Converting Qedit to the Client/Server Model

What is Client/Server and Why Bother?

The Client/Server Model is the standard model for network applications. A server is a process that is waiting to be contacted by a client process so that the server can do something for the client. For example, the server might provide the current time of day, or the weather, or print a file, or get your e-mail messages. A suitable client, whether located on the same host as the server or not, can communicate with the server to request the service it provides.

Client/Server allows separation of the user interface software (the client) from the backend/workhorse software (the server). One advantage of this separation is that different hardware is good at different things: the client is good at providing quick response to a single high-bandwidth graphical user interface with a mouse; the server would be hard pressed to provide this functionality to multiple users at the same time (although the X-Windows software from UNIX attempts it). The server, on the other hand, is very good at dealing with large files and databases.

Another advantage of client/server is that it divides the logic of the application into two or more independent units, which can only interact via a well-defined protocol. A well thought-out protocol makes the project easier to manage and implement.

By letting the client do what it is good at and the server do what it is good at, we ended up with a single Windows interface that is easy to learn—because it is similar to every other Windows program—and which can pull up files on all of our MPE and HP-UX systems, and edit them with exactly the same keystrokes.

Our Experience Converting Qedit to Client/Server

Robelle's Qedit has been available for the past 20 years as a terminal-based, line and full-screen mode editor for HP 3000 and for HP-UX. Over the past 18 months, we have been involved in creating a client/server version of Qedit called Qedit™ for Windows™ or QWIN. The resulting application can cut text from a source file on an MPE system in our Caribbean branch office and paste it into another file on an HP-UX system in Canada. Because we have our own server software, we can use familiar Windows editing keystrokes to browse and edit huge files (such as network traces or database extracts). We can still look at large files even over a slow network connection because the server feeds the client only the pages that it needs, not the entire file. What's more, we can use QWIN for editing local files on our PCs. As soon as we could use the new system ourselves, we began to appreciate the attractions of client/server.

One of the unexpected insights of developing this new tool was that client/server applications do not always cleanly fit the "pure" client/server model. In our case, the client is expected to edit local files in addition to server files. Programming the client to provide edit, search and print functions that were consistent with the server's functions consumed much of our development resources, much more than we expected. Perhaps we should have implemented a copy of the server to run on the client system, although it is just as likely that most real-world client/server applications also lack "purity".

As we went through the process of converting our Qedit to a client/server version, we learned a great deal about the design and implementation of client/server applications. In this paper you will learn one instance of how client/server applications are designed for MPE, HP-UX, and Windows, and how they are then implemented, distributed, tested, and maintained.

Fundamental Design Decisions

We made the most basic design decision early in the project: to use a transaction-oriented design, with the host file primarily resident on the host, rather than copied to the PC.

We judged that this approach would allow the final software to edit the widest range of file sizes, including huge data files, on the widest range of networking speeds, including over slow modem connections.

Project Team

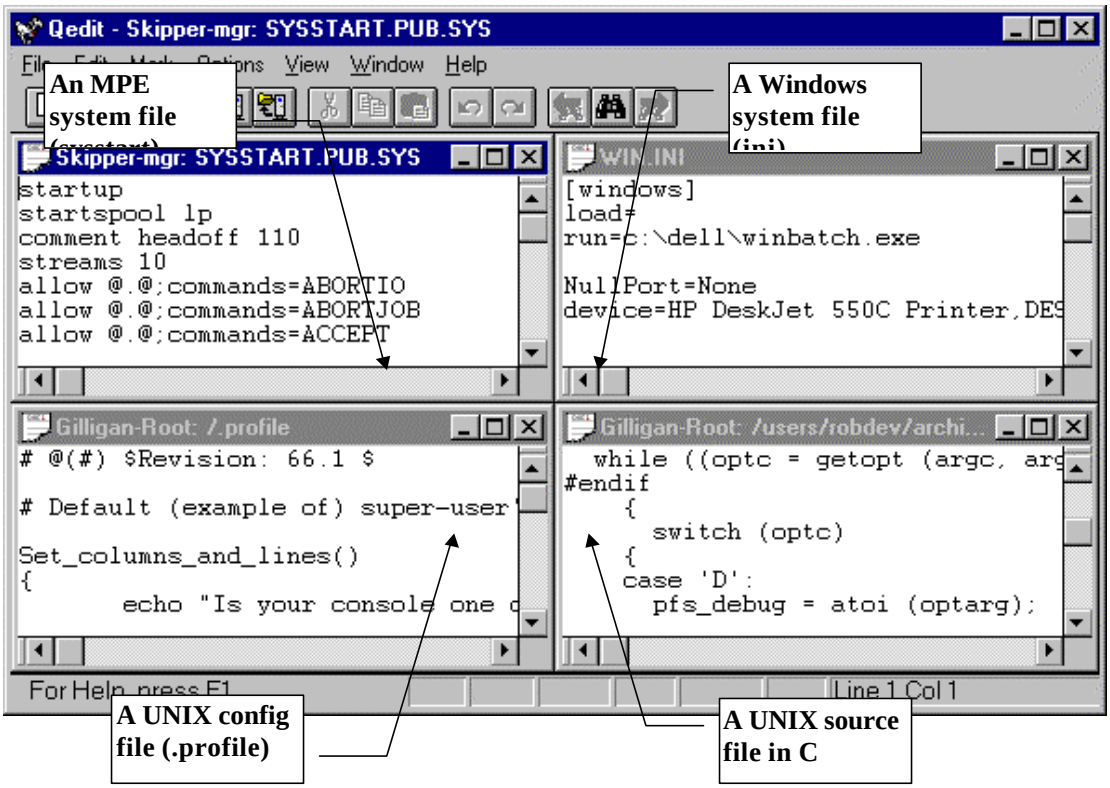
David Greer, President of Robelle, put together most of the networking infrastructure used by the server. We hired Bruce Toback of OPT, Inc. as a contractor for the Windows client. I wrote the server code to do the file manipulation on the hosts. Griffin Webster of Robelle designed the automated test suite, Neil Armstrong of Robelle implemented the client installation procedures, and Dave Lo of Robelle created the in-process CRT-based client. One weakness of the team is that in a pinch only David can find and fix problems in both the client and the server.

The client currently consists of 35,000 lines of source code and the server of about 7000 lines of new code, plus numerous changes to the existing Qedit source code. The team exchanged over 2600 e-mail messages about the design and implementation of the project.

Technical Objectives for the QWIN Project

1. Edit multiple files on one or more host systems (up to ten files at the same time per server), with multiple windows into the same file. Browse the host directory as you would a PC directory.
2. Edit local files on the PC, remote files on PC servers, files on HP hosts connected to the LAN, and files on HP hosts accessible over the Internet or the firm's Intranet.
3. Familiar Windows graphical interface including vertical and horizontal scrolling.
4. Familiar Qedit functions, such as Smart string search, multi-edit Undo, but enhanced for the client platform and extended to a multi-host environment.
5. Dependable server program based on Robelle's existing Qedit technology.
6. Create clients for Windows 3.1, Windows 95 and Windows NT™ which can communicate with servers on HP-UX and MPE/iX.
7. Simplify the process of maintaining source code and scripts in a distributed network.
8. Edit files without loading the entire file into the PC's memory. Use smart caching to make QWIN suitable for editing huge data extracts and network traces, whether local or remote.
9. Cut and paste on a host system, or between host systems, or into local documents.
10. Use standard TCP/IP protocol so that it is not necessary to purchase expensive add-ons for each client PC and/or host.
11. Edit both text files and Qedit work files.
12. Provide full login security on the host (username and password), compatible with Vesoft Security/3000 options such as session name password. Encrypt login data while in transit between client and server.
13. Provide convenient one-click login to hosts, based on a configuration table of hosts, user names, and passwords.
14. Employ a standard UNIX daemon on HP-UX and automatic network login on MPE/iX.

Example: Four Files on Two Servers and on a Local Client



What to Base the Server On?

The existing Qedit has all the functionality we need for examining, editing, and updating files. Therefore, we decided to build a transaction server around the existing Qedit code. Were any changes necessary to the existing Qedit software? Yes, absolutely.

We made four main changes to the existing Qedit design:

1. Most PC and UNIX software assigns relative line numbers to text lines (i.e., 1,2,3...). These numbers change whenever you insert or delete a line, because they merely represent a count of the number of "new lines" since the start of the file. We made QWIN compatible with the PC/UNIX model, instead of with the MPE model of fixed line numbers assigned to each line. The server, however, maintains actual hidden line numbers for compatibility with MPE standards, and renumbers these lines whenever more room is needed. For those users who work from "line-numbered" compiler listings, there is an option to display "absolute" line numbers in the client, but this option is not on by default.

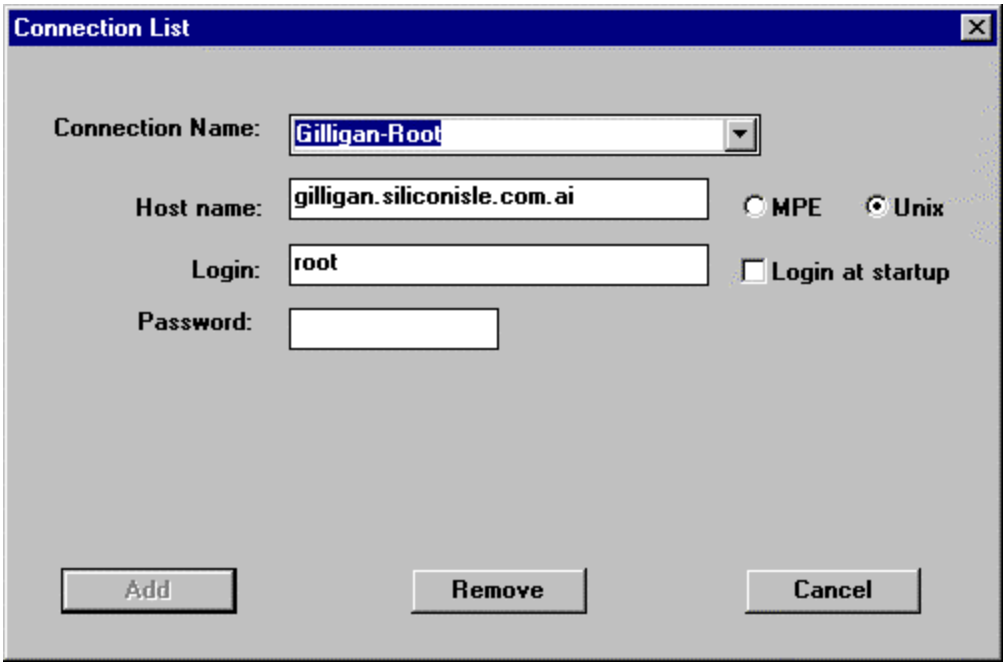
2. The cut-and-paste logic of Qedit is line-oriented. We wrote a new cut-and-paste module for the server to follow the Windows model, which starts at a specific column and ends at another column.

3. Previously Qedit could remember several editing changes and allow you to undo them one by one. But if you undid something and then changed your mind, it was not "redo-able". That is, you could not Undo an Undo in Qedit. When it was pointed out that Microsoft® Word and other

tools on the Windows platform could do both, the Qedit Undo module was enhanced to be compatible. Now both the terminal-based Qedit and the server Qedit can handle "redo" after an Undo in all cases, including that of multiple Undo operations.

4. Rather than create one server process per concurrent file opened by the client, we enhanced the Qedit server to maintain data structures on up to 10 concurrently open files. The terminal-based Qedit can open only one file at a time (it now has the data structures to handle up to 10, but we haven't yet figured out an intuitive command syntax for implementing this).

Defining a Connection to a Host Server



How to Write a Server?

Unless you have been living in a cave for the last year, you have already used a server—the HTTP server that web sites employ to "serve up" web pages upon demand from the browser (client). Another common server you may have used is the POP3 server that collects your e-mail for you.

The QWIN server executes one transaction at a time, in the order received and may send one or more reply transactions. In order to easily enforce the security and access rights assigned by the host to each user, there is one server process per user connected to the host system. A server process can open up to 10 files for a client (this limit is somewhat arbitrary and could be increased by a recompile).

UNIX

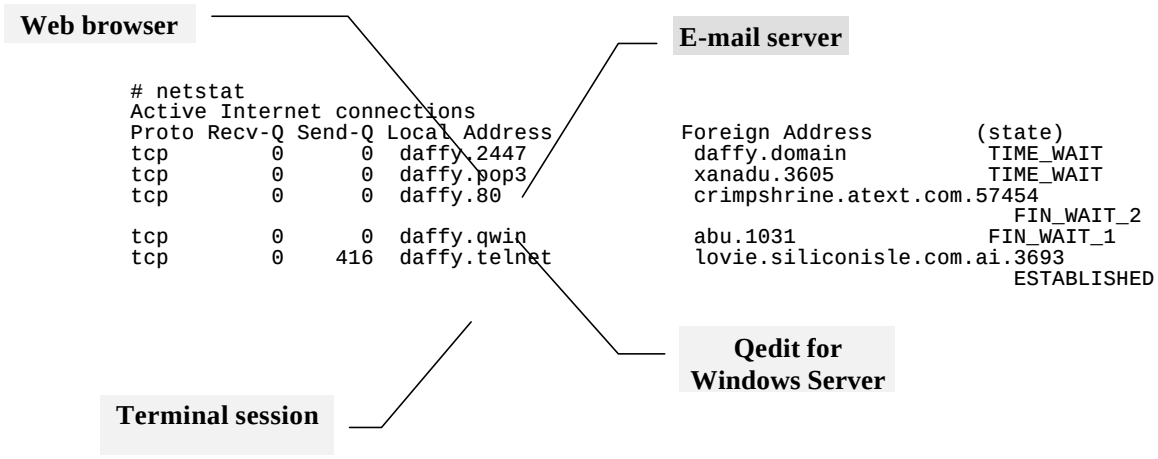
In “UNIX-speak”, a server is called a daemon, a sort of invisible, hidden creature who watches and waits to perform amazing tasks. There are two kinds of server: iterative and concurrent.

The iterative server is a single process that handles all the requests itself in the order in which they come. This is only appropriate for transactions which complete in a “known, short amount of time.” [Stevens] Qedit transactions would not be this short. The concurrent server, on the other hand, invokes a separate process to handle each client request. The original server can then go back to sleep, waiting for the next request.

For Qedit, we create a separate server process for each new client that connects to the server. This new process then handles all subsequent client transactions, rather than creating a new process for each transaction. This is similar to the FTP protocol and opposed to the HTTP protocol which fires off a new process for each web page requested. The connection can handle multiple file Opens and Closes, and actually maintains itself until the user disconnects, or the system is shut down, or the network faults badly for a long period of time (a “time out”).

The original “listener” server is actually just another copy of Qedit in a special mode. UNIX has communication channels called ports which can be used to create a TCP connection between two systems. The Qedit application has been assigned 7395 as its reserved port number, so the server is in a wait state until a request to connect arrives on that port. It then “forks” a copy of itself and “execs” it, launching a child process to take over the new connection. The listener can then return to its quiet, waiting state.

On UNIX, you can use the netstat command to see the connections to a host:

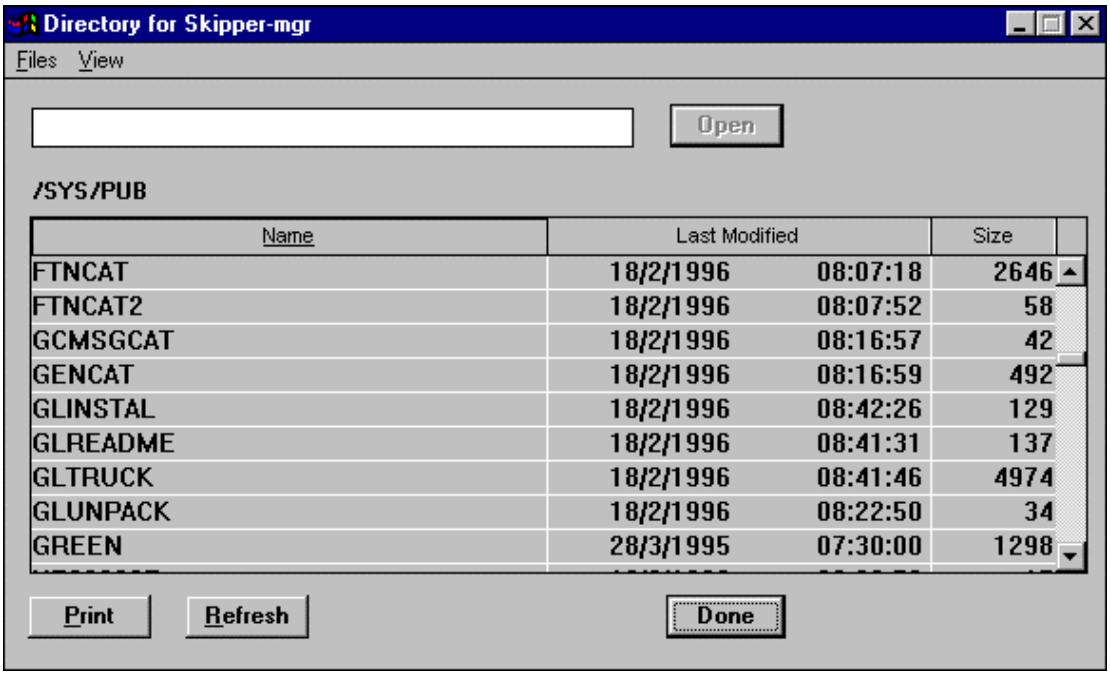


MPE

With POSIX you can implement the same kinds of daemons as on UNIX, but the fork/exec technology was neither complete enough nor fast enough when we started this project. Therefore, we chose to use the standard network logon that is always running on MPE/iX 5.0 (and on earlier systems which have NS/3000). A system listener accepts a logon request, verifies it, and runs a server process specified by the client.

Because QWIN connections do not count as user sessions on MPE or HP-UX, there is no need to upgrade the HP operating system user license.

Select a File from the Host Directory Window



Creating the Client Software

We selected C++ as the development language for our client software because of the breadth of supporting material and tools available to assist us, the ability to get deep into the client OS if necessary, and for the Object Oriented features of C++.

In Object-Oriented Programming (OOP), a program is composed of objects, which can represent real-life concepts such as stocks and bonds, or abstract programming concepts such as linked lists. The goal of OOP is to produce more reliable and inexpensive software by organizing the code in a structure that makes it harder to make errors and easier to think clearly.

C++ is C with OO extensions. The two main concepts of C++ and Object-Oriented Programming are class and object.

A class is a declared structure composed of both a set of data members (like a record) and a set of functions that operate on those data members. Let's consider, for example, the Document, Application, and View classes as they relate to QWIN.

The Application class has data structures that are common to the program as a whole (i.e., not related to any file being edited). It has functions that operate on them, such as terminating the program.

The Document class contains data structures that describe any file being edited by QWIN, such as the filename, links to the cached lines, etc. The class has functions such as Open and Close which operate on the document. QWIN actually has two Document classes: the main or Base Document class and the Local Document class. The Base Document class is designed around the needs of host-based files. The Local Document class is derived from the Base Document class and inherits all of its code and data structures (they are assumed in OO programming, we don't need to respecify them). All we have to describe in the Local Document class is the code and data structures which are *different* for a local file as compared to a host file. For anything we don't override, we get the behavior described in the Base Document class.

The View class is responsible for how a document looks on the screen. In fact, in QWIN there can be more than one "view" (i.e., window) into the same document, each pointing at a different position in the file. The View class handles things like the Page Down key, the resizing of windows, etc. QWIN has only one View class to handle both types of Documents (host and local). In fact, the View class does not even know if it is displaying a local file or a host file. This is where OO programming really starts to pay off.

An object is the instance of a class. For example, one execution of QWIN has only one Application object, but many Document objects (one per file opened). Each instance has its own copy of the data structures, but shares the set of functions with all other instances of the same class.

In OOP terminology, the functions of the object are called methods. In C++ terminology, they are called member functions.

That's more than enough background terminology to get you through the discussions in this paper.

Should We Support Windows 3.11 or Not?

This was a tough decision. If you want to support 16-bit Windows you have to write your code in Microsoft Visual C++® 1.52 and then use Microsoft Foundation Class (MFC) 3.0. It also means that you have to forego some of the Windows 95 user interface features or create them by hand for Windows 3.1. The long and short of it is that if you want to produce a 16-bit version too, you won't be able to use the latest, easiest, most powerful development tools. In Robelle's case, many of our customers had Windows 3.1 systems when we started our project and we thought we were going to produce a product "in a few months". For that reason, we decided to support Windows 3.1, as well as Windows 95 and Windows NT (3.51 and 4.0).

We wrote the client using MS Visual C++ 1.52 and MFC 3.0 (4.0 is much more powerful but does not come in a 16-bit version). A "resource library" is used for all messages and graphics. This "library" will make it much easier when the time comes to produce a version of the client in another language, such as German or French, rather than English.

Two Typical Design Problems

In order to see some client/server design work in action, here are two examples from QWIN:

1. The clipboard question
2. The command question

The QWIN Clipboard Question

Windows systems have a single system clipboard that applications can paste from and copy to. The objective set for QWIN was to build on top of the Windows clipboard model, but extend it to the server hosts. We wanted to be able to copy a large block of text from one server and paste it on the same server, on a different server, or in a Microsoft Word document on the PC client.

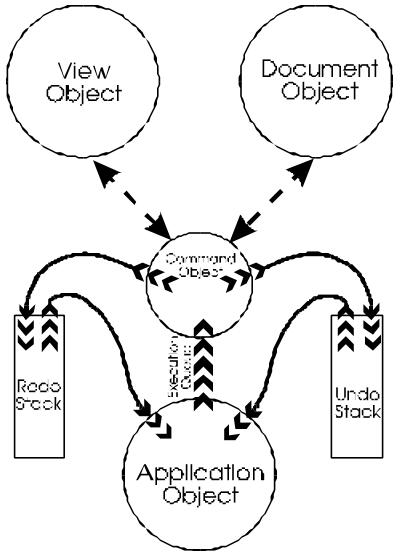
QWIN's clipboard manager has two jobs, which often conflict. It must make the clipboard contents available whenever it is asked to, yet for performance reasons it prefers to leave the contents on the server (and must leave them on the server if they are over 64 kilobytes in size, because that is the maximum size of the real Windows clipboard). Thus the clipboard manager leads a dual life (part of it resides on the server and part on the client), and it hides this fact from the rest of QWIN and from other Windows applications as well.

The QWIN Command Question

A standard C++ program written using MFC receives messages, keystrokes and other events and acts on them immediately. If the user clicks on a task bar icon, a message is sent to the appropriate object within the program which acts on it immediately. The programmer has a relatively simple task of setting up message maps, so that the Edit | Delete menu command is sent to the active View object, the File | Close command is sent to the active Document object, and the File | Exit command is sent to the Application object.

This approach makes for easy programming and is a quick way to construct simple programs. However, the result is that command execution code is scattered throughout the program. Thus, if you want Undo/Redo, there are many places where the Undo context must be saved. Adding a scripting language with this approach requires adding a whole new control system.

The Indirect Execution Model



QWIN uses an indirect execution model for commands instead. User interface events are still sent to the object best equipped to deal with them. However, these objects don't act on the event

immediately. Instead they construct command objects, which in turn interact with other parts of the system to do the user's bidding. The commands are posted to an execution queue maintained by the Application object, which then dispatches them in sequence.

The command object is a package that contains all the data and code necessary to perform a single discrete user action. The context includes the information necessary both to perform the action and to undo it. A command object doesn't rely on any state information in either the View object or the Document object. This has several advantages:

1. All the code that actually performs user actions is contained in one place.
2. All the context necessary to perform the command is contained in one place.
3. A scripting language can operate by creating and queuing command objects. The complex code that maintains the Document and the screen View need not know whether a script is running, and does not contain any script interpretation code.
4. The application can record simply by telling each command object to create a script statement describing itself.

Besides the execution queue, there is also an Undo stack and a Redo stack. The execution queue is where new commands are posted. Once executed, the command is pushed onto the Undo stack. When the user does an Undo, a command is popped from the Undo stack, undone, and pushed onto the Redo stack. When the user does a Redo, the top command is popped from the Redo stack, redone, and pushed back onto the Undo stack. The execution queue is always one-way; commands are added at the tail and pulled off the head (or vice versa, depending on your terminology). The Redo stack is cleared whenever a document-changing command is pulled from the execution queue.

Unfortunately, simple typing doesn't fit into this model. It could, but then each command would consist of a single keystroke. Because the command is also the unit of Undo granularity (not to mention server update granularity), this probably isn't what the user wants to see.

In this case, when the first character is typed, the View object acquires a new Typing Command object and adds the first character to it. On succeeding characters, the View gets the current Typing Command object and adds the character to it. Whenever a user interface event comes in that is not going to be part of the Typing Command, such as a Down Arrow keystroke, the current Typing Command is ended (it becomes a server transaction and an undoable event). The next character typed will instantiate (that is OOP talk for "create") a new Typing Command object. The goal is to gather up all the characters that make up a single typing session into a single object.

As you can see, designing client/server transactions and models can be an intellectually challenging activity. Luckily, many of these considerations go away in a fully-synchronous model. An application that would be perfectly acceptable with a fully-synchronous user interface is a VPlus replacement-type system, where the user enters a key value, gets a record, edits the record, and then sends an update. Both the user and the server view the data as a record at a time. However, for an editor, or for any other application where the user's view of the data differs from the server's, the asynchronous model offers a much more pleasant user experience.

Maintaining Client/Server Applications

The client/server model makes it easier to replace or upgrade a component, because the interaction between components is always a well-defined transaction. The client cannot get inside the server to break anything or vice versa.

For example, in the QWIN project, we were able to write an in-process CRT-based client at the same time as we were writing the Windows-based client. This other client is really a software module that runs on the host as part of the Qedit process, but it only interacts with Qedit through the client/server transactions (i.e., it never calls any of the Qedit subroutines directly). It is aimed at providing screen editing on VT terminals on UNIX, but works on any terminal because it uses the standard Curses library. Dave Lo of Robelle designed and implemented this screen editor using the same transactions that were written for the PC client, giving me two radically different "users" testing my transaction design and implementation at the same time. This was very helpful in ensuring flexibility and reliability of the server.

To make development even more flexible, we implemented something called "tagged fields" in our transactions. Every data item in a transaction is assigned a field number (tag) and is packaged with a length attribute. Clients and servers know how to skip over any field, even when they don't know what a specific field means. This allows us to add fields to a transaction in the server before the client has been programmed to handle them, and vice versa. I can't emphasize how useful this has been in keeping a multiprogrammer, multilocation project moving ahead as quickly as possible.

Previously we used a record structure for each transaction, which meant that adding or deleting a field required a coordinated update on all software components—something that is difficult to arrange in practice with multiple programmers and modules.

By the way, we store the protocol constants for the project, such as these field tags, in a few master source files on our internal web site. Each line defines one constant and ends with a comma (the last line does not have a comma). These files can be "included" without change in the C and C++ source programs (as well as the SPLash! source), in the middle of a statement declaring a list of constants.

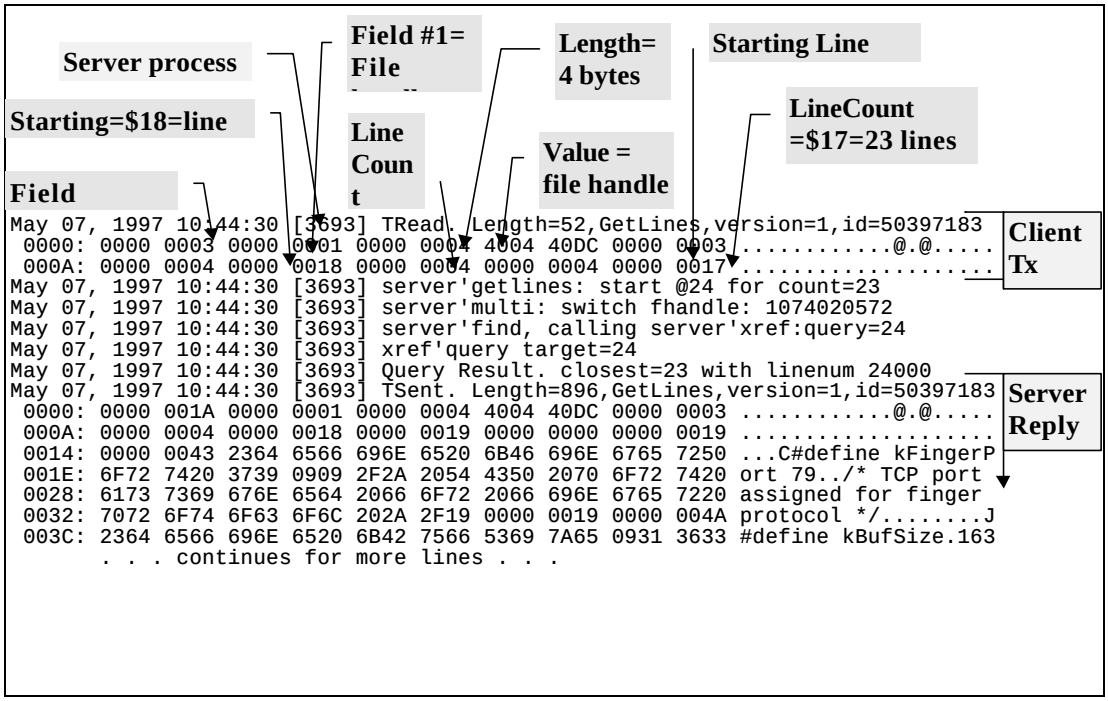
The Difficulty of Thinking Asynchronously

The client responds to external events or messages, such as the user clicking on the scroll down button or the receipt of a Get Lines reply transaction, which may come in at any time. Events do not, of course, occur in a tidy sequential order. In fact, to get the programming correct, you often have to think of events as all happening independently.

What, for example, happens when the user presses the Page Down key?

The client looks for the following lines and can't find them, generating a Cache Miss. The cache software generates a server Get Lines transaction to retrieve the missing lines, then continues to process keystrokes and events. It does not "wait" for the reply transaction. Instead, when the reply arrives with the lines (and there may be several replies coming in if the lines are long and overflow the maximum size of one transaction), the Document object updates the cache with the new data and notifies the View object to see if those lines are currently displayed on the screen and should be refreshed. So, if you press Page Down quickly after pressing Page Down, the lines will not be displayed when they come in—they will just go into the cache.

From the server's point of view, this is actually quite simple. It receives Get Lines transactions, specifying a starting relative line number and a requested line count. The server sends back a transaction containing the lines requested. If the requested lines would create a transaction of more than 8192 bytes, the server breaks the reply up into more than one transaction. Here is a server trace of what happens when you press Page Down:



Debugging Client/Server Applications

Where do you look when a client/server application isn't working properly? The problem could be in the client code, in the TCP/IP system services used to transport the transactions, or in the server code.

Because the server runs as an unattached daemon, you can't see what it is doing. We have added options to our server which allow us to run it from an interactive session and catch the next connection to the system. Using System Debugger we can see where the code is going while it processes transactions. Be aware, however, that time outs occur if you spend too much time in the debugger and do not send replies to the client quickly enough.

The client has a trace option for recording everything it sends to the server and everything that comes back. Here is a typical client trace:

Sample Client Trace

```
MPE logon using 'ci.pub.sys' for user 'bob,bob.green,bob'.
CQServerConn::DispatchReceivedMessage(sync): dispatch txOpen for skipper.
CQServerConn::DispatchReceivedMessage(sync): dispatch txGetLines for
skipper.
CQServerConn::DispatchReceivedMessage(sync): dispatch txOpen for skipper.
CWinqeditDoc::RequestLines: Requesting line 0 for 29.
CQServerConn::DispatchReceivedMessage(sync): dispatch txGetLines for
skipper.
CWinqeditDoc::ProcessGetLinesResponse: message length = 1984;
startingLine = 0; lineCount = 29; errorCode = 0.
ProcessGetLinesResponse: Received 29-line block starting at 0 (was 0);
pf=0.
ProcessGetLinesResponse: exit; pf=0.
CQServerConn::DispatchReceivedMessage(sync): dispatch txCloseFile for
skipper.
ProcessCloseReturn: closeError = 0.
```

This client trace just shows the messages between the host and the client. There are other options to include a Hex dump of the raw buffers (necessary if you want to check exact field values in a transaction), and to trace internal cache management and other functions of the client.

The server also has an option to write transactions to a trace file on the host. These record exactly what the server received, how it processed it, and what it sent back. Tracing is toggled off by default, because it generates a lot of output to the disk, but it is invaluable when trying to determine what happened. There is a sample server trace earlier in the paper under "The Difficulty of Thinking Asynchronously".

Distributed Development

Bruce Toback lives in Phoenix, Dave Lo and David Greer live near Vancouver, Neil Armstrong lives in Alberta, and Griffin Webster and I live on the island of Anguilla in the Caribbean. Our concurrent development efforts were synchronized over the Internet. New versions of the software were delivered via FTP to the various hosts nightly from the site where they were developed.

All documentation was created as web pages and kept on an internal site where everyone involved could check it quickly and easily using his browser.

Installation Procedures

Neil Armstrong of Robelle worked out how to use InstallShield® to package the client executable files, as well as auxiliary files (help, license, etc.), into a single compressed file for distribution and installation. Currently we have a separate file for 16-bit Windows and another for 32-bit, but the new version of InstallShield5 says it will allow us to have one installation "program" that will run on 16-bit and 32-bit machines. InstallShield has an excellent web site, w*ww.installshield.com*, which includes white papers on various topics and access to several newsgroups. Neil found these very helpful because other customers are asking questions that he often asks himself.

Testing Client/Server Software

At Robelle we believe in automated testing of our products. Verifying the correct function of a program after every revision is just too boring for humans to do reliably.

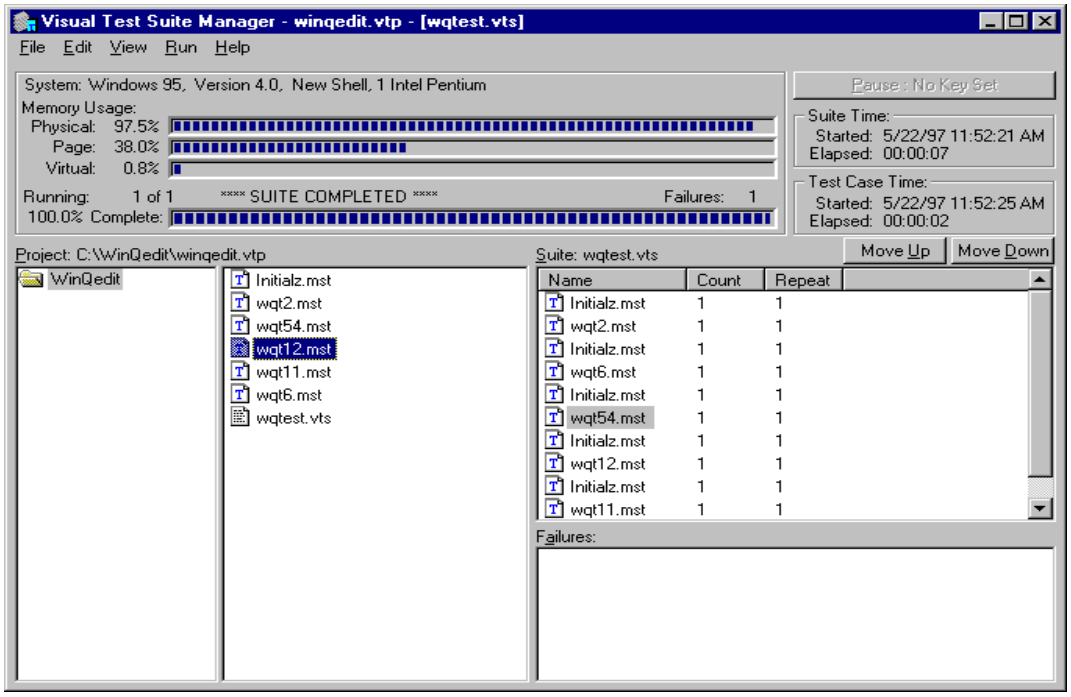
Griffin Webster of Robelle's Anguilla office was asked to find an automated test tool for Windows applications. He evaluated two products: Vermont HighTest and Microsoft®/Rational™ Visual Test.

The criteria for selecting a tool included:

1. ability to construct a test suite that would cover anticipated user interaction,
2. fast, convenient macro recording of individual tests,
3. execution of the complete test suite even if individual tests fail,
4. manual creation/editing of a scripting language for special cases,
5. generation of a test log that clearly identifies which tests failed and where,
6. e-mail notification of developers when a test suite fails,
7. speed of creating tests,
8. compatibility with 16- and 32-bit versions of our client,
9. ease of modification of existing tests to create others,

10. power of scripting language,
11. portability of test suites from one PC system to another.

The log output of both products was suitable for identifying failed tests, but neither product made it possible to automatically notify a user of test failures via e-mail. We have added this feature ourselves, FTPing a customized report on test results daily to a web site where all developers can view it. We are still working on e-mail notification of failures from the test PC to the responsible developers.



Both products were capable of doing the basic recording of test scripts and playing them back automatically against a new version of the software. Our results put Visual Test clearly ahead of Vermont HighTest in creating and managing tests, although it does cost three times as much ($200 versus $600). Visual Test comes with a run-time version that can be installed on any PC to execute tests, but not record or modify them. We considered this a very useful feature because we would like to run the tests on as many client platforms as possible.

Summary

We found writing a client/server application to be a challenging task, requiring more combined skills than we have used for previous projects. We had to become much more capable at C++, learn a lot of Windows technology, delve deeply into UNIX network programming, figure out how to do PC installation scripts, find a program to automate the testing of interactive keystrokes, master PC documentation and help tools, and—it seems to never end.

However, the resulting software is so much more powerful and so much easier to use that we think the effort was worthwhile.

Web Sites

Qedit for Windows Home Page
http://www.robelle.com/products/qeditwin.html

Unofficial Microsoft Test Page
http://www.stlabs.com/mst.htm

Microsoft's Visual Test Page
http://www.microsoft.com/vtest/

Installation Setup Package
http://www.installshield.com

References

UNIX Network Programming, by Richard Stevens, Prentice-Hall Software Series, New Jersey, 1990. ISBN 0-13-949876-1.

Software Testing with Visual Test 4.0, Thomas Arnold II, IDG Books Worldwide Inc. Foster City, CA.: 1996. ISBN 0-7645-8000-0.

A Bookcase Full of C++ and Windows Books. You need at least a dozen books on C++, Windows, and MFC. No one book that we consulted stood out as complete and outstanding. Just keep buying them. When you have a problem you will refer to several books to find the answer.