Protocol Independence Using the Sockets API

Craig Metz
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
cmetz@inner.net

Abstract

The BSD sockets API provides abstractions and other features that help applications be protocol-independent. Unfortunately, not all of the API is abstract and generic, and many programs do not use the APIs in a protocol- independent way. This means that most network programs, in practice, only work with one layered set of communications protocols – usually TCP over IP. This hinders compatibility with older protocols and deployment of new ones, and is making IP a victim of its own success.

During the course of next-generation IP development, implementors worked to convert protocol-dependent applications into protocol-independent applications. Along the way, they defined new interfaces to fix some problems and they found a number of usage problems that lead to protocol dependencies.

This paper explains many of the problems encountered, using examples from freely available software, and how to solve them. It also explains many of the new protocol-independent interfaces.

1 Introduction

The single most painful lesson learned by implementors of next generation IP proposals (such as IPv6) was how deeply most network programs are dependent on the network protocol that they were originally written to use. The widespread success of the IP Internet has put it into the position of being the only network protocol that matters for most network applications, and so there is currently little incentive to support anything else.

Even today, this is a problem. There are other network protocols that are in use today – such as Appletalk, ATM, AX.25, DECnet, Frame Relay, IPX, and OSI – and few applications actually support them. This is also a serious problem for the future, as any research into new network protocols is greatly constrained by the lesson of IPv6: that anything not IP will not be supported by the applications people want to use, and that anything that is not supported by existing applications will encounter great difficulty gaining acceptance.

The core of the BSD sockets API, especially the actual system calls, is not tied to any particular protocol. The problems fall in two major categories: supporting APIs that are protocol-dependent, and poor programming practices that are common. There have been great advances made in fixing the networking APIs in the system libraries though the efforts of the IETF's IPng working group[1] and the IEEE's POSIX p1003.1g (networking API) working group. New library functions, data structures, and pre-processor symbols together allow addresses and other network properties to be treated as variable-length abstract objects whose internal format can be changed without the application's involvement. But there is still a serious problem of programmer education, which in turn requires good documentation of the problems and how to solve them. To date, this documentation still does not exist.

In the NRL IPv6 implementation[2], standard networking applications from BSD and from the Internet were taken and modified to support protocol independence. For most applications, this was straightforward once we had done a few applications and knew what to look for. The end result was not only a conversion to allow the applications to support almost any protocol, but also a significant cleanup of the applications' code.

In this paper, I will first discuss the problems that need to be solved to make programs protocol-independent: what is wrong, why it is wrong, and how it can be done right. I will then discuss the new protocol-independent API functions and compare them with older BSD networking API functions in light of the problems that need to be solved. I will also present in more detail some additional functions that we found necessary to solve certain problems that have not yet been standardized. Specific examples from freely available networking programs will be used.

2 Protocol Independence Problems

Fundamentally, the problem with protocol independence is that software has not been written with the intent of being protocol-independent. Some common programs, such as Sendmail, support multiple protocols, but are still only capable of operating with certain protocols that they know about (and usually only one protocol really works). A protocol-independent application hides away knowledge of particular protocols into run-time abstractions that allow the same operations to apply regardless of what actual protocols happen to be in use. In some cases, it is not possible to make operations completely generic, in which case protocol-dependent code needs to be carefully guarded and some reasonable default actions must be available for other protocols. But most importantly, programs must be tested with several different protocols to prove that they can handle them. The jump from supporting one protocol to two is the biggest hurdle, and clearing that in a reasonable way makes supporting other protocols much easier.

2.1 Hard Coding

The most obvious protocol dependence problem seen in network programs is to hard-code the use of one protocol. Figure 1 shows an example from 4.4BSD-Lite2[3]'s telnet program. There are three major problems here. First, the protocol family to be used is hard-coded as `AF_INET`. That basically prevents protocols other than IP from being used. The family needs to be chosen based on the name resolution information, as will be discussed later. Second, the socket address used is a protocol-dependent address, in this case `sockaddr_in`. This structure is not big enough to hold addresses for some protocols, and in any case manipulating the fields in the structure itself is a protocol-dependent activity. Sockaddrs need to be treated as an opaque buffer manipulated by protocol-independent library functions or carefully guarded code. Third, IP-specific socket options are being used without any guards. That is, if the first two problems were fixed, the IP-specific `setsockopt` calls would still be done and they should always fail. Depending on the particular option being set, the socket option call needs to be replaced with an abstract equivalent or needs to be surrounded by a guard that skips the call if the protocol in use is not IP.

This particular bit of code also carries a common bug: it tries to be slightly protocol-independent and ends up worse off for the effort. It uses the protocol family returned by `gethostbyname()` and copies addresses in a variable-length way, but copies that into a field within a `sockaddr_in` and later tries to `connect()` to that address using an `AF_INET` socket while specifying the length of the `sockaddr_in` as the length of the address information. If the family was something other than `AF_INET`, the `sockaddr_in` would probably not be filled in with something meaningful, and `connect()` call would probably fail regardless because the protocol family of the target address was not the same as that of the socket. As long as the only addresses that ever get returned by `gethostbyname()` are IP addresses, this practice will actually work. If addresses other than IP addresses were returned, programs written this way would break. This creates an interesting problem: interfaces that might be made protocol independent cannot be, because legacy programs don't use them correctly and changing what they return would break software. Using a new interface designed for protocol independence (like `getaddrinfo()`) and using it correctly will solve this problem.

A variation of this problem is hard coding addressing information, such as addresses and ports. Figure 2 shows an example from Sendmail 8.7.6[4]. There are three major problems here. First, the code always treats the address as a `sockaddr_in` without any guards. As in the example above, this is bad for protocol independence. Second, the code hard-codes an address and a port. While this is sometimes useful, it is usually bad practice and always bad practice when not combined with a test to check the protocol family. Third, the code explicitly specifies TCP as the transport protocol being used. This hard-codes a transport protocol and implies that only a small number of network protocols are usable (those that TCP has been made to run over). The second and third problems can be solved by using protocol-independent name resolution functions correctly.

2.2 Inflexible Storage

Another class of problems comes about when storing information needed by various protocols. This was already mentioned in the discussion of Figure 1, where not only does the use of `sockaddr_in` hard-code the address format of a particular protocol, but it also does not provide enough space to store the addresses of many protocols. The most common place where this problem comes into play is when used with `getpeername()`. Figure 3 shows an example of this from the 4.4BSD-Lite2's `fingerd` source; similar code sequences can be found in almost any server program. This code example also shows the assumption that the returned socket will be an IP socket; while originally a fair assumption, this needs to be fixed in order to be protocol-independent.

```
        temp = inet_addr(hostp);
        if (temp != (unsigned long) -1) {
            sin.sin_addr.s_addr = temp;
            sin.sin_family = AF_INET;
            (void) strcpy(_hostname, hostp);
            hostname = _hostname;
        } else {
            host = gethostbyname(hostp);
            if (host) {
                sin.sin_family = host->h_addrtype;
#if     defined(h_addr)         /* In 4.3, this is a #define */
                memmove((caddr_t)&sin.sin_addr,
                                host->h_addr_list[0], host->h_length);

...

        net = socket(AF_INET, SOCK_STREAM, 0);
        setuid(getuid());
        if (net < 0) {
            perror("telnet: socket");
            return 0;
        }
#if     defined(IP_OPTIONS) && defined(IPPROTO_IP)
        if (srp && setsockopt(net, IPPROTO_IP, IP_OPTIONS, (char *)srp, srlen) <
 0)
                perror("setsockopt (IP_OPTIONS)");
#endif

...

        if (connect(net, (struct sockaddr *)&sin, sizeof (sin)) < 0) {
```

Figure 1: Hard-Coding the Network Protocol (4.4BSD Telnet)

```
        if (DaemonAddr.sin.sin_family == 0)
                DaemonAddr.sin.sin_family = AF_INET;
        if (DaemonAddr.sin.sin_addr.s_addr == 0)
                DaemonAddr.sin.sin_addr.s_addr = INADDR_ANY;
        if (DaemonAddr.sin.sin_port == 0)
        {
                register struct servent *sp;

                sp = getservbyname("smtp", "tcp");
                if (sp == NULL)
                {
                        syserr("554 service \"smtp\" unknown");
                        DaemonAddr.sin.sin_port = htons(25);
                }
                else
                        DaemonAddr.sin.sin_port = sp->s_port;
        }
```

Figure 2: Hard Coding Addresses and Ports (Sendmail 8.7.6)

```
            struct sockaddr_in sin;
    ...
            if (logging) {
                    sval = sizeof(sin);
                    if (getpeername(0, (struct sockaddr *)&sin, &sval) < 0)
                            err("getpeername: %s", strerror(errno));
                    if (hp = gethostbyaddr((char *)&sin.sin_addr.s_addr,
                        sizeof(sin.sin_addr.s_addr), AF_INET))
                            lp = hp->h_name;
                    else
                            lp = inet_ntoa(sin.sin_addr);
                    syslog(LOG_NOTICE, "query from %s", lp);
            }
```

Figure 3: Use of a sockaddr_in to Store Arbitrary Addresses (4.4BSD fingerd)

A similar problem is also seen frequently in servers: the use of a generic `sockaddr` to store address information. Like the IP protocol specific structure, it is not big enough to hold addresses for many protocols (on most systems, the two structures are actually the same size). When the size of the address to be stored is known, a buffer of that size can be allocated. When it is not, a maximal-length buffer can be allocated using a `sockaddr_storage`, which will be discussed later.

A particularly bad special case of this problem comes about in some IP-only programs. Because IP addresses happen to be 32 bit unsigned integers and many modern systems have that as a native data type, some programs simply use integers to store IP addresses. Figure 4 shows an example from vat 4.0b2[5], which uses `u_int32_t`s internally to store network addresses (this is a bit less bad than using more generic integer types, but still hopelessly IP- dependent). Due to a particularly common example of this in earlier versions of BSD, this is sometimes referred to as the "all the world's a `u_long`" problem, and has a lot in common with the old "all the world's a VAX" problem. Optimizing assumptions are being made about the size and form of an address that happen to work on most currently interesting systems and protocols. But they're still poor assumptions that break portability, both in terms of supporting different systems and supporting different protocols. 4.4BSD-Lite2 has fixed this problem in many places by using `in_addr` instead, which is still protocol-dependent but at least is the correct type. In general, raw addresses should not be stored – socket addresses should be used instead.

Also, some protocols have variable-length addresses. Most existing programs treat addresses as fixed-length objects and do not store the real length as provided by run-time functions. Programs must store the length of addresses along with the addresses themselves – as with the address type, this can be necessary information for interpreting the address. This also means that the sizes of buffers used to hold addresses should not be arbitrarily bounded.

Using the generic `sockaddr` or the wrong protocol-specific structure also creates problems with alignment. Most network protocols have some alignment requirement for their protocol-specific address structures that may not be satisfied by other structures. Care must be taken to either use the correct protocol specific address structure or to arrange for the buffer used to store addresses to be properly aligned.

The generic `sockaddr` *should* be used as a structure to which an arbitrary socket address can be cast in order to access the `sa_family` and `sa_len` fields. While those fields should have the same type no matter what protocol specific structure is used to access the buffer, it is still good use of types to use the generic `sockaddr` for access where the network protocol in use are not yet known, rather than to using the wrong protocol-specific type.

Finally, many programs assume that a "port" is an integer. The concept of an integer port number is not universal. Some protocols use string service names instead, or use other formats that are at least convertable to a string. Service endpoints should be represented as strings that may or may not end up converted to another format for representation in a socket address.

2.3 Inflexible User Interface

Many programs get address information through some sort of user interface or user parameter syntax. For example, web clients get resource information through URLs, free network programs such as `tcp_wrappers` [6] read configuration files, and some GUI network programs use four three-digit entry blanks to enter a numeric address. In many cases, the user interface or syntax that these clients use is dependent on the syntax of particular kinds of addresses. The use of colons and slashes in URLs, for example, makes it difficult to use those characters in network addresses (colons are the delimiter for IPv6 addresses, so this is a real problem). Similarly, the configuration files for `tcp_wrappers` 7.6 uses colons as the field separator. The generic solution to this problem is to provide support for escaping and/or quoting so that somewhat arbitrary characters can be used in address information. In many cases, a quick-fix solution can be made by changing the fields or delimiters, but this is not backwards compatible and only fixes things until the next new address syntax comes along.

Another problem is that more information may need to be provided to a program in order for it to know how to correctly interpret an address. For example, a host name may be valid in multiple protocol families. If the user wants to use a specific protocol, then this needs to be somehow specified. Programs need to have enough flexibility in their user interfaces to add these sort of options.

2.4 Not Handling Multiple Addresses

Multi-homed IP support in programs is still not as common as it should be. The problem of multi-homing support – supporting multiple interfaces with one or more IP address – is similar to the problem of supporting multiple addresses for different network protocols. In both cases, some subset of the available addresses may need to be listened on (for a server) or be available for an outgoing connection. The selection of these addresses creates interesting problems. In particular, it sometimes (esp. in the case of servers) creates the need to have multiple network sockets open at once and to handle traffic on any of them. Most networking programs are written to use only one network socket, and changing that requires significant work.

Both multi-homing and multi-protocol support requires extra user interface capability. For example, both will cause multiple addresses to be returned from some name lookups, of which a subset might be reachable endpoints. The user should be either be given a choice among this set or some attempt should be made to progress in a reasonable way (for example, trying each in sequence until one succeeds). Whatever actually happens, the user should be made aware of what's happening.

2.5 Protocols Carrying Address Information

Some IP application protocols, such as FTP and talk, pass address information over the network. This means that the application protocol will need to be modified to be multi-protocol, which generally means adding flexibility similar to what has to be done inside a network program. Exactly how to do this is outside the scope of this document, but an example is the approach used for FTP[7]. In general, the best solution to this problem is to change the application protocol to not send address information over the network, because this practice causes problems for many network/protocol translation devices.

3 New Interfaces

The IETF's IPng working group and the POSIX p1003.1g working group have made a good bit of progress in identifying and standardizing new APIs needed to develop protocol-independent programs. Beyond this, the NRL IPv6 implementation found several other interfaces that we felt needed to be present in order to develop fully functional protocol-independent applications. In many cases, the BSD sockets interfaces were almost good enough but in practice misused. The new interfaces extend many BSD interfaces and supersede others.

The new interfaces break down into roughly two categories. The first are functions to perform name resolution operations (name to addresses, address to name) in a clean way. The second are operations to help use socket addresses (sockaddrs) in a clean way.

3.1 Name Resolution

Figure 5 shows a brief summary of the new name resolution interfaces.

The `getaddrinfo()` and `getnameinfo()` functions provide a protocol-independent way of mapping names to addresses information and of mapping address information back to names. Given a host name, a service name, and other information to constrain the lookup, `getaddrinfo()` returns either an integer error or a list of filled in `addrinfo` structures. Each contains the information that needs to be passed to `socket()` to open a socket as well as the information that needs to be passed to `connect()` or `sendmsg()` to reach the named endpoint.

Many programs can simply take the returned list and iterate through it, executing `socket()` and `connect()` calls with the information in each list element, until one attempt succeeds completely or the list has been exhausted. Figure 6 gives an example of how to do this. Notice how the program never needs to manipulate addresses directly. The program only needs to take information out of the `addrinfo` structure and feed it into other functions. This simple block of code is capable of obtaining a connected socket with any stream protocol that is supported in both the kernel and in the runtime library. If the runtime library

```
int Network::dorecv(u_char* buf, int len, u_int32_t& from, int fd)
{
        sockaddr_in sfrom;
        int fromlen = sizeof(sfrom);
        int cc = ::recvfrom(fd, (char*)buf, len, 0,
                            (sockaddr*)&sfrom, &fromlen);
        if (cc < 0) {
                if (errno != EWOULDBLOCK)
                        perror("recvfrom");
                return (-1);
        }
        from = sfrom.sin_addr.s_addr;
```

Figure 4: Using an Integer for an Address (from vat 4.0b2)

```
#define AI_PASSIVE     /* Socket address is intended for bind() */
#define AI_CANONNAME   /* Request for canonical name */
#define AI_NUMERICHOST /* Don't ever try nameservice */

struct addrinfo {
  int ai_flags;             /* input flags */
  int ai_family;            /* protocol family for socket */
  int ai_socktype;          /* socket type */
  int ai_protocol;          /* protocol for socket */
  int ai_addrlen;           /* length of socket-address */
  struct sockaddr *ai_addr; /* socket-address for socket */
  char *ai_canonname;       /* canonical name for service location (iff req) */
  struct addrinfo *ai_next; /* pointer to next in list */
};

int getaddrinfo(const char *name, const char *service,
                const struct addrinfo *req, struct addrinfo **pai);

void freeaddrinfo(struct addrinfo *ai);

char *gai_strerror(int ecode);

#define NI_MAXHOST          /* Maximum host name buffer length needed */
#define NI_MAXSERV          /* Max. service name buffer length needed */
#define NI_NUMERICHOST      /* Don't do name resolution for the host */
#define NI_NUMERICSERV      /* Don't do name resolution for the service */
#define NI_NOFQDN           /* Don't fully qualify host names */
#define NI_NAMEREQD         /* Fail if name resolution for the host fails */
#define NI_DGRAM            /* Service is for a DGRAM socket (not a STREAM) */

int getnameinfo(const struct sockaddr *sa, size_t addrlen, char *host,
                size_t hostlen, char *serv, size_t servlen, int flags);

int nrl_afnametonum(const char *name);       /* (Nonstandard) */
const char *nrl_afnumtoname(int num);        /* (Nonstandard) */
int nrl_socktypenametonum(const char *name); /* (Nonstandard) */
const char *nrl_socktypenumtoname(int num);  /* (Nonstandard) */
```

Figure 5: Summary of New Name Resolution APIs

does not support a protocol, it will not be returned by `getaddrinfo()`. If the kernel does not support a protocol, this function will print an error for those sockets and skip that protocol. This is especially important for binaries to be shipped on systems where the protocols available in the runtime library and/or kernel can be configured by the end user; one binary will be able to work as long as the system is configured so that there is one protocol that the entire system supports.

Note that `getaddrinfo()` and `getnameinfo()` handle both host names and printable numeric addresses, as appropriate. One historical problem with functions like `gethostbyname()` and `gethostbyaddr()` is that on some systems they handle printable numeric addresses and on some systems they do not. Portable programs must be written to attempt printable-numeric conversion separately, just in case – programs that assume the system handles these have encountered portability problems. Some programs have bugs caused by the old printable numeric conversion functions, making this even more of a problem. These new functions should hopefully put these problem to rest.

As shown in the example, the `gai_strerror()` function converts the errors returned by `getaddrinfo()` and `getnameinfo()` into human-printable form. There are also constants for the error values, but few programs need to distinguish between the types of failures beyond giving an appropriate error message. The `freeaddrinfo()` function releases the memory used by the result list, and *must* be called when the result is no longer needed.

The functions `nrl_afnametonum()` and `nrl_afnumtoname()` convert address family names (inet, inet6, local, etc.) to numbers and back. This is needed in order to support user entry of an address family to constrain `getaddrinfo()` lookups. For example, many NRL IPv6-enabled applications support a command line flag that the user can use to specify a family, such as "inet" or "inet6," that selects what protocol to use. The number-to-name function is also helpful for diagnostic output. Similarly, the functions `nrl_socktypenametonum()` and `nrl_socktypenumtoname` convert socket type names (stream, dgram, seqpacket, etc.) to numbers and back. These are less useful for user input, but are still useful for diagnostic purposes.

3.2 Socket Addresses

Figure 7 shows a brief summary of the new socket address interfaces. The major new addition is the `sockaddr_storage`, which is defined as a structure that is big enough to hold any socket address that the system supports or might support in the future, and provides sufficient alignment for any socket address that the system supports or might support in the future. In practice, the size of the structure is bounded on many systems by the capacity of the eight bit integer used in the `sa_len` field of all socket addresses. On other systems, the bound might be provided by other structures' fields. The bound actually chosen can be selected by the systems' authors, but the `sockaddr_storage` is defined to have whatever size is needed. The alignment provided by the `sockaddr_storage` will typically be the largest alignment available on the system, though again the exact choice is up to the systems' implementors.

Note that the `sockaddr_storage` is required to have fields of the same type and in the same place as the `sa_len` and `sa_family` fields in the systems' `sockaddr`s, but that the standards that specify this data structure don not actually require those fields to have a known name and give examples with names that makes them "hidden." While it is hoped that the long term solution will be to fix this problem in the standards, the short term most portable way to use these fields is to cast a `sockaddr_storage` to a `sockaddr` and to use the fields through the latter type.

The `sockaddr_storage` is used where a socket address needs to be stored before its exact length is known. Figure 8 shows some of the example in Figure 3, changed to take advantage of this structure as well as `getnameinfo()`. The code is not very different, but the use of the `sockaddr_storage` guarantees that any protocol-specific socket address can be safely stored in the buffer.

A controversial API extension that was used heavily in the NRL code is the `SA_LEN()` macro. On systems whose `sockaddr` has a `sa_len` field, this expands to return the contents of that field and has the same semantics except that it is only defined to be an rvalue. On systems whose `sockaddr` does not, this expands into an operation that returns the correct value based on the value of the `sa_family` field. This macro solves the problem of needing a sockaddr's length for many function calls well after existing code has lost the length information. It is frequently far easier to replace a hard coded value such as `sizeof(struct sockaddr_in)` with a macro use like `SA_LEN(sa)` than it is to gut an entire program and fix this. Using the macro, this technique is portable to systems with and without `sa_len` support. Authors who have used this technique extensively have been quite supportive of it, while authors of systems that don't have `sa_len` fields have been opposed to it.

3.3 State of Deployment

New API functions are fine as long as they are available everywhere, but deployment does not happen quickly. The interfaces described as non-standard are just that, and are unlikely to be present any-

```
int get_stream(char *host, *service)
{
  int error, fd;
  struct addrinfo req, *ai, *ai2;
  char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

  memset(&req, 0, sizeof(struct addrinfo));
  req.ai_socktype = SOCK_STREAM;

  if (error = getaddrinfo(host, service, NULL, &ai)) {
    fprintf(stderr, "getaddrinfo(%s, %s, ...): %s(%d)", gai_strerror(error),
      error);
    return -1;
  }

  for (ai2 = ai; ai = ai->ai_next) {
    if (error = getnameinfo(ai->ai_addr, ai->ai_addrlen, hbuf, sizeof(hbuf),
        sbuf, sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV)) {
      fprintf(stderr, "getnameinfo(%p, %d, %p, %d, %p, %d, %d): %s(%d)\n",
        ai->ai_addr, ai->ai_addrlen, hbuf, sizeof(hbuf), sbuf, sizeof(sbuf),
        NI_NUMERICHOST | NI_NUMERICSERV, gai_strerror(error), error);
      continue;
    }

    fprintf(stdout, "Trying %s.%s...\n", hbuf, sbuf);

    if ((fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol)) < 0) {
      fprintf(stderr, "socket(%d, %d, %d): %s(%d)\n", ai->ai_family,
        ai->ai_socktype, ai->ai_protocol, strerror(errno), errno);
      continue;
    }

    if (connect(fd, ai->ai_addr, ai->ai_addrlen) < 0) {
      fprintf(stderr, "connect(%d, %p, %d): %s(%d)\n", fd, ai->ai_addr,
        ai->ai_addrlen, strerror(errno), errno);
      close(fd);
      continue;
    }

    freeaddrinfo(ai2);
    return fd;
  }

  freeaddrinfo(ai2);
  fprintf(stderr, "No connections result.\n");
  return -1;
}
```

Figure 6: Using `getaddrinfo()` to Get one Stream Connection

```
struct sockaddr_storage { /* Slightly nonstandard - See text for a warning */
    u_int8_t    ss_len;        /* address length */
    sa_family_t ss_family;     /* address family */
    /* other fields guarantee size and padding */
};

#define SA_LEN(sa) ((sa)->sa_len) /* Nonstandard */
```

Figure 7: New Socket Address Interfaces

```
        struct sockaddr_storage ss;
...
        if (logging) {
                sval = sizeof(sockaddr_storage);
                if (getpeername(0, (struct sockaddr *)&ss, &sval) < 0)
                        err("getpeername: %s", strerror(errno));
```

Figure 8: Use of a sockaddr_storage to Store Arbitrary Addresses

where but systems with the NRL IPv6 code or with the (NRL-derived) Linux inet6-apps kit. The standard new interfaces are supposed to be present now in AIX, BSD/OS, FreeBSD, Linux (with GNU libc 2.1), OpenBSD, NetBSD, Solaris, and Tru64 UNIX. In addition, IRIX and HP-UX will probably adopt these functions very quickly (if they haven't already). In summary, all modern UNIX systems are expected to support the standardized interfaces now or very soon.

But what about legacy systems? The good news is that there are fairly portable implementations of these functions that run on legacy systems and can be included with programs. Several free implementations of the new interfaces exist, and are reasonably portable. There are already some free software packages, such as Zmailer, that have used this approach.

4 Conclusions

The existing BSD sockets API provides most of what is needed to write protocol- independent applications, but there are some things that needed to be added. More important is that the mechanisms for being abstract and generic be used correctly. It is not difficult to do, and the effort pays for itself in flexibility and future-proofing. It is hoped that authors of network programs will take the time to learn about protocol independence and work to use the technique in their programs.

5 Acknowledgements

The NRL IPv6 code, and our exploration of protocol independence, was a team effort. Other than myself, the implementation team includes or has included Randall Atkinson, Ken Chin, Daniel McDonald, Ronald Lee, Bao Phan, Chris Telfer, and Chris Winters.

When I first pushed people to look at protocol independence issues in IPv6 API discussions, W. Richard Stevens, Jun-Ichiro "itojun" Hagino, and the KAME project team provided strong support for the idea, and their support was critical to the development of these APIs.

David Greenman, Kevin Skadron, and Chris Telfer reviewed early drafts of this paper and provided valuable feedback.

References

[1] R. Gilligan, S. Thomson, J. Bound, and W. Stevens. Basic Socket Interface Extensions for IPv6, RFC 2553, March 1999.

[2] Randall J. Atkinson, Ken E. Chin, Bao G. Phan, Daniel L. McDonald, and Craig Metz. Implementation of IPv6 in 4.4BSD. *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[3] Regents of the University of California. 4.4bsd-lite2 software distribution, 1995.

[4] Eric Allman et al. Sendmail, version 8.7.6, 1999.

[5] Van Jacobsen et al. Vat, version 4.0b2, 1996.

[6] Vietse Venema et al. tcp_wrappers, version 7.6, 1997.

[7] M. Allman, S. Ostermann, and C. Metz. FTP Extensions for IPv6 and NATs, rfc 2428, September 1998.