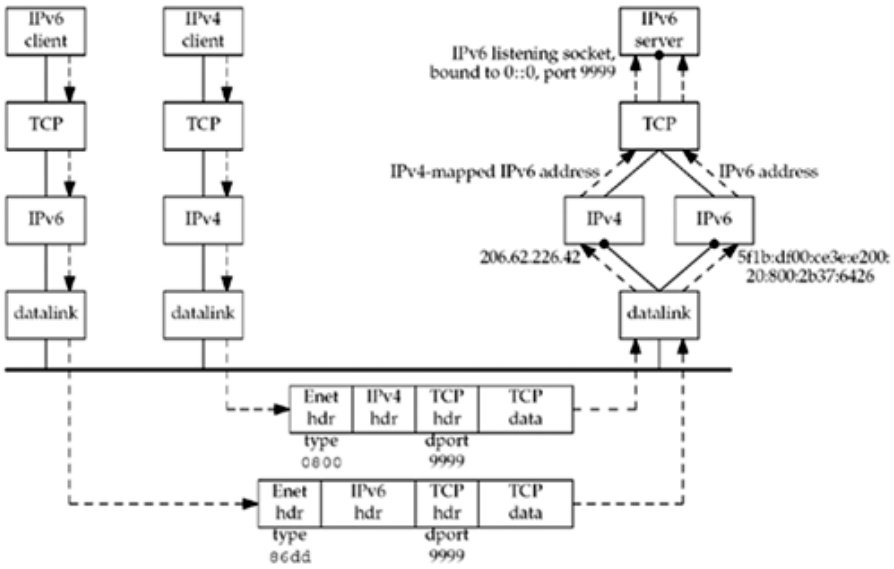
NP Unit 4

1. IPV4 Client and IPV6 server

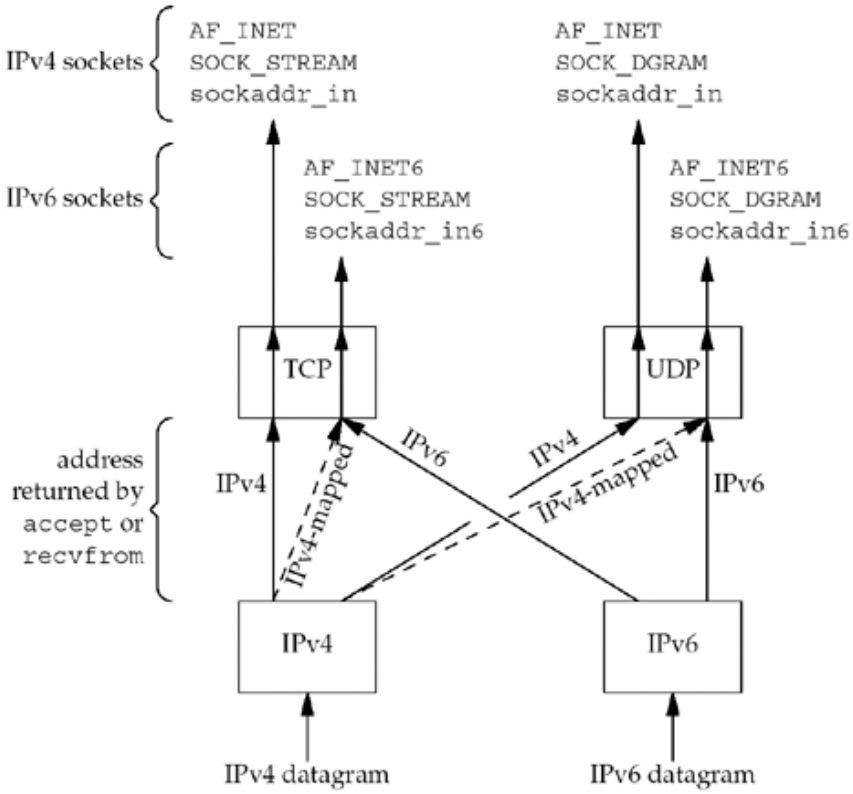


Steps that allow an IPv4 TCP client to communicate with an IPv6 server as follows:

- The IPv6 server starts creating an IPv6 listening socket, and we assume it binds the wildcard address to the socket.
- The IPv4 client calls `gethostbyname` and finds an A record for the server. The server host will have both an A record and an AAAA record since it supports both protocols, but the IPv4 client asks for only an A record.
- The client calls connect and the client's host sends an IPv4 SYN to the server.
- The server host receives the IPv4 SYN directed to the IPv6 listening socket, sets a flag indicating that this connection is using IPv4-mapped IPv6 addresses, and responds with an IPv4 SYN/ACK. When the connection is established, the address returned to the server by accept is the IPv4-mapped IPv6 address.
- When the server host sends to the IPv4-mapped IPv6 address, its IP stack generates IPv4 datagrams to the IPv4 address. Therefore, all communication between this client and server takes place using IPv4 datagrams.
- Unless the server explicitly checks whether this IPv6 address is an IPv4-mapped IPv6 address, the server never knows that it is communicating with an IPv4

client.

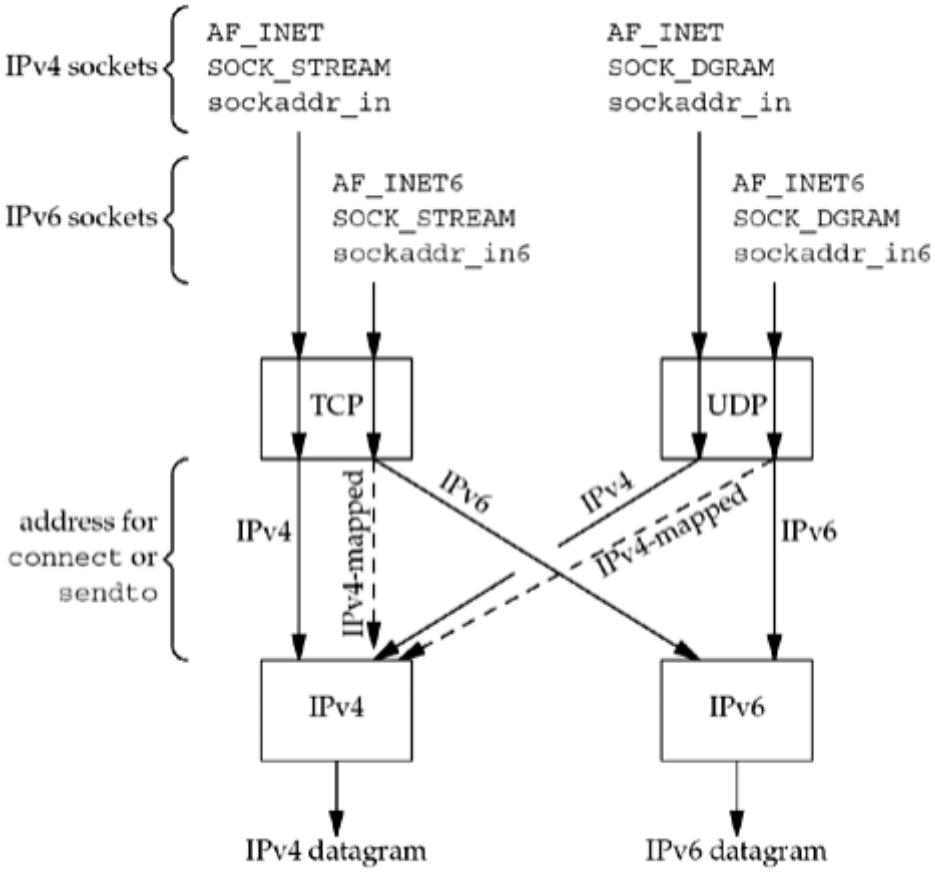
2. Processing of received IPv4 or IPv6 datagrams, depending on the type of receiving socket.



- If an IPv4 datagram is received for an IPv4 socket, nothing special is done. These are the two arrows labelled "IPv4" in the figure: one to TCP and one to UDP. IPv4 datagrams are exchanged between the client and server.
- If an IPv6 datagram is received for an IPv6 socket, nothing special is done. These are the two arrows labelled "IPv6" in the figure: one to TCP and one to UDP. IPv6 datagrams are exchanged between the client and server.
- When an IPv4 datagram is received for an IPv6 socket, the kernel returns the corresponding IPv4-mapped IPv6 address as the address returned by `accept` (TCP) or `recvfrom` (UDP).
- The converse of the previous bullet is false: In general, an IPv6 address cannot be represented as an IPv4 address; therefore, there are no arrows from the IPv6

protocol box to the two IPv4 sockets.

3. IPv6 Client, IPv4 Server



We now swap the protocols used by the client and server from the example in the previous section. First, consider an IPv6 TCP client running on a dual-stack host.

- An IPv4 server starts on an IPv4-only host and creates an IPv4 listening socket.
- The IPv6 client calls `connect` with the IPv4-mapped IPv6 address in the IPv6 socket address structure. The kernel detects the mapped address and automatically sends an IPv4 SYN to the server.
- The server responds with an IPv4 SYN/ACK, and the connection is established using IPv4 datagrams.
- If an IPv4 TCP client calls connect specifying an IPv4 address, or if an IPv4 UDP client calls `sendto` specifying an IPv4 address, nothing special is done.

- If an IPv6 TCP client calls connect specifying an IPv6 address, or if an IPv6 UDP client calls `sendto` specifying an IPv6 address, nothing special is done.

4. IPV6 testing Macros - not that imp, I'm skipping

There is a small class of IPv6 applications that must know whether they are talking to an IPv4 peer. These applications need to know if the peer's address is an IPv4-mapped IPv6 address. The following 12 macros are defined to test an IPv6 address for certain properties.

```
int IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *aptr);
int IN6_IS_ADDR_LOOPBACK(const struct in6_addr *aptr);
int IN6_IS_ADDR_MULTICAST(const struct in6_addr *aptr);
int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4MAPPED(const struct in6_addr *aptr);
int IN6_IS_ADDR_V4COMPAT(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr *aptr);
int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr *aptr);
```

5. Code portability

- Most existing network applications are written assuming IPv4. sockaddr_in structures are allocated and filled in and the calls to socket specify AF_INET as the first argument.
- If we convert an application to use IPv6 and distribute it in source code, we now have to worry about whether or not the recipient's system supports IPv6. The typical way to handle this is with #ifdefs throughout the code, using IPv6 when possible.
- The problem with this approach is that the code becomes littered with #ifdefs very
  quickly, and is harder to follow and maintain.

- A better approach is to consider the move to IPv6 as a chance to make the program
  protocol-independent.
  - The first step is to remove calls to gethostbyname and gethostbyaddr and use
    the `getaddrinfo` and `getnameinfo` functions.
  - This lets us deal with socket address structures as opaque objects, referenced by a
    pointer and size, which is exactly what the basic socket functions
    do: bind, connect, recvfrom, and so on.
- There is a chance that the local name server supports AAAA records and returns both
  AAAA records and A records for some peers with which our application tries to connect.
- If our application, which is IPv6-capable, calls socket to create an IPv6 socket, it will fail
  if the host does not support IPv6.
- Assuming the peer has an A record, and that the name server returns the A record in
  addition to any AAAA records, the creation of an IPv4 socket will succeed.

6. syslogd Daemon - GWAD Question

Unix systems normally start a daemon named syslogd from one of the system initializations scripts, and it runs as long as the system is up. Berkeley-derived implementations of syslogd perform the following actions on startup:

- The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log message that the daemon can receive. These messages can be appended to a file (a special case of which is the file /dev/console, which writes the message to the console), written to a specific user (if that user is logged in), or forwarded to the syslogd daemon on another host.
- A Unix domain socket is created and bound to the pathname /var/run/log (/dev/log on some systems).
- A UDP socket is created and bound to port 514 (the syslog service).

- The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device.
- The syslogd daemon runs in an infinite loop that calls select, waiting for any one of its three descriptors to be readable; it reads the log message and does what the configuration file says to do with that message. If the daemon receives the SIGHUP signal, it rereads its configuration file.

7. syslog function

Syslog stands for System Logging Protocol and is a standard protocol used to send system log or event messages to a specific server, called a syslogserver.

It is primarily used to collect various device logs from several different machines in a central location for monitoring and review.

```
void syslog(int priorityLevel, char msg);
static void err_doit(int errnoflag, int level, const charfmt, va_list ap)
{
 int errno_save n;
 char buf[MAXLINE];
 errno_save = errno;
 if (errno_flag)
   snprintf(buf+n, sizeof(buf)-n, “:%s”, strerror(errno_save));
   strcat(buf, "\n");
   if (daemon_proc) {
     syslog(level, buf);
   } else {
     Fflush(stdout);
     Fputs(buf, stdout);
     Fflush(stdout);
    }
    return;
}
```

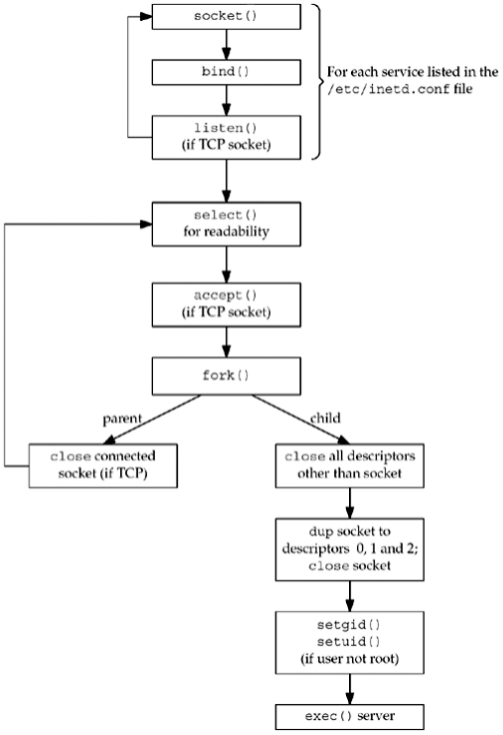
8. daemon_inetd Function

daemon_inetd function: daemonizes process run by inetd.

```
#include "unp.h"
#include <syslog.h>
extern int daemon_proc; /* defined in error.c */
void
daemon_inetd(const char *pname, int facility)
{
  daemon_proc = 1; /* for our err_XXX() functions */
```

```
  openlog(pname, LOG_PID, facility);
}
```

9. inetd Daemon



the inetd daemon. This daemon can be used by servers that use either TCP or UDP. It does not handle other protocols, such as Unix domain sockets.

- It simplifies writing daemon processes since most of the startup details are handled by inetd. This obviates the need for each server to call our daemon_init function.
- It allows a single process (inetd) to be waiting for incoming client requests for multiple services, instead of one process for each service. This reduces the total number of processes in the system.

10. Demon Init function

```
extern int daemon_proc;
int daemon_init(const char *pname, int facility)
{
  int i;
  pid_t pid;
  if ( (pid = Fork()) < 0)
    return (-1);
    else if (pid)
    _exit(0);
    if (setsid() < 0)
      return (-1);
    Signal(SIGHUP, SIG_IGN);
    if ( (pid = Fork()) < 0)
        return (-1);
    else if (pid)
      _exit(0);
```

```
    daemon_proc = 1;
    chdir("/");

    for (i = 0; i < MAXFD; i++)
      close(i);

    open("/dev/null", O_RDONLY);
    open("/dev/null", O_RDWR);
    open("/dev/null", O_RDWR);

    openlog(pname, LOG_PID, facility);

    return (0);
}
```