Some changes are needed to adapt the socket API for IPv6 support: a new socket address structure to carry IPv6 addresses, new address conversion functions and several new socket options that are developed in RFC-3493.

These extensions are designed to provide access to the basic IPv6 features required by TCP and UDP applications, including multicasting, while introducing a minimum of changes into the system and providing complete compatibility for existing IPv4 applications. Access to more advanced features (raw sockets, header configuration, etc.) is addressed in RFC-3542.

This document includes some examples of code porting, used to illustrate the required changes in the client and server components. Porting guidelines are valid for any programming language, however for simplicity, application porting examples are provided only in C language. All these examples have been tested in a SuSE Linux 7.3 distribution, kernel version 2.4.10.

1. Introduction

Today the Internet is predominantly based on IPv4. As a result, most of the end systems and network devices are able to exchange packets all together in a global network. However, there is no a single global IPv6 network on the same scale and it will take some time to get it. Therefore, new applications should be designed to work in all environments: IPv4-only nodes, IPv6-only nodes and dual-stack nodes, see Figure 0.
Dual-stack nodes are one of the ways to maintain IPv4 compatibility. They provide IPv4 and IPv6 stack implementations, allowing the sending and receiving of IPv4 and IPv6 packets. Most dual-stack implementations work following two operation modes:

1. IPv4 packets going to IPv4 applications on a dual-stack node use IPv4 stack. IPv6 packets going to IPv6 applications that are running on a dual-stack node use IPv6 stack. In summary: IPv4 applications run as if they were on an IPv4-only node while IPv6 applications run as if they were on an IPv6-only node.

2. IPv4 packets going to IPv6 applications on a dual-stack node, reach their destination because their addresses are mapped to IPv6 ones (using IPv4-mapped IPv6 addresses) inside dual-stack node, IPv4-mapped IPv6 addresses are IPv6 addresses built from IPv4 ones using the following format: from x.y.z.w IPv4 address the ::FFFF.x.y.z.w IPv4-mapped IPv6 address is built.

In the following, the connections between IPv6 server and IPv6 client applications when they are running on a dual stack node will be analyzed.

If a server application is running bound to the IPv6 wildcard address and a known port on a dual stack node, it will be able to accept connections from IPv4 and IPv6 clients, see Figure 1. When an IPv4 client is connecting to this IPv6 server, the dual stack kernel converts the client IPv4 address to the IPv4-mapped IPv6 address since the IPv6 server can only deal with IPv6 connections. The communication between the IPv4 client and the IPv6 server will take place using IPv4 datagrams, but the server will not know that it is communicating with an IPv4 client, unless the server explicitly checks it.
When running an IPv6 client application on a dual stack node, the client can connect to an IPv6 or IPv4 server, see Figure 2. Since the client node runs dual stack, client will try to open an IPv6 connection to the server. If the server is running over IPv4, the resolved server address returned by the resolver system is the IPv4-mapped IPv6 one. Then, the communication between IPv4 server and IPv6 client, which is running on the dual stack, is carried out using IPv4 but the IPv6 client will not know it, unless the client explicitly checks it.
2. Porting source code considerations

When porting to IPv6, most of changes will be made in the application transport module, which is in charged of establishing communications to remote nodes. First of all it is recommended to separate the transport module from the rest of application functional modules. This separation makes the application independent on the network system used. Then, if the network protocol is changed, only the transport module should be modified. The transport module should provide the communication channel abstraction with basic channel operations and generic data structures to represent the addresses. These abstractions could be instantiated as different implementations depending on the network protocol required at any moment. The application should deal with this generic communication channel interface without knowing the network protocol used. Using this design, if a new network protocol is added, application developers only need to implement a new instance of the channel abstraction which manages the features of this new protocol.

Once the transport module has been designed, there are some implementation details related to the type of the nodes which will run the application: IPv4-only nodes, IPv6-only nodes or both, dual stack.

Within the transport module the use of the new API with extensions for IPv6 is mandatory, but it is strongly recommended to make the program protocol independent (for instance, using the BSD socket API consider getaddrinfo and
getnameinfo instead of gethostbyname and gethostbyaddr). The new IPv6 functions are only valid if this protocol is supported by all installed systems. IPv6 is now reaching maturity and most popular systems provide it as default in their standard distributions. However, IPv6 support does not force to use it.

Besides the application transport module, there are other IP dependencies within the source code:

- **Presentation format for IP addresses**
  IPv4 and IPv6 do not use the same presentation format:
  - IPv4: uses dot as separator a.b.c.d
  - IPv6: uses colon as separator x:x:x:x:x:x:x
  When using Uniform Resource Locators (URLs), IP addresses should be enclosed in brackets, see RFC-2732.
  The recommendation is using FQDN instead of presentation format of IP addresses.

- **IP address selection**
  Choosing IP source/destination addresses pair following some preferential rules, see RFC-3484.

- **Application Framing**
  Some applications prefer implement its own fragmentation mechanisms instead of using IP layer fragmentation in order to obtain better performance. The application fragment size is directly related to the PMTU and applications can implement their own PMTU-D in order to find out its value.

- **Storage of IP addresses**
  If source code stores IP addresses, notice the following considerations:
  - IP addresses can change throughout the time (after a renumbering process).
  - The same node can reach a destination using different source IP addresses, all of them configured in the source network interface.

Dual stack mechanisms do not, by themselves, solve the IPv4 and IPv6 interworking problems. Protocol translation is required many times, for instance when an IPv6 application want to communicate to an IPv4 application exchanging IPv6 packets. Translation refers to the direct translation of protocols, including headers and sometimes protocol payload. Protocol translation often results in features loss. For instance, translation of IPv6 header into an IPv4 header will lead to the loss of the IPv6 flow label. In other cases, tunneling can be used, which is used to bridge compatible networks across incompatible ones.

### 3. Socket Address Structures

Functions provided by socket API use the socket address structures to determine the communication service access point. Since different protocols can handle socket functions, a generic socket address structure is used as argument of these functions for any of the supported communication protocol families, sockaddr.

```c
struct sockaddr {
    sa_family_t sa_family;       /* Address family */
    char sa_data[14];            /* protocol-specific address */
```
Although socket functions handle the generic socket address structures, developers must fill the adequate socket address structure according to the communication protocol they are using to establish the socket. In particular, the IPv4 sockets use the following structure, sockaddr_in:

```c
typedef uint32_t in_addr_t;
struct in_addr {
    in_addr_t s_addr;                    /* IPv4 address */
};

struct sockaddr_in {
    sa_family_t sin_family;             /* AF_INET */
    in_port_t sin_port;                 /* Port number. */
    struct in_addr sin_addr;            /* Internet address. */
    /* Pad to size of `struct sockaddr'. */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           sizeof (sa_family_t) -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
};
```

And the IPv6 sockets use the following structure, sockaddr_in6, with a new address family AF_INET6:

```c
struct in6_addr {
    union {
        uint8_t u6_addr8[16];
        uint16_t u6_addr16[8];
        uint32_t u6_addr32[4];
    } in6_u;

    #define s6_addr                 in6_u.u6_addr8
    #define s6_addr16               in6_u.u6_addr16
    #define s6_addr32               in6_u.u6_addr32
};

struct sockaddr_in6 {
    sa_family_t sin6_family;    /* AF_INET6 */
    in_port_t sin6_port;        /* Transport layer port # */
    uint32_t sin6_flowinfo;     /* IPv6 flow information */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t sin6_scope_id;     /* IPv6 scope-id */
};
```

The sockaddr_in or sockaddr_in6 structures are utilized when using respectively IPv4 or IPv6. Existing applications are written assuming IPv4, using the sockaddr_in structure. They can be easily ported changing this structure by sockaddr_in6. However, when writing portable code, it is preferable to eliminate protocol version dependencies from source code. There is a new data structure, sockaddr_storage, large enough to store all supported protocol-specific address structures and adequately aligned to be cast to the a specific address structure.

```c
/* Structure large enough to hold any socket address (with the historical exception of
```
AF_UNIX). 128 bytes reserved. */

#if ULONG_MAX > 0xffffffff
#define __ss_aligntype __uint64_t
#else
#define __ss_aligntype __uint32_t
#endif
#define _SS_SIZE 128
#define _SS_PADSIZE (_SS_SIZE - (2 * sizeof (__ss_aligntype)))

struct sockaddr_storage
{
    sa_family_t ss_family;      /* Address family */
    __ss_aligntype __ss_align;  /* Force desired alignment. */
    char __ss_padding[_SS_PADSIZE];
};

Hence, portable applications should use the sockaddr_storage structure to store their addresses, IPv4 or IPv6 ones. This new structure hides the specific socket address structure that the application is using.

4. Socket functions

The socket API has not been changed since it handles the generic address structure, independent from the protocol it is using. However, applications should change the arguments used to call the socket functions. First, applications should allocate enough memory to store the appropriate socket address structure. And, before calling the socket functions, the specific socket address structure should be cast to the generic one, which is accepted by the socket functions as an argument.

    int    socket (int domain, int type, int protocol);
    int    listen (int s, int backlog);
    ssize_t write (int fd, const void *buf, size_t count);
    int    send (int s, const void *msg, size_t len, int flags);
    int    sendmsg (int s, const struct msghdr *msg, int flags);
    ssize_t read (int fd, void *buf, size_t count);
    int    recv (int s, void *buf, size_t len, int flags);
    int    recvmsg (int s, struct msghdr *msg, int flags);
    int    close (int fd);
    int    shutdown(int s, int how);

THE SOCKET ADDRESS STRUCTURE IS PASSED FROM APPLICATION TO KERNEL

    int bind   (int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
    int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
int sendto (int s, const void *msg, size_t len, int flags,
    const struct sockaddr *to, socklen_t tolen);

THE SOCKET ADDRESS STRUCTURE IS PASSED FROM KERNEL TO APPLICATION

int accept     (int s, struct sockaddr *addr, socklen_t *addrlen);
int recvfrom   (int s, void *buf, size_t len, int flags,
    struct sockaddr *from, socklen_t *fromlen);
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
int getsockname(int s, struct sockaddr *name, socklen_t *namelen);

Required modifications when porting to IPv6

When porting to IPv6, some modifications related to the socket API are required in
the network applications. Three modification types have been identified:

● Creating a socket.
● Passing the socket address structure from application to kernel.
● Pass the socket address structure from kernel to application.

These are the three kind of operations to be changed in the source code. There are some examples below of these types of operation.

● Creating a socket
The difference between creating an IPv4 and an IPv6 socket is the value of the family argument in the socket call.
IPv4 source code:

    socket(PF_INET, SOCK_STREAM, 0);  /* TCP socket */
    socket(PF_INET, SOCK_DGRAM, 0);   /* UDP socket */

IPv6 source code:

    socket(PF_INET6, SOCK_STREAM, 0); /* TCP socket */
    socket(PF_INET6, SOCK_DGRAM, 0);  /* UDP socket */

● Passing the socket address structure from application to kernel
Socket address structure is filled before calling the socket function.
IPv4 source code:

    struct sockaddr_in addr;
    socklen_t          addrlen = sizeof(addr);
    /*
     * fill addr structure using an IPv4 address before calling socket
     * function
     *
    */
    bind(sockfd,(struct sockaddr *)&addr, addrlen);

IPv6 source code:

    struct sockaddr_in6 addr;
    socklen_t          addrlen = sizeof(addr);

/*
 * fill addr structure using an IPv6 address before calling socket
 * function
 */
bind(sockfd, (struct sockaddr *)&addr, addrlen);

Portable source code:

struct sockaddr_storage addr;
socklen_t       addrlen;

/*
 * fill addr structure using an IPv4/IPv6 address and
 * fill addrlen before calling socket function
 */
bind(sockfd, (struct sockaddr *)&addr, &addrlen);

● **Passing the socket address structure from kernel to application**

When calling this kind of socket functions, the socket address structure is
filled in with the address of the source entity.

**IPv4 source code:**

```
struct sockaddr_in addr;
socklen_t       addrlen = sizeof(addr);
accept(sockfd, (struct sockaddr *)&addr, &addrlen);

/*
 * addr structure contains an IPv4 address
 */
```

**IPv6 source code:**

```
struct sockaddr_in6 addr;
socklen_t       addrlen = sizeof(addr);
accept(sockfd, (struct sockaddr *)&addr, &addrlen);

/*
 * addr structure contains an IPv4 address
 */
```

**Portable source code:**

```
struct sockaddr_storage addr;
socklen_t       addrlen = sizeof(addr);
accept(sockfd, (struct sockaddr *)&addr, &addrlen);

/*
 * addr structure contains an IPv4/IPv6 address
 * addrlen contains the size of the addr structure returned
 */
```
5. Address conversion functions

The address conversion functions convert between binary and text address representation. Binary representation is the network byte ordered binary value which is stored in the socket address structure and the text representation, named presentation, is an ASCII string.

The IPv4 address conversion functions are the following ones:

```c
/*
 * From text to IPv4 binary representation
 */
int inet_aton (const char *cp, struct in_addr *inp);
in_addr_t inet_addr( const char *cp);

/*
 * From IPv4 binary to text representation
 */
char *inet_ntoa(struct in_addr in);
```

The new address conversion functions which work with both IPv4 and IPv6 addresses are the following ones:

```c
/*
 * From presentation to IPv4/IPv6 binary representation
 */
int inet_pton(int family, const char *src, void *dst);

/*
 * From IPv4/IPv6 binary to presentation
 */
const char *inet_ntop(int family, const void *src,
                      char *dst, size_t cnt);
```

IPv4 source code example:

```c
struct sockaddr_in addr;
char *straddr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;        // family
addr.sin_port = htons(MYPORT);    // port, networt byte order

/*
 * from text to binary representation
 */
inet_aton("138.4.2.10", &(addr.sin_addr));

/*
 * from binary to text representation
 */
straddr = inet_ntoa(addr.sin_addr);
```

IPv6 source code example:

```c
struct sockaddr_in6 addr;
char straddr[INET6_ADDRSTRLEN];
memset(&addr, 0, sizeof(addr));
```
However, the use of these functions should be avoided, using the new resolution ones instead: getnameinfo and getaddrinfo. The inet_* functions work with IPv4 and IPv6, however they are protocol dependent, it means you need to specify which kind of resolution should be made, v4 or v6. Besides these inet_* functions do not work with scoped addresses, so getaddrinfo and getnameinfo are highly recommended.

6. Resolving names

Applications should use names instead of addresses for hosts. Names are easier to remember and remain the same, but numeric addresses could change more frequently.

From applications point of view the name resolution is a system-independent process. Applications call functions in a system library known as the resolver, typically gethostbyname and gethostbyaddr, which are linked into the application when the application is built, see Figure 3. The resolver code is the burden of making the resolution dependent of the system configuration.
There are two new functions to make name and address conversions protocol independent, getaddrinfo and getnameinfo. Besides, the use of these new ones instead of gethostbyname and gethostbyaddr is recommended because the latter are not normally reentrant and could provoke problems in threaded applications.

The getaddrinfo function returns a linked list of addrinfo structures which contains information requested for a specific set of hostname, service and additional information stored in an addrinfo structure.

```c
struct addrinfo {
    int     ai_flags;           /* AI_PASSIVE, AI_CANONNAME */
    int     ai_family;          /* AF_UNSPEC, AF_INET, AF_INET6 */
    int     ai_socktype;        /* SOCK_STREAM, SOCK_DGRAM ... */
    int     ai_protocol;        /* IPPROTO_IP, IPPROTO_IPV6 */
    size_t  ai_addrlen;        /* length of ai_addr */
    struct sockaddr ai_addr;    /* socket address structure */
    char    ai_canonname;       /* cannonical name */
    struct addrinfo ai_next;    /* next addrinfo structure */
};

/* function to get socket address structures */
int getaddrinfo(const char *node, const char *service,
                 const struct addrinfo *hints,
                 struct addrinfo **res);

/* Note: Check the error using !=0 instead of <0 */
On FreeBSD, a failing resolve because the local nameserver is not reachable or times out, it will result >0 which must be also considered as an error.

When writing a typical client application, node and service are normally specified. When writing a server application, they both can be specified too, but in many cases only the service is specified allowing clients to connect to any server interfaces.

Applications should examine the linked list returned by getaddrinfo to use the adequate structure, see Figure 4. In some cases, not all the addresses returned by this function can be used to create a socket.

Figure 4. Linked list of addresses returned by getaddrinfo function.

The getaddrinfo function allocates a set of resources for the returned linked list. The freeaddrinfo function frees these resources.

```c
/* function to free the resources allocated by getaddrinfo */
void freeaddrinfo(struct addrinfo *res);
```

Next, an example of getaddrinfo and freeaddrinfo usage.

```c
error = getaddrinfo(hostname, service, &hints, &res);

/*
Try open socket with each address getaddrinfo returned, until getting a valid socket.
*/
```
if (error !=0 ) {  
  /* Note: Check the error using !=0 instead of <0 */  
  /* On FreeBSD, a failing resolve because the */  
  /* local nameserver is not reachable or times */  
  /* out, it will result >0 which must be also */  
  /* considered as an error. */  
  /* handle getaddrinfo error and return */
}

resave = res;

for (; res; res = res->ai_next ) {  
  sockfd = socket(res->ai_family,  
                  res->ai_socktype,  
                  res->ai_protocol);

  if ((sockfd < 0)) {  
    switch errno {  
      case EAFNOSUPPORT:  
      case EPROTONOSUPPORT:  
        /* last address family is not supported,  
         * continue with the next address returned  
         * by getaddrinfo */  
        continue;

      default:  
        /* handle other socket errors */  
        ;  
    }
  } else {  
    /* socket succesfully created */  

    /* check now if bind()/connect(). If they return errors,  
     * continue the while loop with the next address returned  
     * by getaddrinfo. If they don't return errors, break */  

    /* ... */
  }
}

freeaddrinfo(ressave);

The getnameinfo function provides from a socket address structure the address  
and service as character strings.

char clienthost [NI_MAXHOST];
char clientservice[NXI_MAXSERV];

/* ... */

/* listenfd is a server socket descriptor waiting connections  
   from clients */

connfd = accept(listenfd,  
               (struct sockaddr *)&clientaddr,  
               &addrlen);
if (connfd < 0) {
    /* handle connect error and return */
}

error = getnameinfo((struct sockaddr *)&clientaddr, addrlen,
    clienthost, sizeof(clienthost),
    clientservice, sizeof(clientservice),
    NI_NUMERICHOST);

if (error != 0) {
    /* handle getnameinfo error and return */
}

printf("Received request from host=[%s] port=[%s]\n",
    clienthost, clientservice);

Typically applications do not require to know the version of the IP they are using. Hence, applications only should try to establish the communication using each address returned by resolver until it works. However, applications could have a different behavior when using IPv4, IPv6, IPv4-compatible-IPv6 or IPv4-mapped, etc. addresses.

There are defined some macros to help applications to test the type of address they are using, see Table 2.

Table 2. Macros for testing type of addresses

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN6_IS_ADDR_UNSPECIFIED</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_LOOPBACK</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_MULTICAST</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_LINKLOCAL</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_SITELOCAL</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_V4MAPPED</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_V4COMPAT</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_MC_NODELOCAL</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_MC_LINKLOCAL</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_MC_SITELOCAL</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_MC_ORGLOCAL</td>
<td>(const struct in6_addr *)</td>
</tr>
<tr>
<td>IN6_IS_ADDR_MC_GLOBAL</td>
<td>(const struct in6_addr *)</td>
</tr>
</tbody>
</table>

7. Appendix: Real examples of porting process

This appendix is devoted to show some simple applications which can be used to illustrate porting procedures. This appendix is used to review characteristics and methods described in previous sections.

Migration guidelines are valid for any programming language, however for simplicity, application porting examples are provided only in C language. All these examples have been tested in a SuSE Linux 7.3 distribution, kernel version 2.4.10.
7.1. Original daytime version

Daytime is a simple utility, which returns the time and date of a node in a human-readable format. It is defined in RFCs-867 and it works in the port number 13.

It is a good example to show porting guidelines with a very simple application. Like most of distributed applications, daytime is composed of a client and a server programs. It can operate over TCP and UDP.

7.1. The unicast daytime ported to IPv6

In the following sections it is shown how daytime service is ported from IPv4 to IPv6. Following previous described porting guidelines and after previous program analysis, the code should be reviewed to support IPv4 and IPv6 without compilation.

IPv6 sockets library is similar to IPv4 version, therefore server or client code does not change for the operation phase. The connection phase maintain the same connection model, however the code to set up connections should be adapted.

7.1.1. IPv6 Daytime server (TCP/UDP)

See source code here.

IPv6 Daytime Server

The TCP and UDP versions of the daytime server program use a common function to create the server socket, the listen_server function. This function generates a server socket from a hostname, the service, socket family (IPv4 or IPv6) and socket type (TCP or UDP) parameters.

File "listen_server.h"

```c
#ifndef listen_server_h__
#define listen_server_h__

/*
 * listen_server
 * creates a server server socket listening at a hostname:service
 * using the family and socket type specified in the function
 * arguments.
 */

int
listen_server(const char *hostname,
               const char *service,
               int         family,
               int         socktype);

#endif
```

File "listen_server.cpp"

```c
#include <types.h>
#include <socket.h>
#include <b.h>
#include <o.h>

#include "listen_server.h"
```
const int LISTEN_QUEUE = 128;

int
listen_server(const char *hostname,
               const char *service,
               int         family,
               int         socktype)
{
    struct addrinfo hints, *res, *ressave;
    int error, sockfd;

    memset(&hints, 0, sizeof(struct addrinfo));

    /*
     * AI_PASSIVE flag: the resulting address is used to bind
to a socket for accepting incoming connections.
     * So, when the hostname == NULL, getaddrinfo function will
return one entry per allowed protocol family containing
the unspecified address for that family.
     */

    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = family;
    hints.ai_socktype = socktype;

    error = getaddrinfo(hostname, service, &hints, &res);

    if (error != 0) {
        fprintf(stderr,
"getaddrinfo error:: [%s]\n",
gai_strerror(error));
        return -1;
    }

    ressave = res;

    /*
     * Try open socket with each address getaddrinfo returned,
until getting a valid listening socket.
     */
    sockfd = -1;
    while (res) {
        sockfd = socket(res->ai_family,
                        res->ai_socktype,
                        res->ai_protocol);

        if (sockfd >= 0) {
            if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0)
                break;

            close(sockfd);
            sockfd = -1;
        }
        res = res->ai_next;
    }

    freeaddrinfo(ressave);

    if (sockfd < 0) {

The TCP daytime server uses listen_server with SOCK_STREAM and PF_UNSPEC parameters to obtain a server socket, which will accept connections to all of the interfaces. When clients connect to the server, it will answer with the daytime information and close the client connection.

File "tcp_daytime_server.cpp"

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#include <unistd.h>
#include "listen_server.h"

const char *DAYTIME_PORT="13";

int main(int argc, char *argv[]) {
    int listenfd, connfd, error;
    socklen_t addrlen;
    char timeStr[256];
    struct sockaddr_storage clientaddr;
    time_t now;
    char clienthost[NI_MAXHOST];
    char clientservice[NI_MAXSERV];

    /* local server socket listening at daytime port=13 */
    listenfd = listen_server( NULL, DAYTIME_PORT,
                              AF_UNSPEC, SOCK_STREAM);

    if (listenfd < 0) {
        fprintf(stderr,
                "listen_socket error:: could not create listening "
                "socket\n");
        return -1;
    }

    for ( ; ; ) {
        addrlen = sizeof(clientaddr);

        /* accept daytime client connections */
        connfd = accept(listenfd,
                        (struct sockaddr *)&clientaddr,
                        &addrlen);
```
if (connfd < 0)
    continue;

memset(clienthost, 0, sizeof(clienthost));
memset(clientservice, 0, sizeof(clientservice));

error = getnameinfo((struct sockaddr *)&clientaddr, addrlen,
    clienthost, sizeof(clienthost),
    clientservice, sizeof(clientservice),
    NI_NUMERICHOST);
if (error != 0) {
    fprintf(stderr,
        "getnameinfo error:: can not print the received"
        "request \n");
} else {
    printf("Received request from host=[%s] port=[%s]\n",
        clienthost, clientservice);
}
/* process daytime request from a client */
memset(timeStr, 0, sizeof(timeStr));
time(&now);
sprintf(timeStr, ":%s", ctime(&now));
write(connfd, timeStr, strlen(timeStr));
close(connfd);
}
return 0;

The UDP daytime server uses listen_server with SOCK_DGRAM and PF_UNSPEC parameters to obtain a server socket, which will receive connections to all of the interfaces. When clients connect to the server, it will answer with the daytime information and close the client connection.

File "udp_daytime_server.cpp"
#include <types.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#include "listen_server.h"

const char *DAYTIME_PORT="13";

int main(int argc, char *argv[])
{
    int listenfd, error;
    socklen_t addrlen;
    char *myhost;
    char timeStr[256];
    struct sockaddr_storage clientaddr;
    time_t now;
    char b[256];
char clienthost[NI_MAXHOST];
char clientservice[NI_MAXSERV];

myhost=NULL;
if (argc > 1)
    myhost=argv[1];

listenfd= listen_server(myhost, DAYTIME_PORT, AF_UNSPEC, SOCK_DGRAM);
if (listenfd < 0) {
    fprintf(stderr,
        "listen_server error:: could not create listening 
        "socket\n"
    );
    return -1;
}

addrlen = sizeof(clientaddr);
for ( ; ;) {
    error = recvfrom(listenfd,
        b, 
        sizeof(b),
        0,
        (struct sockaddr *)&clientaddr,
        &addrlen);

    if (error < 0)
        continue;

    memset(clienthost, 0, sizeof(clienthost));
    memset(clientservice, 0, sizeof(clientservice));

    error = getnameinfo((struct sockaddr *)&clientaddr, addrlen,
        clienthost, sizeof(clienthost),
        clientservice, sizeof(clientservice),
        NI_NUMERICHOST);

    if (error != 0) {
        fprintf(stderr,
            "getnameinfo error:: can not print the received" 
            "request \n"
        );
    } else {
        printf("Received request from host=[%s] port=[%s]\n", 
            clienthost, clientservice);
    }

    memset(timeStr, 0, sizeof(timeStr));
    time(&now);
    sprintf(timeStr, "%s", ctime(&now));

    error = sendto(listenfd, timeStr, sizeof(timeStr), 0,
        (struct sockaddr *)&clientaddr,
        &addrlen);
    if (error < 0) {
        ;
    }
}

return 0;
Migration process is simple because all main changes are grouped inside listen_server function. If application is not correctly structured porting effort increases. Sometimes, it is much better to review the program structure that only to change functions calls to make the adaptation.

This is the reason why scripts to change code automatically are not recommended. Automatic scripts look for concrete functions and change them by new function version however, program structure is not analyzed and many times the result is very poor.

7.1.2. IPv6 Daytime client (TCP/UDP).

See source code here.

IPv6 Daytime Client

The TCP and UDP versions of the daytime client program use a common function to create the client socket, the connect_client function. This function generates a client socket from a hostname, the service, socket family (IPv4 or IPv6) and socket type (TCP or UDP) parameters.

File "connect_client.h"

```c
#ifndef connect_client_h
#define connect_client_h

int
connect_client (const char *hostname,
                const char *service,
                int         family,
                int         socktype);
#endif
```

File "connect_client.cpp"

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include "connect_client.h"

int
connect_client (const char *hostname,
                const char *service,
                int         family,
                int         socktype)
{
    struct addrinfo hints, *res, *ressave;
    int error, sockfd;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = family;
    hints.ai_socktype = socktype;
    error = getaddrinfo(hostname, service, &hints, &res);
```
if (error != 0) {
    fprintf(stderr,
            "getaddrinfo error: [\%s]n",
            gai_strerror(error));
    return -1;
}

ressave = res;

sockfd=-1;
while (res) {
    sockfd = socket(res->ai_family,
                    res->ai_socktype,
                    res->ai_protocol);
    if (sockfd >= 0) {
        if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
            break;
        close(sockfd);
        sockfd=-1;
    }
    res=res->ai_next;
}
freeaddrinfo(ressave);
return sockfd;

The TCP daytime server uses connect_client with the following input parameter values: SOCK_STREAM, PF_UNSPEC, the hostname and the port where TCP daytime server is listening. The client socket connects to this server using IPv4 or IPv6 depending on if the server hostname is resolved to an IPv4 or an IPv6 address. Clients will wait for the daytime answer from the server.

File "tcp_daytime_client.cpp"

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include "connect_client.h"

const char *DAYTIME_PORT="13";

int main(int argc, char *argv[])
{
    int connfd;
    char *myhost;
    char timeStr[256];

    myhost = "localhost";
    if (argc > 1)
        myhost = argv[1];
The UDP daytime server uses connect_client with the following input parameter values: SOCK_DGRAM, PF_UNSPEC, the hostname and the port where TCP daytime server is listening. And the same as in the TCP case, the client socket connects to this server using IPv4 or IPv6 depending on if the server hostname is resolved to an IPv4 or an IPv6 address. Clients will wait for the daytime answer from the server.

File "udp_daytime_client.cpp"

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include "connect_client.h"

const char *DAYTIME_PORT="13";

int main(int argc, char *argv[]) {
    int connfd, n, m;
    char *myhost;
    char timeStr[256];
    char letter;

    myhost = "localhost";
    if (argc > 1)
        myhost=argv[1];

    connfd = connect_client(myhost, DAYTIME_PORT, AF_UNSPEC, SOCK_DGRAM);

    if (connfd < 0) {
        fprintf(stderr,
            "client error:: could not create connected socket "
            "socket\n");
        return -1;
    }
```

```c
memset(timeStr, 0, sizeof(timeStr));

while (read(connfd, timeStr, sizeof(timeStr)) > 0)
    printf("%s", timeStr);

close(connfd);
return 0;
```
letter = '1';
memset(timeStr, 0, sizeof(timeStr));

n = read(connfd, timeStr, sizeof(timeStr));

printf("%s\n", timeStr);

return 0;
}

8. Glossary and Abbreviations

<table>
<thead>
<tr>
<th>ADU</th>
<th>Application Data Unit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Program Interface.</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange.</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley System Distribution.</td>
</tr>
<tr>
<td>FQDN</td>
<td>Fully Qualified Domain Name.</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol.</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum Transfer Unit.</td>
</tr>
<tr>
<td>PMTU</td>
<td>Path Maximum Transfer Unit.</td>
</tr>
<tr>
<td>PMTU-D</td>
<td>Path Maximum Transfer Unit Discovery.</td>
</tr>
<tr>
<td>TCP</td>
<td>Transport Control Protocol.</td>
</tr>
<tr>
<td>TU</td>
<td>Transfer Unit.</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol.</td>
</tr>
</tbody>
</table>