Distributed Application Handbook

By Paul Dickson

Business and Office Systems Engineering

Distributed Application and Architecture Group

First edition

Draft of December 6, 1985

Table of Contents

Preface

Potential computer users have problems. They want a computer to help them with these problems. They don't want a collection of unrelated components, and the *last* thing they need is another set of problems.

The "micro-mainframe connection" gets a lot of attention these days, but there is no concensus as to what it really is. Is it just copying files between computers? That is about all that most companies can deliver today. Yet even this is presented in such a complicated way that the user has to understand computer concepts in order to use it.

There are many computer products in the market that, while attempting to solve one set of problems, create another. Personal computers have saddled the end user with unexpected system support tasks, while making integration with the shared environment of centralized databases more difficult. Some observers say the the users' discovery of these shortcomings is one of the causes of the current industry slump.

The purpose of this paper is to describe a particular way of looking at computer applications, and a particular engineering approach, that leads toward efficient and versatile system products. The tool described here is the *distributed application*, a particular kind of computer program that splits the processing load into two or more parts.

Distributed processing is any processing in which programs are executed cooperatively in separate computers, and communicate with one another. There are five senses in which an application can be said to be distributed:

Functionally
: Separate functions are in separate components.

Physically
: The components are in separate computers.

Logically
: The specifications of the client and server are under separate control, although the specification of the client necessarily references that of the server.

Operationally
: Each component is under separate operational control, with some degree of local discretion. Operational control is the control of program execution and physical operation of the equipment.

Administratively

The installation and management of the clients is separated from the (often complex) administrative arrangements for the servers. This has ramifications for pricing, installation procedures, and licencing.

The focus of this paper is on the *functional*, *logical*, and *physical* aspects of distribution.

Cost per desk

If we divide a system's total cost by the number of desks served, we get the *cost per desk* for that system. This is a useful metric for measuring the economic viability of various solutions.

The costs associated with shared resources (large disks, printers, etc.) can be spread over all the desks, while costs associated with individual desks (CRT display, keyboard, communications hardware, personal computer) apply directly to the cost per desk. It is important to include not only the initial capital and training costs but also the continuing costs of operation, in both time and money.

For example, the choice of application design can affect communication requirements, thereby affecting the kind of wiring to use between the shared (centralized) and unshared (dispersed) components. A design calling for high-bandwidth communication will typically require more expensive communication facilities, and more compute power in the shared component, than a design calling for lower-speed links.

The goal of a distributed design is to assign functions to the components of a system in such a way as to *minimize* the cost per desk while simultaneously *maximizing* the visible performance of the system.

That is the *goal* of distributed design, but there can be additional *benefits*. Among these are increased modularity of the resulting systems, and ease of extension to handle new problems. (That is a good test for an elegant solution: it solves other problems in addition to the one you thought you were solving.)

Structure of this paper

This paper is organized in several chapters. The early chapters are conceptual while the later ones are technical. Readers with limited interest in implementation details can stop after 3 chapters. Development managers should also take a look at chapter six.

In the System Engineering chapter we look at how to analyze requirements and entire systems.

In the Structure of Distributed Applications chapter I present a way of looking at the functions to be performed in a distributed application, and how to choose which functions should be performed where. The model is similar to the "Open System Interconnect" model for communications systems, but talks about information representation, manipulation, and presentation functions, rather than routing, re-try, and so on.

In What the user sees we look at some human factors concepts that lead the way to a style of presentation that is both *technically efficient* in a distributed environment and *operationally efficient* for the person using it.

The Implementation Techniques chapter presents a "toolbag" of programming techniques you can use in designing high performance distributed applications.

Chapter five contains some **Examples** of real distributed applications, and the final chapter is a discussion of how many applications, with a single user interface, can be built from the same components.

At the back is a glossary of terms and a bibliography.

What makes you the expert?

Many of the ideas presented here are distilled from my experiences in designing and implementing early versions of three distributed applications: the Message Router, VAX VTX, and the ALL-IN-1 Workstation Server, as well as a few experimental projects that never became products. At the time, I was not aware of any conceptual foundations for what I was doing. I just did what felt right.

So writing this handbook has been a process of "reverse engineering" for me, trying to remember what I was thinking of back then and attempting to discover the underlying concepts. It turns out that there *were* common themes running through those designs - I just had not put them into words. At times it has been a considerable challenge to come up with clear explanations for what is, in some respects, a new way of looking at computer applications design.

Scattered though the paper there are some questions about how things might be done differently from the way they are now. These half-finished suggestions are intended to get you thinking about how to apply the concepts to your own projects. I hope that you will find the ideas presented here to be useful.

Chapter 1
System Engineering

Computers are not solutions to business problems. At most computers can be tools to make people more productive, and then only when appropriate application software is available. The task of computer engineering is to make effective tools using computer technology.

A system is a group of interrelated elements forming a collective entity. If we take a bunch of existing components and stick them together and call the result a "system", we are just fooling ourselves. If the components are not working consistently together toward one end, it is not really a system.

Making low-level system components like printers and disks work on a network (with "printer servers" and so on) does not make the applications themselves distributed. As I will show in the next chapter, distribution has to take place at high levels in the program in order for all the performance benefits to be seen.

The system designer must consider each individual component's effect on the over-all system as seen by the user in terms of response-time, cost, and management effort. From the manufacturer's point of view, the maintainability, expandability, ease of selling, and profits must be considered.

In this chapter we will be looking at how product requirements are best determined. My view is that the only meaningful source of requirements is an analysis of the tasks the user needs to perform. Therefore our analysis will be from *front to back*, starting with the user.

As we look at the requirements we must keep an open mind about what they mean and how they might be met. So first we will be looking at some common pitfalls in the analytical process.

How to look

When we look at how systems are put together, or when we analyze problems to see how they might be solved, we should be careful not to go in with our minds locked on one way of looking at things. Specifically, we should try not to confuse what we have chosen to *call* things with what those things really *are*. "Naming" is only one of the ways in which a mind can be closed, but it is a very easy trap to fall into. As Alan Watts writes in *The Way of Zen:*

> "To serve their purpose, names and terms must of necessity be fixed and definite like all other units of measurement. But their use is - up to a point - so satisfactory that man is always in danger of confusing his measures with the world so measured, of identifying money with wealth, fixed convention with fluid reality. But to the degree that he identifies himself and his life with these rigid and hollow frames of definition, he condemns himself to the perpetual frustration of one trying to catch water in a sieve."

Watts is talking about confusion between the name of a thing and the thing itself. A name restricts your knowledge about something to a particular viewpoint, and if you pay too much attention to the name, and not to the named thing, you will lose contact with reality. In particular for this discussion, if two things are similar, but have different names, you might be tempted to handle the two things differently. Why implement the same function twice?

Another aspect of naming that can cause a lot of trouble in software design is the effect names have on how we build hierarchies. As Robert Pirsig puts it in *Zen and the Art of Motorcycle Maintenance:*

> "There is a knife moving here. A very deadly one: an intellectual scalpel so swift and so sharp you sometimes don't see it moving. You get the illusion that all those parts are just there and are being named as they exist. But they can be named quite differently and organized quite differently depending on how the knife moves."

Focusing on the wrong names can lead you to a wrong analysis of organization. Names can be given according to where something is in some organizational view, but if you pay too much attention to the names and forget the organization from which they spring, you might not think to look for *other* organizations.

The trick in applying this principle to the analysis of requirements is to take a step back from the problem to see the general context in which it exists. You may be able to pick a better viewpoint from which to approach the solution than where you started.

To clarify this point, consider the following examples.

Constellations

A stellar constellation is a group of stars. In ancient times, names were given to these groups (Gemini, Ursa Major, and so on) because the patterns reminded people of something. But these patterns are only visible from our location in the galaxy. Viewed from another position, the stars will form entirely different patterns.

The stars that make up a constellation are at various distances from the Earth, but the constellation itself is right here, inside our heads.

Animals in a zoo

Imagine you are setting up a small zoo, and you have just three kinds of animal: lions, snakes, and rabbits. There are five ways you could organize your zoo:

Scheme	Description
No groups	One big buildings
By name	Three separate buildings
By class	Lions and rabbits in the mammal building, snakes in the reptile building
By size	Lions in the "large animal" building, snakes and rabbits in the "small animal" building
By diet	Snakes and lions in the "carnivore" building, rabbits in the "herbivore" building

Each scheme has advantages and disadvantages, depending on cost, and what educational message you, as zoo director, are trying to get across to the visitors.

Books in a library

As another example, consider a library. Most libraries put the books on the shelves according to some analysis of what the book is *about:* non-fiction by subject category, fiction by author's name. This facilitates browsing in the stacks.

But the medical library at the Ohio State University School of Medicine stores the books according to their *physical size.* This library does not allow browsing in the stacks. In fact, the only "person" in the stacks is a robot that retrieves whole boxes of books at a time. Using an index, a computer tells the robot which box to fetch.

By eliminating the need for efficient browsing in the stacks, the designers of this library were free to choose a more efficient shelving arrangement: the boxes are filled most efficiently if all the books in a box are the same size. Browsing is done in various abstracts and index collections, which is a more productive way of finding something in such a vast technical collection.

Examples of wrong hierarchies

Hierarchies are inventions of the human mind. They represent a particular view of the thing under study. Taxonomies and hierarchies may help us to understand a complex structure, but they are not necessarily inherent in that structure. It all depends on how you look at it.

In this section we will look at a few examples of hierarchies in computer products, and how they may not have been appropriate.

Office functions on a menu

In an attempt to shield end users from traditional computer command languages, we have often resorted to *menus.* When there are more alternatives than will legibly fit on a screen, we have several menus, linked into a tree structure.

How often have you heard computer menu interfaces decried as slow or unwieldly? This happens because the designers have put the functions into a hierarchy that has nothing to do with the way in which the customer is going to use them. These hierarchies also do not necessarily represent what the user will have in mind. If you want to visit all the carnivores in the zoo, but the animals are grouped by size, you are going to do a lot of walking between buildings.

Perhaps the end-user should be allowed to move things around in the menus, to suit his own idea of how things should be organized. Job requirements change over time, and a fixed menu scheme does not let the user adapt the environment to meet the requirements or his own changing capabilities or assignments. After all, we usually don't bolt the furniture to the floor.

Wrong menu organizations also have a human factors impact. If the goal menu is more than one choice away from the current menu, we are asking the user to remember and visualize the hierarchy *we* used.

Office projects on a menu

Another consideration the kind of things that are on the menus. Is the choice from a list of computer tasks to perform, or a list of projects to work on? That is, does the menu reflect the organizational chart of the end-user or of the development team?

Working from Front to Back

The system starts on the desk, right in front of the user's eyeballs. That is where we start building. But what the user wants to accomplish starts *behind* his eyeballs. That is where we should start analyzing potential computer applications.

The only meaningful source of functional requirements is an analysis of the tasks the user needs to perform, but we can not rely just on what people tell us they want, or what the "market" says it wants. We should solve the customer's problem, not just provide the customer's solution.

It is somewhat naive to think that users know what they want and that all we have to do is ask them and they will tell us. If you sit down with a potential user and describe just about any solution based on your knowledge and research into how he does business, the person will say "Yeah, that sounds good. Build me one."

Conversations between engineers and customers and salesmen often suffer from the different point of view each has of the problem. A "romanticist" looks at the *function* of a thing, while the "classicist" looks at the *form* of a thing. End users and salesmen typically take a romantic view. They want a computer to accomplish a particular task and they are not particularly interested in how it works. The engineer typically takes a classic view, and is concerned with the form the solution takes. There can be a great deal of confusion if somebody confuses a desired function with the form in which that function is delivered.

To the user, the object of attention is usually the solution to some business problem. To the engineer, the object of attention is typically the tool used to solve the problem. There is a conflict here that would arise even if both the user and the tool builder were

both either romanticist or classicist: there is a conflict in purpose. One thing we should work on is trying to bring the tool builders closer to the problems their tools are being used for. And we should bring the users closer to the tools they use.

We must discover the real problems and tasks, then step back to get an over-all perspective. This discovery process includes customer interviews, but also learning directly about their offices, business operations, and industries.

So called *technology driven* products are not responsive to customers' needs, but to engineers' egos. Just because something *can* be done does not mean it *should* be done. It is all too easy to be so caught up in making better and better electric drills that one loses sight of the fact that what the customer really wanted was holes.

Perhaps we should not concentrate so much on "functionality" as on usefulness. You can probably buy an electric apple peeler, and it does indeed perform that function. But how *useful* is it, considering it takes longer to set up before, and clean after, than using a simple paring knife, is more prone to failure, and costs more?

A paring knife is much, much, simpler than an electric apple peeler, but it embodies much more useful functionality. A collection of simple tools, usable in various combinations for each task, may be easier to understand than large and complex tools specialized for single tasks.

How are things the same?

A good technique for finding other views of a problem is too pick some *other* problem and see what the two problems have in common. If you can find a point of view from which the problems are similar, perhaps you have found a method for solving both problems at once.

This is why I brought up "stepping back from the problem" earlier. You have to step away from the immediate goals and look at other goals before you can find a pattern. There is no pattern in a singularity.

A mental trick for "stepping back" is to temporarily stop naming and classifying all that you experience and simply feel life as it is. This is a method of getting away from any preconceptions you may have about how things are organized.

A Data-flow model

Sometimes it is useful to analyze a computer problem by looking at the way information flows through the system rather than how and where various procedures are performed.

Don't look at the problem from the point of view of processes that read data in and write data out. Instead think of the processes as simply waypoints through which the data flows, like filters. Filters are passive; they take what is given to them, process it, and pass it out again, transformed in some way.

The procedure can only process information at a certain speed. The client (or clients) can generate information into this procedure faster than it can be processed. A buffer is needed to smooth out the flow. To store the information until the function can process it. This buffer is that functions *input queue*.

If there are a chain of several functions, each with an input queue, it is unreasonable for the first function to wait for an answer back from the second function, before it starts processing another input request. So after the first function processes the request and passes information on back, it can immediately start processing the next input.

When the response comes back forward again, it has to match the response with the context of the suspended first request and performs the next step in the procedure.

Some input going backward requires a response before that thread can continue in the client. Every transaction is not of this type. Short ones might be, but if there is a large amount of information to pass, it is better to send rather long streams of it, several sub-transactions, without waiting for an acknowledgement on each one.

The client can work the same way as the function boxes: that is, proceed as far as possible. The acknowledgements will be coming back, but they will come some time later. Be prepared to have the operation suspended until the response comes back. This does not mean it should stop doing everything else.

Bandwidth and performance

Communication links are also bottlenecks between functions. Therefore a communication link needs to have an input queue of messages waiting to go out. So as a function unit prepares a message to send to another unit, which might be on another node somewhere, information may go into an input queue for a function that does nothing more than transport the information to the other end.

A procedural function has a certain thoughput, which can often be expressed in bits per second. This involves the complexity of the procedure. A communication link also has an effective bandwidth.

If a function contains calls on another function which can take an amount of time, it should be broken down into more steps. For example, if the function has to read from a disk file, the disk latency is very long compared to what the CPU can do. So the action of reading a record from the disk should be a seperate action, with an input queue. Procedural functions put requests in this queue, the disk services them as fast as possible, information is retured later to the next next step of the procedural function, which matches it up with the suspended context.

When effective, this decoupling of functions by queues allows each resource to saturate at its own rate. This makes it much easier to predict what the service time will be with a mathematical model.

Multi-server queues

Sometimes there is a performance advantage in having more than one instance of a function available to service a single queue. An example of this is the HSC-50 disk subsystem controller, which is capable of *reducing* the effective seek time of the disks if more than one request is presented to it simultaneously.

Multiprogramming not required

The procedures do not necessarily operate in parallel or consecutively. If you take the procedure-centered view, you worry about parallelism and so on. If you take the data-flow point of view, with the distinct function units separated by queues, then whether the proceures operate in parallel is not important. Any complex procedures are broken down into steps, and each step is indivisible.

There is no point in interrupting one step in order to execute another step, so there is no need to have a time-slice scheduler involved.

Designed in this pipe-lined fashion, the functional units can be on other processors or the same processor. When units do execute on the same processor, they can be within the same host system process, with no need for any kind of "multiprogramming" within the program. No operating system support for "multiprogramming within a process" is needed.

This same technique will work on Ultrix, VMS, or MSDOS.

Recapitulation

In this chapter we looked at the insideous trap of *naming*, and how it can mislead us into building inappropriate hierarchies.

We discussed how requirements have to flow from front to back, starting with an analysis of the user's tasks.

Finally we examined a *data-flow* method of analysis.

In the next chapter we will look at an analytical model of the functions performed in computer programs in general, and interactive programs in particular.

Chapter 2
The Structure of Distributed Applications

The most important decision in the design of a succesful distributed application is the choice of which functions go on which end of each communications link. As an aid to the understanding of program structure, so we will know the ramifications of introducing a communications link in one place or another, this chapter describes an analytical tool called the Layered Abstraction Model.

The Layered Abstraction Model is similar to the "Open System Interconnect" model for communications systems, but here our focus will be on information representation, manipulation, and presentation functions, rather than routing, re-try, and so on.

Functional layers in interactive programs

While acknowledgeing that all such categorizations are artificial (see "How to Look"), I find that analysis of applications according to two viewpoints to be useful for the purposes here. The first view is that programs contains components that deal with different levels of abstraction in the information being processed.

The second view is that programs contain both *visible* and *invisible* components.

The Concrete and the Abstract

Now let's take a look at the different kinds of things going on in an application program. I find that five layers prove useful in this analysis:

Specific	Abstract	5	Coordination
		4	Manipulation
		3	Representation
		2	Style
General	Concrete	1	Physical

The more *concrete* layers deal with the mundane issues of particular I/O devices and data structures. The more *abstract* layers deal with particular views of problems to be solved.

Any particular implementation can leave the boundaries between layers indistinct, but if you are building a sharable function module, the boundary should be well-defined. Such compartmentalizing is important when it comes time to split an application between layers.

Layer 1 - The Physical layer - This is a very general-purpose layer, and it is also the most constrained by the hardware design. This layer is very *concrete*, like the symbols in the preceding chapter. It is not really "physical", but it is as close to physical as a programmer typically gets.

This is the "QIO" level of interaction with disk drives, where the only structure is a series of blocks. (We will gloss over the details of disk controller interfaces and MSCP.)

In a terminal, this layer includes the *display list*, a section of memory from which the video display is regenerated by hardware. This is the most visible part of a computer program. So-called *bit map* terminals have very simple display-lists, while some text and vector graphics terminals have very rich display-list formats.

Layer 2 - The Style Layer - This software superimposes a general purpose structure on top of the fixed blocks available from the previous level. For example, the RMS, DBMS, and RdB database facilities operate at the Style layer. Alhough still general-purpose, this layer is a bit more focused on particular ways of organizing information than is the QIO level.

In terminals, this layer includes the *display style*, which controls the generation of the display-list from the Representation Data. The display style is a set of rules for the translation of data from the Representation Layer into its concrete visible form on a specific display device. Forms drivers and window managers are examples of functions operating in this layer.

Layer 3 - The Representation Layer - This software superimposes a special purpose structure on top of the general purpose structure available from the Style Layer. Code at this level is reposible for maintaining the integrity of the information. A paragraph of text is an example of data at this level, perhaps coded according to the DDIF (Digital Document Interchange Format) specification.

For terminals, this is a hardware-independant representation of visible information. An examples is the data in the fields of a form. The elements in this layer are common to many applications, as was shown in the earlier visible/invisible table.

Layer 4 - The Manipulation Layer - These processes operate on the data maintained in the representation layer. Examples of operations at this layer include moving information from one document into another, wrapping a paragraph, inserting and removing text, and so on).

Layer 5 - The Coordination Layer - This is the *most specialized* layer of an application, the *least constrained* by hardware, and the most *abstract*. This is the "glue" that holds the pieces together, forming a particular instance of an application program out of more general functions. Code in this layer keeps track of what manipulations are being performed, how many windows are active at once, which files are open for what purpose, and so on. It also coordinates the activites of the other layers.

The Visible and the Invisible

The second viewpoint I find useful is that functions can be classified as either visible or invisible. Let's take a look at five common office computer application families. For each family we will list the functions that are *visible* to the end user and those that are *invisible*.

Word processing - This is generally text editing. The invisible operations of word processing are the manipulation of the on-disk structure of the document. Other background operations include formating a document to be printed.

Electronic mail - This is really just a special "post" operation added to a text editor and a filing system. Text editing was covered above, and the filing system needed by mail can be exactly the same one used in most word processing packages. The "post" operation is itself invisible.

Transaction processing - The interactive style of transaction processing (order entry, for example) is pretty much fixed forms and menus. Of course, the processing of the transaction varies from one application to the other, and the layout of the forms may differ.

Information retrieval - This can be just another form of transaction processing, but might include more unstructured information, such as videotex pages.

Spreadsheets - A spreadsheet's *visible* aspect is of a multi-windowed text editor with a whole lot of windows, mapped to different parts of the same document.

When we summarize these applications in a table, one of the first things we notice is that the same set of relatively well defined visible functions is used by all the applications. It is even feasible to put them all into a terminal.

Application	Visible functions	Invisible functions
Edit	Scroll, select, wrap, move, insert, replace	File update, indexing, print
Mail	edit, forms	Filing and posting
TP	Forms, menus	Special processing, database access
Info ret	Forms, menus, videotex pages	Database access
Spread	Scroll, select, forms, move, insert, replace	Database access, computation

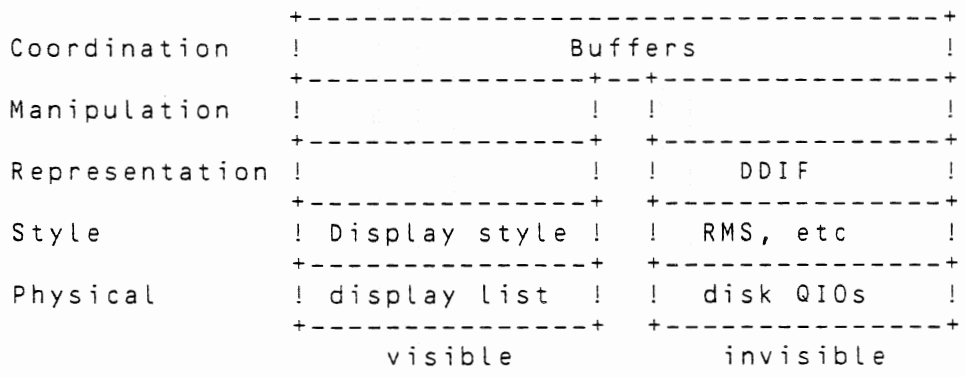
A nice thing about invisible operations is that they can be placed in a host computer with minimal performance penalty, because they *are* invisible; that is, *noninteractive*. This fact suggests that the visible/invisible boundary is a good place to make the cut in a distributed application.

To the extent visible and invisible operations can overlap, response time is reduced. If the invisible operations can be sufficiently decoupled from the visible operations, the visible, perceived responsiveness can be greatly improved. This is easily visible in a spreadsheet when you turn off "automatic recalculation".

Combining the two views

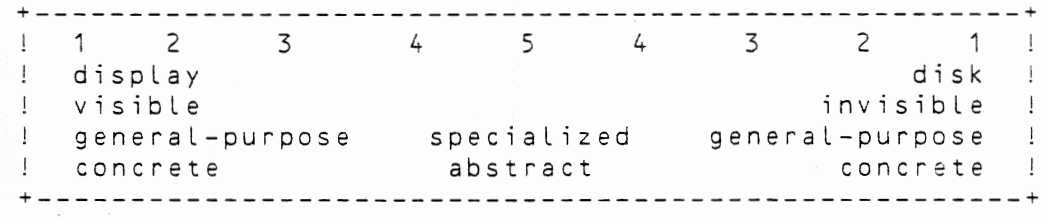
An *interactive* program has *two* concrete ends: one in the disk drives and one in the terminal display screen. Using the five layers just described, we see that such a program must consist of two sequences of layers one through four - one for the front or *visible* part of the program, and one for the back or *invisible* part of the program.

For a text editor, it might look like this diagram. (Readers with knowledge of the Open System Interconnect communication model will see a certain similarity between this diagram and those used in OSI.)



Remember we will be looking for a good place to make the *physical* cut between the functions in one (local, private) computer and another (remote, shared) computer.

Let us step back a moment, and look at the over-all form of an interactive program. I'll draw it sideways this time.



Thus the interactive program completes the cycle from concrete symbols, to abstract application concepts, and back to concrete symbols again. What you see on the screen is just the *presentation*. Moving information between windows must move more than the presentation; it must move the real data. A window manager package does not know where the real data is, so the application itself must implement the movement. If the application is split between the desktop and a server, this means that the protocol connecting the two must have a way to express such a movement.

Examples of application splits

In this section we will look at a few ways of splitting programs in distributed environments. Between each layer we have the opportunity to define a *protocol* and to split the layers apart and put them in different computers. For example:

- Corvus Omninet disk servers define a protocol between the invisible Style and Physical layers.
- VT220 terminals are controlled with a protocol between a host program at the visible Style layer and the terminal microcode at the Physical layer. This protocol conveys operations on the terminal's display-list.
- DECnet's Data Access Protocol (DAP) operates between RMS in the invisible Style layer and a program implementing the invisible Representation layer. The protocol conveys operations on the disk "storage style".
- The Videotex Access Protocol of VAX VTX (see chapter on **Examples**) operates in the middle of the Coordination layer, between the server and the Terminal Control Program. Both the server and the TCP implement Manipulation functions, one invisible, the other visible, on different representations of the same data structure.

Time-sharing with dumb terminals

"Traditional time-sharing" applications split the program between visible-layer one and visible-layer two. Essentially everything except the maintenance of the display-list is done on the shared computer. A "dumb terminal" is, after all, a computer. What differentiates a VT220 from a Rainbow PC100 is the particular program being run.

Communications bandwidth is not the bottleneck. The real bottleneck is in the server computer. Put another way, the problem for the server is not in speaking at high speed, but in thinking of something to say.

The responsiveness of traditional (dumb terminal) time-sharing systems becomes unacceptable when many terminals are in use at the same time. This is because the single computer is involved in making many low-level "visible" computations for each user, and any load beyond a certain point slows down those computations.

And looking at it the other way, keeping the computer busy with visible functions slows down the background computations for the invisible functions. If the invisible functions are not divorced from the visible ones, so they can each proceed at their own speed (invisible functions at a lower priority), they effect each other even more.

Putting more intelligence in the terminal would relieve the host of the visible computation load, allowing it to support more simultaneous users.

It is probably misleading to think of a computer terminal as a "peripheral device" at all. Peripheral to what? If it is peripheral, then it is not at the center. But in a distributed design, there is no center.

Personal computers

Applications designed for personal computers typically split the program between invisible-layer three and invisible-layer two. An examination of the numbers will show that the effective speed of such an arrangement is not much faster than a floppy disk, even on Ethernet, especially when there are a large number of clients for the same server.

If the PC has its own files, and the server is not merely simulating a PC disk structure, there will be two filing systems. Files will almost always be in the wrong place, because some of the programs to be run will be on the server, and some will be on the PC. Files that need to be shared must be in one place. In many cases, the end user is forced to be aware of two computer systems, with different command languages.

Distributed Applications as compromise

The problem is shortage of cpu power at the human interface, and we can best solve that problem by simply putting cpu power at the human interface. There is no need to put a complete computer system on the user's desk.

In a *distributed application* of the kind I am encouraging, the physical split between desktop and back-end comes right in the middle, in the Coordination Layer.

Predicatable response time - With the interactive component of the applications in the PC, short-term response time will be essentially independant of host load.

Single file system - There is only one file-system: on the host computer. Information is kept in a safer place, and is easily shared.

Cheaper desk-top - A fully-configured PC is not required on the desk. This saves money as well as not requiring "system manager" activities on the part of the user.

High-level vs. Low-level communications

There has been much interest recently in communication techniques that attempt to tie computers together at a very low level, in the Physical or Style layers of the model. The so-called *remote procedure call* is one of these techniques, as are ideas like *file servers* and *printer servers*.

This technique can work in a homogeneous environment, where all of the components are built out of the same cpu and are running the same operating system. But that is not the environment we are working in today.

When there is not an exact match in the low-level concepts of two enviroments, any gateway or translation between those environments is going to lose something. Either performance, functionality, or both are going to suffer, and the user of the heterogeneous system will be worse off than the user of the homogeneous system.

If communication between computers is in terms of abstract concepts, beyond the range of operating system convention, then there is no need for performance or functional penalties.

For example, consider the issue of "file types". The VMS file system maintains some "type" information about each file. The records in a file may all be the same specified length, or may be of varying lengths. In either case, a record may have implicit "carriage control", explicit but encoded carriage control, fixed prefixes of unspecified control information, or no special format at all.

This is all very well, but what if I want to store a FORTRAN source program? What is the proper VMS file format? Given that we pick one, how can VMS convey enough information to, say, an Ultrix system, so that the file is stored in the appropriate format *for FORTRAN source files* there? Too easy? Make that word-processing documents, and have the other system run IBM's MVS operating system.

Design in abstract terms first

If the same abstract concept exists in both environments, communication between those environments should be in terms of the common concepts, not low-level issues like record formats. Each environment is then free to process those concepts in the way that makes sense locally.

If a distributed application is built assuming that both components are on the same kind of computer, it may be very difficult later to have the components on different kinds of computer and still deliver the same performance and functionality.

Recapitulation

This chapter presented the layered function model and its application to distributed interactive programs. Important concepts were the difference between visible and invisible functions, and the observation that what makes one program different from another is in the most abstract functional layers.

We also looked at the results of making inappropriate splits between components of a distributed application, and why the communication between components must be at abstract levels.

In the next chapter we will look at some human factors issues and how a distributed implementation can help.

Chapter 3
What the user sees

Now that we have decided that it is possible to have a single piece of code handle the human interactions for all applications, we should consider whether it is acceptable, or even desirable, to have a single interface. This is important, because the separation of function necessary for distributed applications is feasible only if there are not a lot of different interfaces that have to co-exist in the terminal, PC, or whatever.

There are human factors things to consider in designing a user interface package, but there are other issues as well, and they all affect each other. If you design something with the user as the primary contraint, you end up with perhaps a different architectural model than otherwise.

No matter how nice a "user interface" package might be, it won't be as good as it could be if you graft it on to the front of some random program that is not itself well behaved.

Building a *technically* efficient program is always a concern, but we also have to consider whether it is *operationally* efficient. That is, does it let the user work as efficiently as he could, or does it slow him down or make it hard for him to understand what is going on?

Clearly, different styles of interface should be available for different markets or terminal equipment capabilities, but why should an individual user have to put up with different interface styles when dealing with the varying aspects of the job at hand?

We should not approach the task of designing the visible parts of a computer program looking for a way to "interface" the user to the computer, as though the user were just another component under our control. A company buys computer equipment to increase the productivity of its employees in accomplishing their jobs, so it should be our ultimate goal to increase the user's productivity.

The user must feel in control at all times. This does not mean just that the computer waits for commands from the user. If we put an untrained airline passenger into the pilot's seat, that passenger will be in control, but will not *feel* in control, because he will not understand what is going on, or what is expected of him.

We don't want to be too constrained by what the user has seen in other computer products. We want to be able to lead the user "toward the light". But if we force a strange way of working that is driven by the way we choose to implement something, we create an unnesessary wall of "learning curve" between the user and the product.

If there is a single user interface, a person will become so familiar with that one style that he will become fluent in its use. And he will be in a supportive environent because his neighbors in the workplace will be using that same interface, even if they are doing different jobs.

What does a terminal terminate?

In computers, "terminal" is the boundary of the computer. [1] A human being sits on the other side.

> Terminal: Pertaining to, situated at, or forming the end or boundary of something. A terminating point or limit.

But the printers have humans on the other side, so it makes sense to look for a more general term.

> Peripheral: (1) Of or on the periphery.
> (2) Relatively unimportant.

I think we can agree that meaning (2) is not appropriate. Maybe to a designer of big central processors a VT220 is unimportant, but certainly not to the person sitting in front of it. And if it is important to the user, it should be important to us. Lets look at (1), and what it means to be "on the periphery".

> Periphery: (1) The outermost region within a precise boundary.
> (2) The region immediately beyond a precise boundary.
> (3) A zone constituting an imprecise boundary.
> (4) Perimeter.

The first 3 look good, especially (3) when we start talking about workstations. And this stays with historical computer usage where the world ends where the electrical signals change to light, sound, or motion. I do not think this is a healthy point of view any more, because users are not interested in where the electrical signals start or stop. Those impulses are not important to the job at hand, and the fact that some of the job is inside the computer and some of it is outside should always be in our minds. Consider this:

> Gateway: Something that serves as an entrance or means of access.

The computer definition of gateway is similar: a point of access between two domains, the domains having different rules for name management, security management, coding, etc.

Certainly the rules and procedures on my side of the VT220 are not the same as those on the the other side, yet we want the richest possible communication between my side and the computer side.

---

[1] Definitions are from the *American Heritage Dictionary of the English Language*, published by Dell.

When we think in terms of gateways, we are acknowledging some responsibility for what happens on the other side, and I think that is a point of view we should encourage in the design of software products, what with the importance of "human factors" these days. The semantics of "gateway" is rich enough to encompass the most powerful workstations as well as the simplest dumb terminal. The word "interface" suggests only the surface where two domains meet (in the mathematical sense of "surface"), and does not suggest the processing that takes place in the kind of equipment we are talking about. Leave "interface" to the board-level hardware engineers.

A computer will be perceived as "difficult to use" if it forces the user to make conscious decisions to manipulate information that he is only aware of at a sub-conscious level. Our goal should not be to make the computer "user friendly", but to make it *inconspicuous*. A good computer gateway is transparent to the processes using it.

All is One

One of the central beliefs of many eastern philosophies is that "All is one". Paraphrasing for our purposes here, "All applications are one". This is *not* the same thing as is meant by the name of ALL-IN-1, DEC's office product. In ALL-IN-1, all of the application programs or functions are in, or are available through, a single *product framework*.

The point I want to make here is that the distinction we make between different office applications is a result of the way we have built them, not in anything inherent in the functions to be performed. We have optimized each *view* of the user's problems, resulting in separate programs. Then we have the task of putting these programs back together under a single interface style, when we could have done it the other way around and designed the common interface *first*.

That all applications are really the same is what we are constantly learning whenever we talk to customers about what they want. The *perceived* differences are simply examples of Pirsig's "knife" being used. The users are romanticists and the engineers are classicists, and finding the common ground between the two views of the world is not simple.

There are two issues here: the first is the mental frame that each uses when looking at the world. The second is the object of the contemplation. Simply making the distinction between the two views of the system does not get us very far. We have to understand a bit about why the differences exist and if they really are differences.

One of the real differences is in the role that the system plays for any particular person. To the "user", the system is a tool, a means by which some end can be acheived, an end which has meaning only in a greater context. To the engineer, the system itself is the object of attention. It is its own justification. It is a thing which needs to be understood as an end in itself, distinct from its utility or role.

As a tool the system is an extension of the user's self. It is, or is intended to be, an augmentation of the user. And, as a distinct augmentation, the system has its place in a universe of many other distinct augmentations. The user views these things as a set of all things outside of himself that can become, for a moment, part of himself in order

to accomplish some purpose. With most computer systems, the user is "one" but the extensions are "many" - a different extension for each function. Perhaps we can do better than this.

The search for one-ness

"Becoming one with the environment" seems to be a constant in many of life's experiences. A child must, over time and by experience, learn to differentiate between that which is itself and that which is not. Self awareness is largely the result of success in that differentiation. There is a certain amount of tension that is experienced in learning about this division. Then people spend their lives joining groups, assuming religions, and taking analysis in a never ending attempt to become one again: to end the loneliness.

Sometimes, users are unconsciously limited by the systems, changed by them at the same time they are augmented by them. This is probably the case with text editors. A long time user of the EDT editor program reports:

> "My entire concept of text is largely limited by the capabilites of EDT as I have learned them. I do not think about EDT when I use EDT: I simply have thoughts and express them via the keyboard and my fingers. There are all sorts of new ways of thinking and new modes of expression that I have accepted after years of 'becoming one' with this editor. It is to the point where I find myself feeling claustrophobic whenever I try to use paper and pencil.
>
> "Now, EDT is not the perfect editor. But to a certain extent it has become part of me in that I now accept it as a natural extension of myself. It is an essential component of the means with which I communicate with the world. I think this is in part why the debates between the supporters of various editors are so emotional. What is going on is not just a simple objective comparison of different tools. The discussion is about who the users are and how they express themselves."

Changing contexts considered harmful

One of the big problems with the kind of systems we build is that they emphasize their *separateness* from the user. The user must be constantly aware of what the system is and needs in order to acheive the user's goals. The user also needs to be constantly aware of the limits of the system as distinct from the user's limits. Thus, the users are constantly choosing between temporary extentions of themselves. Perhaps we could design programs in such a way that we eliminate the need for making these choices and thereby reduce the awareness of self as separate from the computer.

In ALL-IN-1 when we integrate Datatrieve (DTR) and other tools into the editor an exciting thing happens. Suddenly, the distinctions between the different tools are lessened. (Unfortunately, the differences in syntax still make the distinctions obvious.) This integrating of the tools is exciting because it means that I am extending my Self.

Put another way, we are am extending the capacity of the tool that we have allowed to share a part of our selfness. We can now use a data analysis tool to extend our capacities. The editor becomes capable of handling the same sort of integrated thought that we do inside our heads. Not only can it help in recording (or remembering) the data, it

also helps in generating it. We *could* do this without the integration, but if we had to leave the editor and then run DTR on its own, create files, and include them back into the document, we would become entirely too aware of the separate worlds, the separate tools that we are using. We begin to see again the little islands of other things which are not us. We begin again to deal with "Me" vs. "the other".

We want the user to remain focussed on the business problem at hand and be unaware of the tool. When we make the user leave the editor to run DTR, he has to back away from his problem and see the computer again, and Computer Anxiety has a chance to creep in. This need not be the case. If we build our programs so that the user is not forced to be aware of himself as distinct from the computer, we reduce the opportunity for him to be aware of the computer at all. Remember, our goal is *transparency*.

Why just one

We do not conciously make distinctions between the different capacities of our own intellects. For instance, we are usually not really aware that at one moment we are "calculating" and at the next "analysing" or "communicating". Perhaps, when we contemplate the past, or plan the future, we can see the distinctions, but we are not aware of them when we are actually acting. The best tools, the best systems, will be those that do not require us to make these distinctions when using them. The closer these tools can come to reflecting the way we think, and amplifying that thought, the more effective they will be and the less tension they will cause. Since we are *one* the best tools will also be *one*. Then, there is hope of a coming together.

This is one reason we must strive to acheive such goals as "The network is the system." For the network to be maximally useful it must become *one*. We must not be able to notice the boundaries between the components that are combined to construct it. We must perceive a *one* that we can then begin to become one with.

Fluency

Fluency is having facility or aptitude in the use of something. The result is an effortless, polished excecution of the task at hand.

You can only begin to type rapidly when you stop thinking about your fingers on the keys and start thinking about what you want to say. You can only play a good game of tennis when you stop thinking about getting your racquet on the ball. You can only speak a language fluently when you stop trying to translate from your native lanaguage.

The tennis player does not think (consciously) about how to hold the racket or when to swing; she thinks about strategy. The musician does not think (consciously) about individual notes, but about tempo and phrasing. You do not think (consciously) about what your feet are doing as you walk down the street. In fact, if you do think about your feet, you will stumble.

So not only is fluency possible at a subconscious level, but it is nearly *impossible* at a *conscious* level. The conscious mind, because it is focused, cannot think about enough things fast enough to maintain fluent performance. Fluency demands sub-conscious thinking, and without fluency, productivity will not improve.

A book to read on this subject is "Inner Tennis", by Timothy Galwey. He has written "Inner" books on other subjects as well, including skiing and business management.

When the computer makes the user stop to think about what he is doing, especially when his mental model of what is going on does not match system behavior, he will slow down. (See also Hersh and Rubenstein's book on "The Human Factor".)

The conscious and unconscious are not two separate worlds; there is a gradual fading from one into the other, and where we consider the boundary to be at any given moment, and how sharp the boundary is, can change according to circumstances. Once we have learned how to perform some mental task, and practiced it enough, the doing of that task starts to drop into the subconscious.

This is not to say that you can *learn* to do things without conscious thought. You really have to focus your attention while you are still learning. But while you are slaving away, the subconscious part of your mind is "looking over your shoulder", and after a while it says "ok, I can do that now", and from then on it takes less and less conscious thought.

(The mind can be subconsciously creative, as well as analytical, if it has been previously provided with the knowledge and techniques to use. Somebody once asked Albert Einstein how he came up with his theories on relativity. His answer was, "it just came to me." This is *insight*. The conscious mind may later work out the detailed description and justification, but the original idea was developed subconsciously.)

Familiarity breeds fluency

We spend so much time in editors that we become fluent and merge with the editor, as was reported by the user of EDT. If you do not become fluent, your productivity will not increase.

If there is a single interface to all applications, so that the person re-uses the same set of tools no matter what the task, that person cannot help but become fluent in the use of those tools. Also, people in neighboring offices, using the same tools, are available for support.

A world of things

When we look out at the world, we see *things*, not actions. We know the actions we are capable of performing, and we just do them as needed. The world has the things, but the actions are internal to us.

"I seem to be a verb." - Buckminster Fuller

But when we sit down at a computer and look at a menu, often we are presented with a list of *actions*.

One of the nice things about the Apple Macintosh™ is that when you use it you are not thinking verbally. The wastebasket is just there, and you drop your old document into it; you don't have to think of the word "DELETE".

If a computer puts names onto things (like commands), the user must learn those names, and the documentation must talk about them. If the *user* puts names onto things (like documents), no learning, documentation, or translation of documentation about those names is required. A manufacturer of paring knives does not have to come out with foreign language versions of his product.

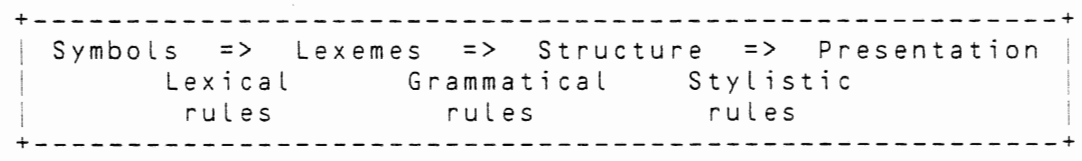
Representation of Information

The central layer in the Abstraction Model is the Representation Layer. It is here that what the computer "knows" about the problem at hand is stored. The Style Layer uses the information stored in the Representation Layer to build the visual display. The functions of the Manipulation Layer are defined in terms of the concepts stored in the Representation Layer. Obviously, the choice of what goes into this layer is going to be important.

Information in a language is represented by a linear sequence of *symbols*. The sequence is analyzed into *lexemes* by a set of *lexical rules*. ("A", "B", and "C" are symbols. "DOG", "THE", and "HYPOTHESIZE" are lexemes in English, whose lexical rules say that a sequence of letters bounded by non-letters is a word.)

Lexemes are then interpreted according to a *grammar*, resulting in a *structure*. The structure is NOT inherent in the lexemes, but is only a particular view of the lexemes imposed by the grammar. A "prepositional phrase" and a "paragraph" are elements of structure in English.

A *style* is the collection of rules by which an external visual *presentation* of the structure is generated. A rule that paragraphs should be separated by one blank line is a style rule.



Since the presentation is manifested to the user by symbols ("A", "B", and "C" again), we have finished where we started. This progression from concrete to abstract to concrete came up in the last chapter when we examined the internal structure of programs.

Consequences of the storage format

If the storage represents the stream of lexemes, different grammars can be employed to analyze the information for different purposes. If the storage represents the structure, the grammar cannot be changed, only the style. If the storage represents the presentation, not even the style can be changed.

Is an editor program to work on the presentation, the structure, the lexemes, or the symbols? Which does the user want to edit? Perhaps he wants to work at different levels at different times. But he is not *conscious* of these levels, so how do we work at the right level without forcing the user to verbalize what he wants?

Human beings are very good at analyzing linear streams for their structure and in generating streams from a structure. In fact they are so good at it that most people are unaware they are doing it, and being made to think about it slows them down.

As an intermediate step in editing, the sequence of lexemes may not be grammatical. Should the editor prevent this from happening, or merely detect when it does happen and alert the user?

Recapitulation

We want to reduce anxiety in the user. A good way to do this is to present a "mythology" that is consistent with the way the system behaves [Hersh and Rubenstein].

We want all applications to present the same mythology. Yet different kinds of users might need different styles of interaction. If we just publish a standard for user interaction and require all application developers to adhere to it, we will end up with as many implementations as there are applications.

This lead us to the conclusion that the "user interface" should be a separate component, with a standard interface to the application software.

A separate computer must be thought about. If you think about your tools, you can't be fluent in their use.

In the next chapter, we will look at some implementation techniques for actually building efficient distributed client and server programs.

Chapter 4
Implementation techniques

In this chapter we will look at various programming techniques that are helpful in actually building the components of distributed applications. Some of these techniques will require cooperation from underlying service layers (like operating systems) in ways that current products do not support. If products for end users are to be based on standard components (to avoid duplicate efforts in engineering, services, and sales), how are we to determine what those base components should be?

System engineering has to be more than designing products out of existing parts. It has to include determining what those parts should be. If the available parts do not fit together well regarding performance, cost, or functionality, pouring glue over the mess will not make them into a system that is easy to describe, sell, install, learn, use, and evolve.

Here is a list of the basic techniques I have been leading up to in this paper. The rest of this chapter is an in-depth discussion of each one.

- Asynchrony between layers
- Pipeline operations
- Multi-thread servers
- Communicate high-level concepts
- Extend sharing backward
- Avoid time-sensitive server operations
- Minimize procedural programming on PC's
- Separation of visible and invisible functions

Asynchrony between layers

Make the code in each functional layer operate asynchronously from code in other layers. If you used the data flow analysis technique described in chapter one, you are halfway toward implementing asynchronous code anyway.

Pipeline operations

Pipelining is particularly useful in servers, where as the number of clients increases, it quickly becomes desirable to overlap the I/O operations on behalf of one client with the computations or I/O on behalf of another.

In a pipelined implementation, each request is broken down into several steps, each involving a single I/O operation. The operations are all done asynchronously, and as each completes with an AST, the next step is initiated.

In situations where there are several disk files, and some requests (or steps of requests) use different files, servicing these requests can proceed in parallel. Also if the disk controller would benefit from having more than one operation to perform at a time (such as an HSC-50), setting up parallel pipes can increase effective disk speed.

One aspect of this is that a simple *remote procedure call* (RPC) communications scheme is generally not appropriate, unless the RPC provides for asynchronous returns. With a multi-threaded pipelined implementation, the RPC mechanism cannot use the server's single stack to hold the stub call context. A way around this is for the server to return from the first remote call immediately, then call back to the client later when the results from the operation are available.

Multi-thread code

Hand in hand with the pipelined technique is the concept of *multi-threading*. Multi-threaded code can handle requests simultaneously from more than one "context". Again, this is particularly useful in servers.

If a functional unit is to be multi-thread, each function call must pass a "context handle". Where the functional units are separated by a communications link, it is a good idea to maintain sufficient context information on each side that only the handle needs to be passed across.

The context for all the clients is kept inside the one program.

Advantages

The advantages of a multi-thread server are numerous.

Opportunities for data sharing. If more than one client are getting at the same base information, the server only needs to work with one copy of the information.

Less task-switch overhead. Because all operations of the server are contained within a single host task, there is no overhead from the operating system switching contexts between users.

Fewer locks required. There is often no need for a locking mechanism to prevent attempts at simultaneous changes to data structures, because the code never gets interrupted.

Simpler account management. Because the server acts on behalf of all clients, the host operating system does not have to know about each individual user.

No start-up delay. As the single server task will be running all the time, and there is no need to start up another task as each client request comes in, any host operating system delays in task spawning are eliminated.

More predictable response pattern. With the small transaction. pipeline model of processing. it is reasonsble to talk about the number of transactions per unit time. an easily measurable parameter. And since each resource saturates at its own rate. it is easier to come up with an accurate mathematical model of performance under load.

The multi-thread technique can also be used in the client program. A multi-thread client is a single task which contains context for more than one simultaneous operation. For example. a multi-window text editor might treat the activities in each window as a separate thread.

Disadvantages

There might be a mismatch with underlying security facilities. This comes about because the server is running in its own "account" on the host system. with its own set of rights and priviledges. Yet it is acting on behalf of several distinct users. each with potentially different rights.

If the server has to call on services which do not allow for this possibility. the implementation might be forced to be very inefficient. The server may know the rights of each user. and can keep them straight. but if the called-on service is not prepared to handle multiple contexts in a single activation. additional tasks may need to be started.

Implementation

In Appendix A you will find a description of a software package called DASL (the Distributed Application Services Link) which handles the management of multiple contexts of communication links.

Most multi-tasking operating systems will attempt to give equal cpu time to all tasks of the same priority. If there are two compute-limited tasks. the operating system will periodically interrupt one to run the other. This can cause problems if there are common data structures between the tasks.

Since the application designer knows more about the nature of his transactions than does the operating system. he can put the points of loss of control at places in the program that will not cause damage to the data structures.

If there are two tasks to be completed. interrupting one in order to run another does not improve the response time of either one. All it does is make them both degrade equally.

Communicate high-level concepts

Protocols and gateways are very difficult to build between domains at very low levels. Nobody would attempt to gateway between a VAX and an IBM 370 at the cpu backplane level. It is much easier to build gateways at higher levels of abstraction. where you have enough context to know what goal is intended and not just the way this or that domain goes about accomplishing it.

For example, the VAX RMS disk structure only stores attributes about the physical representation of information (fixed length, variable length, this or that carriage control) and nothing about the information itself (source, object, executable, DDIF, postscript). When you try to build gateways or servers, and you don't know what the high-level concept is, you can't deduce it from the low-level implementation.

The proper way to build protocols and other interfaces between functions, especially in a distributed environment, is to encode only the high-level concepts, at let each domain represent those concepts however it wants.

Extend sharing backward

Separate visible and invisible operations

Avoid time-sensitive server operations

Remember that any operation the user has to wait for is visible to some extent. It is possible to make some operations on a server overlap in such a way that, although they take finite time to complete, do not actually cause a long wait.

Some examples:

- Pull in the next page of a document being edited *before* the user scrolls off the bottom of the current page.
- Overlap sending back changes to the last page with pulling out the next page.

As we move back from the user, the bandwidth requirements to deliver a given responsiveness to the user *decrease* as far as the support of that one user is concerned. There is an opportunity for a step reduction in bandwidth wherever memory and computing power is located in the system. An analysis of the bandwidth requirements in the data flow model of an application can show up the good (and bad) places put a communications link to a shared resource.

Display list refreshes the screen - A raster display monitor capable of displaying 24 rows of 80 characters at a resolution that is comfortable for long periods needs 240 scan lines of 800 pixels per line. At 60 frames per second, this means that the video circuitry of the terminal must generate pixels at a rate of 12 million per second.

In a character-cell display, the video chip provides the 100 pixels required for each character, so the text display list in video memory typically needs only 8 bits for each character to be displayed. Therefore information is read from the display list at an average of only 960 kilobits per second. Still not a good place to put a communications link.

Style code updates the display list - The display list only needs to be changed as the visible part of the current document is changed. The worst case might be when the entire display list (say 2000 bytes) must be rebuilt in one second, requiring 16 kilobits per second. A more typical case is where one text row (say 80 bytes) is changed 3 times per second as the user types in insert mode. This requires only 1920 bits per second, assuming no assist from the display-list layer.

Some dumb terminals have an "insert mode" in which microcode in the terminal takes care of shuffling the row over for each new character. These terminals need only 40 bits per second, until the line overflows. The style layer (in the host for dumb terminals) must keep track of when this will happen.

Exchanges between visible and invisible representations - The visible text buffer representation might not be large enough to hold the entire document at once. Assuming the user wants to see the beginning of the document first, the we need deliver information no faster than it takes to fill the screen in one second, or 16 kilobits per second, in a short burst. Taking an average document size of 5 pages of 4000 bytes each, and an average working time per document of 20 minutes, the *average* speed requirement might be $5 \times 4000 \times 8 \div 20 \div 60$ or only 135 bits per second, assuming the entire document needs to be seen by the user.

The following table summarizes the information rate in the five communication links just described. Speeds are given in bits per second.

Link between components	Average	Peak
Video chip to screen	12M	12M
Display list to Video chip	960K	960K
Visible buffer to Display list	1920	16K
Invisible to visible	135	16K
Invisible rep to disk	135	16K

From this table it looks like a communications link running at 19,200 bits per second, placed between the visible and invisible components of a text editor, will deliver adequate performance. Provided the visible maniipulation and style code is powerful enough to maintain the screen for serveral minutes of editing at a time, the load on the back-end will be minimal.

Some functions require a higher bandwidth again as we get closer to the disk storage subsystem. For example, a text search unaided by pre-built indices must read great quantities of data from the disk in as short a time as possible. But since this search is an *invisible* operation, it can be built on a server, close to the disk. There is no need to convey the raw data to the visible portion of the program.

Minimize large procedural programming on PC's

Use lightweight protocols

Lightweight protocols are protocols designed to support specific application services and to do so with minimum run-time overheads.

An implementation that improves response-time by sending fewer bits between components will be cheaper in the long run than one that requires a faster communication channel. This is because hi-speed interconnects are more expensive than low-speed interconnects, and also because the principle bottleneck in a client/server architecture is the server's inability to communicate meaningfully at high speed with a large number of clients. *Fewer bits* is cheaper than *faster bits*.

Recapitulation

In this chapter we have examined various specific programming techniques and tools that help in the construction of distributed applications.

In the next chapter we will examine some real distributed application systems and see how they used these techniques.

Chapter 5
Examples

In this chapter we will examine three examples of distributed applications and how they are implemented. The three applications are a videotex service, a shared filing system for personal computers, and a text editor.

For each application a description will be given of how the protocol passes functional requests between the client and server components.

Videotex Service

Videotex is a means of delivering pages of textual or graphic information with a very simple menu selection scheme. It usually also includes the ability to use simple transaction processing programs through a combination of menus and forms. VAX VTX™ is DEC's videotex implementation.

Separation of functions

For videotex there are three main categories of functions: those pertaining to the user at the terminal, those pertaining to the retreival of fixed information from a videotex database, and those pertaining to special application programs. VAX VTX puts these three functions into three separate components.

Terminal operations - The Terminal Control Program is the only component of VAX VTX that knows what kind of terminal is being used. It translates keyboard input into *generic codes* that get passed on to the database server.

The TCP is also the only component that knows how to clear the screen of the terminal, how to display error messages and prompts, and so on. If something goes wrong, the server sends a generic code to the TCP, which then takes the appropriate action. Since the prompt and message strings are in the TCP, it can display the messages in any of several languages.

Although it is possible to do presentation code translation here, none of the existing VAX VTX products do so. It was decided that performance is better if the pages are translated just once, and a duplicate database kept for the different terminal types.

Database operations - The architecture or VTX identifies three roles for a database server: the *home* server, the *current* server, and *intermediate* servers.

The home server is the one to which the Terminal Control Program is directly connected. For any given user, there is only one home server. This is where accounting is done for that user, where authorization information is stored, and where the "Main Menu" function always returns.

The current server is the one from which videotex pages are retreived. The home server can also be the current server, but the home server may connect to another server and pass user requests back. The remote server then becomes the current server *for that user's session*, until the link is broken and the home server takes over again, or until the current server passes control on to yet a third server. The current server is sometimes called the *rearmost* server.

When one server passes control for a session on to another server, the data flow for the session still passes through the first server, and if the link is broken, control returns there. If a server is neither the home nor current server, then it is said to be an intermediate server. All it has to do is keep passing messages back and forth until one side or the other disconnects the session.

Using a multi-thread database server leads to the need for a multiplexed communication link between servers, as there is no particular advantage to having tens or hundreds of network links between two tasks. The DASL communications package was derived from this requirement.

Application operations - The same protocol used between terminal control programs and the server is used between the server and "back end" application programs. The TCP and server work together to provide a request-level forms and menu interface to specialized transaction processing programs.

Access control

VAX VTX uses a "rights forwarding" authorization model, to avoid the disadvantage of multi-threaded servers described in the preceding chapter. Each user's *home server* has a list of the *closed user groups* to which that user belongs. Each page in a server database can be tagged as belonging to such a group. Only members of that group may retrieve the page.

When one server connects to another, acting as agent for the user, a list of the groups of which that user is a member is passed along so that, if the two servers are set up to use the same group codes, the called-on server will immediately have that information available. And the list of groups is stored at only one place in the network: on the user's home server.

An additional mechanism called *base page determination* is available to direct incoming connections to particular places in the server's database. Various attributes of the user can be matched against a template to control which pages will be made directly available to that user.

Single protocol

The generic requests which tie the three components of VAX VTX together are encoded in a single communications protocol, the *Videotex Access Protocol* (VAP).

INPUT Conveys user requests to the server

OUTPUT	Conveys information to the user
LOG	Carries accounting information from back-end application programs to server logging files
FORCE	Used by application programs to cause a particular database page to be displayed to the user
FORM	Used by application programs to cause a data collection form to be displayed to the user, information collected, and returned to the program
WHERE	Used by home servers to request context information from other servers
HERE	Used by rear servers to report context to the home server

The INPUT messages convey all direct user input back to the servers and application programs.

MAIN	Request main menu page
CHOICE	Select from a menu
BACKUP	Go back to last menu
NEXT	Go to next continuation page
PREVIOUS	Go back to previous continuation page
DIRECT	Request a page by its number
KEYWORD	Request a page by its keyword
FORMDATA	Report field contents from filling out a form

The OUTPUT messages convey all user-visible information from the servers and application programs.

PRESENT	Displays database information
Errors	Reports various failures
FORMDEF	Describes a form layout

Implementation

The VAX VTX database server makes use of the full pipelined multi-thread technique described in chapter 4. The DASL package (described in appendix A) takes care of manageing the communication links.

To increase performance when an HSC50 or similar disk controller is available, VAX VTX uses multi-server queues for requests for pages from the database.

ALL-IN-1 Workstation Server

The File Cabinet server of A1WS is a single VMS process, and as it serves several users at once, and since it cannot simultaneously be in many accounts, it can not rely entirely on the VMS protection scheme. There was a product requirement that a VMS account not be required for each user, so an independant access control mechanism was required. Since we had to build one anyway, we designed it to have exactly the features we needed in an office filing system.

In particular, the ability to share documents and folders was built into the server. All sharing and access control is handled by the File Cabinet server.

Protocol summary

FIND	Select one folder from a group of folders, or one document from a group within a folder.
CREATE	Create a new folder.
DELETE	Get rid of folders and documents.
READ	Transfer a document to the client.
WRITE	Transfer a document to the server.
MODIFY	Change various attributes of a document.
ACTION	Invoke special outboard application functions.

The outboard functions available though the ACTION message are:

GIVE	Gives a reference to the current document to some other file cabinet user
PRINT	Queues the current document for printing
MAIL	Mails a copy of the current document to one or more other users anywhere in the network

Implementation details

The File Cabinet Server uses the same pipelined multi-thread concepts as does the VAX VTX database server, although without multi-server queues. (The goal configuration was much smaller than a typical VTX installation, and the load needed to justify multi-server queues would not be present.)

This server uses the same DASL package as does VTX, although the first version does not make use of the agent concept to tie servers together.

Distributed Editor

This is a protocol design for a possible future editor. The protocol is designed in such a way that the server (where the files are) is not aware of things like cursor position, insert mode, screen layout, etc. The FORM of editing does not affect what goes on in the protocol, which is only concerned with the FUNCTION of editing.

This is accomplished by having every command carry all the state information necessary to give the proper effect. In terms of the model, all *visible* concepts stay in the front-end, and the server deals only with *invisible* concepts.

Protocol summary

One DASL session corresponds to one "document" being edited. The protocol has provisions so that, if two documents are being edited at the same time by the same workstation, using the same server, data may be moved between the two documents without the need to copy the information out to the workstation and back again. This is indicated by a session reference in the COPY messages..

Control - These messages set up and terminate workspaces, and manage the copying of information across the link.

Open	Creates a workspace and binds it to a particular backing store or program.

Close	Disolves a workspace.
Find	The server will search the workspace for a string, starting at a particular bucket. A Failure or Success report will be sent in return.
Request	Specifies a range of buckets which are to be sent from the server to the client. The server will send one or more Write Bucket messages.

Ranges - These messages operate on ranges of one or more complete buckets.

Write	Insert or overlay bucket
Remove	Remove contiguous buckets
Copy	Copy (destructively or not) contiguous buckets, inserting or overlaying at destination.

Strings - These messages operate on strings of bytes within a single bucket.

Write	Insert or overlay bytes
Remove	Remove contiguous bytes
Copy	Copy (destructively or not) contiguous bytes, inserting or overlaying at destination.

Reports - These messages report activity by the server.

Failure	Reports that an operation has not been able to complete normally.
Success	Reports that a FIND operation has located a matching string. Supplies bucket number and byte offset of the match.
Summary	Reports general information about a workspace, such as the number of buckets present.

A position in a workspace is given by a bucket number and sometimes a byte number. The first bucket in a workspace is number one, and the first byte within a bucket is numbered one. When a command refers to bucket 65535 in a workspace, it really means "the last bucket", whatever its number.

Chapter 6
A Framework for many Applications

All the preceding chapters have been analytical and conceptual. presenting analytical tools. analyzing the nature of applications and then how best to distribute them across processors while achieving the maximum advantage from that distribution. We have looked at some existing programs in the light of these analytical tools. Now it is time to be creative. In this final chapter I will present a single design calling on most of the concepts presented earlier.

We will look at a way of building a framework on which *many* products can be built. The framework will embody the layered model from chapter two. and will take care of the operational details. The application programmer then need only deal with what is special about the particular problem at hand.

An interpreter in each layer

In each functional layer. we should be able to define a suite of primitive functions from which more complex specialized functions can be constructed. This has already been done for some layers - wherever a protocol or command language has been defined.

Once the functions are identified. we can define a *programming language* in which to describe how these functions are used within the layer. Communication and coordination *between* the layers will be taken care of by the underlying framework.

Event-triggered procedures

"Programming" in this environment can be reduced to writing small action routines which will be invoked by the framework when certain events occur. These events might be a keystroke. a mouse click in a particular place. a message arriving from another layer. and so on.

Primitive Style functions

The visible style layer combines the functions of a forms driver. a window manager. and a text formatter. Taking a step back we can see how these are three different implementations of the same concept. So if we capture the *concept* in the primitive functions. the application designer should be able to put the functions together to get a wide range of effects.

Primitive Representation functions

Primitive Manipulation functions

Anyone who has used a DEC WPS word processor or VT173 terminal with "User Definable Keys" has had a taste of a programming language for manipulation functions. Parts of DCL ("Digital Command Language") and the ALL-IN-1 script language can also be considered examples of such a manipulation language.

A user-defined-key is a small program, triggered by an event (a key press), and executed only in the context of some larger process.

Implementation of the Visible Interpreter

Appendix A
DASL

DASL is the Distributed Application Service Level protocol for communicating between the distributed components of an application. The DASL interface library implements this protocol, as well as provideding context management and communications management functions.

The concepts in DASL were developed as part of the VAX VTX product, but have been split out and made more general.

Application definition

A single DASL server task can support more than one application at the same time. The programmer must provide DASL with a description of the communication and context requirements of each application. This information is kept in an *Application Control Block*.

Protocol overview

The DASL protocol [1] consists of six message types. Each message type has an inherent direction, expressed in terms of whether the message moves toward the client ("forward") or toward the server ("backward").

Connections are always initiated by the client. In fact, that is the definition of being the client - the one that initiates the connection. Connections can be broken from either end.

All DASL messages start with the same three fields:

COMMAND, one byte
: carries a message type identifier to indicate if the message is a CONNECT, DISCONNECT, etc.

REFERENCE, four bytes

---

[1] This description is for version 1.0 of DASL.

an identification of the session on whose behalf this message is being sent. It is composed of two parts, each of two bytes. The *Front Reference* carries a number by which the client identifies the session, and the *Rear Reference* carries a number by which the server identifies the session.

DATA_LENGTH, two bytes
carries the number of data bytes in the remainder of the message.

Each particular DASL message (described below) may have additional fields, and then the particular application can add more fields at the end.

CONNECT message - The CONNECT message (backward-going) tells a server to begin processing input from a particular session.

APPLICATION, one byte
Indicates which application protocol will be used over the new session link. Values up to 127 are assigned by Bostac. Values of 128 and above are for customer use.

VERSION, one byte
Indicates the version of the DASL protocol being used.

The Rear Reference must be zero in a CONNECT message.

CONFIRM message - The CONFIRM message (forward-going), sent in response to a CONNECT messsage, tells the client a little about the environment in which the new session will be operating.

VERSION, one byte
Indicates the DASL protocol version that will be used by the server.

The value of the Front Reference must be the same as was used in the CONNECT message for which this is a confirmation. The Rear Reference must be non-zero, so the client will know the reference number to be used in future messages for this session over the same link.

All subsequent messages for the same session over the same link will use the same value for REFERENCE.

DISCONNECT message - The DISCONNECT message (backward-going) tells a server to stop processing for a session. The client may send a DISCONNECT at any time, asynchronously. After receiving a DISCONNECT, the server must send no more messages for the session.

REASON, one byte
Indicates the reason the session is being terminated. Possible values (explained later) are APPL, OPER, and UNLINK.

BREAK message - The BREAK message (forward-going) has the same effect on the protocol as the DISCONNECT message, but is sent in the other direction.

The rear program may send a BREAK at any time, asynchronously. After receiving a BREAK, the forward program must send no more messages for the session over the same link (without first re-establishing the connection with a CONNECT message).

REASON, one byte
: Indicates the reason the session is being terminated.

If the reason code is APPL, the application can send additional information about the reason in fields beginning after REASON.

BDATA message - BDATA messages (backward going) convey purely application information toward the rear, that is, away from the end-user. BDATA messages do not have any required fields after DATA_LENGTH.

FDATA message - FDATA messages (forward going) convey purely application information forward, that is, toward the end-user. FDATA messages do not have any required fields after DATA_LENGTH.

Description of Reason Codes

The reason codes in DISCONNECT and BREAK messages are taken from the following list:

Symbol	Reason
APPL	Application requested BREAK
RESOURCES	Insufficient resources to create a session
PERMIT	User does not have permission
OPER	System operator requested BREAK
NOAPPL	Application not available
BADVER	Protocol version not supported
UNLINK	Comm link broken or not established

Note that the UNLINK code is usualy not transmitted over a communications link, but is faked up by the DASL support routines.

Context management

To make it easier to write multi-thread programs, the DASL library takes on the job of matching incoming messages to application context areas.

Communication management

Since every DASL message carries a context identifier, a large number of DASL sessions can be run through a single transport-level connection. Under DECnet there is no performance advantage to having a separate link for each session between two tasks, and there are several disadvantages.

Protocol procedures

To start a session -

1 Client sends CONNECT.
2 Server performs access checks.
3 If a session can not be started, server sends a BREAK message. Otherwise, the server Rear sends CONFIRM.

FDATA and BDATA messages *may* be exchanged before the CONFIRM or BREAK is sent. These are used by the application to collect additional information necessary for starting the session. (Such as passwords, etc.) But the first message exchanged after the CONNECT must be one of the forward-going messages, BREAK, CONFIRM, or FDATA, with the Rear Reference filled in.

To discontinue a session - If the client sends DISCONNECT or the server sends BREAK, the session immediately stops using the link over which the message was sent. The server can discard any context it was keeping for the session.

To exchange application information - Once the session connection is set up as described above, the client may send BDATA messages at any time. Likewise, the server may send FDATA messages at any time.

The DASL protocol provides no acknowledgement or flow control. If the functions inherited from the lower layers are not sufficient for a given application, that application must implement its own flow control on top of the DASL protocol.

Glossary

Client

A consumer of resources in a network.

Client-Server model

Cognitive workload

Co-computing

A form of distributed processing (see below) in which the two programs are in fact parts of the same program, and execute at the same time.

DASL

Distributed Application Service Link. A lightweight session protocol on top of which multiple application protocols can be built.

DECnet

Distributed processing

Processing in which programs are executed cooperatively in separate computers.

Lightweight protocol

A protocol designed to support specific application services and to do so with minimum run-time overhead.

Server

A provider of resources in a network.

VAP

Videotex Access Protocol. This is the application protocol used between components of the VAX VTX service.

WYSIWYG

What You See Is What You Get, pronounced "wizzy wig". A kind of editor program which makes the document on the screen look just the way it will when finally printed.

Bibliography

Whiteside, J., *et al, Usability Engineering Handbook*, DEC-TR-347, November 1985.

Galwey, T., *Inner Tennis*

Magers, C., *Systems that grow with the User: User Description Document*, DEC-TR-136, October 1980.

Pirsig, R., *Zen and the Art of Motorcycle Maintenance*. Bantam Books 1974.

Rubenstein, R. and Hersh, H., *The Human Factor: Designing computer systems for people*, Digital Press, 1984.

Saltzer, J., Reed, D., and Clark, D., End-to-end Arguments in System Design, *ACM Transactions on Computer Systems*, Vol. 2, No. 4, November 1984, pages 277-288.

Watts, A., *The Way of Zen*, Vintage Books (Random House), 1957.