PORTABILITY ISSUES FOR IPv4 to IPv6 APPLICATIONS

K.Ettikan and Tan Wei Chong,
Faculty of Information Technology,
Multimedia University,
Jalan Multimedia, 63100 Cyberjaya,
Selangor, Malaysia.

Abstract

Since the introduction of IPv6 as the new potential standard for the Internet compatibility, scalability and portability of existing applications from IPv4 have been a major issue for Application Developers and Application Service Providers. Various steps have been taken in order to reuse existing applications and infrastructures to reduce reconstruction cost and to save investment. Internet Engineering Task Force has produced various documents to guide the developers on the migration issues. However, not much information is available to share practical experience [1,2,3,4,5] in porting existing IPv4 application to IPv6 which crucial. This paper is aimed to discuss about various issues when porting an IPv4 application to IPv6 with focus on issues that an application developer would face rather than a complete API reference. The porting of codes has been grouped into four main categories based on complexity of the work involved.

1.0 Introduction

IPv6 as the successor to the two decades old IPv4 [7] has been designed to be a evolutionary protocol which will solve the address shortage problem besides adding new functionality. The deployment of this new protocol brings set of new challenges to the Application Developers and Application Service Providers. Besides having new reliable and stable network to carry the IPv6 traffic [9] all the application need to ported to be able to be used in the new environment [10]. Currently many vendors, manufacturers, and Internet Service Provides are working hard to provide IPv6 connectivity and support for the users. Meanwhile IPv6 aware operating systems are being developed to enable applications to run on these platforms, such as Sun Microsystems's Solaris 8, Microsoft's msripv6 [11] and FreeBSD's Kame [8].

Although porting an application from IPv4 to IPv6 is not a difficult task there are many hidden problems that must be considered and carefully analyzed. Most of the third party applications are not aware of the change of IP versions and may not be well designed for easy porting. These old applications need to be carefully analyzed and ported to avoid any functionality and interoperability problems on the new OS. Besides that applications need to be designed to run on IPv4 and IPv6 during the transition period. This needs great flexibility on the application itself to be aware of IP versions and adapt accordingly for reliable communication.

This paper is aimed to discuss about various issues when porting an IPv4 application to IPv6 with focus on issues that an application developer would face rather than a complete API reference. The porting of codes has been grouped into four main categories based on complexity of the work involved. The first category refers to the most protocol independent codes where the codes can be ported easily while second category refers to codes that can be ported by introducing new API to replace the old ones. The third category group's codes that need modification for some of the system calls and finally forth category covers codes that can only be ported if there is a modification of the program logic.

Through our experience porting of an application from IPv4 to IPv6 may fall in any of these categories or combination of the categories based on the previous code implementation. Experience of porting IPv4 Trivial File Transfer Protocol (tftp) application, which comes with the standard FreeBSD distribution kit to IPv6, will be used as the base for discussion in this paper. The following four sections in this paper will analyze the codes that need to be ported. Finally we conclude our paper with our experience in porting the client TFTP [12] application.

2.0 Category 1 : Protocol Independent Codes

In most applications, there are some portion of codes that deals mainly with logic and algorithmic processing without calling any particular system calls or API to carry out their job. These codes usually can be written using standard language such as C/C++ language. An example would be the codes in Figure 1 that has been extracted from main.c of the client TFTP program.

```
struct	modes {
	char *m_name;
	char *m_mode;
} modes[] = {
	{ "ascii", "netascii" },
	{ "netascii",   "netascii" },
	{ "binary",     "octet" },
	{ "image",      "octet" },
	{ "octet",     "octet" },
/*      { "mail",       "mail" },       */
	{ 0,                    0 }
};

void
modecmd(argc, argv)
	int argc;
	char *argv[];
{
	register struct modes *p;
	char *sep;

	if (argc < 2) {
		printf("Using %s mode to transfer files.\n", mode);
		return;
	}
	if (argc == 2) {
		for (p = modes; p->m_name; p++)
			if (strcmp(argv[1], p->m_name) == 0)
				break;
		if (p->m_name) {
			settftpmode(p->m_mode);
			return;
		}
		printf("%s: unknown mode\n", argv[1]);
		/* drop through and print usage message */
	}

	printf("usage: %s [", argv[0]);
	sep = " ";
	for (p = modes; p->m_name; p++) {
		printf("%s%s", sep, p->m_name);
		if (*sep == ' ')
			sep = " | ";
	}
	printf(" ]\n");
	return;
}
static void
settftpmode(newmode)
	char *newmode;
{
	strcpy(mode, newmode);
	if (verbose)
		printf("mode set to %s\n", mode);
}
```

Figure 1. Codes from client TFTP main.c of FreeBSD program.

The code snippet in modecmd( ) in Figure 1 involve no major calls then the standard C language functions such as printf( ), strcmp( ) and the programmer defined setftpmodes( ) function. The rest of the program constructs use only standard C language construct. These types of code need not undergo any major modification while porting into IPv6 unless some strategy of processing has changed. These changes could due to either cater for new demand on new application or to improve the performance of such application in IPv6. Under most normal circumstances, it is recommended that the programmer just do the usual "cut and paste" practice to port the code over to the new application. This gives two major advantages that is to save time and also to guarantee the highest degree of compatibility since the original code has been in used and debug for some time.

3.0 Category 2: Changes for New API

In this category, the code can be ported by just rigidly substituting some of the API and data structures that IPv4 application use to establish and carry out the communication. Such kind of porting is most interesting in its rigid nature of API because its probability of automatically ported is higher than the rest. Let's have a look into some code examples to study some attribute. Below is a table consist of API and data structures from main.c:

IPv4 system calls and data structures	*Possible IPv6 system calls and data structures*
struct sockaddr_in peeraddr;	struct sockaddr_in6 peeraddr;
peeraddr.sin_family = AF_INET;	peeraddr.sin6_family = AF_INET6;
hp = gethostbyname(cp);	hp = getipnodebyname(cp, AF_INET6, AI_DEFAULT, NULL); [1]
peeraddr.sin_port = port;	peeraddr.sin6_port = port;

Table 1: API and Data Structures from main.c of TFTP program.

Among others, *struct sockaddr_in* is the most frequently encountered data structure in IPv4 network application. This structure is known as IPv4 socket address structure or commonly also known as Internet socket address structure. In BSD implementation, this structure is defined in the <*netinet/in.h*> header file. There are multiple ways to deal with this structure. One possible alternative is to change *sockaddr_in* into *sockaddr_in6* structure and leave the rest of the code alone with minimal modification. This is because socket address structures are always passed by referenced to any socket functions. (The socket functions are designed to accept argument by reference of type generic *sockaddr* structure to enable handling of various protocol families. This is also the reason why RFC2553 [1,2,3] identify the core socket functions as portable and do not need any changes of system calls, just modification on the system calls' macros). There are other alternatives besides *sockaddr_in6*, for example using the *sockaddr_storage* structure to store all the information and utilize the casting into generic *sockaddr* structure.

Besides the system calls and data structures, macros are another issue. For example, to port to IPv6, the macro AF_INET which identify the IPv4 address family can be changed to either AF_INET6 which is the IPv6 specific address family or AF_UNSPEC which is the unspecified address family. Changing to AF_INET6 usually make the porting easier but AF_UNSPEC provides the flexibility of being able to handle multiple address family including both IPv4, IPv6 and others.

One popular API that is frequently called in IPv4 is *gethostbyname( ).* Since this function does not allow address family to be explicitly specified, it is not suitable to be used in IPv6 application. In the original RFC2553, *getipnodebyname( )* was introduced to handle the same job but it allows for address family specification. However, in IETF draft [2,3] document, all references to *getipnodebyname()* was removed and programmers are advised to use *getaddrinfo( )* to obtain the same information.

4.0 Category 3: System Calls Modification

The third group of code involves modification of network calls, which then affect other portion of code such as network independent system calls. These codes usually interact with other system calls or rely on other system data structures to function. As far as the experience of the author is concerned, the group of code in such group is not much encountered in the context of TFTP implementation. However, there are still some small portion exist. Figure 2 shows the small portion of code from main.c

```
struct hostent *host;
host = gethostbyname(argv[1]);
if (host) {
                peeraddr.sin_family = host->h_addrtype;
                bcopy(host->h_addr, &peeraddr.sin_addr,
                        MIN(sizeof(peeraddr.sin_addr), host->h_length));
                strncpy(hostname, host->h_name, sizeof(hostname));
        }
```

Figure 2. Codes from main.c of FreeBSD TFTP program shows codes that need system call modification

From the above codes, one can see that it involves the *hostent* structure which store the result from *gethostbyname()*. Later on, information is copied to another member in a *sockaddr_in* structure using *bcopy()* and *strncopy()*. Although here it is not a major dependency as the related function just act as a copier. However, the programmer still needs to be careful whenever another non-network related system call is involved.

5.0 Category 4: Modification of the Program Logic

Finally, there are certain portion of code that not only affect modification of function calls but also the logic behind the their usage. Such code is the hardest to deal with and cannot be automatically ported most of the time. Evidence for such type of code can be found in the beginning portion of main.c:

```
struct sockaddr_in sin;

        sp = getservbyname("tftp", "udp");
        if (sp == 0)
                errx(1, "udp/tftp: unknown service");
        f = socket(AF_INET, SOCK_DGRAM, 0);
        if (f < 0)
                err(3, "socket");
        bzero((char *)&sin, sizeof(sin));
        sin.sin_family = AF_INET;
        if (bind(f, (struct sockaddr *)&sin, sizeof(sin)) < 0)
                err(1, "bind");
```

Figure 3. Codes from main.c of FreeBSD TFTP program shows codes that need program logic modification

These codes can be misleading if the programmer does not consider the logic of the program. For instance, the *bind()* system call is usually being used by the connect-oriented or TCP server application[6] and if such assumption holds true, a usual approach to port such code would be as below.

```
struct addrinfo hints, *res, *res0;
        int s, i;

        memset(&hints, 0, sizeof(hints));
        hints.ai_family = PF_UNSPEC;
        hints.ai_socktype = SOCK_DGRAM;
        hints.ai_flags = AI_PASSIVE;
        error = getaddrinfo(NULL, "tftp", &hints, &res0);
        if(error){
                fprintf(stderr, "%s", gai_strerror(error));
                exit(1);
        }
                i = 0;
        for(res = res0; res; res = res->ai_next){
                s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
                if(s < 0) continue;
                if(bind(s, res->ai_addr, res->ai_addrlen) < 0){
                        close(s);
                        continue;
                }
                listen(s, 5);
                socktable[i] = s;
                sockfamily[i++] = res->ai_family;
        }
        freeaddrinfo(res0);
        if(i == 0) errx(1, "no bind() successful");
```

Figure 4. Codes from main.c of FreeBSD TFTP program shows misleading that can happen due to code misinterpret.

However, since tftp is a UDP client program, such assumption is already wrong in the first place and a more suitable suggestion would be:

```
int af = AF_INET6;
int i, error;
struct sockaddr_in6 bla6;
bla6.sin6_family = af;
bla6.sin6_flowinfo = 0;
bla6.sin6_port = 0;
bla6.sin6_addr = in6addr_any;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
sp = getservbyname("tftp", "udp");
f = socket(AF_INET6,SOCK_DGRAM ,0);
if(f < 0) err(3, "socket");
if(bind(f, (struct sockaddr*) &bla6, sizeof(bla6)) < 0) err(1, "bind");
```

Figure 5. Codes from main.c of FreeBSD TFTP program shows codes that need program logic modification

Let's analyze what is the difference between the two porting approaches. Because there is a call to *getservbyname()* in the IPv4 implementation, this can be misleading. However, apparently the call has nothing to do with the *bind()* and *socket()* calls. The *getservbyname()* call just return the port number that is stored in the usual /etc/services which in effect identifies the port number used by the target server in the normal standard. The *socket()* call in the IPv4 implementation is intended to create a socket descriptor to be bind with an ephemeral port (this can be known by the *bzero()* call to clear the *sockaddr_in sin* structure before *bind()* was called). However, in the first attempt of porting, the wrong assumption of associating the *bind()* call with the usual convention of server code makes the application bind() its sockets with the standard tftp port instead of an ephemeral port. The wrong assumption of server code worsens the issue by creating an array of sockets instead of one single socket descriptor. Also the call to *listen()* would be unnecessary in the case of a client either TCP or UDP.

The second attempt solve all the problem by studying the meaning and logic of the code and clearly identify that the code is intended for a client application, in addition to factoring out the *getservbyname()* call from all the unrelated code portion. In this attempt, the code is nearer to the IPv4 implementation and the socket descriptor is *bind()* with an ephemeral port associated with a sockaddr_in6 structure. Also, since the programmer knows that it is intended for a client software, he can opt for the easier option of supporting just AF_INET6 instead of PF_UNSPEC.

6.0 Conclusion

From the discussion above it is clear that the ability for a particular portion of code to be ported to IPv6 depends on its own context and not so much defined rigidly. However, it can be safely said that for category 1 and 2 of codes, the porting is quite direct and possibly can be automated but not 3 and 4. It is our aim to write an automated code porting application that will port codes in IPv4 to IPv6 considering the codes in categories 1 and 2, while as in categories 3 and 4 the application will indicate to the programmer to analyze further to manually port the codes.

References

[1] RFC2553 - Basic Socket Interface Extensions for IPv6
[2] draft-ietf-ipngwg-rfc2553bis-00
[3] draft-ietf-ipngwg-rfc2553bis-01
[4] RFC2292 - Advanced Sockets API for IPv6
[5] draft-ietf-ipngwg-rfc2292bis-02
[6] Richard Stevens, Unix Network Programming – Volume 1
[7] Richard Stevens, Gary Wright, TCP/IP Illustrated – Volume 2
[8] www.kame.net
[9] Ettikan Kandasamy Karuppiah, Gopi Kurup and Takefumi Yamazaki - "Application Performance Analysis in Transition Mechanism from IPv4 to IPv6"
[10] K. Ettikan and V. Ganapathy – "IPv6 Performance Analysis on FreeBSD Workstations Using Simple Applications"
[11] Microsoft Windows 2000 Server – Introduction to IPv6 White Paper
[12] TFTP Application on FreeBSD (www.freebsd.org)