Proxynet - A Mechanism For Creating Connectivity Realms

Paul Vixie
<paul@vix.com>
Vixie Enterprises

24 October, 1995

Abstract

Proxynet is a set of library routines and network servers that can create multiple *connectivity realms.* Multiple connectivity realms exist wherever direct network connectivity is impossible for toplogy reasons, or proscribed for administrative or security reasons. Applications which must be able to reach servers (or be reached by clients) in the other realms can use Proxynet as their agent.

1. Problem Statement and Definitions

Sometimes the full connectivity model of the Internet is nonideal. In these cases it has become common to install firewalls, route filters, application layer gateways, and other devices designed to provide connectivity only within some narrowly defined set of constraints.

Given RFC 1597, it is even becoming common for network architects to violate the original IP model by using what we call "ambiguous addresses," where the same network number is used by multiple unrelated, discontiguous networks. In that situation, a network's location in the graph of interconnects becomes a crucial element of its identity. A host which can reach more than one network with the same network number considers such a network number to be "ambiguous," and some means must be employed by which a host selects and reaches the appropriate network for any given connection or transaction.

No matter whether the connectivity constraint is due to network number ambiguity or to administrative policies (to implement security, for example), any set of networks which does not permit the full interconnectivity provided for in the IP model can be said to have multiple *connectivity realm*s.

We expect the reader to have a passing familiarity with the Internet Domain Name System, Internet routing concepts, and BSD system calls.

2. Introduction to Proxynet

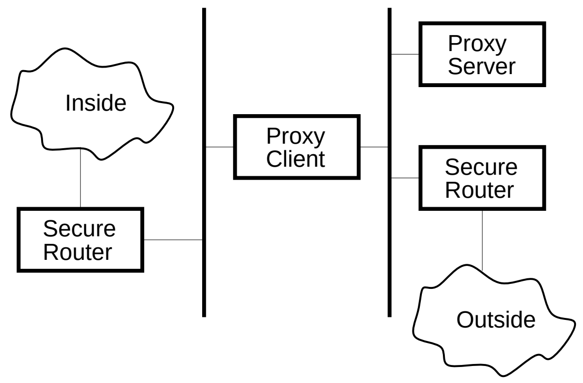
Proxynet is an application layer tunnel. A network server or client which wishes to do business in connectivity realms other than its native one can use Proxynet as its mechanism for reaching the other realms.

Proxynet consists of a set of library functions that intercept system calls in a BSD style operating system, and a daemon that runs on the gateways to other connectivity realms. An application linked against **`libproxynet.a`** will have several of its system calls intercepted, such that rather than being direct downcalls into the kernel, these calls give control to the Proxynet library long enough for Proxynet to decide whether the system call has any relevance to it. Some system calls are allowed to proceed unimpeded, some are given new side effects, and some have their results cached inside Proxynet for use in future decisions. One of Proxynet's chief design goals was to maintain all relevant invariants in the system call set; to that end, most applications we have tested against have not required any source changes. We succeeded pretty well at this goal; exceptions are **`ftp-gw`** from TIS, which had a bug, and **`ftp`** from BSD, which violated one of our assumptions. Also, many applications do not call **`openlog`**, which renders Proxynet's internal **`syslog`** calls mute; we consider this inconvenient.

Recent BIND resolvers have *hooks* which are called by **`res_query`** before sending a request and after receiving a response. These hooks permit the query or the response to be modified in transit, or to have the resolver's name server list modified depending on the query contents so that certain queries can be directed to the appropriate name server. We use this latter feature so that we can be sure to ask a name server whose root is in the same connectivity realm as the object.

The Proxynet daemon must run on some *gateway* host which can reach only a single instance of the ambiguous network number. That is, while the network is in fact ambiguous, due to routing policies it is unambiguous to its gateway host. The gateway host must be unambiguously reachable from the client host. It is possible for all instances of an ambiguous network to be reached via Proxynet and therefore served by gateway hosts. However, that level of complexity and cost is not necessary. Once the routing has been arranged so that all but one instance of the ambiguous network number's route are not advertised to the client host, that remaining instance is not, from a routing perspective, ambiguous. The design philosophy we have used in our networks is to choose the "nonproxy" instance of an ambiguous net such that most traffic from the Proxynet client hosts does not have to pass through a Proxynet gateway.

Consider the following diagram:



Imagine that many hosts on the *Inside* network are using addresses in the internet class A network 42.0.0.0, but that this network has been officially allocated to a completely separate organization out on the *Outside* network. The *Secure Router*s reject packets that do not reflect the organization's communications requirements, and also ensure that the route for the *Inside* 42.0.0.0 is never advertise to the *Outside* network. The *Proxy Client* has an ambiguity problem, in that when it discovers a need to reach a host on net 42.0.0.0, it has more than one choice, since from its point of view, there is more than one net using the 42.0.0.0 address. If an application on *Proxy Client* sends a packet or opens a connection to a host on 42.0.0.0, it will reach the one on the *Inside* network. Similarly, if *Proxy Server* (or any other host on the *Outside* network) sends a packet to a host on 42.0.0.0, it will reach the 42.0.0.0 on the *Outside* network.

The function of Proxynet in this situation is two fold: first, it must disambiguate connection requests such that when a connection is made to a host on 42.0.0.0, the "right" 42.0.0.0 is chosen. Second, it must provide the means by which the "other" 42.0.0.0 is reached, when required.

3. Theory of Operation

Fundamentally, all BSD network connection endpoints are brought into existence via the (**`bind`**, **`listen`**, **`accept`**) or **`connect`** system calls. By intercepting these calls, we can redirect outbound connections or ensure that corresponding inbound listener sockets are set up in other connectivity realms. The system calls **`getsockname`** and **`getpeername`** are an application's only reasonable means of learning the addresses of the endpoints of a connection; by intercepting these calls, we can make the application believe what we want it to believe instead of having it learn the details of internal Proxynet connections whose addresses might be surprising to the application.

Let's take a look at each system call that we intercept, and explain what we do and why. While we're at it, let's show the resolver hooks and learn what's going on inside them.

3.1. `res_qhook`

When the resolver is making a query, either as a result of an application like Sendmail that calls **`res_query`** directly, or as a call from the high level **`gethostbyname`** or **`gethostbyaddr`** routines, we are given an opportunity to examine and perhaps modify the query before it goes out. We do not presently modify the query, but we do examine it to the extent of unpacking its query name and looking that name up in the library's configuration file. If this name is known to be *proxied*, then we temporarily overwrite the resolver's list of name servers with the list of servers known to be able to answer queries for the proxied name. In our configurations so far, these *proxy name servers* have always been the same as the Proxynet gateway hosts themselves, though this is not a requirement.

3.2. `res_rhook`

We do not currently make any use of this hook. **`res_rhook`** is called after a reply has been received by **`res_query`** and we could use this opportunity to modify or examine the (*name*, *address*) tuples in the reply, but so far we have not needed to do so.

3.3. `bind`

When a socket is bound to a *wildcard* port number in IP/TCP or IP/UDP, the kernel's response is to select a port number at random. By the time we intercept the **`listen`** call (see below), it is not possible for us to determine via **`getsockname`** whether the port number was originally wildcarded or not since the kernel has "filled it in." Therefore we capture this bit of information when the **`bind`** call is made. Later when we intercept the **`listen`** call, we will propagate the "wildcard" semantic to the Proxynet daemon so that if the application did not care what the port number was, Proxynet won't care, either.

3.4. `listen`

Here we use **`getsockname`** to find out the name of the socket, and if the address is one we should (according to our configuration file) be proxying for, then in addition to allowing the system's **`listen`** call to proceed, we contact the appropriate Proxynet daemon(s) and have them set up a *proxy listener*. Those listeners will then forward all incoming connections to our application's real listener socket.

Sometimes a listener socket needs to be proxied even though it does not use an address or port number which is registered as proxiable in the Proxynet library's configuration file. For instance, an FTP data connection is established by the server toward the client; the client's socket is just opened on some arbitrary TCP address and port number whose name is sent to the server over the FTP control connection. We handle this condition by maintaining an association tree of sockets, such that when a **`listen`** is done on a socket which is *associated* with a proxied socket, then the listener socket is proxied also. This requirement bent our "no client source code modifications" model since in the case of FTP, there just was not enough hint information without the following patch, which we include here to show the worst thing we've had to do:

```
***************
*** 1030,1034 ****
    perror("ftp: setsockopt (ign)");
  len = sizeof (data_addr);
! if (getsockname(data, &data_addr,
                 &len) < 0) {
    perror("ftp: getsockname");
--- 1030,1037 ----
    perror("ftp: setsockopt (ign)");
+ proxhintFDset(fileno(cin), data);
+ if (listen(data, 1) < 0)
+   perror("ftp: listen");
  len = sizeof (data_addr);
! if (getproxname(data, &data_addr,
                 &len) < 0) {
    perror("ftp: getsockname");
***************
*** 1036,1039 ****
  }
- if (listen(data, 1) < 0)
-   perror("ftp: listen");
  if (sendport) {
--- 1039,1040 ----
```

The reason this patch is needed is that the FTP protocol has a **`port`** verb that requires the address of the client's host to be encoded into a text stream and transmitted to the server. Since Proxynet cannot see or edit the TCP stream used to transmit this encoded address, some other means must be employed to ensure that the address received by the server is of the Proxynet server rather than the real client. Furthermore, Proxynet needs to know when it sees the **`listen`** call that this listener ought to be proxied. We had originally thought that just intercepting the **`getsockname`** call would be enough, but it turns out that the **`listen`** was being done too late, and that the proxy status of the socket was ambiguous anyway. We added a simple call into our Proxynet library and changed a **`getsockname`** call to a **`getproxname`** call. Naturally, a correct patch would use However, Proxynet is smart enough to stay out of the way if its services are not required – so the above patch does not interfere with nonproxied operations.

3.5. `accept`

We intercept **`accept`** because the application is capable of blocking here waiting for an incoming connection. Because the proxy listener runs asynchronously, we have no easy way to tell whether it has died or even if its host has been rebooted and all context lost. Therefore, rather than block in a system call that waits for connections, we enter a **`select`** loop with a short timeout. If the file descriptor of the intercepted **`accept`** ever becomes readable, this means that an **`accept`** would not block and we can permit the real system call to proceed. Otherwise, we wake up every short while and ensure that all control connections to our Proxynet daemons are still healthy. If any become sick, we redo the connection and start up another proxy listener for our application's socket. In this way we become resilient to gateway host reboots or Proxynet daemon crashes. The actual details are more obscure since the Proxynet daemon and the proxy listeners are unsynchronized. But these are essentially the facts.

The other thing we do, after the system's **`accept`** call has completed and we know we have a new connection on our hands, is to look up the actual initiator of the connection using **`getpeername`**. If the peer address is one of our Proxynet gateways, then we ask that host's Proxynet daemon whether this was a proxied connection or not. In this way, we can distinguish between real connections that happen to come from a host which has a Proxynet daemon, vs. proxied connections where the peer information is actually way out in the other realm somewhere.

3.6. `select`

For the same reasons stated above for **`accept`**, we must intercept **`select`** and if it could block, we loop in our own **`select`** with a short timeout, periodically verifying the health of any proxy listeners we have running. Note that we do not depend on TCP *keepalives* since they take several minutes before closing dead connections, and we need faster service.

3.7. `connect`

In **`connect`**, we need only look up the destination address to see if it is something we should be proxying for. If it is, then the appropriate Proxynet daemon is told the destination, and after it sets up its own listener and bidirectional data forwarder, we **`connect`** the application's socket to the proxy forwarder. If the destination is not proxied, we run the real system call with the application's given destination.

3.8. `close`

We use the opportunity of a **`close`** call to clear out all our internal hints about a file descriptor (e.g., whether it is a proxied connection, and so on). Since **`close`** is often called very early in the application's initialization (in fact, the resolver initialization is known to call **`close`**), this also gives us a chance to ensure that the resolver hooks are installed before the resolver is actually called.

3.9. `getsockname`

`getsockname` must be made to return an address on the Proxynet gateway if the socket is connected to its destination through Proxynet, since the application may encapsulate this address in messages its sends to its server (e.g., FTP's "PORT" verb). This can cause consternation inside applications, since it is usually reasonable to assume that the address shown by **`getsockname`** is one of the interfaces on the local host. We have not encountered such an application yet but we recognize the possibility that it may become a problem in the future.

3.10. `getpeername`

`getpeername` must be made to return the address of the remote client in the alternate connectivity realm, if the socket is connected through Proxynet. This can, like our new semantics for **`getsockname`** (see above) cause some confusion inside an application since the address returned by **`getpeername`** may not be in the same IP routing domain as the local host. However, since we will intercept DNS lookups for PTR RRs for this address and we will intercept **`connect`** calls to this address, we consider it mostly safe. The only area of concern is an application like INN which maintains many open TCP connections and might end up using potentially ambiguous peer addresses as if they were unique. So far this has not been a problem, even for INN.

4. Configuration Files

Proxynet follows the tradition that if a system is complicated and hard to understand, then its configuration files ought to be complicated and hard to understand, too.

Proxynet's configuration files follow the usual conventions with regard to comments ("**`#`**" anywhere on a line introduces a comment which lasts to the end of the line) and blank lines (which are ignored). Blanks and tabs which follow a delimiter (such as a "**`,`**") are ignored. A run of spaces, tabs, and/or newlines are the same as a single space character.

Configuration elements are of the form:

key value-list [...] **`;`**

A *value-list* is of the form:

value [**`,`** *value*] [...]

4.1. Client Side

/etc/proxynet.conf contains elements whose *key* is **proxy** and whose *value-list*s are ordered as *addresses*, *domains*, *proxy servers* and *name servers*. Each **proxy** element describes an alternate connectivity realm. An example appears in the appendix XXX.

addresses
: The set of addresses which we will proxy for if we encounter them. For IP, there are two forms here. A *wildcard* address with a *nonwildcard* port number indicates a *listener* address; this matches attempts to **listen** on a socket which is bound to the 0.0.0.0 address on the specified port. For example, the address **ip/tcp/0.0.0.0/119** would tell an NNTP server which had been linked against **libproxynet.a** to open a proxy listener in the connectivity realm this address appears in. The other IP form is where the address is a *network* or *subnet* and is followed by an "**&**" and a *mask*. Either form can be preceded by an "**!**" which tells the Proxynet library that if the address matches, no further searching should be done and the address should *not* be proxied. This is useful if there is a wildcard entry further down the list, which for IP would look like **ip/tcp/*/0&ip/tcp/*/0**. For completeness, an example of a negated network entry would be **!ip/tcp/42.0.0.0/0&XXX**. Blanks around the **&** are not allowed, due to parser limitations.

domains
: The domain list tells the DNS resolver whether or not to use a proxy name server. If no match is found, the client's usual name servers will be used to resolve the name. If a proxy name server is used to resolve a name to an address, then that address is marked *proxied* for the duration of that application's execution, even if it does not appear on the *addresses* list. Like *addresses* list elements, each *domains* list element can be preceded by an "**!**" to stop the search with an immediate negative answer, in case a wildcard appears later on. A wildcard on the domain list is simply "*****". Shell style "*****" patterns are permitted, with an implicit "***.**" on the left hand side. Thus **!*.sony.com** is the same as just **!sony.com**. Don't forget to put in your IN-ADDRs; for example, **!*.42.IN-ADDR.ARPA**.

proxy servers
: This is a list of Proxynet gateways. If there is more than one, each application will use "round robin" ordering to effect a primative form of load balancing. An example would be **ip/tcp/198.93.3.1/555**.

name servers
: This is a list of DNS servers to be used when resolving names that match something in the domains list. Strictly speaking, only the **ip/** elements are of interest to DNS, since the resolver won't try to reach name servers using any non-IP address family. We expect the definition of the *name servers* list to be expanded somewhat when Proxynet is ported to a non-IP protocol.

4.2. Server Side

5. Things We Did That Were Cool

Every project should have beneficient side effects or little surprise packages in the form of useful artifacts or new methods that can be put to other uses in the future. We believe that this is very much true in Proxynet's case.

5.1. Server Template

In the process of implementing the server side of the Proxynet protocol, we realized that we had written the same program a lot of times previously, but with different protocol details. We therefore separated out the initialization, command loop, parsing, reporting, and error propagation in their own private module. The part of the program that knows protocol details is completely isolated. We believe that this template will be of use to other protocol implementors who want to concentrate on implementing the protocol, not a server infrastructure.

5.2. Multiprotocol Readiness

While the current implementation only supports IP/TCP, every effort has been made in the protocol and in the code to leave room for other protocols such as DECnet, OSI/ISO, or even IPng. Naturally, the proof of our success at this goal will come when at least one other protocol is supported.

Within that constraint, however, we are intrigued by the possibility of using Proxynet as an infrastructure around which application layer tunnels can be built. Some connectivity realms could be islands of IPng or DECnet surrounded by an existing network of just plain IP.

5.3. Memory File System for Metadata Storage

We determined that the Proxynet daemon needed a reliable, easily examined database with strong atomic locking for its list of currently proxied connections, and we chose to use the BSD file system since it has all of these attributes. However, performance was mediocre since every connection opened or closed caused an inode and/or directory update, which are synchronous operations (that is, the disk head has to wriggle and the platter has to rotate for a while before the application is allowed to continue).

BSD provides a *Memory File System* which has every semantic of a normal file system except persistence across reboots. MFS keeps its data blocks in a process address space, which is pageable if the system runs low on real memory. This turns out to be exactly what we need for our database, since inode and directory operations that are usually synchronous to mechanical events are instead synchronous only to a few context switches. Our Proxynet gateway system's **`df`** display contains two *MFS* file systems rather than just the customary one for **`/tmp`**, to wit:

```
Filesystem KBytes  Avail Mounted on
/dev/sd0a    7924    377 /
mfs:16      31419  28273 /tmp
/dev/sd0d  201636    269 /usr
/dev/sd0e  242432 163563 /var
/dev/sd1a    8423   7579 /bak
mfs:18       3711   3334 /var/run
```

5.4. Generic Socket Address Library

Since our protocol uses printable text for its verbs and responses, we found a need quite early on to standardize on the print format of network endpoints. Our results in this area may have applicability well beyond Proxynet, since the Internet community itself has no single standard for endpoint display and entry formats. Each protocol tends to have a standard form for printing and entering host addresses; in IP we have the "dotted quad", while in DECnet we have the *area***.***node***::** format. But when it comes to complete endpoints, there is no universal specifier. The closest thing we have is the leading component of a WWW URL, but it assumes TCP and as such, was not suitable for our needs.

First, we chose delimiters that were not valid or customary characters in any of the protocols or implementations we had on hand, or which were already delimiters in those protocols. Our delimiters are slash ("/") and space (ASCII SP or HT). Slash is safe because the only address family we know of that uses it is the UNIX Domain, where it is already a delimiter. Spaces are safe because only Appletalk, of the protocols we know of, uses it; we expect to define an escape mechanism if we ever port Proxynet to the Appletalk address family.

Second, we chose a hierarchy for several common protocols, with the high order terms being given earliest in the string. Ultimately, all we need to define is the leading term (which is a constant in each address family) and the structure of the following terms. For IP we chose, not surprisingly, **`IP`** as our leading term, followed by the *protocol* (**`TCP`** or **`UDP`**), followed by the *host address* (in dotted quad form – no hostnames allowed), followed by the *port number* if the protocol requires one (as UDP and TCP both do). In practice this looks like the following:

```
ip/tcp/127.0.0.1/23
ip/udp/*/53
```

The first example is the Telnet server on the local host (127.0.0.1 being a BSD convention for reaching the local host, and 23 being the Telnet port number). The second example is a wildcard: it is the DNS port on *all* interfaces of the current machine. Wildcards of this form are useful for server ports, where one wishes to receive packets sent to any interface of a multihomed host. Wildcard port numbers work, also, and are used in the degenerate case of speaking of a host rather than an endpoint. In fact, a wildcard port number will be imputed on entry if none is given.

Some other protocols and their *socket specifiers* (also called *sockspec*s) are:

```
dn/10.853/11
dn/11093/11
un/dev/printer
```

We have a very complete library of C functions which parse, format, and otherwise manipulate these *sockspecs.* We recognize a need in the Internet community for this specification, and as time permits, we will write an RFC on the socket specification syntax and the C API of our library.

6. What We Didn't Do And Why

6.1. UDP and Datagrams In General

To do Proxynet for datagrams, we would need to intercept the **`sendto`**, **`send`**, **`write`**, **`writev`**, and **`recvfrom`** system calls. Each datagram sent would need to have its alternate realm source and destination addresses encapsulated and then peeled off by Proxynet when forwarding into the remote realm. Each datagram received would need to have its alternate realm address put into the **`recvfrom`** data so that the application would not be confused by seeing a Proxynet gateway address where it expects to find a remote client address. Datagram applications which use unconnected sockets and scatter their destination addresses could end up running the “proxy decision” logic on every outbound packet. The Proxynet gateway would have to have some kind of timeout since there is no way to identify the end of a datagram session – and there may not be one. Any timeout we put in would ultimately help us find some application that uses extraordinarily long delays between adjacent packets.

In other words, we didn’t do UDP because it’s *too hard*.

6.2. Source Routing

Source routing seems attractive since at least in the IP case, there is a form (“strict” source routing) where successive addresses in the source route need not be from the same connectivity realm. We were concerned, however, that many routers on the wide open Internet implement source routing poorly, or in Screend’s case, not at all. Then also, we wondered if we could make our own routers secure against source routing *attacks* if we enabled it for use in our production system. Finally, we considered the possibility that IPng might deprecate source routing altogether, and while Proxynet’s role in an IPng universe is less than clear, we thought that this was sufficiently risky to void this approach.

6.3. Stackable System Calls

In order to intercept a system call in a BSD system, you must have your library provide a function with that system call’s name, so that when the application is linked against your library, it gets your versions of various system calls instead of the “real” ones. If at some point in your version of the system call, you determine that you need to call the “real” one, you do it with the **`syscall`** system call.

While the ability to do this is a wonderful thing, it does not scale well. If another library (say, for example, Prospero) wanted to intercept some of the same system calls that Proxynet does, it would be a case of “first one wins”, with respect to the ordering of the **`-l`** options on the **`cc`** or **`ld`** link command. This could lead to chaos if some of a library’s system call intercepts were taken, and others not.

What we need is a way to dynamically push system call intercept functions onto some kind of a stack, so that each is given an opportunity to work its magic and each has the easy option of deferring to the next layer of the stack if it discovers no present use for its unique talents. We considered implementing this sort of stackability, but we considered the chances of its wide adoption to be rather slim, and the consequences of Proxynet being its only user to be rather grim. A solution is needed, but Proxynet is blazing enough *necessary* new trails and this was one that could be left for another day’s work.

6.4. Intercept `fork`

When a library uses static variables, it is subject to a mind boggling hoard of thread safety and multiprocessing issues. We have not invested the effort in making Proxynet “thread safe,” since the usual method of doing that requires changing the calling interface, which is a course explicitly denied to us. However, we did run into some problems with multiprocessing, and as this is a more common programming tool than multithreading, we had to solve it.

What happened was that as applications that were linked against Proxynet would **`fork`**, all of the internal file and socket descriptors used by Proxynet would suddenly be shared by the parent and child. This led to protocol errors where a child could and did consume responses intended for the parent, and vice versa. Our first cut at a solution was to intercept the **`fork`** system call so that we could take the opportunity to close all the descriptors in the child, forcing the child to reinitialize Proxynet and form its own connections as needed.

However, we then encountered Sendmail, which has a feature called “freeze files,” wherein the process’ data space is committed to disk and later restored. Since Proxynet’s static variables were also committed and restored, this put us in the untenable position of having seemingly initialized internal state but no open file or socket descriptors. Sendmail has since lost this interesting feature, but we recognize this as a general class of error that we wish to avoid.

So we removed our intercept for **`fork`**, and instead, added code to detect the case where the process identification (PID) suddenly changed. As soon as Proxynet notices that this has happened, it reinitializes all of its internal state, including **`free`**ing all of its dynamic memory, closing all of its possibly-still-open file descriptors, reloading configuration files, and so on. We are not happy about this but know of no alternative.

6.5. Kernel Implementation

We recognize that doing our magic in user space has certain pitfalls, not the least of which is that all applications must be relinked and a few must be modified. If the kernel could maintain the "proxy" attribute as part of its internal connection state, many applications could run unchanged.

Attractive as the kernel is in this instance, we elected to work in user mode for several reasons:

1. One of the tenets of the UNIX kernel philosophy is that anything which can be done in user space, should be done in user space. We respect and agree with this philosophy.

2. On systems with shared libraries, ubiquitous proxying can be had by editing those libraries. Dynamically linked applications would begin proxying automatically.

6.6. Use SOCKS

The SOCKS system (by Koblas, et al) accomplishes much the same goals as Proxynet. We do not agree with many of the design decisions made in SOCKS, such as its binary protocol (which saves precious little space but makes debugging quite a bit harder), and its requirement that the application source code be modified (to use **`rconnect`** rather than **`connect`**, among other things). However, the real reason we found SOCKS unsuitable is that it does not solve the fundamental "multiple universe" problem that Proxynet does.

Proxynet assumes that there can be multiple connectivity realms using names or addresses that overlap and are therefore ambiguous. It integrates the selection of remote objects at the Resolver level, retaining state about (*name*, *address*) tuples known by the resolver so that the "proxy" decision can be made automatically by the libraries. SOCKS, to the best of our knowledge, assumes that each application will explicitly make the "proxy" decision and make the appropriate library or system calls to get where it wants to go. We were not willing to extensively modify every application, since it would have created a maintainance nightmare with respect to integrating new releases of the vendor's operating system into the gateway.

In all fairness, we did not realize how much overlap there would be between Proxynet and SOCKS until we were almost finished. Had we known, we might have chosen to extend or revamp SOCKS rather than starting from scratch. Having finished our own system, we now feel that literally everything SOCKS can do, Proxynet can do better; the reverse is definitely not true. As a parting shot, SOCKS does not provide any facility for load balancing; if the traffic load warrants multiple proxy gateways, Proxynet will automatically load balance incoming and outgoing connections between them. SOCKS can use multiple servers for "fallback" but not for load balancing.

7. The Proxynet Protocol

The Proxynet protocol is in the usual ARPA style, with verbs and arguments flowing in one direction, numeric result codes and detail elements flowing in the other direction, and CR-LF pairs acting as record separators. We did a few of the usual extra UNIXy things like permitting bare LFs as record separators. Continuations are, as usual, indicated by a dash ("`-`") immediately following the numeric result code. Numeric result codes on continuation lines are transient or meaningless – the last one "wins."

We did an unusual thing, which was using the **`<`** and **`>`** characters in responses as delimiters to tokens of information which may be of interest to the client. Clients are expected to ignore the text outside the (**`<`**, **`>`**) characters rather than treating it as a template. The protocol specifies only the numeric response codes, and the order and meaning of the response tokens, not the intervening text.

Protocol details are shown below. Note that in our protocol trace examples, we have broken the example text with \ characters, which are not part of (or permitted by) the protocol.

7.1. `test` – test the address parser

```
test ip/udp/127.1/53
250 <ip/udp/127.1/53> \
    is <ip/udp/127.0.0.1/53>
```

`test` allows a developer to test her assumptions about how a *sockspec* will be parsed. As you can see above, the result is just an echo of the input, along with an expanded version of the input. **`test`** is not a required verb, but we like it. The response tokens for **`test`** are the input address, and the parsed/regenerated version of that address.

7.2. `conn` – connect to a remote server

```
conn ip/tcp/16.1.0.2/23
201 <ip/tcp/42.128.1.1/1023> \
    listening (pid 6824)
```

`conn` causes the Proxynet server to initiate a connection to the specified destination, **`gatekeeper.dec.com`**'s Telnet service in this example. The response token in this case is the *client local address* (*cla*) of the proxy connection. The appropriate action on the client's part at this point is to initiate a new connection to TCP port **`1023`** at IP address **`42.128.1.1`**. Indeed, executing the shell command `telnet 42.128.1.1 1023` in another window after running the above example gives the `login:` prompt from Gatekeeper.

The proxy connection will time out in a minute or so if no client connects to it. The connection must come from the same host that initiated the Proxynet session, for security reasons. The Proxynet server itself is not blocked by this operation, and so a client can initiate multiple simultaneous connections, and need not "consume" them in any particular order. We feel a need for some kind of quotas in this area but we're still considering the details.

7.3. `list` – list all open connections

```
list
250-<ctl ip/tcp/192.68.129.10/1024\
     cla ip/tcp/192.68.129.10/119 \
     cpa ip/tcp/*/119 \
     spa ip/tcp/*/119 \
     sra ip/tcp/*/119 \
     flg 0x3>
250-<ctl ip/tcp/*/119 \
     cla ip/tcp/192.68.129.10/119 \
     cpa ip/tcp/198.93.3.1/4712 \
     spa ip/tcp/*/119 \
     sra ip/tcp/16.1.0.18/1366 \
     flg 0x0>
250 <>
```

The above example has been deliberately truncated by us since the actual list of connections was quite long. **`list`**, as you can see, shows a list of all proxy connections currently open through the Proxynet gateway. Each response token is a complex string which is a list of (*name*, *value*) pairs. The end of the list is denoted by an empty response token. Note that the existing implementation happens to use a separate numeric response code for each list element, with continuations; this is not required other than by the ARPA style 990 character line limit. Client implementations must be prepared to parse multiple list elements per response line.

You will need to follow along very closely with the example to understand this discussion. The (*name*, *value*) pairs are as follows:

`ctl` *control address*. This is either the address of the client's of the Proxynet session that created this connection or listener, or, on incoming connection created through a proxied listener, it will be will be the `spa` of the listener.

`cla` *client local address*. This is the address of the endpoint on the client host.

`cpa` *client proxy address*. This is the address of the endpoint on the Proxynet gateway, in the client's connectivity realm.

`spa` *service proxy address*. This is the address of the endpoint on the Proxynet gateway, in the remote connectivity realm.

`sra` *service remote address*. This is the address of the endpoint on the remote host, in its connectivity realm.

`flg` *flags*. This is an internal mask of boolean bit masks. `0x3` happens to mean that this is a *listener* (as opposed to the default, which is a *connection*), but conforming clients are not allowed to interpret this field at this time.

7.4. `find` – locate open connections

```
find sra ip/tcp/16.1.0.18/1366 ctl
250 ctl <ip/tcp/*/119>
```

`find` is the underlying mechanism of the client's proxified versions of the **`getsockname`** and **`getpeername`** system calls. The parameters are (*key*, *value*, *result*) where *key* and *value* identify the connection being inquired of, and *result* is the field whose contents we want to learn. It is an error to specify a *result* which is equal to the *key*. In the above example we asked the control address of the first connection whose remote address is **`16.1.0.18`**, and we found that it was an NNTP server.

Care must be taken with **`find`** since several fields are context dependent, having meaning only in the connectivity realm they came from. Also, since **`find`** stops on its first match, care must be taken to specify a *key* and *value* which will be unique if present. We recognize the need to specify multiple (*key*, *value*) pairs so that the search can be narrowed and unique matches found among a wider input set. **`find`** does what we needed for our **`getsockname`** and **`getpeername`**, but it is not as generally useful as we would like.

7.5. `lstn` – create a listener

```
lstn ip/tcp/42.1.3.2/119 \
     ip/tcp/*/100
201 <ip/tcp/*/100> listening \
    (pid 85)
```

`lstn` creates a *listener*. The parameters are *cla* and *spa*. The above example will cause connections to TCP port 100 on any interface on the Proxynet gateway to be accepted and then forwarded to port 119 on the local host (whose IP address happens to be **`42.1.3.2`** in our private connectivity realm. It is possible to create listeners that point to other hosts than the client's, but this is better done by clients running on each host that requires a proxy listener. The hard part of using **`lstn`** turns out to be finding the address which is "closest to" the Proxynet server when the client is multihomed. Generally the result of **`getsockname`** on the Proxynet session connection is suitable for this use.

If the Proxynet server cannot bind to the given *spa* due to a specific error condition called "address already in use," it will sleep for 60 seconds and try again. If it tries and sleeps three times, it reports a failure. The reason for this is that when a TCP connection dies, its port number becomes unusable by other TCP connections on that host for a period of about three minutes. This is called the "TIME_WAIT" feature in BSD networking. An example of this is shown below. Note that the client pays no attention to any numeric code other than the last one.

```
lstn ip/tcp/*/119 ip/tcp/*/119
231-EADDRINUSE, sleeping
231-EADDRINUSE, sleeping
505 Address already in use
```

7.6. `help` – show a brief help message

```
help
250-test    a           \
    (test sock addr 'a')
250-conn    sra         \
    (connect to 'sra')
250-list                \
    (list connections)
250-find    srch a sel  \
    (find 'srch', report 'sel')
250-lstn    cla spa     \
    (lstn as 'spa', fwd to 'cla')
250-help    [cmd]       \
    (show command syntax summary)
250-noop    noop        \
    (do nothing)
250 quit                \
    (close connection)
```

When running Proxynet by hand (usually via `telnet`), it is sometimes necessary to jog one's memory as to exact command syntax and details. As can be seen from the above example, **`help`** also takes a command name as its optional parameter. If specified, only the help text for that command will be shown.

7.7. `noop` – do nothing

Every protocol needs one. 'Nuff said.

7.8. `quit` – end the protocol

```
quit
250 Goodbye
```

This terminates the Proxynet session, causing the Proxynet server to close its end of the control connection.