PAPER #4016

The Legacy Continues: MPE/iX in an Open Systems World

Mike Yawn
Senior Consultant
Hewlett-Packard Commercial Systems Division
19447 Pruneridge Avenue MS 47UA
Cupertino, CA 95014
myawn@cup.hp.com

Abstract

With the changes that have transformed the computer industry in the past years, where does the HP 3000 fit? Is the move to UNIX unavoidable, or do customers have a choice? How can you make the right investments to run your business today, and still be confident that tomorrow won't turn your efforts into money down the drain?

This presentation will focus on Information Technology Architecture as a way of enhancing your ability to respond to the changes your company is facing. Through a well defined IT Architecture, MPE, UNIX, and Windows systems can peacefully coexist in an environment that preserves legacy system investments, while allowing the introduction of new technologies that may not be available for the legacy platform.

The role of open system systandards, client/server computing, and object-oriented technologies will all be covered as they relate to the creation of a successful IT Architecture.

What is an IT Architecture, and why do you need one?

Is calling a system analyst or programmer an 'IT Architect' just a case of title inflation, such as calling garbage collectors 'sanitation engineers'? What is the difference in creating an architecture, as opposed to designing an application? There are several aspects we will emphasize of good architecture design. Adaptability to future changes, in either the functional requirements of the application or in the data processing infrastructure, is part of the design of an IT Architecture. An IT Architecture also attempts to define and meet requirements beyond the functional definition of software requirements, in such areas as high availability, security, and performance. While it is true that the best systems analysts have always done these things, we think the distinction between enterprise-level strategic IT planning and application level design is a valid one.

If you're going to operate a single system, dedicated to running a single off-the-shelf application package, then you may not need an IT architecture. The instructions from your hardware and software vendors are your blueprints. While topics such as high availability are no less a concern with off-the-shelf software than they are with your own development efforts, the approach must be different. You must approach these areas from a system management perspective, rather than incorporating features as part of the overall application design. If you're going to running multiple applications, with each one independent of the others, the same is true: you have more work to do from a system administration point of view, but you still don't require the services of an "IT Architect".

Consider a case that's more typical of IT challenges of the typical IT department. You have a combination of application software packages, which may be in-house developments or third-party

purchases, and they need to interface with each other. The financial packages need to know how much inventory is in the warehouse and what payroll expenses are. The production scheduling program needs to know what the marketing forecast is. If you design interfaces between these based on the assumption that none of the applications will ever change, you're operating in the traditional programmer's role. But if you design interfaces knowing that the applications will change, the systems they run on may change, and the requirements for what data is shared may change, then you're thinking like an IT architect.

Now, we add in the complexities of today's IT challenges. The systems that you must interface are not homogeneous, but represent several different operating systems and vendors. They aren't all located in a single computer room, but are geographically dispersed. Some of them -- the servers -- are under control of the IT department, but the majority of them (clients) are on user's desks, where they may or may not be backed up and are almost certainly not secure. Software you do not control and have never heard of may be introduced onto these systems at any time. Business requirements are much more aggressive than they once were. Perhaps your system used to batch up transactions for interfacing on a weekly or even monthly basis. Now you're probably interfacing the systems on a daily basis, but even this may not be good enough for some applications. You may have to provide up-to-the-minute access to any of your company's data, regardless of what system it resides on, to any other system on an as-needed basis.

While there has always been some need for IT Architects, the changes caused in the industry by open systems have increased severalfold the complexity of designing an IT architecture. In the early days of the HP 3000, for example, most of the choices were simple. For languages, FORTRAN and COBOL were the most popular choices. For a database, there was IMAGE; KSAM provided indexed files. For the user interface, you most likely used VPLUS. Best of all, you were assured that all of these pieces would work together: the job of a 'system integrator' had never been heard of. Everything came from the same vendor, and had been designed and tested to ensure it all interoperated correctly.

Today, you can still make those same choices: all the products mentioned above are still available. But the resulting application will not be portable. It will not provide the ease of use of a graphical user interface, or the flexible inquiry capabilities of a relational data base. Your programmers' productivity will be less than with more recent tools that facilitate rapid prototyping and code reuse. So if you are starting out today, your choices are almost certainly going to be different. The products you choose will come from more than one vendor, and possibly encompass more than one hardware platform. It used to be that except for a few limited choices (COBOL or FORTRAN?), the tools you had to work with were preordained by the hardware platform you chose. Your job was then to develop and deploy the applications. Today, your job is to build the technology infrastructure: you should select the applications first, then the tools, letting those choices determine the hardware platforms required. It will be up to you to make the pieces fit, and work, together. Without an IT Architecture -- a blueprint for this infrastructure -- you aren't prepared to make the choices required.

You want to be able to pick the best tool for each job: the best program development environment, the best database management system, the best applications -- based on the criteria that matter to you. With a robust architecture, you can do this, and make all the pieces play together.

Why Standards aren't a substitute for a software architecture

Everybody wants Open Systems. But before you jump on the bandwagon, find out what it is you really want when you say you want open systems.. Is it application portability? Interoperability? The ability to change hardware vendors? Database vendors? Application vendors? Lower acquisition cost (which may be offset by higher operating costs)? "Open" has so many definitions that a blanket

statement of "we want open systems" says nothing. Define the criteria by which openness will be measured, and to what extent openness is to be favored over other requirements. (Will you still prefer the "open" solution if it is 20% slower than the "proprietary" solution? If it fails twice as often?)

There is undoubtedly a move in the industry towards increasing adoption of open systems. These systems are most frequently implemented in addition to, rather than in place of, existing proprietary systems. Despite the increasing market share of open systems, many IT professionals have not yet realized that truly open systems cannot be bought -- they must be built. It isn't just running on top of a UNIX-derived operating system. If you are committed to open systems, every decision you make -- hardware, software, languages, applications, tools -- will be restricted by the requirement to be "open". Open software is more portable; there is no question on that point. But there is a question about whether portability is as important as the current emphasis suggests. Open systems do not inherently provide any advantage in adaptability, performance, functionality, reliability, security, or availability. You will have the most flexibility in your future IT directions if you set portability requirements for all of your new purchases and new development. But, you should also develop architectural requirements in the areas of reliability, performance, productivity, system administration difficulty, and any other areas that you feel are important. There should be specific goals in each area, and a prioritization of the requirements so that when goals are found to be mutually exclusive it will be clear which goal should receive priority.

What are the factors that need to be considered in developing a software architecture? You should be considering at least the following:

- Functionality
- Data Integrity
- Performance
- Security
- Availability / Fault Tolerance / Resilience
- Support of Development Environment
- End-User Productivity
- Difficulty of Administration
- Interoperability
- Migration

These criteria can be applied to the selection of hardware platforms, operating systems, databases, tools, and applications. An organization that says it will only choose UNIX-based solutions has decided to put the final two items, migration and interoperability, above all others. It is far better to specify what is required for each of these areas than to assume that a particular solution strong in a few areas will meet your requirements in all of them. There are always trade-offs to be made; there is no one system or architecture that will be best in every area. Although there will be no one "ideal" solution, there are enough technologies available that you should be able to create an IT architecture that balances these attributes in the way that best serves the needs of your business.

Characteristics of Good Software Architectures

In descriptions of software architectures, you will often hear the terms ***modular***, ***layered***, ***object-oriented***, or ***tiered*** (as in two-tiered or three-tiered). These represent complementary, but different, aspects of how the software is divided into pieces. An architecture may have all of these characteristics or none of them. In general, the more of these characteristics an architecture possesses, the more flexible it will be, and the less it will cost to maintain over the long run.

- ***Modular*** software is function-oriented. A modular design facilitates making changes in the business processes supported by the software.
- ***Layered*** software isolates application logic to the greatest extent possible from the implementation details of operating systems, databases, networks, etc. Layered software facilitates software portability.
- ***Object-oriented*** software is modular, incorporates data hiding, and includes software reusability as a design objective.
- "***Tiered***" is often used to describe a distributed hardware configuration, rather than a software implementation. In this paper, we will use tiered to describe the software characteristics that allow the software to be distributed across multiple systems in a client/server environment.

We'll see how each of these characteristics can be implemented in HP 3000-based software.

Evolution of HP 3000-based Software Architectures

As we develop our ideas for an application software architecture, it helps to recap briefly what types of architectures have been used in the past. It is, after all, the limitations of these architectures that drive us to seek something better for new applications. Understanding where we have come from gives us a clearer understanding of where we would like to go.

This paper will present a brief trip through the history of software architectures on the HP 3000. As you study the progression, don't be disturbed if you find that the model which best describes your current architecture is the first one described -- the oldest model. The HP 3000 and its application development subsystems -- TurboIMAGE and VPLUS -- are so well suited to a particular architectural model that many, if not most, users have never seen a strong benefit to moving to a later model. Indeed, after studying everything we can present to you about what today's software architectures look like, you may still feel that the "classic HP 3000" model is the one best suited to your business needs. That must be the key consideration -- not moving to the latest and greatest for the sake of change, but finding the model which best suits your business needs. If your business needs have changed little, then a radical shift in architecture probably isn't warranted.

The Classic HP 3000 Architecture

Figure 1 presents the classic HP 3000 application architecture. The program code is most likely implemented in COBOL. Data storage is most frequently TurboIMAGE, although KSAM and a number of flat-file formats are frequently used as well. VPLUS is most commonly used to provide the user interface. Various system services, such as process management, are accessed via the MPE intrinsic mechanism.

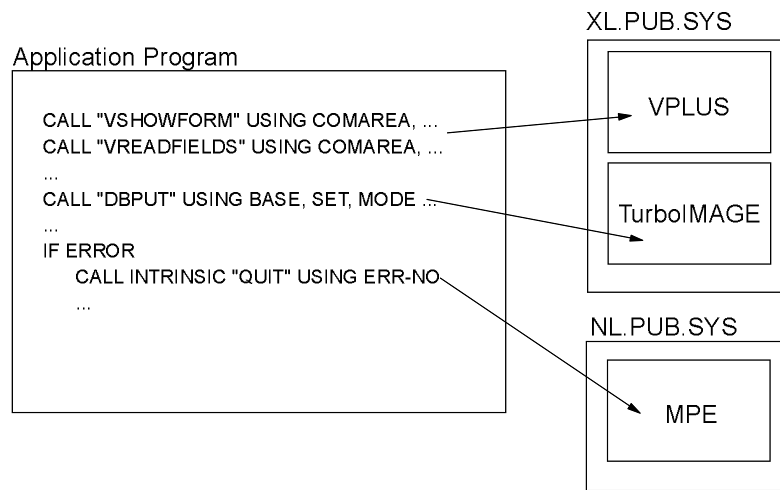

Figure 1: The Classic HP 3000 Architecture

This architecture is widely used for a number of reasons. One is its simplicity; there is nothing in the architecture which doesn't have to be there. Thus, development and maintenance are straightforward. The subsystem technologies are extremely well tuned to the HP 3000 environment, such that even with all the advances in past twenty years, the combination of components shown above will generally outperform systems built on a newer technology base.

The problem arises when there is a desire to change any of the components in this model. If you want to replace TurboIMAGE with a relational database, you are hindered by the fact that database access code is scattered throughout the entire application. If your application grows so large that it cannot run well on a single system, there is no straightforward way to distribute the application across multiple systems. If it ever became desirable to move the application entirely to another platform, you would be faced with a near rewrite to remove the dependencies on MPE intrinsics, TurboIMAGE, and VPLUS. The classic application architecture provides great functionality and performance, but very limited flexibility and no portability.

The Layered Architecture

A layered application architecture will address many (but not all) of the limitations of the classic architecture. The main difference between the classic and layered architecture is the creation of a software layer that separates the application code from the subsystems and operating system that provide the technological base for the application.

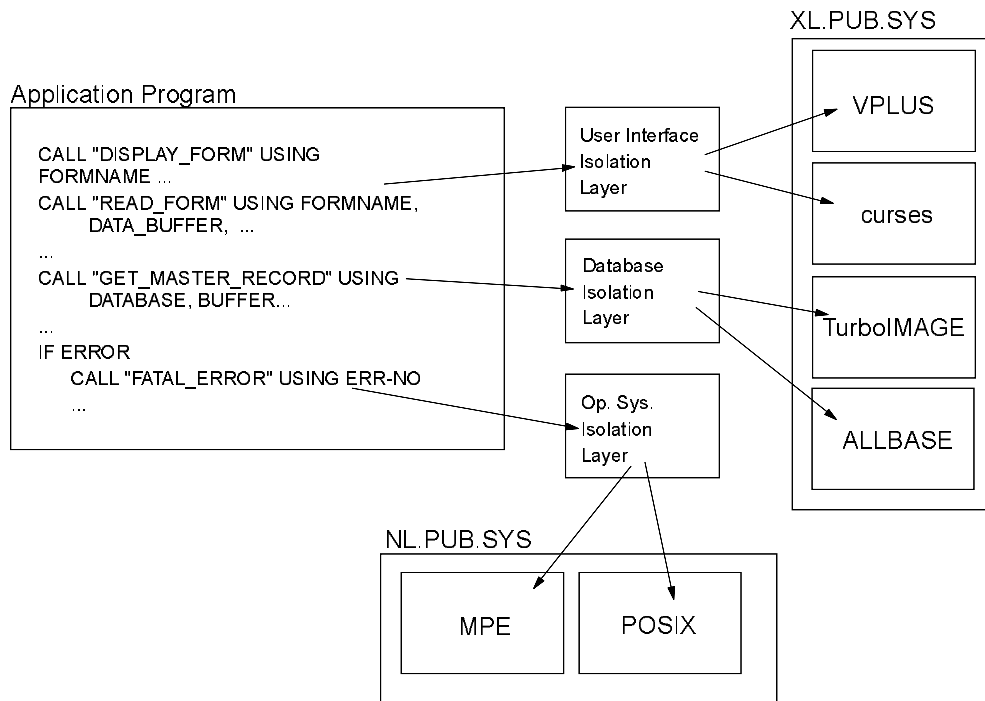

Figure 2: The Layered Architecture

Figure 2 depicts a layered application architecture which could be used for the same application. The insertion of a software isolation layer removes any direct dependencies between the application logic and the underlying subsystems. There are a number of advantages realized from this architectural change.

- Much of the code in the intermediate layer will be leveragable. For example, to display and collect data from a VPLUS form requires a relatively large number of calls (VGETNEXTFORM, VINITFORM, VSHOWFORM, VREADFIELDS, VFIELDEDITS, VFINISHFORM, VGETBUFFER). This entire sequence could be replaced by a single call, passing a form name and returning a data buffer. The intermediate user interface layer can be leveraged for all forms within the application, and even used in other applications.
- All data structures and logic specific to the subsystems can be moved out of the application logic, making the application more compact and maintainable.
- Application programmers need not be trained in the intricacies of VPLUS, TurboIMAGE, or MPE; they can simply write to a defined set of APIs (Application Program Interfaces) that are provided by the intermediate layer.
- If the interfaces to the intermediate layer are made sufficiently abstract, the underlying subsystems can be changed without affecting the application. For example, a VPLUS user interface could be replaced with a curses user interface, without changing the API used by the program. (This requires thoughtful design of the API. For example, do not pass the VPLUS COMAREA between the program and the intermediate layer; this is a VPLUS specific construct.)

- The application code itself is now portable. Components of the intermediate layer may be portable, depending on the subsystems accessed. Database access code for Allbase will be portable to HP-UX; code for TurboIMAGE will not be. For non-portable components, it will only be necessary to re-implement the specific module in question; compare this to the need to rewrite the entire application for the classic architecture.

As you can see, there are a number of advantages to using the layered architecture. The cost of such an architecture is the run-time penalty of processing additional procedure calls each time a subsystem is accessed. Most applications will be able to absorb this cost without a noticeable increase in response time or any measurable degradation in overall system performance. In reality, performance is more likely to improve as a result of this architecture. Rather than all programmers coding their own database access routines in varying fashions, you can have your best database programmer develop the intermediate layer. It will also be more practical to perform performance measurement and tuning on this single module than on the entire application.

The Client/Server Architectures

The Gartner Group describes five different models of Client/Server architecture: Distributed Presentation, Remote Presentation, Distributed Logic, Remote Data Management, and Distributed Database. We will look at three of these in detail (the remaining two will be presented as variations of these three).

The Remote Presentation architecture, as shown in Figure 3, is the most widely used of the client/server architectures. This is perhaps because it solves the problem that forced the evolution of client/server architectures in the first place; the demands placed on the host system to support increasingly CPU intensive user interface technologies. If your users are satisfied with the look-and-feel of VPLUS forms based interfaces, then there may be no need for you to adopt a client/server architecture. But most users are demanding drop-down menu bars, scrollable pick lists, context sensitive help, and other features found on Graphical User Interface (GUI) based systems. If you try to implement these features on dumb terminals, you will find that they consume more CPU resources than VPLUS by a factor of 10 or more. Thus, you will be forced to either upgrade your CPU, support fewer users, or offload the processing of the user interface to a less expensive processor. It is this last option that is most economical, plus provides the greatest flexibility. CPU-intensive user interface processing is performed on the client system, while data access and application logic continue to be provided by the host (server) system.

In the Layered code model, we created a software layer that isolated the application logic from the specific user-interface technologies. This layer could provide transparent portability between, for example, VPLUS and curses. This is possible because both VPLUS and curses are screen-based interfaces; and while the mapping between functionality is not exact, a workable abstraction can be created to allow support of both environments.

Graphical interfaces such as Windows, Motif, or the Macintosh are quite different. This difference is more than just in appearance; there is a difference in the philosophy of how an application is controlled. Non-GUI applications typically have a standard program flow that is determined by the application designer. Users can influence this somewhat; for example, function keys can be used to navigate within the application. Compare this to the typical GUI application. Actions selectable from the application’s menubar may give access to several dozen functions. Other than disabling selections that are not currently available, the application imposes no ordering of actions upon the user.

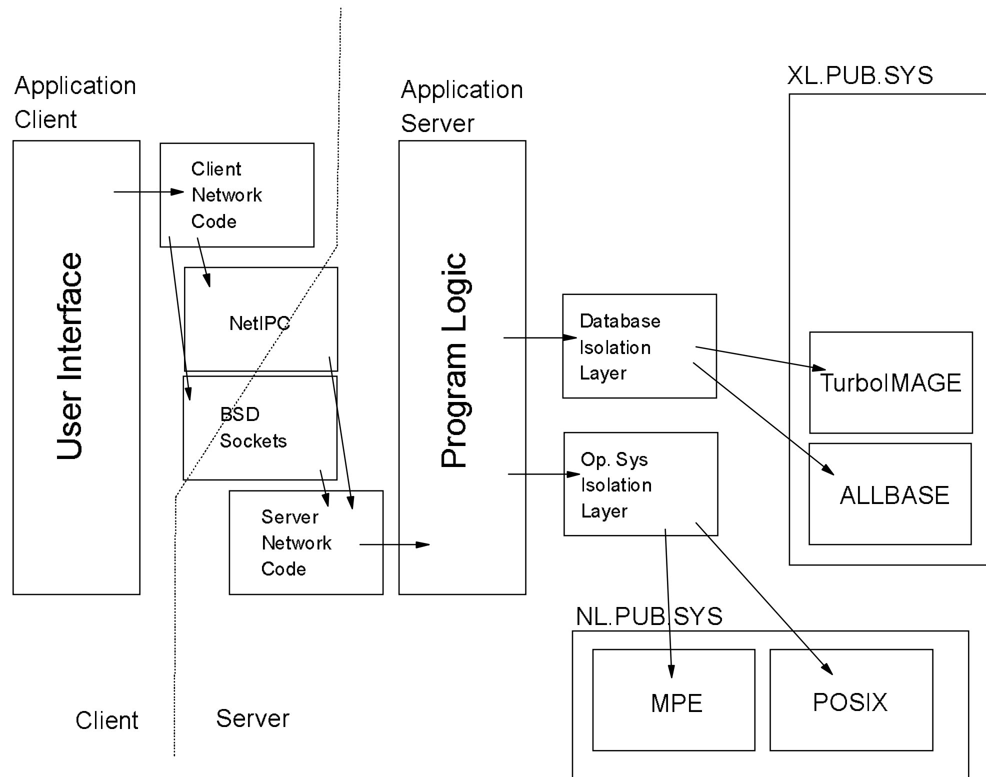

Figure 3: Remote Presentation Client/Server Architecture

This different view of how an application is controlled leads to a different way of constructing applications. The older architecture is based on a prompt-and-response model; the application prompts for input (by displaying a form in the case of VPLUS), the user inputs data, and the cycle repeats. The new architecture is based on an event-action model. The application waits for the user to do something (an event), which may be selecting a menu item, entering data into a dialog box, or clicking on an icon, among others. The application responds to this by performing the indicated action, and then waits for the next user event. The application is controlled by the user interface; not the other way around. This change is indicated by the reorganization of the application modules in Figure 3; instead of the program code calling functions within the User Interface when user interaction is required, the User Interface now calls functions within the application logic module when an action is requested. This control of the application from the user interface is characteristic of all modern architectures, and will be seen in all the models we examine from this point forward.

The Distributed Presentation model (not shown) is a variant of this model in which a portion of the user interface logic continues to reside on the server.

The Distributed Logic model, as illustrated in Figure 4, moves portions of the application logic to the client in addition to the user interface. This permits tasks which are CPU intensive to be offloaded from the server system, or for certain tasks for which response times are particularly critical to be performed locally. For example, a production controller in a manufacturing environment may download a production schedule, and then perform analysis of various what-if scenarios on the client.

If changes in the schedule are made, the results can then be transmitted back to the server. In a catalog order entry environment, the server would be accessed to determine the availability of merchandise, but processing such as calculating an extended price and adding sales tax can be performed locally in the client.

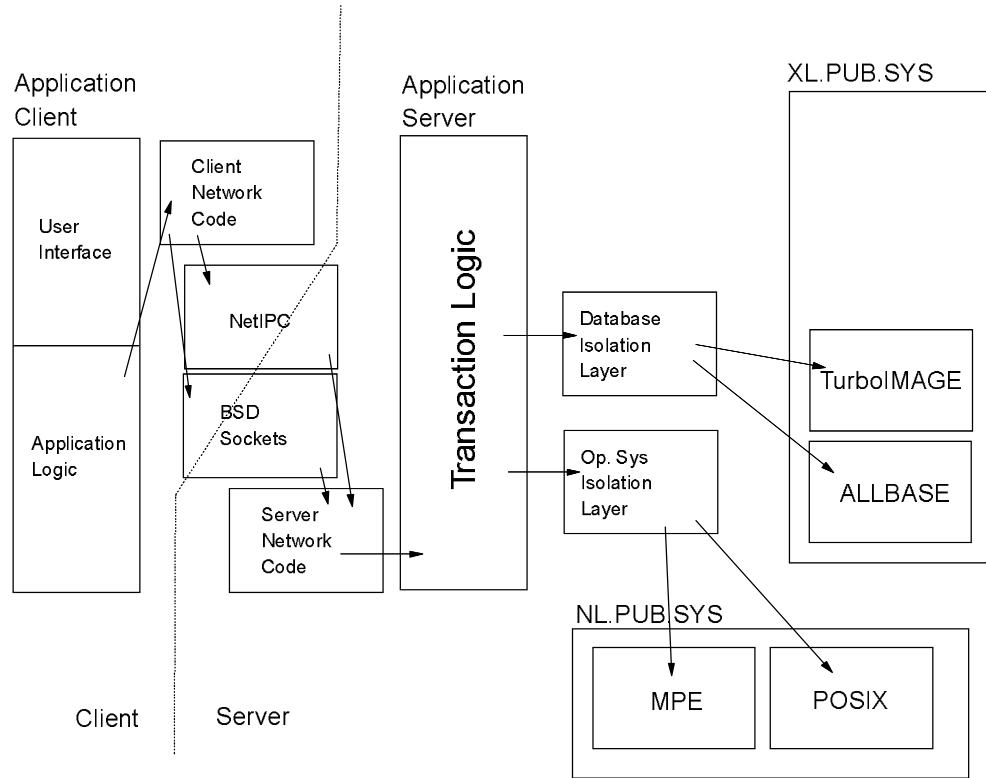

Figure 4: Distributed Logic Client/Server Architecture

Applications built upon the distributed logic model are frequently thought to be more complex than those built upon other client/server models. This is not necessarily the case. The perception arises primarily from the fact that in the other models, there is widely used middleware that handles the network interfaces. In a distributed logic implementation, the programmer is much more likely to be writing directly to a network API, rather than through a middleware layer. We feel that this is not as intimidating as it may at first seem.

In the Remote Data Management model (Figure 5), virtually all application logic is moved to the client. This is also referred to as the Database Server model, since all that remains on the server in this configuration is the database and code directly associated with data access. A database front-end provides the illusion of local database access; calls are made to this front-end as if it were the database engine. The front-end then sends the database requests over the network, using either a general-purpose mechanism such as BSD sockets or, more commonly, an API designed specifically for database communication, such as ODBC. A database process on the server receives messages from the client, performs the actual database operations, and returns requested information and status to the client. An advantage of this model is that the programmer can now take advantage of the latest generation of Rapid Prototyping / Rapid Development tools, such as Visual BASIC, Delphi, and

PowerBuilder, while still enjoying the high reliability and performance of mainframe or minicomputer based database systems.

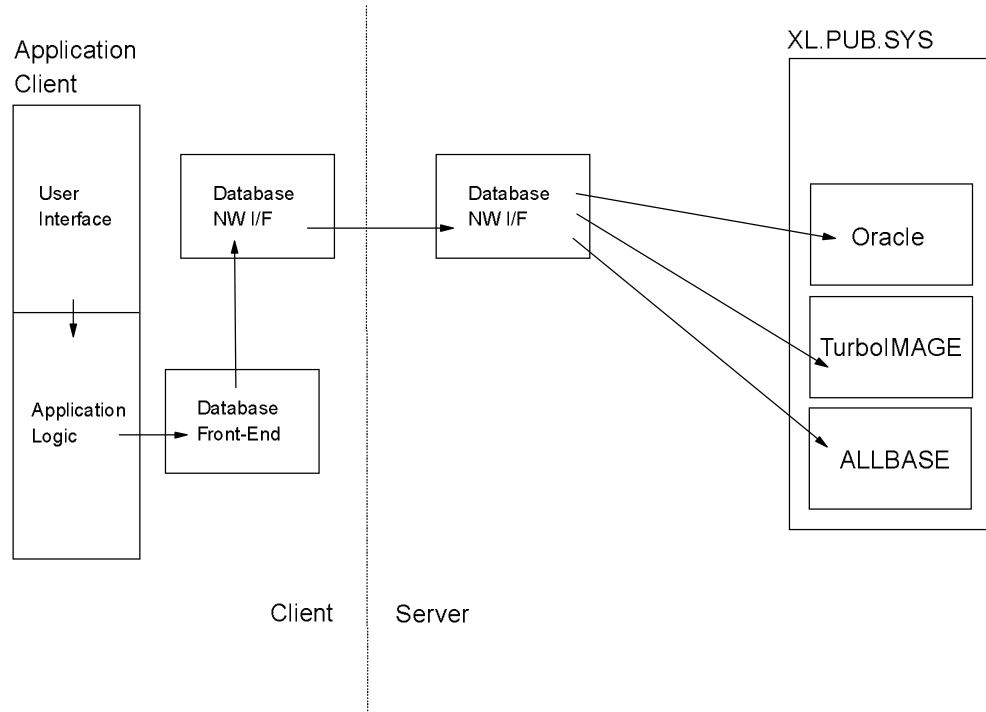

Figure 5: Remote Data Management Client/Server Architecture

The Distributed Database architecture (not shown) is a variant of the Remote Data Management architecture in which the database is itself distributed. It may be distributed between the client and server, or across multiple server systems. Maintaining the integrity of a distributed database in case of processing interruptions such as system failures is a far more complex task than maintaining integrity on a single system. Although it is possible to write code to manage this task yourself, organizations implementing a distributed database should consider implementing a Transaction Manager designed expressly for this purpose.

The various flavors of client/server architectures provide many advantages over the traditional host-terminal architectures. In each of the cases we have discussed so far, the distribution of work between the client and the server is fixed at the time the architecture is designed. An application architecture based upon the Remote Presentation model, for example, may not be easily adapted to allow additional processing to be added to the client at a later time. Since it is possible that the client/server model which best suits your processing needs may change over time, our final evolutionary step is to create an architecture which allows any of these client/server architectures, or a combination of them, to be used, with the capability of switching between models fairly easily when required.

The Maximum Flexibility Architecture

We'll refer to this final architecture as the Maximum Flexibility architecture, since that is its distinguishing characteristic. The design of this architecture, shown in Figure 6, includes the following features:

- To the programmers on the client side, all resources appear to be local.
- Local (client-side) isolation layer accesses local objects directly, and accesses remote objects via the messaging layer.
- Messaging middleware layer provides capability to send messages over various types of networks using different APIs. Ability to support asynchronous (non-blocking) messages.
- Single monitor process at server to handle incoming traffic from clients; can be replicated if needed to support workload.

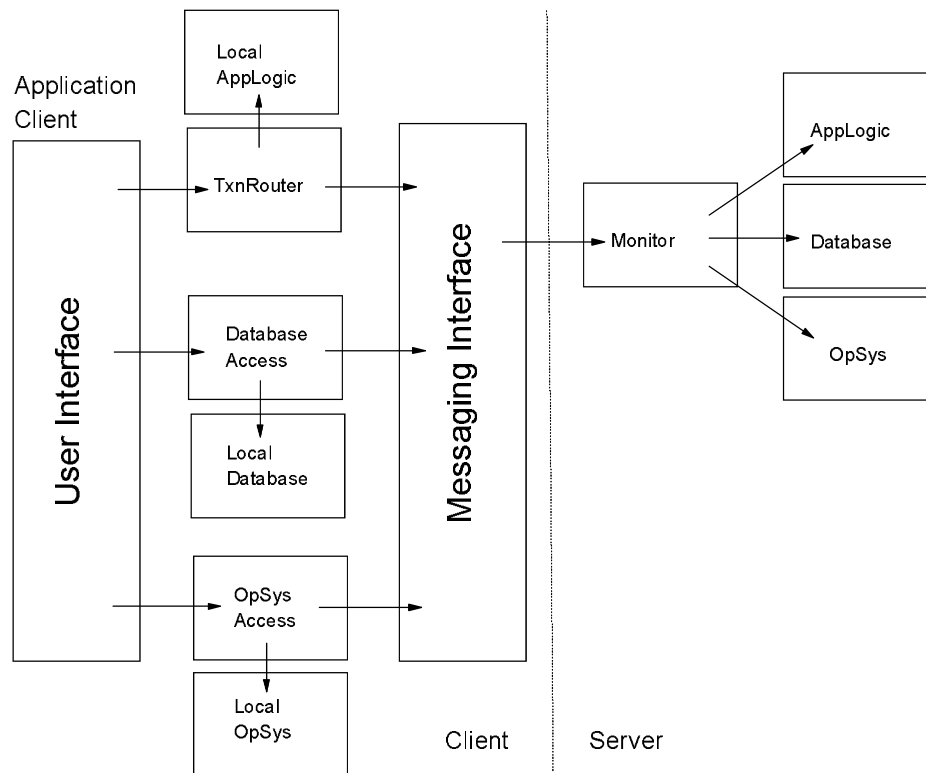

Figure 6: Maximum Flexibility Architecture

The design of this architecture was driven by two overriding objectives. First, it should work in the largest possible variety of configurations; flexibility is the number one requirement. This means that at every design point, the architecture strives to present the greatest number of alternatives. An architecture can be made simpler by limiting the choices at each of these design points; for example, if only MPE/iX and HP-UX need to be considered as operating systems, or if only one database technology needs to be supported. The architecture also provides the capability to be scaled anywhere from a single-user model on a PC, to a single multi-user system with terminal connections, to a client/server architecture based on any of the previously described models..

The second overriding objective is vendor independence. While many people today equate “open systems” with running on a UNIX-derived operating system, we have chosen to take the extreme position (at least within this architecture) that the required degree of openness is to be completely vendor independent. Thus, the architecture is designed to provide independence from hardware vendors, operating system vendors, database vendors, tool vendors, and stress relief medication vendors.

Exceptions will be noted where portions of the code are viewed as “throwaway”; that is, they can be redeveloped using another technology when required at a lower cost than designing them up front to support multiple underlying technologies.

In the client/server architectures described previously, the application always knew where to look for each component. In a Remote Data Management model, the database was on the server, while application logic was local. In the maximum flexibility architecture, the location of all components is flexible. For each component, there will be code on the client which receives requests from the user interface. This client code will then determine, by accessing configuration information which may be dynamically changing, where to go for the required service. It may be locally on the client, or on any of a number of servers which can be accessed over the network.

One component that is shown explicitly in this architecture for the first time is a monitor process running on the server. There is some such process in any client/server implementation, but depending on the architecture selected and middleware products used, it may be transparent to the programmer. The monitor is responsible for receiving the various messages that can be received from clients, and either processing them or passing them along to another process which will handle them.

Trends in Software Architecture

The evolution described above was not the random emergence of new ideas, but rather movement along a well-defined path. We will examine three of the most important ways in which software architectures have been changing throughout this evolution.

Monolithic to Modular

One trend that has been prevalent in programming for some time is a move toward smaller compilation units. A typical COBOL program written for a mainframe environment in the 1970s might be tens of thousands of lines of code in a single monolithic program. The logic flow through the program might be very difficult to follow: most COBOL programmers learned their craft before it became unfashionable to use GO TO as the preferred method of controlling program flow. As the benefits of structured, or modular, programming began to be realized, the look of the typical program changed. The program was now broken down into smaller logical units. PERFORMing these discrete sub-units, and not using GO TOs indiscriminately to control the program flow, allowed the program to be seen as a collection of smaller well-defined units rather than as one huge program. In some cases, programmers took the next logical step of actually moving some of this code into separate programs altogether that would be CALLed from the main program, although COBOL did not facilitate this step as well as later languages.

Procedure based languages, such as Pascal and C, became much more prevalent during this time, because their structure is well suited to this new way of modularizing code. But nothing prevented the COBOL or FORTRAN programmer from incorporating the new thinking into his programs. Then, as now, it was not necessary to throw away your existing technology investments to take advantage of the latest technologies.

The breaking down of code into smaller functions was mirrored by changes in the way the program's run-time data was handled. Again, a large monolithic chunk of code tended to treat its data as monolithic, as well -- everything went into a COBOL Working-Storage section (there was no alternative), usually with no indication of what parts of the program would access which data. With procedure-based languages, it was possible to create variables that were local to a particular function, as well as data that was global to an entire compilation unit or process. Making data local to a function helped provide greater control over it; the programmer could be confident that only the intended code could be modifying data values. This worked well when the data was only needed by a single function. For data used in several functions, some advocates of structured programming recommended that global variables always be passed explicitly into any module that might modify them. Although popular modular languages never enforced this restriction, those programmers who followed it would have an easier time understanding where global data was being used within a program. This set the stage for the introduction of object-oriented programming practices, which we'll come to shortly.

Imbedded Interfaces to Layered

When dinosaurs roamed the earth and the caveman programmers chiseled their programs onto stone tablets, portability wasn't a concern. The early mainframe era wasn't much different; computing resources were far too expensive to allow for elegant software architectures that could be ported to another system that probably hadn't been invented yet. But the minicomputer era brought very different economics to the industry, and for the first time it became practical to think in terms of creating software that might someday run on a different environment than the one for which it was initially created.

The most important design factor for creating portable software is having a layered software architecture. A modular software architecture, as described in the previous section, breaks software into small functional pieces. But each of these pieces may still interact with many different system-dependent or middleware components -- the operating system, the database, the network, etc. In a layered software architecture, each of these interfaces to the system on which the application resides would be separated from the application itself. With a separate database interface layer, for example, if it becomes desirable in the future to change the database implementation, only this layer needs to be changed.

A properly layered software architecture provides all of the following benefits:

- Greater portability between platforms
- Easier adaptation to new technologies (different database, different networking protocols, new middleware layers, etc.)
- More flexibility in distributing tasks between client and server (or peers)
- Improved ease of maintenance

Procedure-Centric to Data-Centric

As discussed previously, the change in how code was organized was accompanied by a change in the organization of data. There was a further evolution in the relationship between code and data that went beyond merely reorganizing data into "global" and "local" units. Moving data that was unique to a task into the function responsible for that task removed much clutter from the set of data managed by a program, but did little to help manage the data that needed to be global in scope. Explicit passing of any global variables into any routine that modified them helped answer the question of "who's using the data," but did nothing to guarantee that the data was used in a consistent

fashion (for example, one routine that modified a variable might perform a check to not allow a negative number to be stored, whereas another routine in the same program might omit this test). The large number of potential accessors, with no guarantee of consistent behavior, made maintenance and troubleshooting of such programs difficult. Another frequently encountered problem was that if it was necessary to change the format of the data for any reason, it became difficult to discover all the places in the code that might be affected by the change. Many IT organizations are now tackling the problem of expanding date fields to accommodate 4 digit years, and are encountering this problem head-on.

Thus, our next milestone in the evolution of flexible programming is the nearly-object-oriented concept of "data hiding." The reasoning is straightforward: the fewer ways in which a given piece of data can be modified, the smaller the chance of inconsistency in the way modifications are handled. Similarly, the fewer routines there are that know the physical representation of a piece of data, the smaller the impact of any change to that physical representation. The objective of data hiding is to minimize the number of routines that a) know the physical representation of a given data item, and b) may modify the contents of the data item.

There are numerous ways in which data hiding may be implemented. Many organizations will implement data hiding by agreement -- there is no enforceable prohibition against directly accessing a data item, but routines are provided to read and write values to the data, and it is agreed that everyone will use these routines instead of bypassing them. In other environments, there may be security implemented on the actual data items such that only the approved routines will be able to access or modify them. This is particularly common with data that is sensitive or where the opportunity for mischief is great (e.g., payroll records).

There are several good examples of data hiding on any MPE system. In moving from the classic MPE V system to MPE XL, the internal representation of many data items changed from 16 bits to 32 bits to match the new architectural word size. Yet because most programmers never access these data items directly, but only through the MPE intrinsic interfaces, programs that represent these fields as 16 bit fields can in most cases continue to function without changes. (In those cases where the value stored in the field cannot be represented in 16 bits, the programmer will have to make modifications.) Another example is the TurboIMAGE database system, which implements data hiding at the record, rather than the data item level. If you use a particular field in a TurboIMAGE database, then any changes to the physical representation of that field will need to be reflected in your program. However, if you use a list parameter that returns only the fields you specify, then your program will be unaffected by any changes to other fields in the record. The record can be completely reordered, fields can be added or dropped, and your program will continue to work.

Why Objects are a key technology for today's software architectures

Whether you adopt a true object-oriented language such as Smalltalk or C++, or develop in a traditional third-generation language such as COBOL or C, or use fourth-generation development tools, you should be incorporating object based thinking into the design of your applications. The rate of change in business today is faster than it has ever been, and all indications are that in the future, change will occur even more rapidly. If your development cycles are still measured in years, then the applications you are developing will be obsolete before they are ever deployed. Not obsolete in terms of the technology they use -- you can always live with systems that aren't on the bleeding edge of technology -- but obsolete in terms of being able to meet business requirements, a far more damaging limitation that you simply cannot accept.

Object technologies help you meet the business need of being able to re-engineer your applications to meet changing business conditions. A set of well-defined objects can be implemented once, and will

persist with only minor changes through a number of re-engineering cycles. These business objects, which change little, are accessed through applications which can be rapidly deployed through the use of high-productivity development environments such as Visual Basic. The front-end applications can be revised or completely redeveloped as often as need dictates, while the business-critical data contained in the objects remains stable.

Why Client/Server is a key technology for today's software architectures

There are several trends driving the current push toward client/server architectures. As with the trends driving the market toward object-oriented development, these trends show no sign of slowing. If your current development activities do not include any support for client/server architectures, you are almost certainly creating a competitive disadvantage for your business.

The most visible trend is the evolution of user interfaces to graphical, window based systems. The display devices required for these types of interfaces -- bit-mapped displays -- are not directly supported by the HP 3000, so if you want such an interface for your MPE applications, client/server is the only alternative. Furthermore, even if the hardware didn't require such an architecture, performance demands make it far more economical. Block-mode interfaces such as MPE's VPLUS or IBM's CICS place very little processing demand on the system, allowing a single system to support a large number of users with excellent response times. Character-based interfaces such as the UNIX curses system are far more CPU intensive. This is a primary reason why large UNIX installations almost always include some flavor of client/server architecture; the load created by of hundreds of character-mode users cannot be handled by even the largest available CPUs. Graphical interfaces, such as Microsoft Windows or OSF/Motif, represent another significant increase in processing requirements, such that a dedicated CPU per user is the norm for these systems. Thus, the demand for these graphical interfaces requires a move to a client/server design. We frequently hear requests from users to add more "GUI-like" features to VPLUS; for example, drop-down selection lists. Users want these capabilities made available on terminals so that they can provide a more modern user interface without incurring the expense of upgrading users to personal computers. Ironically, however, such capabilities will not avoid the need for such an upgrade, but rather accelerate it. These changes would make VPLUS far more CPU intensive, making it necessary to either upgrade the host system or move to a client/server configuration so that user interface processing could be offloaded from the CPU.

A second driving factor is, once again, the increasing rate of change required to meet business needs. Because of business changes, a company may require a new application, only to find that the application doesn't run on the current hardware platform(s). Users may demand new office automation or decision support tools that require changes in desktop systems. These needs can only be met by bringing in new hardware. But it is not economical to convert all of the existing software to the new platforms. Instead, what is needed is a way for the various platforms to interoperate. Users can have the desktop system that best meets their needs; legacy applications can continue to run on the system best suited to support them; and new applications can be deployed on the platform which makes the most sense. Only through client/server technology can these systems be integrated into a cohesive "system" from the user's point of view, rather than a collection of incompatible systems.

Bibliography

Material for this paper was extracted from the book "The Legacy Continues: Using the HP 3000 with HP-UX and Windows NT", by Mike Yawn, George Stachnik, and Perry Sellars; Prentice-Hall, 1996.