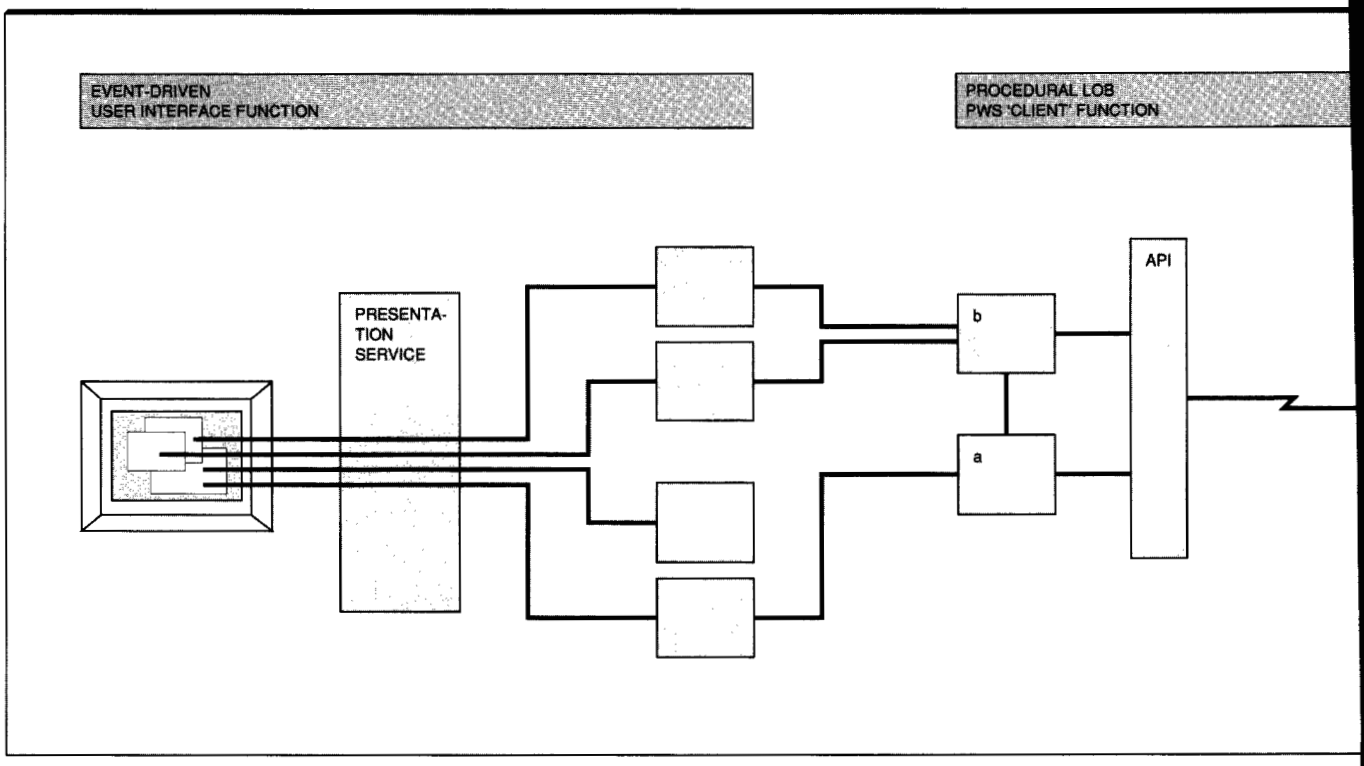


not. In general, we want to place those parts of the application that deal with the end user on the programmable workstation. It is also desirable to place those parts of the application that require host processing or database sharing on a host system where they can run most efficiently. The LAN environment is also a "host." If the database sharing needs can be satisfied on a LAN server, then that becomes the host to the particular application. Two models for application design are discussed next, one of which might be better suited for distributed applications.

Hierarchical application structure. Procedure-driven or "panel-oriented" applications are hierarchical by design. They interact with an end user on a screen-by-screen basis, usually in a hierarchy of menus (Figure 4). Each procedure within the hierarchy that needs input from the user, actively seeks it on its own behalf. The application displays a selection menu that may call a procedure that displays its submenu, which calls another procedure that displays the data entry screen, and so on. To select a different menu item, the user must back out of the chain of screens and start down a new path. This is referred to as modal operation because the interaction proceeds in one mode at a time, i.e., the user may be using the "data entry mode" or "report mode" or "electronic mail mode." To look at a report as a result of some mail that was received, the user has to back out of mail mode and go into report mode. One could argue that a jump key or fast path can change the status in the menu tree, but it must be preplanned by the programmer.

The problem with this design is that user input is solicited by each menu procedure that is called. Although structured programming and top-down design procedures have been used, no central point is specified to gather user input. It is very

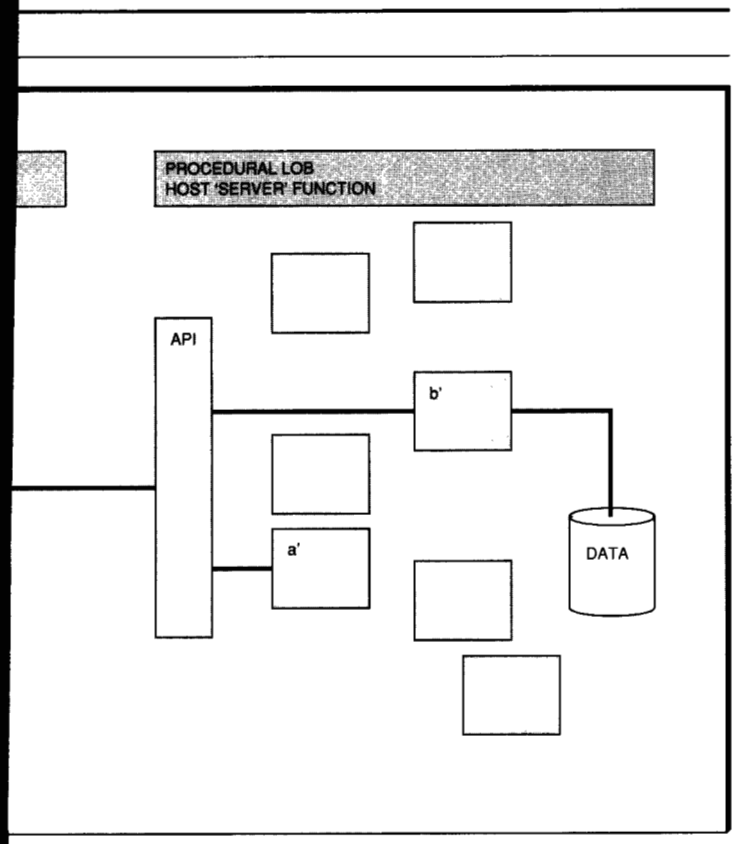
Figure 5 Event-driven application structure



difficult to write a distributed application with the user interface on the PWS and compute-intensive data processing on a back-end host or server using this monolithic design. It is also almost impossible to take an existing panel-oriented application and change it to be distributed. There are too many routines that want to get at both the end user and the data from the same block of code. How can these functions be placed on a back-end host when they need to prompt for input at the PWS?

The answer is to structure the program so that application functions have the data passed to them by a "front-end" procedure. This procedure collects the data from the user on the function's behalf. The functions then become reactive to input instead of actively seeking it. But how is front-end processing accomplished? Obviously a new structure is needed for distributed applications. A solution is to write event-driven applications.

Event-driven application structure. The application structure for the event-driven model is quite different, as seen in Figure 5. This model breaks the modal barriers of the hierarchical design by exploiting the graphical user interface and presenting the user with simultaneous choices that can be accessed in a modeless fashion. Multiple options are available simultaneously and are represented as windows or icons on the screen. Rather than a function actively seeking input from the user as in the hierarchical model, user input is solicited by a front-end code via the presentation services, and a back-end function reacts to these user events. This not only enhances usability by offering the end user the freedom to work more naturally in a modeless fashion, but it also takes the burden of program navigation off of the application programmer. Each function in the code is atomic and simply performs when called upon. This is called an event-driven application structure.[4] There is nothing really new about writing event-driven applications; programmers have



written them to monitor events on manufacturing floors for years. The new aspect is using them in line-of-business applications to monitor the end user!

The application code is structured in three sections (Figure 5): code that handles the event-driven graphical user interface, code that performs line-of-business (LOB) functions on the PWS, and code that performs line-of-business functions on the host. The PWS-to-host communication will probably be somewhere within the two line-of-business code sections, but we could certainly have an interactive interface that communicates directly with host code. Ideally, this communication should be as transparent to the application as making a local procedure call.

As this model is explored further, it is shown that only the end-user interface code really needs to be event-driven. The line-of-business code on the workstation and on the host can be modal for the most part. In fact, this structure lends itself quite easily to having icons represented on the desktop that actually initiate transactions running on a remote host. With the exception of the end-user interface code, host programmers will be writing pretty much the same style code in the same languages as they do today. They just will not need to solicit input.

This means it will not matter how back-end transactions get the user input they need to process. They also no longer need to perform simple range checking or input validation. All this can be done once in the end-user interface code and will no longer need to be part of each and every transaction. This will yield a savings in host programmer time and contribute to reuse of code at the workstation. Because of this structure, the same transaction can process in both real time and batch mode. One of the added values in designing an application using this technique occurs when a communication link goes down. The system allows the user to keep working on the front end and do a batch and upload when the link comes back up. The host transactions will not know that there is no end user at the other end of the line, because they now react only to input.

Each function should be implemented as a "black box." Functions are only required to know what another function does, not necessarily how it does it. This is the object-oriented programming concept known as *data encapsulation*. A function that needs access to data should do so via a request to a function that manages all requests for that data. The controlling function would be passed all the parameters needed for data extract or update, perform the function needed, and then return the results. In this way one could change the physical data access method and the requesting function would not be affected, because it is unaware of how the data are physically accessed (the requesting function has only a logical view).

Alternate views. The capabilities of the programmable workstation change the manner in which one views the computer systems. The move is from a view of applications looking out from the mainframe to drive dumb terminals, to an alternate view of the intelligent workstations looking back at and driving a host. This view is opposite that of most host programmers. The basis for client/server computing is to expand the role to the

workstation to include other services in the enterprise. Now the workstation is master and requests services from an upstream server.

It is very important to understand how an application developer thinks. Does the developer view him or herself as a host programmer or a PWS programmer? If the developer asks, "What functions should I move down to the workstation?" then the developer is probably a host programmer. Workstation programmers are looking at what function to distribute up to the host. How the developer thinks will no doubt influence the developer's decisions on where to place function and data. It is well to be aware that there is an alternate view of the world, in which the host computer is sometimes not the obvious place to run the bulk of the application.

There are three approaches to communications between distributed functions.

Distributed design considerations. There are three approaches to communications between distributed functions. One uses multiple conversations based on a program-to-program communications protocol such as APPC. This requires both functions to be executing at the same time in a "synchronous" fashion. Another uses a messaging model, such as a datagram, which requests a service. This service may or may not be running at the same time the request is made. When the service processes the request, it sends back a reply, if any, as another message to the original requestor. This requires a queuing mechanism that can store messages and ensure their delivery. The third is to design a requestor/server interface to act as a remote procedure call (RPC) on behalf of the application. This design allows calls for a service similar to local function calls within the application. It looks like a local procedure call to the application, and the distribution is transparent.

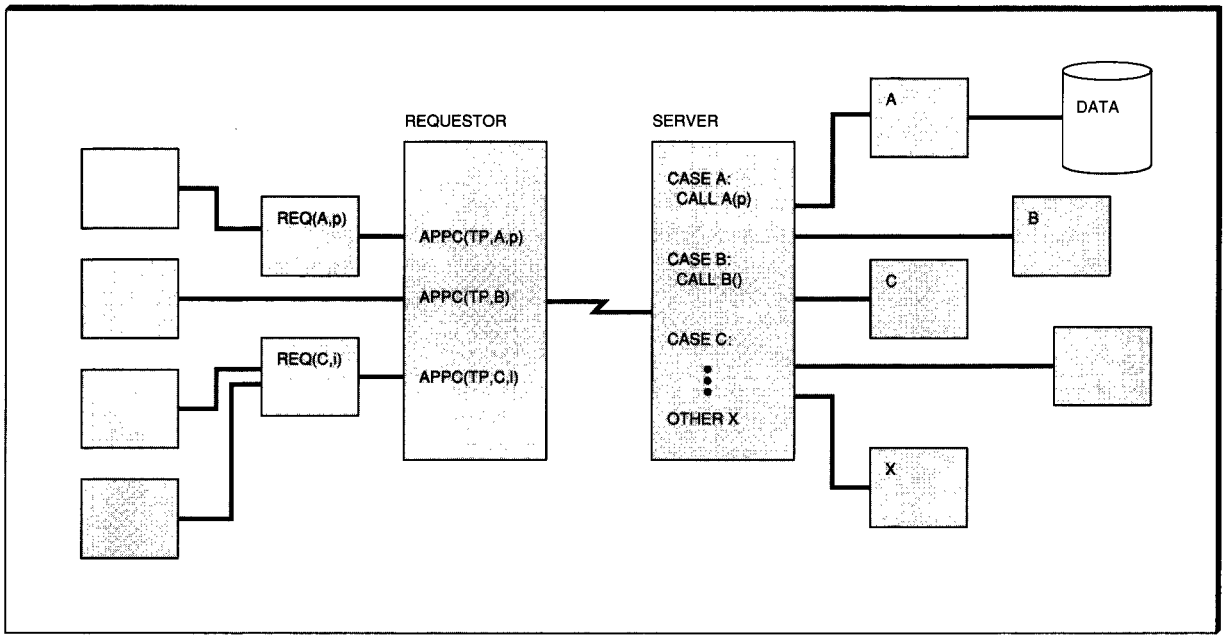
A well-architected interface that allows programs to make requests for services and data without being involved in how the service functions is very desirable. These servers may become requestors (or clients) to other servers in the process of servicing the original request. Servers should not care if the requestor is an end-user program or another server making a request on behalf of its client. This style of application interface is the easiest approach for most programmers to use.

If this type of interface does not exist on a development platform, it can be built from what is available today: RPC on AIX, APPC in SAA, NetBIOS (local area network basic input/output system) on a LAN, or TCP/IP, which is available on all SAA and AIX environments. If the API does exist (remote data and print services on a LAN), then it should be used. The objective is not to reinvent the wheel, but simply to place a layer of code between the application functions and the underlying communication transport so that the transport can change across environments and the application is not affected.

Redirection. Environments that employ redirection as a distribution means are the most transparent. An example is a virtual file on a LAN. Redirection allows the application to access data as if the data were local when they are actually on a remote server. If the network can absorb the traffic of multiple single I/Os, then the system will perform, but this approach does not allow for upward scalability. What if this same application were run outside of the LAN environment where there are no redirection services or where not enough bandwidth exists to use those services? One must be able to take an application that processes 100 transactions per minute today and use it in 10 000 transactions-per-minute environments tomorrow, without rewriting the application because of redirection at rates that can no longer be handled by the underlying topology. The designer must be sensitive to bandwidth considerations when using redirection as a distribution mechanism. A more robust mechanism is to use some form of distributed data service.

Requestor/server. We now examine the building of a requestor/server interface in more detail. From the software design shown in Figure 6 one can see a server program on the back-end processor, which sits and waits for commands from

Figure 6 Example requestor/server interface



the requestor program on the PWS. This particular example uses APPC as its communications protocol. The user should architect personalized commands to perform the functions needed and pass the parameters along with the command. The advantage is that programmers only have to learn the command set that is implemented, not the underlying APPC. The APPC could be changed to TCP/IP or RPC and not affect the calling application functions.

In the example, a request is being made for function *A* to be performed passing a pointer *P* as a parameter. The requestor code would then take the data pointed to by *P* and ship it via APPC to the server function. The server code is just a router switch that calls the actual function on the host, passing it the parameters shipped to it by the requestor function. When the function is complete, the server code will return the results to the requestor code via APPC, which would then place it back into a data structure so that the application code could extract it via a pointer. This is how the Server Requestor Programming Interface (SRPI) works on PC-DOS and OS/2 via LU 2 communications instead of APPC.

The result is a callable interface that keeps the communications transparent to the user program. A programmer writes the APPC code once, and everyone else uses that interface. In environments where there exists a remote procedure call (RPC) interface, such as with TCP/IP, this layer of code may or may not be necessary, depending on whether the programmer may want to move the application to other environments that may not have TCP/IP available. If so, RPC would be substituted for APPC and the original interface syntax left the same.

Conversational. If the requestor/server approach is not appropriate for the given application, then one may want to implement a simplified verb set interface to APPC for programmers to code to (i.e., START, STOP, SEND, RECEIVE, ERROR_CHECK, TOGGLE_STATE). This way only a few programmers have to write the error handling code in APPC and everyone else uses a higher level API to access them. Finally, if this does not yield enough function for the program the programmers will have to code at the native APPC level. This is not as desirable because it makes the application "platform-specific" and it will not move easily to other environments.

Rules for data placement

A common question that developers ask is, "How do I determine data placement?" The use of distributed data promises access to local data regardless of where the data are placed in the network, but what are the performance implications of accessing data remotely? The following are some simple rules-of-thumb for data placement.

Data placement is a function of the following parameters:

- Level of sharing
- Update/access frequency (timeliness)
- Security
- Capacity
- Business needs

If other contributing factors for data placement exist in the enterprise, then these factors should be added to this list as considerations.

Optimally, data should not be moved. If the platform in use has distributed data services, then this may be the easiest way to design the application. It may not, however, yield the most robust design or even perform well, unless there is network bandwidth to support the data traffic. Networks with low latency such as LANs are much more forgiving than WANs.

Given the lack of bandwidth to support distributed data for a specific application function, for performance reasons the data should be placed as close to the user as possible. On the other hand, the data may be required to be in a central place for sharing reasons. The key is to find a solution that satisfies both requirements.

Level of sharing. The first step is to get the data placed at the lowest level that meets the application's sharing requirements. If the data are private, keep the data at the workstation. Data that are shared at the department or work group level should be on departmental or work group systems, usually on either a LAN or midrange system. Data that are shared throughout the corporation should be on a central host processor. This sounds very simple in theory, but when an examination is made of the existing applications, it is easy to think that most data are corporate data. That does not help in this equation. The first observation then, is to change the way we think about data sharing.

Now, consider the following example: An order entry application running on one centralized processor probably views an order in process (one being entered into the system) as data that are shared by the corporation. After all, someone in another department is probably taking that order and filling it somewhere else in the enterprise. Based on this fact, the data may be thought to be shared at the corporate level. Now, consider the ownership and level of sharing of the data.

How was order entry performed when a manual system was employed? Probably a physical paper invoice was thought of as a private piece of work until the clerk actually sent it off to be processed. Once submitted, the ownership and level of sharing changed based on whose desk it landed on next. Therefore the order can be thought of as a private piece of work, and thus is private data to the order entry clerk until the order is ready to submit for processing. So the question is not so much, "Are the data private?" as it is, "Can the data be *thought of* as private for that application function?" If so, that function should treat the data as a private entity and work with the data on the PWS.

Data ownership must be evaluated and it must be brought down to the personal or work group level where ever possible. If all the data remain on a central host processor, the processor may not yield optimal performance for distributed data access.

Update/access frequency. There will be times when data really are shared at a corporate level. Then to get that data to a lower level, the following questions should be addressed:

- How often are the data accessed?
- How much is required?
- How often are the data updated?
- How critical is it to have the latest copy of the data?

How often are the data accessed? If the data are not accessed very often by an application, then a remote request can be used to get to the data on the host. Using some form of distributed data services is the easiest way to access remote data. There still will be times, however, when distrib-

uted data are not the right solution. If the data are accessed often, or the amount of data requested cannot be accurately determined, distributed data may have a serious performance and network impact. If this is the case, the developer might consider downloading the data to a lower level platform.

How much is required? If downloading is chosen, the next question is how much should be downloaded? If only an extract is needed, then extract and download only the data needed. If the whole database is needed, how often is it updated?

How often are the data updated? If the data are not updated that often, perhaps a snapshot is required that is refreshed when the data are refreshed. If the data are updated often, and refreshing at the same rate of update is not feasible, then the question arises as to how critical it is to have the latest copy.

How current are the data? In a personal experience, the author worked with one customer on the design of an executive information system. In this application the data were being used for trend analysis. The designers insisted that the executives needed the latest data. But, how current did the data really need to be? It is always nice to have current data, but when comparing quarter-to-quarter, or even week-to-week, how much impact will last night's data have compared to including this morning's data too? For most applications that show trends, last night's data are good enough for weekly trends and last week's data may be good enough for quarterly trends. We need to think about how timely the data really need to be.

If the data are updated often and current access to the whole database is needed in real time, then there are two options: Either leave the data on the host and split the application so that the functions that manipulate the data execute on the same host as the data, or re-evaluate the need for current data and ask these same questions again.

It is important to keep one's focus on the business problem that is being solved, not the technical problem. The end user should be asked to clarify the business need for timely data. Can the end user use yesterday's data? Last week's? Last month's? At some point the business process breaks and the business need is not satisfied anymore. At that point, it is well to back up one step to set the ultimate timeliness required for that data. If technology allows the delivery of better results, fine, but what the user views as timely and what the business demands may be two different things. Keep focusing on the business need for current data.

It is important to keep one's focus on the business problem that is being solved.

Security. Security is a concern when dealing with multiple copies and extracts. The obvious rule of thumb is that if the platform being extracted does not meet security requirements, the data must be kept on the next upstream host that does meet those needs. The classification of data is important.

For example, consider a personnel database. If the database contains a field for employee's salary, which is usually considered confidential, the whole database would be labeled confidential. The rule-of-thumb has always been to classify the database at the level of the highest element. When taking extracts of this personnel database, what is the security level of the extract? To say the entire database is confidential would be excessive. If all that is extracted are names and addresses, then the extract may not be confidential. This implies that the developer must assign security at the "field" level for databases that are being extracted.

Capacity. Capacity is becoming less of an issue these days. As disk space on PWSs and LANs increases, there is more we can store for use at the personal and work group level. The data to place at lower levels in a network should be files like rate tables, schedules, and any type of reference item that does not change too often. These are generally small enough to fit in a LAN or PWS environment and are read-only items, so there are not many data synchronization problems to deal with.

Figure 7 Function placement methodology

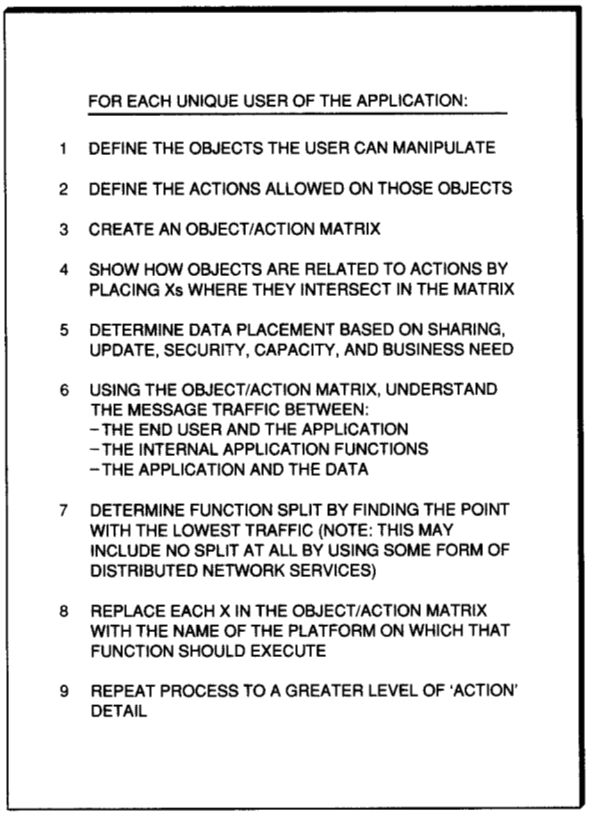


Table 1 Database descriptions

Database	Description
Customer details	Company-wide customer information
Inventory of parts	Warehouse inventory
Price catalog	Company-wide prices for parts
Daily orders	Current orders in process
Pending orders	Orders committed but not filled

The rule-of-thumb is: if the developer determines the best placement for data, only to find that there is not enough capacity for the data on that platform, then the developer must either increase the capacity on that platform or move the data back to the next up-stream platform that has the capacity. In a distributed environment, LANs play an important role to fill the resource sharing role that mainframes now perform. It does not make sense to have 500 workstations directly attached to a mainframe, with 500 copies of all the applications duplicated at each workstation. Installing LAN servers is a good way to solve workstation capacity problems in a workstation-to-mainframe-only environment.

Business needs. There may be business needs for data placement. A company may have a need to decentralize and want all databases to be managed on a midrange or work group LAN system. Or there may be a policy that all data are to reside in one location for ease of backup, so centralization is key. Whatever the need, the business need must be balanced with what is available via current technology to satisfy the local data requirements. Quite often, the business needs prevail and data placement will be determined by factors outside of the application. In these cases, tradeoffs must be made as to where to place the application function given the current data placement.

Determining function placement

The general rule for function placement is to keep the function near the object it manipulates. If the function manipulates screen objects, it is kept on the workstation where the display is being done. If the function manipulates data objects, it is kept on the node with the data. This can best be shown by using an application sample, applying the rules for both data and function placement as previously discussed. A tool called an *object/action* matrix is introduced to help determine the best platform for function placement. The steps for determining function placement can be seen in Figure 7.

We now take a set of functions in a sample application and apply this methodology. The example is the order entry function of a parts supplier warehouse. It is important to note that this application may, in fact, perform more than just this one function or be used by more than one user group. This methodology takes the perspective of a single user of the application for a particular application function. In the example we are only concerned with the order entry function. There may also be an order analysis function or an inventory function, but the order entry clerk does not use it in the process of taking an order. Those functions should be addressed as a separate matrix. The key is to break the functions down into simple steps.

Table 2 Work flow

Step	Procedure Name	Description of Activity
1	Get a form	CREATE a blank order form (invoice)
2	Take customer details	New customer? CREATE information and ADD to customer file Existing customer? SEARCH customer file, verify information, and UPDATE if necessary
3	Enter parts needed	SEARCH catalog file for part ADD to invoice
4	Check inventory	SEARCH inventory file and UPDATE accordingly
5	Check customer credit	CHECK total amount of order and any pending order against credit If OK, process order If not, cancel or SAVE invoice for later approval
6	Process order	ADD invoice to pending order file

These steps should be performed for each user of the system. It is very important to take one user's view at a time. One matrix is made for the order entry clerk and another for the stockroom clerk. Trying to combine several user's views of the business in one matrix yields too much detail.

For a new system, the first step is to gather requirements. For modification of an existing system, however, there may be a tendency to assume that the existing application functions are the base requirements. This is a dangerous assumption. Perhaps the existing functions perform as they do because of the limits of technology 10 or 20 years ago when the application was first written. The developer should take the attitude that he or she is designing a new system and gather the user's requirements for existing functions all over again. When technology poses limitations, we tend to make the requirements fit the solution. Taking a new look at the application ensures that the solution will fit the requirements.

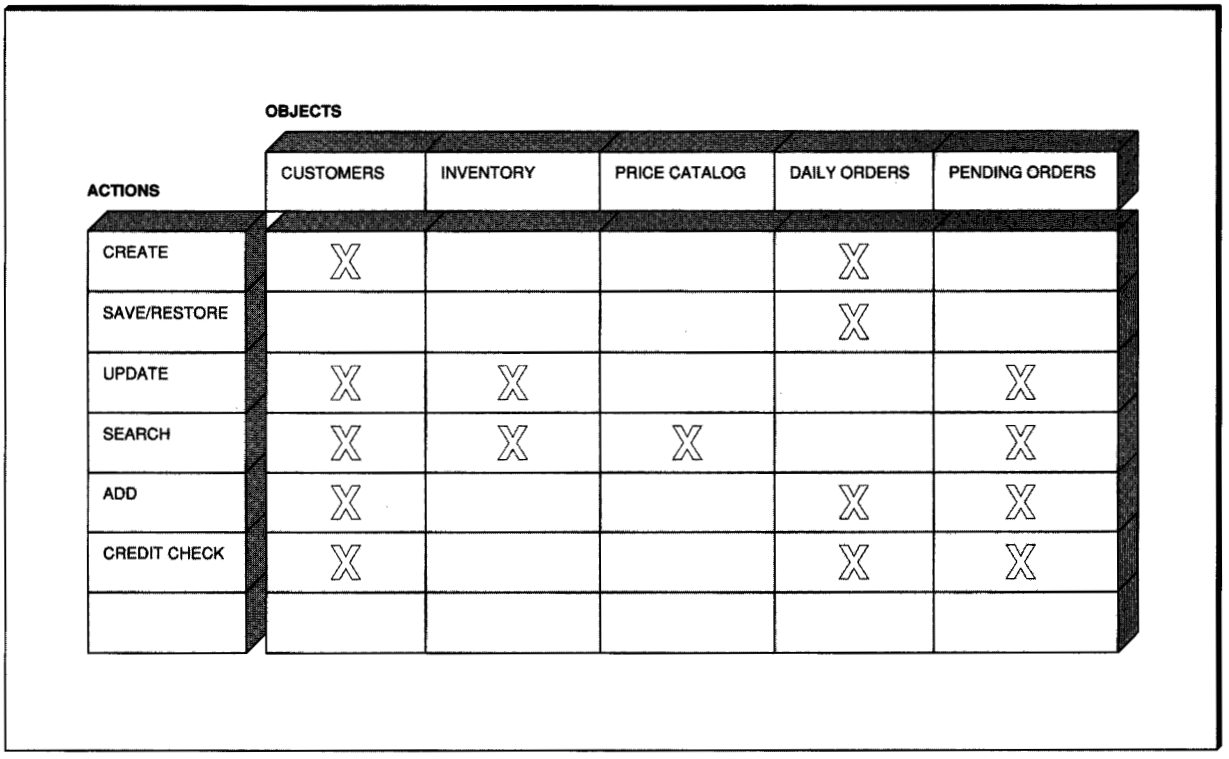
Work flow and data definition. The first thing we need to do, as in any application design process, is to understand the current business process. In our example, this is the order processing procedure. The data associated with entering an order can be found in Table 1. We start with this example and decide how we can make use of distributed processing.

In response to a phone call from a customer the order entry clerk begins the order entry process. The steps that the order entry clerk follows to enter an order into the system can be seen in Table 2. In computer terms, getting a form is simply creating a blank order form that will become the invoice. Taking the customer details involves two options. For a new customer, the clerk creates the information and adds it to the customer file. For an existing customer, the clerk searches the customer file for information and verifies over the phone that the information is still correct; if not, the information is updated, if necessary. In both cases, this information is added to the invoice.

Next the clerk asks the customer for the items to be ordered, searches the catalog for the parts and adds them to the invoice. At some point the clerk will check the warehouse inventory to make sure the parts are available; to process this order the clerk will need to update the inventory accordingly to commit those resources. The clerk will also want to check the customer's credit against the final invoice amount to be sure the customer is in good standing. The algorithm for this is simple: Check the total amount of the order, plus any pending orders the customer has not yet been billed for, against the customer's credit limit. If the credit is good, then process the order; if not the clerk may want to save the order details for further credit authorization from a manager or supervisor.

Finally, when the checking is complete, the clerk processes the order by adding it to the pending order file. Each action in the activity portion of Table 2 is highlighted by using all uppercase let-

Figure 8 Object/action matrix



ters. This defines all of the actions the order entry clerk performs.

Object/action matrix. Now that the actions that are carried out during the order entry process have been defined, and the data files are identified, it is time to understand the interaction between the two. By thinking of the data as "objects" and the things done to the data as "actions," a matrix can be built to show the interaction between the two. We call this the object/action matrix (see Figure 8).

Across the top of this matrix are listed the data objects that the order entry clerk is allowed to manipulate to process an order. These objects are defined at the database or file (record) level for simplicity. In reality, objects may have to be defined at the field level. Perhaps the order entry clerk is not allowed to see all the information in the customer file. In this case the data object would be the order entry clerk's "database view" of the customer file.

On the left side of the matrix are listed all the actions that are allowed on those data objects by the clerk. The terminology used for these actions is not of primary importance. It is best to use colloquial terms that reflect the user's perspective. While it is very easy to get caught up in the definition of terms, what is important is that the clerk has logged the action needed.

Once the matrix is set up, we place an "X" in each box where an action is allowed on an object, to show a relationship between the two. This serves a dual purpose. First, the developer can quickly see which functions will need to be written. Second, the developer can see that actions like "search" and "update" are used quite often. This is an indication that some code can be reused by making the core of these functions generic enough

to be used by multiple data objects. This could mean a tremendous savings of programmer time for the project.

For the first pass, the developer should start with data objects and broadly defined actions. It is best not to get too entrenched in detail the first time through. This is an iterative process that will be repeated several times, refining the actions to smaller entities until the developer is left with the actual functions (i.e., subroutines) that will be coded.

Current data placement. At some point the developer must look at data placement. We assume that most companies already have their data defined. If this is a totally new application, then the system designer will have to perform normal data modeling before going on. The current data placement for the sample corporation is shown in Figure 9.

In the sample, corporate headquarters maintains its own computer with the master copies of the customer details file and price catalog for the entire company. There are three regional warehouses, each having its own host computer that has a copy of the corporate customer details and price catalog, along with its own inventory of parts, pending orders, and daily orders that are maintained for the warehouse. The host computer is the warehouse computer.

The order entry department is one of several order entry departments connected to the warehouse host via a LAN. This LAN also has a file server attached that is currently unused by the existing application.

Data placement scenario. It is now time to make a decision about data placement in our sample. The options for placement are the warehouse host, work group LAN, or personal PWS. We take each data file and analyze it using the rules outlined earlier in this paper. The results are displayed in Table 3. Following is a discussion of the data objects.

The level of sharing for the *customer details* file is at the warehouse level so the tendency is to leave it on the warehouse host. It is updated daily and the system may not have the capacity on the LAN server to download it and then refresh it each night. Also there will be updates during the day as new customers call, so it is necessary to keep it synchronized with the other order entry department LANs throughout the day.

The next choice is to try to extract a subset of the data. The designer wants to find some subset of the data that may not change very often. Analyzing the demographics of the customers, the designer may find that because they are retailers who depend on the warehouse for a steady flow of parts, 90 percent of the customers are return business. This means that 90 percent of the time the calls are from customers within the local region. Therefore, an extract can be taken of the regional customers from the corporate database down to the LAN server. This subset would not take up as much LAN resource and would take less time to refresh. For the remaining 10 percent of outside and new customers, the system would use remote data access to the host that contains the master file.

This is an example of how the solution may actually be a combination of data access techniques. The system assumes a 90 percent hit ratio on the LAN. If a customer calls who is outside the local region, the system is willing to take the additional communications overhead of distributed data access across a WAN to the host database, because this should only be 10 percent of the time. Also, when a new customer calls, the host is updated immediately so that other LANs will find them if they call back twice within the same day (i.e., between LAN refreshes).

The *inventory of parts* is shared by the whole warehouse and is updated in real time to reflect the current parts committed to orders and parts remaining. Trying to manage distributed updates in a real-time system across multiple LANs is not recommended since the developer has to write all the synchronization code. It would be easier to keep this file on the host and distribute the application functions to access it so the system would have the host integrity that is needed for real-time updates at the warehouse level.

Although the *price catalog* is another file whose sharing scope is company wide, it is only updated weekly, so we download a copy to the LAN server and access it via a remote server function or locally from the PWS. Since the order entry clerk is

Figure 9 Current data placement

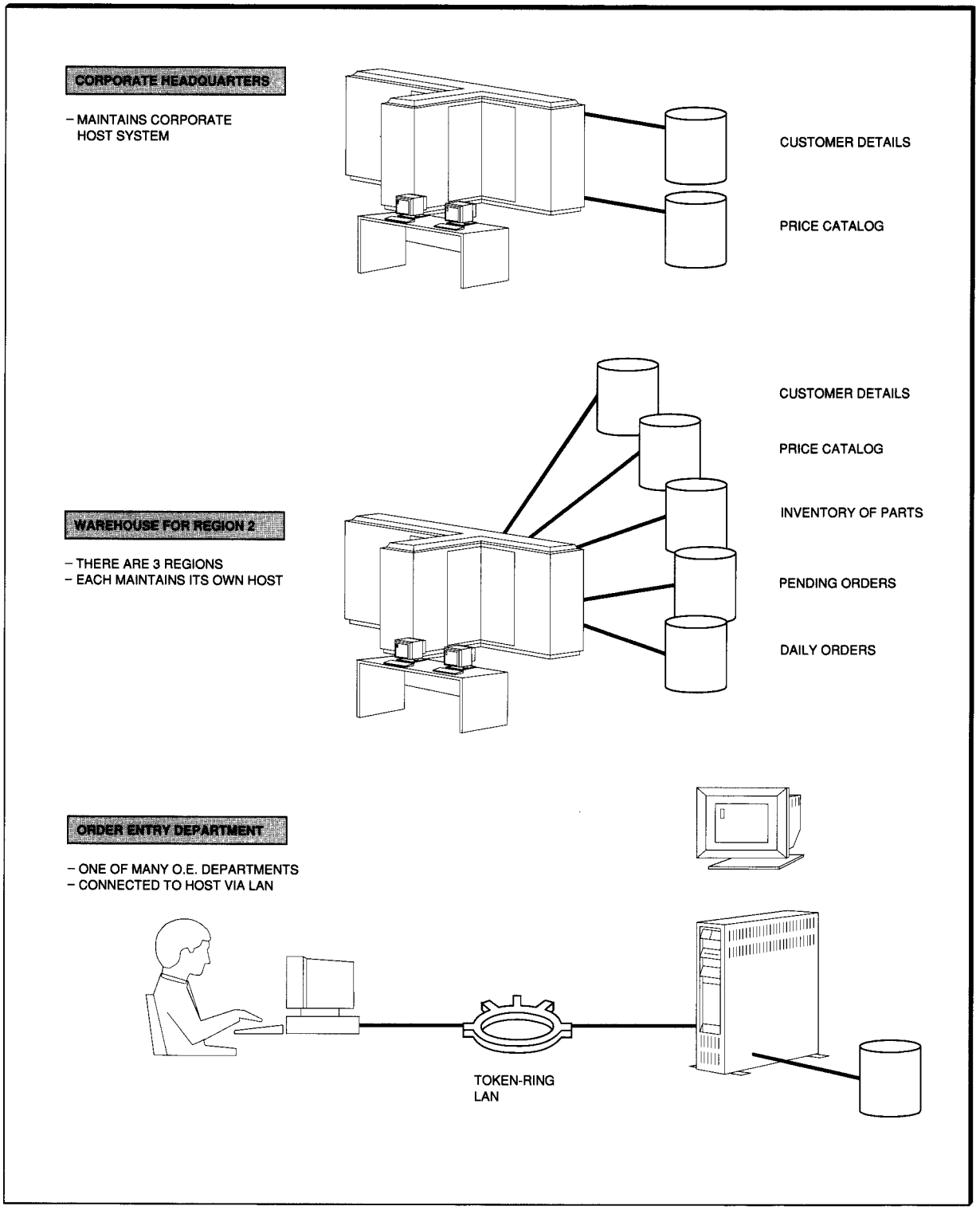


Table 3 Data sharing vs timeliness

Data Objects	Suggested Placement	Suggested Access
CUSTOMER DETAILS Shared by company Updated daily	Extract for warehouse and download to LAN server (access too frequent for remote data access)	Local access to extract Use remote data access for exceptions
INVENTORY OF PARTS Shared by warehouse Updated in real time	Maintain on host (extraction and update not feasible)	Split application
PRICE CATALOG Shared by company Updated weekly	Batch download weekly to LAN server	Local to LAN server
DAILY ORDERS Private to order entry clerk Updated in real time	Maintain on PWS	Local to PWS
PENDING ORDERS Shared by warehouse Updated as needed	Maintain on host	Batch upload at predetermined intervals

not allowed to update prices, the database is a read-only database that should not be a problem, given that the system has the disk capacity on the LAN server. This is the approach that should be taken for most read-only files like rate tables and other static files. We get them as close to the user as possible for performance and refresh them only as needed.

The *daily orders* file is created on the host and stays just long enough for some authorization process before it is appended to the pending orders file. The file is currently viewed as data that are shared at the warehouse level because other workers need to share the data for authorization purposes or to fill the order. As suggested earlier, while the data are in the hands of the order entry clerk, they can be viewed as private data until ready for submission. It is for this reason that we place the data on the workstation and then add the data to the pending orders file at some predetermined interval.

Since the shipping department needs the *pending orders* file to fill the orders, it is shared at the warehouse level. The developer could also probably add to it in real time from the daily orders file today, but does the business really require real-time access? If the business policy is next-day shipping, then perhaps this is true, but if the business policy is to allow six-to-eight weeks for delivery and a three-day backlog exists, then one must ask if a batch update at the end of the day is really going to cause an impact. In the sample, this company's policy is the latter, so the decision is to keep the pending orders file on the host for sharing reasons, but add to it from the daily orders file on the PWS in batch mode each evening.

This is a case where the designer questions current practices based on business need. It may have been easy to design the application with real-time updates in the past because the application ran on one system, but was there a business need for it? The author strongly suggests that designers look at the business requirements and not rely on how the previous application handled a particular problem.

New physical layout. Figure 10 shows the new physical layout of the data placement. The parts inventory file is maintained at the host, as is the pending orders file. Rather than updating the pending orders in real time, we batch and upload the updates on a daily basis. A full copy of the price catalog is downloaded to the LAN server, as well as a regional extract of the customer details file. The daily orders are maintained on the PWS until they are batched and uploaded to the host.

Application function placement. Now that the functions (or actions) that need to be written have

been identified and the designer has taken a first look at data placement, the designer can begin to look at where to place those functions in the enterprise. Using the object/action matrix, the location of the data objects is added at the top of the chart. Now for every "X" made in the matrix, the designer must go back and determine the location for that function, based on the message traffic it generates to manipulate its data. Keep in mind that almost all of these functions will have a PWS component to trigger them off. What is being determined here is where the "core" function will execute.

Remember it has been stated that the function should be placed close to the object it manipulates for good performance, preferably on the same processor. It is unlikely that anyone will debate that any code that interacts with the end user belongs on the PWS. The problem is with code that accesses data. The question is when to use distributed services and when to move the application function. This can only be determined by understanding the amount of traffic caused by a transaction and whether the distributed service, line speed, and network bandwidth are sufficient to handle it. Since tools are not available to monitor the physical traffic caused by transactions, all this is based on manual calculations using the size of the data record, communications link, and approximate frequency of access. By using the matrix a column at a time, leaving the credit check function for last, we obtain the results shown in Figure 11.

The customer file has a regional extract on the LAN server and a full copy on the warehouse host. The CREATE function can be written on the PWS to take advantage of the graphical user interface and excellent editing capabilities of the workstation. Keep in mind that wherever possible the designer will rethink data ownership and scope of sharing so the designer can consider a new customer record a private work until it is completed and submitted for processing. Also keep in mind that when the word "create" is used, it is not the intention to conjure up any database definition of what create means. It simply means to collect the information about a new customer that is needed for the customer file.

All the information for an order is taken over the phone, and the capabilities of the PWS are used to enter it and edit it. When the processing is done, the new record must be added to the database. Because the system is in a distributed environment, protection against two people adding the same customer at the same time must be ensured. It is for this reason that the ADD function (the actual adding to the database) is placed on the host for execution. This allows the integrity of the host system to perform this function. Likewise if there is any updating of the customer's existing record to be done, this should take place on the host also to allow record locking or what ever is needed to ensure integrity. Because a full copy of the customer database is not available at the LAN server, the SEARCH function must be written for both the LAN and host environments.

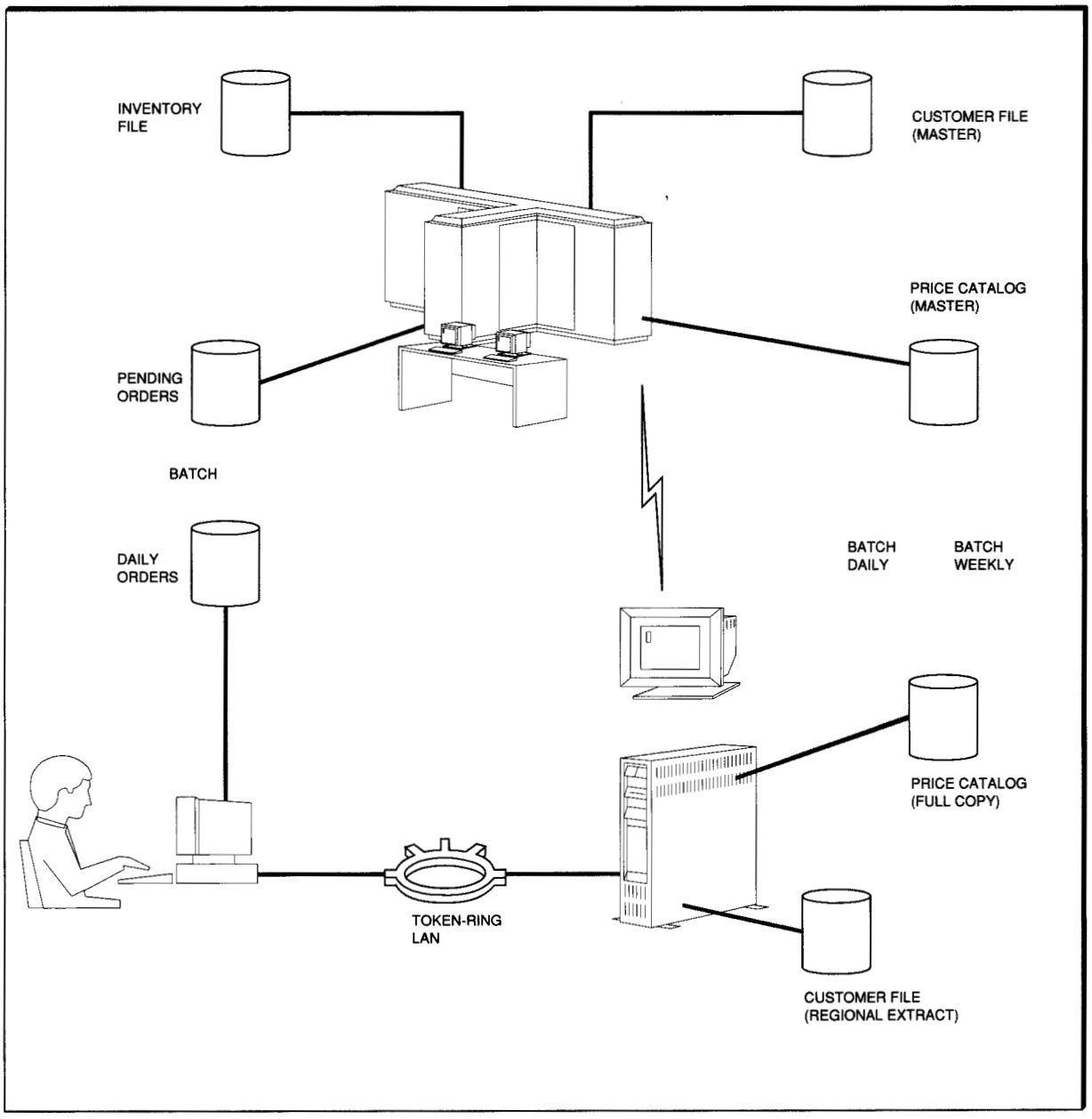
The inventory file is a rather simple matter. Since we elected to maintain this purely as a host file, the UPDATE and SEARCH functions are placed on the host with the data. It will be necessary to architect a way to invoke these host functions from the PWS later. Remember each of these will probably have a PWS component even though the *core* function runs on a host.

The price catalog is really a read-only file for the order entry function, so the SEARCH function is placed to access the catalog on the PWS and use the distributed LAN services to access it via file redirection or distributed structured query language (SQL). It is possible that many people searching this file at the same time on the LAN may cause too much traffic even for NetBIOS. If this happens, the SEARCH function should run in the LAN server machine and requests should be made from the PWS.

Since the daily orders file is really private data to the order entry person until it is ready to be submitted, all functions that manipulate this file are placed on the PWS and then the results are batched and uploaded to the pending orders file. Also, since multiple departments need access to the pending orders file at the warehouse level, the function that manipulates this file resides on the warehouse host with the database.

Example: Credit check function. The credit check function is left until last for a reason. There are two possible ways to implement this function, as

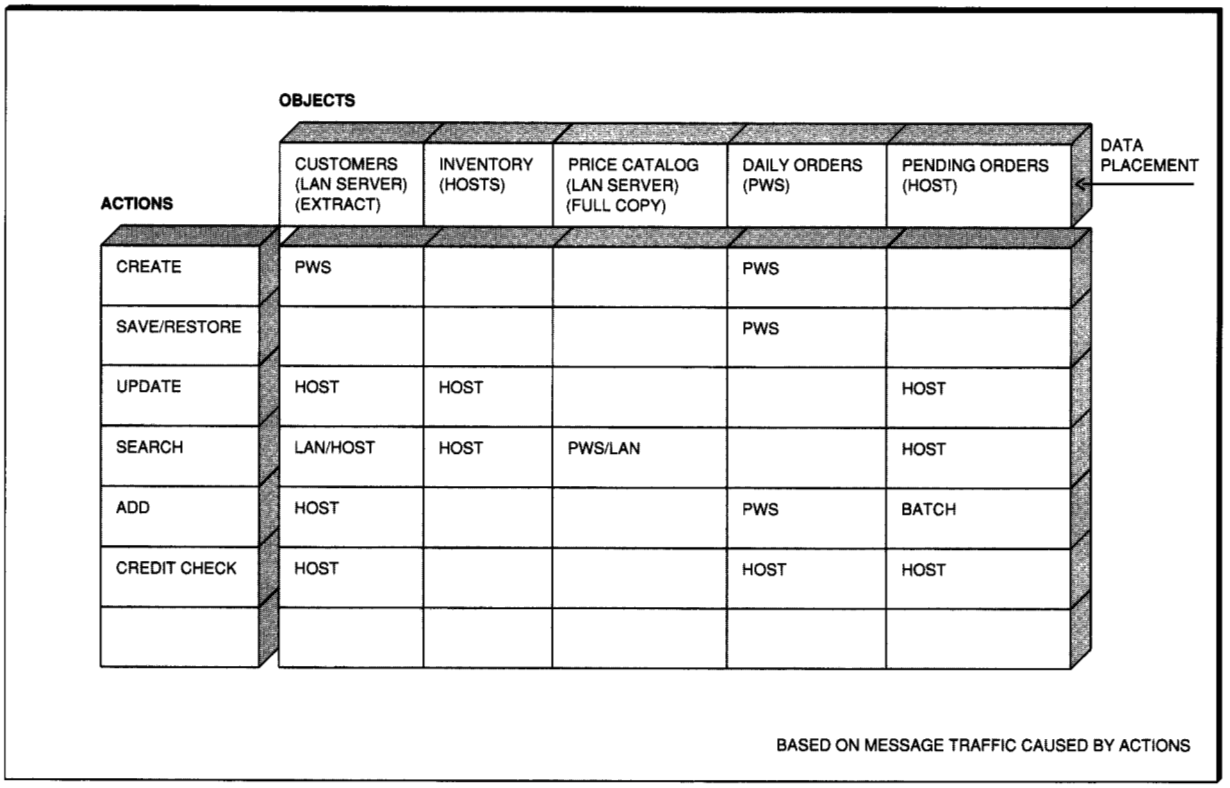
Figure 10 New physical layout of data



shown in Figure 12. Since the customer file and the daily orders file are both on the PWS, the function can run on the PWS and access the pending orders file remotely. If this is not a lot of data, this may still be a good idea; but if the data are too large, or the frequency of credit checks is such that the traffic is too much, this is not a good choice.

This is an example of how using some form of remote request works to a point. If the system is in a small company that does not have many

Figure 11 Object/action matrix with function placement



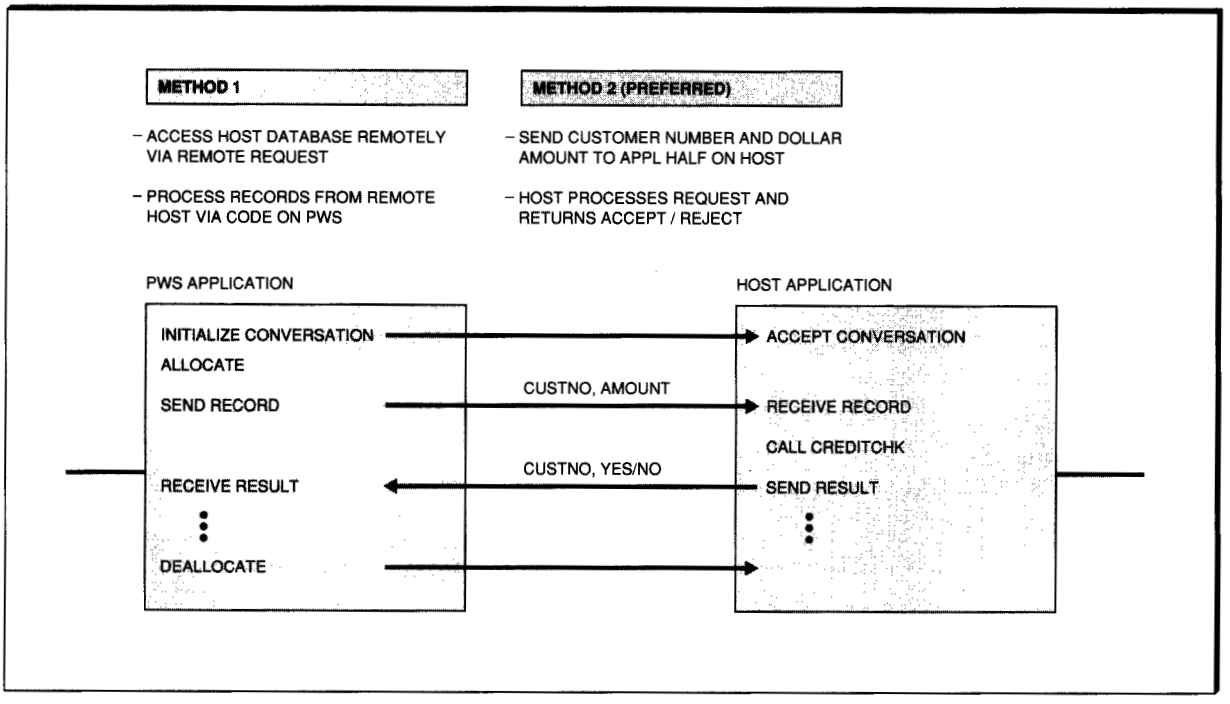
pending orders, this method might work fine; but if we have a large corporation with hundreds of pending orders, this method might break if the data coming across the link from a distributed request are too much to handle. This is not a robust design.

A better solution is to write the credit check function to run on the host with the master customer file and pending orders file. After all, the only thing needed from the PWS is the customer number and the amount of the invoice. The only response required is the customer number and an accept or reject notification. If the credit check function is run as an synchronous task, the customer number is not needed because the user is waiting for the answer. This also eliminates the worry of how many data are returned from the credit search, because this is all contained within the host environment (i.e., the size of the results are always predictable). This design will not be affected by a growing order backlog or increased transaction rates and is preferred for this particular type of request.

It is important to only use distributed data in a line-of-business application when one can safely predict or limit the volume of data to be transmitted from a given request. The network may be able to absorb a few ad hoc queries, but businesses do not want all the order entry people making voluminous queries all day, every day, for every customer invoice.

Methodology summary. To summarize, we start by taking the view of one end user. This may be the data entry person's view, the analyst's view, or the executive's view, but we hold to that view. We then go back and make a new matrix for each different user of the application, avoiding the temptation to combine all actions and objects for all users into one large matrix.

Figure 12 Credit check implementation



First define the objects that the user can manipulate. These should not be restricted to data objects, as in the example. Data objects can also be defined at the record or field level if needed. In the example they are defined at the database level for simplicity, but in reality some people may not have access to a whole record.

Once the objects are defined, next define the actions allowed on those objects. These can be derived from an examination of the work flow, much the same as one would do for any application development. What is important is the next step, which is to define how the objects and actions are related through the use of an object/action matrix.

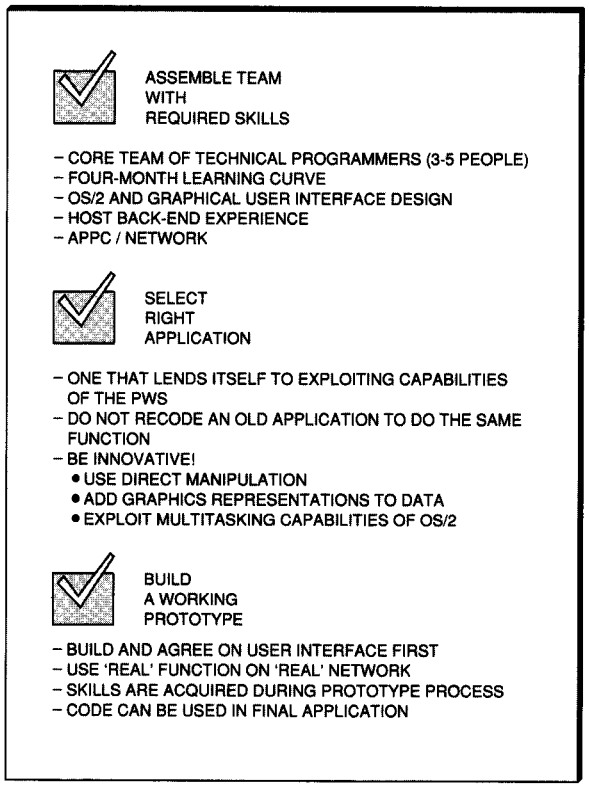
At this stage, examine the data placement and see if the data can be placed as close to the user as possible for performance reasons. Remember, data placement is primarily a function of sharing. It is desirable to keep the data at the lowest level of sharing possible. Go back to the object/action matrix once this is done and add the location of the data under each object heading.

Now the designer is ready to analyze and understand the message traffic between the end user and the application, application-to-application, and the application and the data. The traffic between the application and the data is probably the highest. This is why we suggest that the parts of the application be kept close to the objects they manipulate. If one is manipulating a screen object, keep the code on the workstation; if one is manipulating a data object, keep the code on the processor that contains the data. Also, be very careful to understand the size of the results from a distributed request if distributed access is the preferred choice. Since the traffic between the application functions is probably the lowest, the designer will want to choose a point within the application that makes sense to distribute.

Once the matrix is built, the designer must explore the possibilities for making the connections.

Checklist for getting started. A checklist for getting started is shown in Figure 13. First and foremost, assemble a team with the proper skills. The core team should consist of three to five of the

Figure 13 Checklist for getting started



best people available. Allow for a four- to six-month learning curve in the respective areas of PWS, APPC, and host back-end skills.

Next, select an application that lends itself to exploiting the capabilities of the PWS. It should be something small enough to manage easily. Do not simply recode an old application to do the same function in a distributed manner. No one will see the benefit. Add additional function that exploits the PWS so people will say, "You can't do that with a nonprogrammable terminal!" Be innovative, use direct manipulation, add graphic representations to data that were only shown in tabular format before. Exploit the multitasking capabilities of OS/2 so that things like credit checks and table lookups are performed asynchronously and work goes on in the foreground while back-end processors operate.

Build a working prototype. Because all application function will be driven by the end-user interface, build and agree on the user interface first. Having an action bar that controls the pop-up windows with no code behind them will give the user a good feeling for how the application will flow. Since functions will be more atomic in this object/action environment, you can plug the functions behind the windows later with greater ease than in the old hierarchical model, where the function called the user interface rather than the reverse.

Take one distributed function and code it from start to finish. Then use it on a "real" network to understand the implications of network traffic. Network traffic is going to change. We used to think of PWS traffic as infrequent but occurring in large chunks. This is because we mostly downloaded files, worked with them, and uploaded the results. The trend in distributed processing is to have a much shorter message duration but with increased message traffic. This can be equally devastating to a network. It is better to work on one function and refine it until the calculations for traffic can be trusted, than to code all functions first and then find out the network cannot handle the result in final test stage.

Skills are acquired during the prototyping process. Although you may need to recode your first prototype, most others can be used in the final application. An object/action application is designed with the user interface (objects) first and function (actions) added later.

Conclusion

When applications are developed that are distributed across multiple platforms, there are always tradeoffs to be made. Clearly if one must capitalize on system services that are specific to one platform, the design will suffer if it needs to be moved to other platforms. Still, distributed processing can be a powerful base for application design, where the resulting application should provide a better result than could have been achieved with either the PWS or mainframe technology alone. All this is achievable with today's technology.

Applications can be written to perform well regardless of the underlying topology. The application design should provide scalability, acceptable performance, and reconfigurability for future growth in the enterprise. A good application de-

sign will work well across a wide range of usage/loads without change, and execute across a wide range of configurations. It should also allow flexibility in data placement.

While there are considerations today that have to do with bandwidth implications, these can be overcome by designing the application to be insensitive to network latency by executing functions that require high data access of the same platform with the data.

As a common remote procedure call becomes widely available across systems, it will be much easier to implement distributed applications. In the mean time, the user can design applications today to take advantage of tomorrow's technology when it arrives.

Acknowledgments

I would like to thank Allan L. Scherr who not only provided the basis for this work, but from whom I have learned so much in the time I have worked for him. I would also like to thank the customers who worked with me in validating my methodology for function placement. Finally I would like to thank Oliver Simms, IBM United Kingdom Technical Support, for the use of some of his charts and figures and for sharing his ideas on the event-driven application model.

Cited references

1. *SAA Guide for Evaluating Applications*, G320-9803, IBM Corporation; available from IBM branch offices.
2. A. L. Scherr, "SAA Distributed Processing," *IBM Systems Journal* **27**, No. 3, 370–383 (1988).
3. R. N. Manicom, *The Mainframe Versus PC Battle*, White Paper, IBM Canada Limited (1990).
4. O. Simms, "The New World," unpublished, IBM UK Technical Support.

Accepted for publication December 4, 1991.

John J. Rofrano, Jr., *IBM Application Solutions, Route 100, Somers, New York 10589.* Mr. Rofrano is currently a senior programmer on the Application Solutions architecture and development technical staff and is working on the definition and implementation of an application services architecture. He joined IBM in 1984 in the System Products Division in White Plains, New York, as an information center analyst. There he specialized in the support of Personal Computer products. In 1986 he became manager of the Information Center for the Information Systems and Products group and a year later became manager of Customer Services I/S at IBM Corporate Headquarters in Purchase, New York. In 1989 he joined Application Solutions, working in the Cooperative Processing Cluster as a technical planner. He was responsible for strategy and plan evaluation of future products that would enable cooperative processing. During this time Mr. Rofrano served as IBM's representative to the GUIDE Cooperative Processing Project and worked with members of GUIDE, SHARE, and numerous other customers on his distributed design guidelines. Mr. Rofrano received a B.S. in computer science in 1984 from Mercy College, New York.

Reprint Order No. G321-5487.