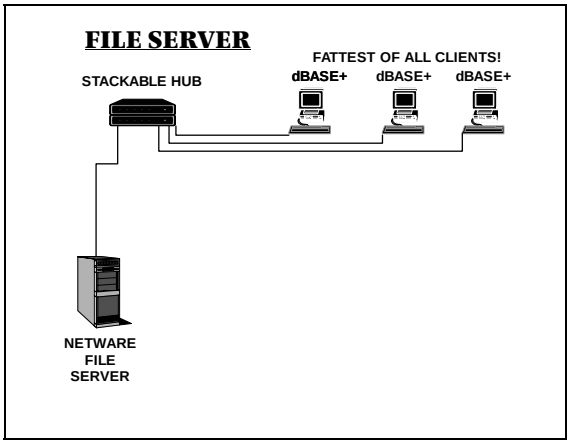
Client/Server, from dBASE to JAVA: Is it Over, or Just Beginning?

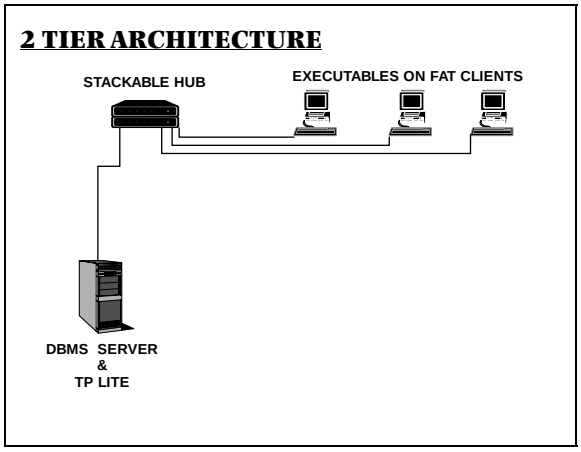
by George Schussel

It's the purpose of this article to explain how the "Client/Server" architecture is really a fundamental enabling approach that provides the most flexible framework for using new technologies like the World Wide Web, as they come along. The old paradigm of host centric, time shared computing has given way to a new client/server approach, which is message based and modular. The examples below show how most new technologies can be viewed as simply different implementation strategies built on a client/server foundation.

Even though most people use the term "client/server" when talking about group computing with PC's on networks, PC network computing evolved before the client/server model started gaining acceptance in the late 1980's. These first PC networks were based on the file sharing metaphor illustrated in the figure entitled FILE SERVER. In file sharing, the server simply downloads or transfers files from the shared location to your desktop where the logic and data for the job run in their entirety. This approach was popularized mostly by Xbase style products (dBASE, FoxPro and Clipper). File sharing is simple and works as long as shared usage is low, update contention is very low, and the volume of data to be transferred is low compared with LAN capacity.



As PC LAN computing moved into the 90's two megatrends provided the impetus for client/server computing. The first was that as first generation PC LAN applications and their users both grew, the capacity of file sharing was strained. Multi-user Xbase technology can provide satisfactory performance for a few up to maybe a dozen simultaneous users of a shared file, but it's very rare to find a successful implementation of this approach beyond that point. The second change was the emergence and then dominance of the GUI metaphor on the desktop. Very soon GUI presentation formats, led by Windows and Mac, became mandatory for presenting information. The requirement for GUI displays meant that traditional mini or mainframe applications with their terminal displays soon looked hopelessly out of date.



The architecture and technology that evolved to answer this demand was client/server, in the guise of a two-tiered approach. By replacing the file server with a true database server, the network could respond to client requests with just the answer to a query against a relational DBMS (rather than the entire file). One benefit to this approach, then, is to significantly reduce network traffic. Also, with a real DBMS, true multi-user updating is now easily available to users on the PC LAN. By now, the idea of using Windows or Mac style PC's to front end a shared database server is familiar and widely implemented.

In a 2-tier client/server architecture, as shown in the figure entitled 2-TIER ARCHITECTURE, RPC's or SQL are typically used to communicate between the client and server. The server is likely to have support for stored procedures and triggers. These mean that the server can be

programmed to implement business rules that are better suited to run on the server than the client, resultingin a much more efficient overall system.

Since 1992 software vendors have developed and brought to market many toolsets to simplify development of applications for the 2-tier client/server architecture. The best known of these tools are Microsoft's Visual Basic, Borland's Delphi, and Sybase's PowerBuilder. These modern, powerful tools combined with literally millions of developers who know how to use them, means that the 2-tiered client/server approach is a good and economical solution for certain classes of problems.

The 2-tiered client/server architecture has proven to be very effective in solving workgroup problems. "Workgroup", as used here, is loosely defined as a dozen to 100 people interacting on a LAN. For bigger, enterprise-class problems and/or applications that are distributed over a WAN, use of this 2-tier approach has generated some problems.

Client/Server in Large Enterprise Environments

What typically happens with client/server in large enterprise environments is that the performance of a 2-tier architecture deteriorates as the number of on-line users increases. The reason for this is due to the connection process of the DBMS server. The DBMS maintains a thread for each client connected to the server. Even when no work is being done, the client and server exchange "keep alive" messages on a continuous basis. If something happens to the connection, the client must go through a session reinitiating process. With 50 clients and today's typical PC hardware, this is no problem. When one has 2,000 clients on a single server, however, the resulting performance isn't likely to be satisfactory.

The data language used to implement server procedures in SQL server type data base management systems is proprietary to each vendor. Oracle, Sybase, Informix and IBM, for example, have implemented different language extensions for these functions. Proprietary approachs are fine from a performance point of view, but are a disadvantage for users who wish to maintain flexibility and choice in which DBMS is used with their applications.

Another problem with the 2-tiered approach is that current implementations provide no flexibility in "after the fact partitioning". Once an application is developed it isn't easy to to move (split) some of the program functionality from one server to another. This would require manually regenerating procedural code. In some of the newer 3-tiered approaches to be discussed below, tools offer the capability to "drag and drop" application code modules onto different computers.

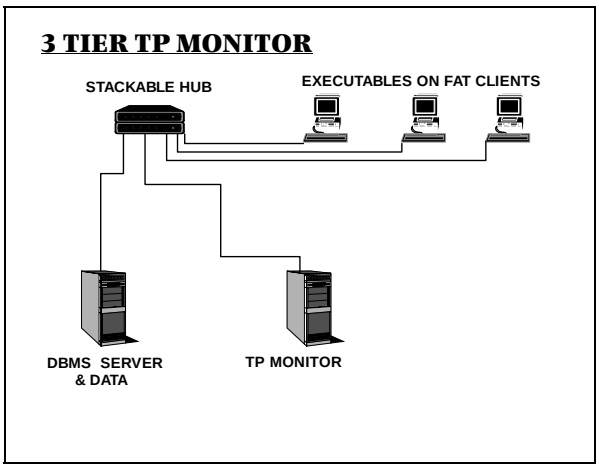
The industry's response to limitations in the 2-tier architecture has been to add a third, middle tier, between the input/output device (PC on your desktop) and the DBMS server. This middle layer can perform a number of different functions - queuing, application execution, database staging and so forth. The use of client/server technology with such a middle layer has been shown to offer considerably more performance and flexibility than a 2-tier approach.

Just to illustrate one advantage of a middle layer, if that middle tier can provide queuing, the synchronous process of the 2-tier approach becomes asynchronous. In other words, the client can deliver its request to the middle layer, disengage and be assured that a proper response will be forthcoming at a later time. In addition, the middle layer adds scheduling and prioritization for the work in process. The use of an architecture with such a middle layer is called "3-tier" or "multi-tier". These two terms are largely synonymous in this context.

There's no free lunch, however, and the price for this added flexibility and performance has been a development environment that is considerably more difficult to use than the very visually oriented development of 2-tiered applications.

3-Tier With a TP Monitor

The most basic type of middle layer (and the oldest, the concept on mainframes dating from the early 1970's) is the transaction processing monitor or TP monitor. You can think of a TP monitor as a kind of message queuing service. The client connects to the TP monitor instead of the database server. The transaction is accepted by the monitor, which queues it and then takes responsibility for managing it to correct completion.



TP monitors first became popular in the 1970's on mainframes. On-line access to mainframes was available through one of two metaphors - time sharing or transaction processing (OLTP). Time sharing was used for program development and the computer's resources were allocated with a simple scheduling algorithm like round robin. OLTP scheduling was more sophisticated and priority driven. TP monitors were almost always used in this environment, and the most popular of these was IBM's CICS (pronounced "kicks").

As client/server applications gained popularity over the early 1990's, the use of TP monitors dropped by the wayside. That happened principally because many of the services provided by a TP monitor were available as part of the DBMS or middleware software provided by vendors like Sybase, Gupta, and Oracle. Those embedded (in the DBMS) TP services have acquired the nickname "TP Lite". The "Lite" term comes from experience that DBMS-based transaction processing works OK as long as a relatively small number (<100) of clients are connected.

TP monitors (TP Heavy) have staged a comeback because their queuing engines provide a funneling effect, reducing the number of threads a DBMS server needs to maintain. The client connects with the monitor, which accepts the message and queues it for processing against the database. Once the monitor has accepted the message, the client can be released for further processing. The synchronous session based computing of a 2-tier architecture, then, becomes asynchronous through the insertion of the TP monitor into the equation. The monitor smoothes out and lowers the overhead of accessing the database server.

Some other key services a monitor provides are: the ability to update multiple different DBMS in a single transaction; connectivity to a variety of data sources including flat files, non relational DBMS, and the mainframe; the ability to attach priorities to transactions; and robust security, including Kerberos. The net result of using a 3-tier client/server architecture with a TP monitor is that the resulting environment is *FAR* more scaleable than a 2-tier approach with direct client to server connection. For really large (e.g., 1,000 user) applications, a TP monitor is one of the most effective solutions.

As you might expect, however, there is a downside to network-based TP monitors. At this point in time, the major problem with using this approach is that the code to implement TP monitors is usually written in a lower level language (like COBOL), and support for TP monitors is not (yet) widely available in the most popular visual toolsets like PowerBuilder or Visual Basic.

3-Tier With a Messaging Server

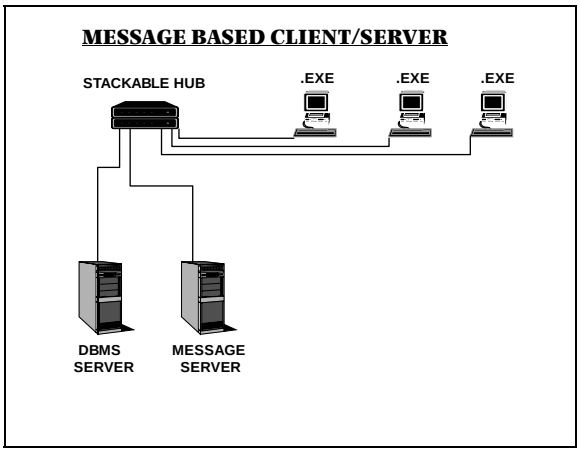
Messaging provides still another technology to implement 3-tier computing. It is available today from companies such as IBM, DEC, Sybase, and Oracle. A messaging server can be thought of as a kind of "second generation" TP monitor and provides the same funneling process. Messages are processed asynchronously with the appropriate priority level. And, like a TP monitor, a messaging server provides connectivity to data sources other than RDBMS.

A message is a self contained object that carries information about what it is, where it needs to go, and what should happen when it reaches its destination. There are at least two parts to every message; the header contains priority, and address and an ID number. The body of the message contains the information being sent, which can be anything including text, images or transactions.

A primary difference from TP Monitors is that a message server architecture is designed around intelligence in the message itself as opposed to a TP monitor environment which places the system intelligence in the monitor or the process logic of the application server.

In a TP monitor environment the transactions are simply dumb packets of data. They travel over a pre-existing and pre-defined connection to the TP Monitor. The TP Monitor interrogates and processes the transaction, usually submitting the request to a server tier application. If the TP Monitor doesn't understand the data, it doesn't get processed. Ultimately, the TP Monitor needs to know as much about the transaction as the server tier does.

Contrasting with this, in a message-based architecture there's intelligence in the message itself. The message server just becomes a container of messages and their stored procedures. The operations performed by the message server on the message are communications related (e.g. encrypt message over one service and decrypt message sent over another service). For the most part, messages are treated as discrete objects. The message contains all the information needed to transverse network services (i.e. network addresses, both logical and physical). Because the message contains the intelligence, the middle tier of a message-based system is more flexible than a TP monitor. For one kind of message, the middle tier may simply serve as a routing point between two kinds of network services. For another kind of message, the middle tier may execute a stored procedure or business rule as directed by the message. This abstraction of the middle-tier away from the contents and behavior of the information flowing through it makes the system more portable to different environments and networks. The specifics of communicating the information are hidden underneath the messaging service.

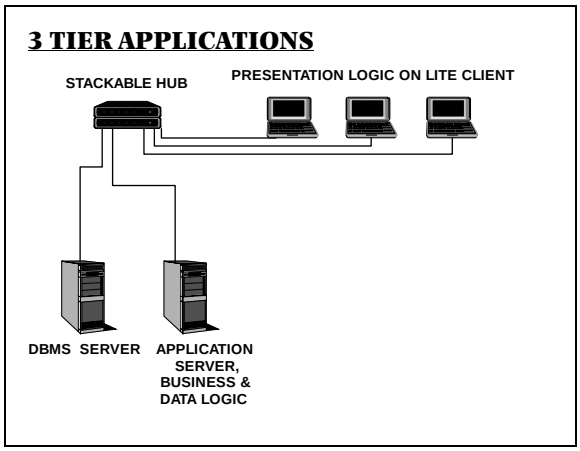


Messaging systems are designed for robustness. By using store and forward logic, they provide message delivery after and around failures. They also provide independence from the enabling technologies such as wired or wireless or protocols. They don't require a persistent connection between the client and server. They are robust because message delivery can be programmed to occur after or around failures. Because messaging systems support an emerging wireless infrastructure, they should become popular for supporting mobile and occasionally connected workers.

A typical message server architecture would look like the figure entitled MESSAGE BASED CLIENT/SERVER, which of course, looks just like any of the other 3-tier approaches we're going to discuss. The architecture of an application that uses messaging services will turn out to look similar to an approach that depends on distributed objects and object request brokers (ORB's) for communication. If you're unwilling or unable to wait for the arrival of distributed object technologies to build your application (widespread popularity probably won't happen with ORB's until the end of the 1990's), you can construct a reasonable clone using the messaging approaches that are now available. When distributed objects are a reality, you can migrate your application if that seems like the best move.

3-Tier With an Application Server

When most people talk of 3-tier architectures, they mean the approach of an application server (illustrated below). With this approach most of the application's business logic is moved from the PC and into a common, shared host server. The PC is basically used for presentation services - not unlike the role that a terminal plays on a mainframe. Of course, because we are talking about a real PC here, it still has the advantages of being used for client side application integration (via OLE or other approach) if desired.



The application server approach is similar in overall concept to the X architecture that was developed at MIT in the 1980's. In X the goal is to allow host-based computing with graphical interfaces on the desktop (I'm using the term "desktop" here because in the X architecture, the term "server" refers to the graphical server which sits on the desktop and the term "client" refers to where the application runs - on the shared host). The similarity between X and a 3-tiered client/server architecture with an application server is that both architectures have the goal of pulling the main body of application logic off the desktop and running it on a shared host.

The application server is also similar to a mainframe in that it doesn't need to worry about driving a GUI, and therefore it's a shared business logic, computation, and data retrieval engine. This server normally operates under a 32 bit multitasking OS like NT, OS/2, NetWare or UNIX. As an option, these OS' all run on symmetric multiprocessing (SMP) configurations. In addition, some are available on massively parallel hardware. Therefore, the server is very scaleable in terms of performance. As new versions of the application software are developed and released, the installation of that software occurs on the one server rather than hundreds or thousands of PC's.

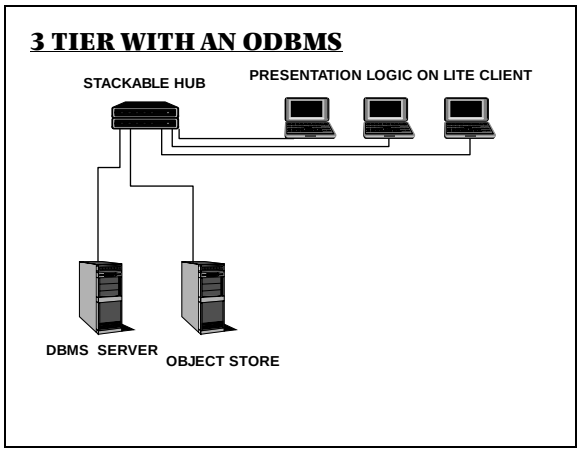
The approach of putting business logic on a server offer a number of important advantages to the application designer:

- When less software is on the client, there is less worry about security since the important software is on a server in a more controlled environment.
- The resulting application is more scalable with an application server approach. For one thing servers are far more scalable than PC's. While a server could be a single Pentium based Compaq or Dell, it could also be a symmetric multiprocessing Sequent, with 32 or more processors. Or, it could be a massively parallel UNIX processor like IBM's SP2.
- The support and installation costs of maintaining software on a single server is much less than trying to maintain the same software on hundreds or thousands of PC's.
- With a middle application server tier it's much easier to design the application to be DBMS-agnostic. If you want to switch to another DBMS vendor, it's more achievable with reasonable effort with a single multithreaded application than with thousands of applications on PC's.
- Most new tools for implementing a 3-tier application server approach offer "after the fact" application partitioning. This means that code and function modules can be reallocated to new servers after the has been built. This offers important flexibility and performance benefits. (e.g. This technology is available today in toolsets from Dynasty Technologies and Forte Software).

The major downside to an application server approach to client/server computing is that the technology is much more difficult to implement than a 2-tier approach.

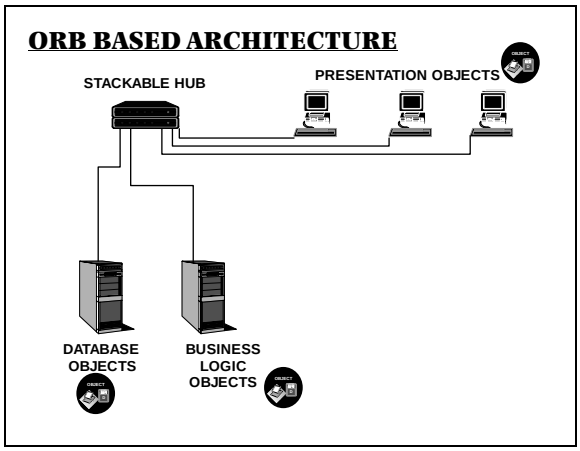
3-Tier With an Object DBMS

A variation on this theme of application server is the idea of using an object DBMS (ODBMS) as the middle layer. This is illustrated in the figure entitled 3-TIER WITH AN ODBMS. In this sense, the ODBMS acts as an accelerator or "hot cache". Data in a relational DBMS is usually stored in normalized fashion across many tables and for access by different applications and users. This generalized form of storage may prove inadequate (performance wise) for the needs of any one particular application. An ODBMS can be used to retrieve the data from the common store, assemble it for efficient usage by your application, and provide a persistent store for that data as long as your application might need it. Since extended data types like video or voice are not typically supported in today's RDBMS, those data types might also be stored in the ODBMS, which could then associate the appropriate multimedia data with the data retrieved from the RDBMS.



Distributed Components & the 3-Tier Architecture

This brings us to distributed object computing and components. Many software pundits are predicting a software future with the creation of application systems through assembly of software components as is illustrated in the ORB BASED ARCHITECTURE figure. That kind of software approach is available today in a few proprietary object environments like NeXT's NeXTStep and ParcPlace's VisualWorks. The emergence of a broad based industry for component based software will require the prior emergence of industry standards for interchangeable parts. For components to be assembled like tinker toys, they are going to have to match up in terms of connectors. Translated, that means that all vendors who want to create software components are going to have to agree on the software object bus. There are only two real candidates for such a standard backbone: Microsoft's OLE and OMG's (Object Management Group) implementations on CORBA and OpenDoc. It isn't the purpose of this article to explore this issue, but it can be mentioned that not enough of either network OLE or CORBA technology is currently available for ordinary mortals to build with. By the end of 1997, however, it's probable that both will be available and that they may even achieve some level of interoperability.



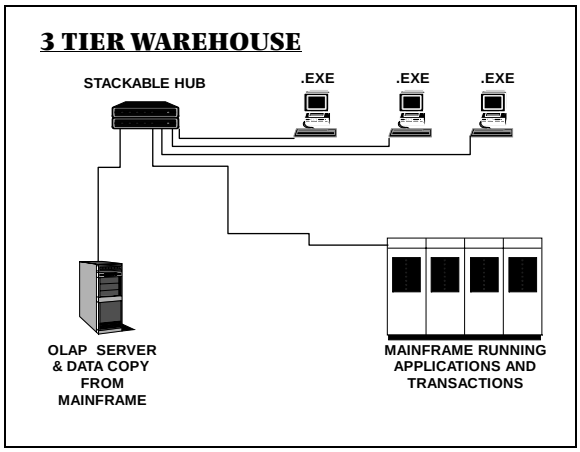
The distributed object implementation of client/server computing is going to change the way applications are built. There should be some very interesting advantages to observe. For one, if we needed fault tolerant computing, we could implement copies of objects onto multiple servers. That way if any were down, it would be possible to go to another site for service. With distributed objects being self contained and executable (all data and procedures present) it will be possible for a systems administrator to tune the performance of the network by moving those objects from overloaded hardware to underutilized computers. This approach is called tuning through "drag and drop", referring to the metaphor the administrator uses on a workstation to move the components.

A distributed object architecture should also offer other benefits for application developers. For example, consider the following:

- The same interface will be used for building a desktop, single location application or a fully distributed application.
- The application can be developed and tested locally and you'll know that it will work fine when it's distributed - you depend on the known services of an object request broker for distribution.
- Since the application developer is dealing with an object request broker for transmission services, technical issues like queuing, timing and protocols aren't an issue for the application developer.

Data Warehouse & 3-Tier

A 3-tier architecture is also useful for data mining or warehouse types of applications. These applications are characterized by unanticipated browsing of historical data. The databases supporting this type of application can sometimes be huge (up to a few terabytes -$10^{(12)}$ bytes) and have to be structured properly for adequate performance (a few second turnaround).



Data mining and decision support applications typically need response times of a few seconds. If the system can't provide that kind of performance, the thought process of the human analyst is disrupted and the overall purpose of the system is foiled. A production database established for multiple users isn't typically in a form that can support ad-hoc inquiries. The approach to support this browsing is then to make data copies available for that browsing and to organize the data in those copies in the best supporting fashion. This typically means that the data is denormalized, summarized, and stored in a multidimensional table - all of which is very non-relational. IT systems and operations managers usually don't want access to those tables to be on the mainframe. Unpredictable performance from ad hoc browsing can have a nasty impact on production OLTP systems that require predictable response times.

For cost, management, security, and other reasons, it makes sense to load this data copy on its own server rather that leaving it on the mainframe. Often this server is called OLAP - on-line analytical processor. In other circumstances this server can be a symmetric or massively parallel processor running an RDBMS. Since the OLAP server is typically a UNIX or PC-based technology, the MIPS costs are much lower than the same cycles executed on a mainframe. The figure entitled 3-TIER WAREHOUSE illustrates this approach. (The graphic for mainframe is a little different, of course, but the reader has probably noticed that nothing has really changed architecturally here from any of the other multi-tier approaches already discussed!)

3-Tier and the Future

By now the point is made. Client/server architectures are flexible and modular. They can be changed, added to, and evolved in numbers of ways. All of the above described 3-tier approaches could be mixed and matched in various combinatorial sequences to satisfy almost any computing need. As the Internet becomes a significant factor in computing environments client/server applications operating over the Internet will become an important new type of distributed computing. (This is probably an understatement, since the use of Internet and intranet based applications will very shortly dwarf all of the distributed computing initiatives of the past)

The Internet will extend the reach and power of client/server computing. Through its promise of widely accepted standards, it will ease and extend client/server computing both intra and inter-

company. The movement in programming languages to the technology of distributed objects is going to happen at light speed - because of the the Internet.

Client/server still remains the only and best architecture for taking advantage of the Internet and other new technologies that come along. We'll have to add "changes in client/server computing" to death and taxes in our inevitable list. But, regardless of what comes, client/server computing is likely to remain the underpinning for most computing developments we'll see over the next decade.

George Schussel

George Schussel has been a CIO, consultant, industry analyst, writer and lecturer on computer topics for 30 years. His lectures are held before more than 20,000 professionals a year. He is the founder and Chairman of Digital Consulting, Inc. (DCI) in Andover, Massachusetts and Chairman of the *Database & Client/Server World* trade show. He has published over 50 technical and analytical articles and his latest book, Rightsizing Information Systems, co-authored with Steve Guengerich, was published by the SAMS Publishing Division of MacMillan. Reach him at 74407.2472@compuserve.com or http://www.dciexpo.com/.

