by MICHAEL O'BRIEN

"I am sorry," said Holmes. "I am accustomed to have mystery at one end of my cases, but to have it at both ends is too confusing."
–A. Conan Doyle, *The Adventure of the Illustrious Client*

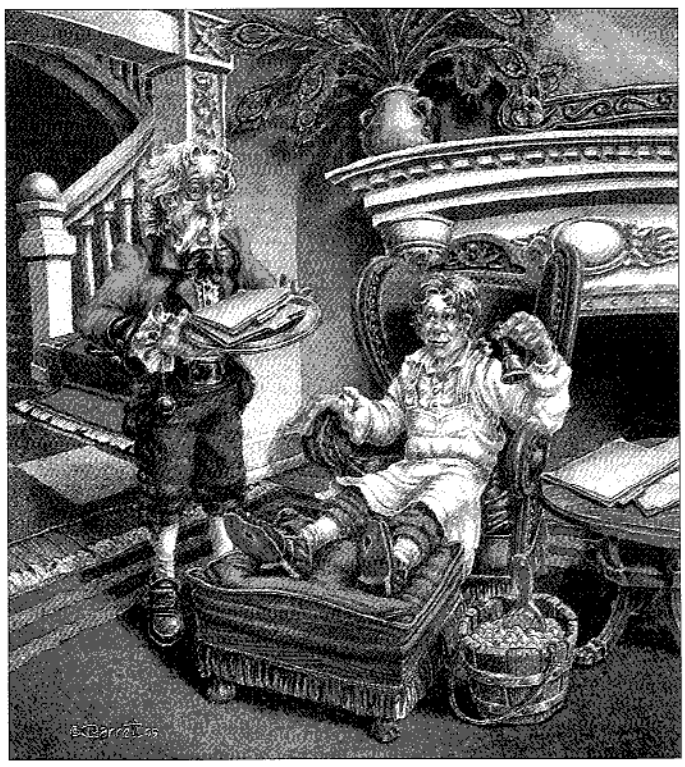
"A starship cannot run without protocols."
–Mr. Tuvak, evidently chief network engineer of the U.S.S. Voyager

"She read a book and decided that the client and server were the other way around."
–A real-world quote. Honest.

Mr. Protocol Serves a Distinguished Client

Q: *Faugh! I can't breathe! The air in here is the worst I've ever seen except for the air outside, which is even worse! What in the world is going on?*

A: I'm afraid you're in the thick of it, if you'll pardon the expression. Mr. Protocol is having one of *those* weeks, and we'll just have to humor him until the fit passes. For now, we're in London, on Baker Street. The year is 1891, and the air outside is in the grip of one of the infamous London "yellow fogs." It's a sulfurous, industrial brew. It'll kill hundreds, and Mr. Protocol is attempting to dispel it by repelling it with Turkish tobacco.



What really has me ready to resign as amanuensis is that he's got me tottering around with a distinct limp, carrying a medical bag and muttering something about a "jezail bullet." And to top it all off, he's going on and on about a client so distinguished that he can't be named. On the other hand, I've got a deal on Victorian wool trousers that'll last me until the 1990s the long way, so I suppose I shouldn't complain.

Really, I wish the computer press would just shut up about all this client/server stuff. If it weren't for that, we wouldn't be having these problems. I'm sure that's what's set him off.

As usual, Mr. Protocol's fit has its basis in the distant past of the Internet. In fact it goes back to the very beginnings of the ARPANET, when the "client/server model" was chosen as the basis for just about all network interaction. It's such a fundamental concept that Mr. Protocol is even less than usually rational in its vicinity, and even experienced programmers can be led astray by following it too slavishly.

The basic idea is simple. I'll refrain from saying "too simple" in Mr. P.'s hearing range, since he can make anything seem far from simple (I don't let him balance the checkbook since the kitchen table disappeared in what has to have been a dimensional warp). But it's not very hard. You're using a computer because you want it to do something for you. On the way to the big thing, you want a bunch of smaller things, like windows, files, a connection to another net machine and so forth. The network delivers these "resources" (a fine word, that) to you via "servers." You accept these resources through the use of "client" programs. When you run Mosaic, Netscape, telnet, ftp, tf or whatever, you are running one half of the resource delivery mechanism: the client. The server isn't (usually) run by a person. It, or some avatar of it, runs in the background all the time on the target system.

You ask the client program for the resource you want, and the server sends it down to you. The details of this depend on the nature of the thing you're trying to get, or get done. File transfer happens in one big chunk. In ftp, the resources are files: Ask for file, get file. Telnet is more interactive, and it's harder to

TOM BARRETT

define exactly what the resources are. That's why telnet has such a funny name. What it's serving is access to another machine. The interaction is continuous, though. You shovel input into the client, and the server shovels output back at you, in a (seemingly) endless stream. Netscape is more interactive than ftp, but less so than telnet—it supports one or a few transactions per mouse click in the client. `irc` is a client program that uses a special-purpose IRC server to exchange blither with other clients, on a line-at-a-time basis.

The problem, as Mr. Protocol sees it, is that as systems become more complex, and the client/server model undergoes inevitable elaboration, some people expend too much energy trying to shoehorn all possible architectures into the simple one-client, one-server, one-transaction model with which the ARPANET began.

Consider, for example, good old NFS. This is definitely a client/server relationship, but it has become skewed. No special client program is needed. Any program that tries to access an entity in the file system that turns out to be on an NFS-mounted file system is suddenly put in the position of being an NFS client. And while an NFS server can seem to be running many instances of `nfsd`, in fact many transactions take place mostly in the kernel of the server, for the sake of efficiency. So here we have a client program that isn't a client program, making requests that don't make it as far as the server. Yet the interaction is still obviously a clear-cut client/server model. It just has an identity crisis.

The client/server model has served well over the years, but it does require one peculiarity. An ordinary program (where "ordinary" is defined as "the kind you used to write in your beginning programming class") has a linear flow of control. Most FORTRAN programs are still written this way. Read the input file, do the magic, write the output file, exit. Client/server programs have to be written in a different way, though. They're implemented as loops. Mr. Protocol threatens to strangle me if I call them "infinite loops" since they do have definite terminations, but only under outside influence (i.e., the user commands them to exit).

The loop structure for a program is familiar to anyone who has written an editor, or other interactive code, especially for a windowing system. Client programs look very much like this. They read commands from the user, and based on these commands, send traffic to the server. In the case of ftp these are file transfer commands; in the case of telnet, these are lines of text, or single characters. In the case of Mosaic or Netscape, mouse clicks by the user translate into requests to the Web server on the other end of the Net connection to send pictures, more text, a new page or whatever.

Remote procedure call code involves so much standardized boilerplate that it is almost never written directly these days.

Server programs are written in the same way, but they operate in a different environment since they are not under the control of a local user. They are spawned when a client opens a network connection and requests a service, and their command loop involves reading requests from the Net and responding.

More complicated interactions are possible. Mr. Protocol, for several years, never seemed to write a program that did not involve the use of remote procedure calls, about which he has written before. A remote procedure call translates something that looks like a function call into a request over a network. A "return" from the function actually represents the receipt of a response from a network server.

Client and server programs are generally written at the same time, since they amount to the implementation of a special-purpose protocol, which is probably why Mr. Protocol has such a major reaction whenever the subject comes up. In particular, writing code that uses remote procedure calls almost always involves writing the server and the client programs at the same time, since they share much of the same code.

Remote procedure call code involves so much standardized boilerplate that it is almost never written directly these days. Instead, a sort of ad hoc language is used that specifies the name and type of the called function, the name and type of the arguments and the name and type of any returned values. This is translated by a program (Sun's version is called `rpcgen`) into a mass of C code that performs the scut work of locating the server on the network, opening a connection to it, marshaling the arguments (a fancy term for "converting the arguments into a form suitable for squirting over the Net"), shipping the request off and waiting for the response. The response must in turn be converted from network format to local machine format.

Obviously this takes considerable work, so remote procedure calls are nowhere near as fast as local function calls, unless you are using something like a Myrinet plus special software to ensure that roughly the same number of machine instructions are executed in each case. If you are running such special hardware and software, you are almost certainly aware of it already, since you are now more deaf than Quasimodo from all the people telling you how special your system is.

Absent such special stuff, remote procedure calls trade speed for flexibility and distribution. They are ideal, for instance, for writing software that pretends that a distributed database is local to the client program. This is how NFS works. System calls such as `open`, `read`, `write`, etc. are translated into remote procedure calls to the NFS file server where, as we have said, most of the easy stuff is handled directly by the kernel. Obviously this is a case where

rpcgen was not used to implement the protocol.

Mr. Protocol has found, over the years, that it is often the case that rpcgen alone is not enough to do the job, even in cases that are obvious candidates for the remote procedure call type of server/client interaction. This is because if it were that easy, someone would have done it already. It is usually necessary, Mr. P. sadly admits, to dink with, diddle and generally mess with the output of rpcgen in order to accomplish whatever peculiar and extraordinary network interaction is required. This is perfectly OK. rpcgen is a tool, not Holy Writ. One look at the code it generates and you will see that the ante-cedents of rpcgen have very little to do with anything holy.

The only messy thing about this system is that it makes automatic maintenance and generation of code difficult. The source to your program is no longer contained in the (relatively) tiny file of RPC specifications. Instead, you must provide for some automatic script in perl, sed, awk or something similar to perform the dinking and the messing with. This is just another example of the fact that the easier a programming tool is to use, the more it limits your choices. Alas.

One other common occurrence in Mr. Protocol's experience is the complete blurring of the distinction between a client and a server. People initially find the X Window System confusing because the notion of having a server do the drawing on the screen seems foreign. It would seem that the thing we see and talk to would have to be a client. However, it is a server, because the resource it is managing is your screen real estate, as well as your input devices. Paradoxically, you have to use the server to see your clients, such as xterm. Mind you, people still find the X Window System confusing even after a number of years' experience, but this is a different and more tragic matter.

Mr. Protocol has taken a flier at writing graphics programs in the past as well, and in one such case, the distinction was blurred even further. One program was responsible for display, the other for accessing the database, and these programs exhibited the sort of back-and-forth interaction generally found only in coroutines. Each program, it seemed, needed data from the other to accomplish its task. The result was a straightforward blend. Each program became simultaneously a client and a server. Each was a server in its domain of expertise. Each was a client of the other as required. The only danger in this approach is that of any recursive structure involving two entities: deadlock, where each half requires something from the other before it can act. In this particular case, the recursion was bounded and no deadlock was possible. It pays to draw pictures when you try something like this, though. If your pictures are pretty enough you can dump the program,

publish a book on how to draw the pictures, and get rich as a pundit on correct software design. If you come across well teaching seminars, your career is assured. Mr. Protocol even gives you a free name for it: Structured Object-Oriented Recursive Client/Server Programming.

What's Mr. Protocol's point in all this? He's glad you asked, and yes he has one.

The point is that it is a mistake to become too hung up on the question of which is the client and which the server. Consider one of today's typically monstrous applications: a system with a graphical user interface designed, for instance, to push files over a network to a variety of destinations. Let's add some glamour and say that we have everything in this stewpot: satellite data streams, uplink facilities, ATM switches, serial buffers, routers, crypto boxes and all sorts of people in uniform running around alternately panicked and trying to be helpful. The files are pushed up to the satellite from the uplink facility and are sprayed down to various sites around the world. (Any resemblance between this and an actual interservice joint exercise is purely coincidental. Far be it from Mr. Protocol to ever work on anything practical, though calling such a system as this practical is an obvious stretch.)

The point is that it is a mistake to become too hung up on the question of which is the client and which is the server.

Now, consider that we have several independent pieces of software working here. One is the GUI, which takes commands from the user, and another is the network module, which takes a file and pushes it out the uplink, after first converting it to some typically wacko packet format, probably made up on the spot. These two programs must communicate: the GUI must tell the file module what files to push, and the file module must inform the GUI when a file transfer has been completed, or has at least had its component bits sprayed into outer space.

If these two program pieces are going to rendezvous on, say a UNIX-domain socket, which is one of those little wonders that looks like a file but is really a socket, then which one is going to act like a server and create the socket, and which one is going to be the client and connect to it? The file module looks a bit like a server, after all: It takes requests to push files, and then pushes the files. But on the other hand, it's started by the GUI, and it's usually more convenient in

cases like that to have it turn around and connect to the "master" program right away, thereby indicating that it's begun execution and is ready to go.

The correct answer is: whatever works. There is no right or wrong way. In most situations the file module is indeed a server. Requiring it to be the one to receive the connection from the GUI, however, makes program synchronization more difficult. These things are hard enough to get working without adding needless complication to the design in a misguided attempt to conform to a model. Models exist to help us think about problems, not to force us to smash the solution until it fits the model.

For a final example, consider the new programming language, Java. Now, this one's a pip. You use a client program to connect to a Web server. The server turns around and jams executable Java code down the throat of the client and tells it to do all the work. Sort of sounds like the new model of the federal government, doesn't it? Your Web client program becomes all-singing, all-dancing and suddenly the work is being accomplished on your local workstation instead of on the machine where the server lives. People are touting this as the greatest thing since the invention of the transistor.

Certainly, Java will allow Web clients to do things they couldn't do before, if only because they can now do things that were not preprogrammed. It's a fascinating new development, but it does further blur the lines between client and server.

The correct answer is: whatever works. There is no right or wrong way.

Mr. Protocol doesn't think this should bother you much, however. By the time you get a look at what the new Java browsers can do, you'll be too busy surfing the Web to be bothered by definitions. ➸

Mike O'Brien has been noodling around the UNIX world for far too long a time. He knows he started out with UNIX Research Version 5 (not System V, he hastens to point out), but forgets the year. He thinks it was around 1975 or so.

He founded and ran the first nationwide UNIX Users Group Software Distribution Center. He worked at Rand during the glory days of the Rand editor and the MH mail system, helped build CSNET (first at Rand and later at BBN Labs Inc.) and now works at an aerospace research corporation.

Mr. Protocol refuses to divulge his qualifications and may, in fact, have none whatsoever. His email address is amp@cpg.com.