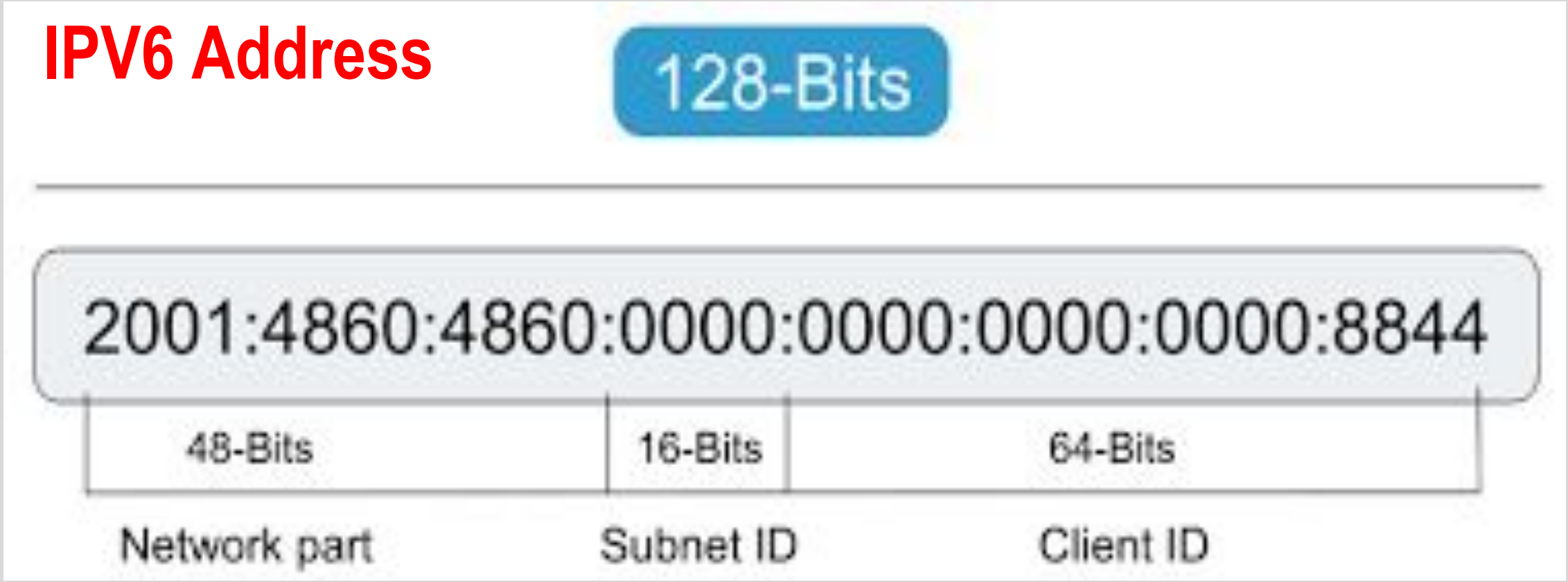
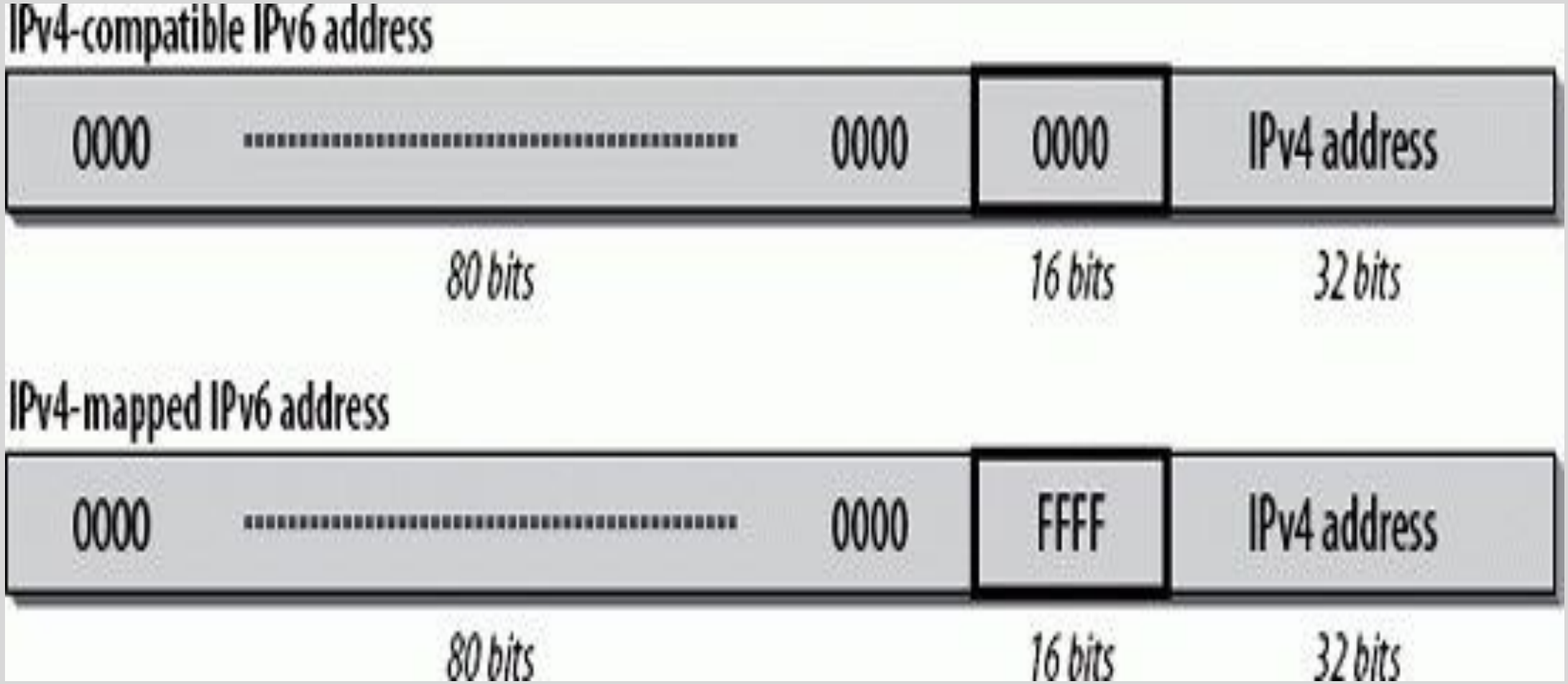
IPv4 vs. IPv6

IPv4	IPv6
Deployed 1981	Deployed 1998
32-bit IP address	128-bit IP address
4.3 billion addresses Addresses must be reused and masked	7.9×10^{28} addresses Every device can have a unique address
Numeric dot-decimal notation 192.168.5.18	Alphanumeric hexadecimal notation 50b2:6400:0000:0000:6c3a:b17d:0000:10a9 (Simplified - 50b2:6400::6c3a:b17d:0:10a9)
DHCP or manual configuration	Supports autoconfiguration

IPV6 Address

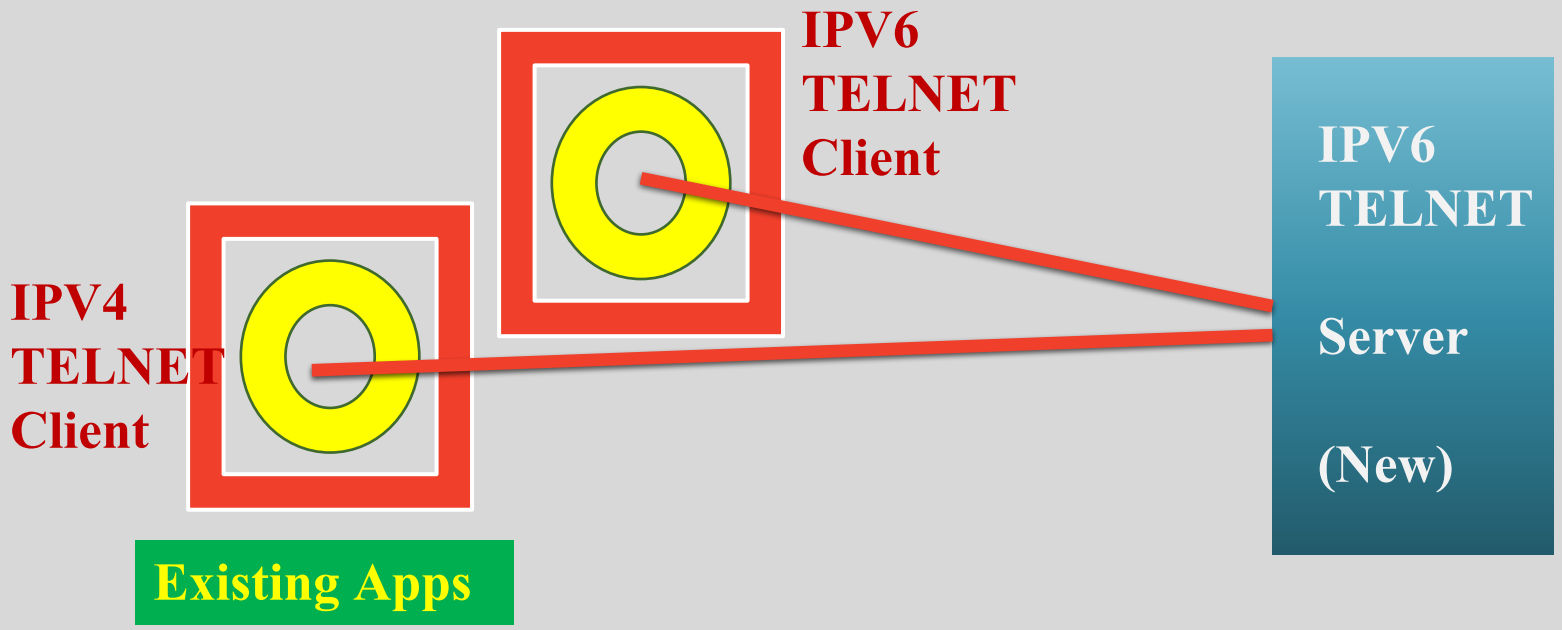


- **2001:0000:3238:DFE1:63:0000:0000:FEFB**

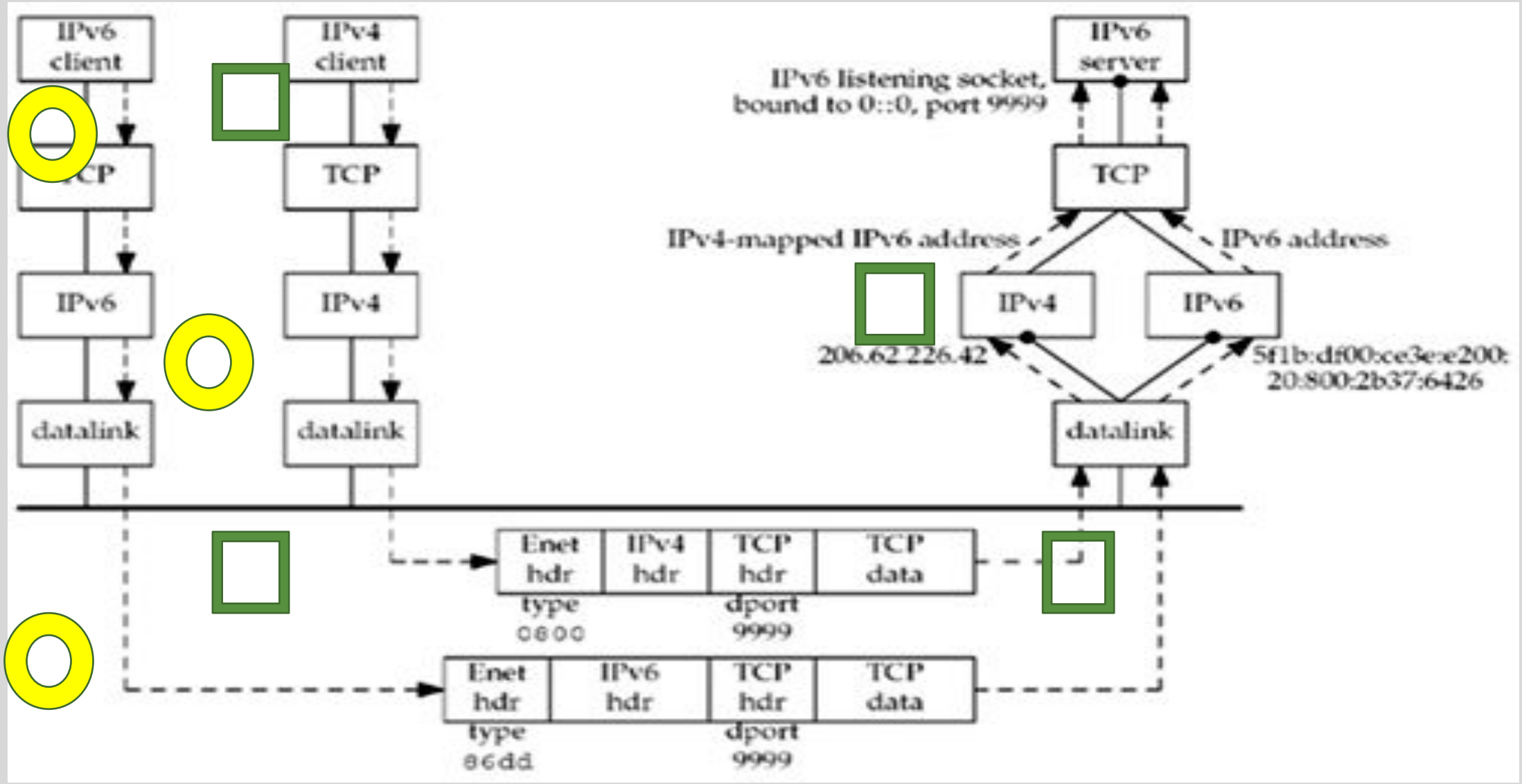

IPv4-compatible IPv6 address
0000
0000
0000
IPv4 address
80 bits
16 bits
32 bits
IPv4-mapped IPv6 address
0000
0000
FFFF
IPv4 address
80 bits
16 bits
32 bits

IPV4 and IPV6 Interoperability

- **Transition will be gradual from IPV4 to IPV6**
- **During this transition phase, existing IPV4 applications must continue to work with newer IPV6 Applications**



IPV4 Client and IPV6 Server



IPV4 Client and IPV6 Server

Step 1. IPV6 Server starts, creates a listening IPV6 socket and it binds wildcard address to the socket

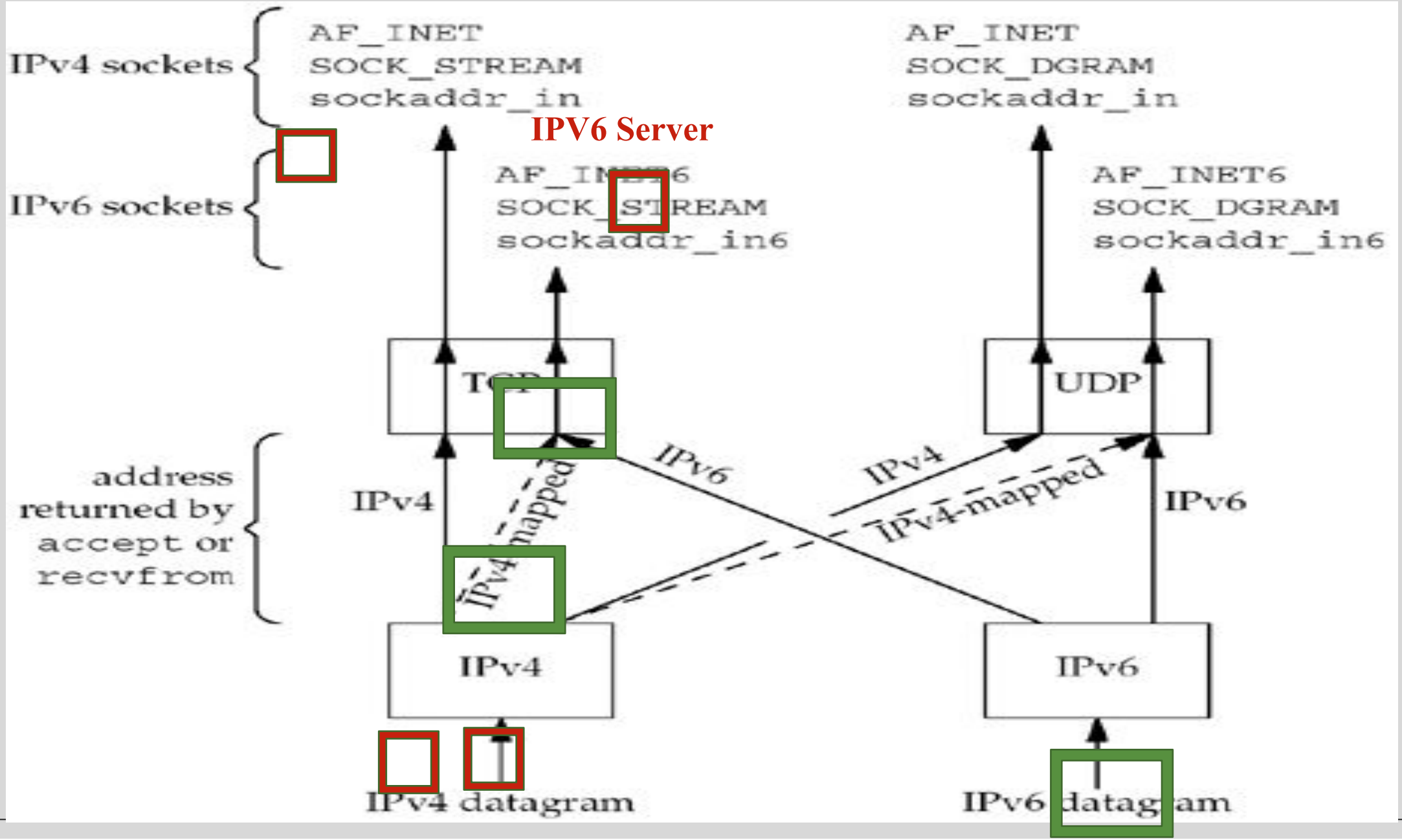
Step 2. IPV4 client cakks gethostbyname and finds an A record..

Steps 3. The Client calls connect and client's host sends an IPV4 SYN to server

Step 4. The Server host receives IPV4 SYN directed to IPV6 socket, sets a flag indicating the this connection is using IPV4 mapped IPV6 address and responds with IPV4 SYN/ACK

Step 5. When the server host sends IPV4-mapped-IPV6 address, IP Stack generates IPV4 datagram to IPV4 address

Step 6. Dual Stack handles all the finer details and the Server is unaware that it is communicating with IPV4 client


IPv4 sockets
IPv6 sockets
AF_INET
SOCK_STREAM
sockaddr_in
AF_INET
SOCK_DGRAM
sockaddr_in
IPV6 Server
AF_INET6
SOCK_STREAM
sockaddr_in6
AF_INET6
SOCK_DGRAM
sockaddr_in6
TCP
UDP
address
returned by
accept or
recvfrom
IPv4
IPv4-mapped
IPv6
IPv4
IPv4-mapped
IPv6
IPv4
IPv6
IPv4 datagram
IPv6 datagram

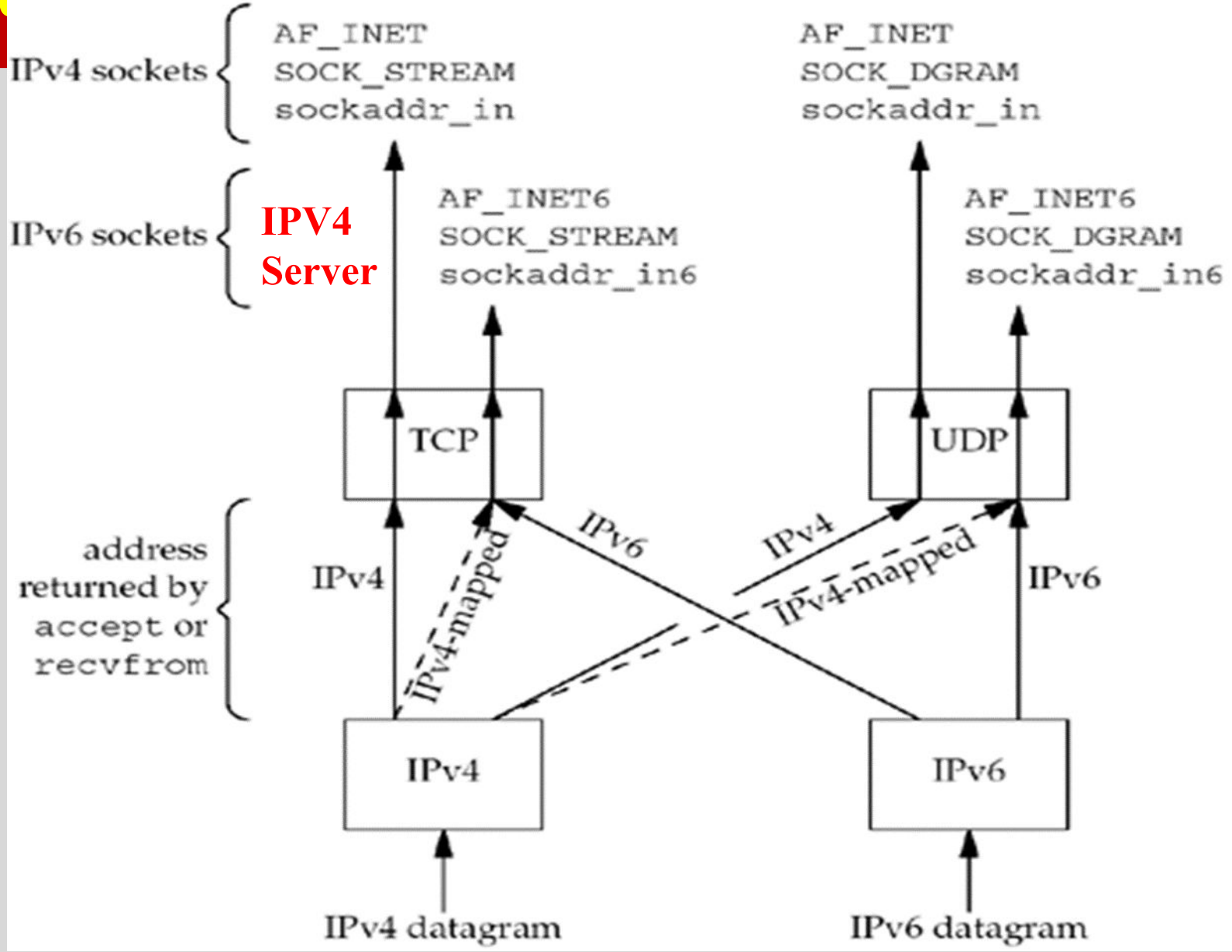
IPV6 Client – IPV4 Server

- **An IPv4 server starts on an IPv4-only host and creates an IPv4 listening socket.**
- **The IPv6 client starts and calls getaddrinfo asking for only IPv6 addresses**
- **it requests the AF_INET6 address family and sets the AI_V4MAPPED flag in its hints structure.**
- **Since the IPv4-only server host has only A records, we see that an IPV4-mapped IPv6 address is returned to the client.**
- **The IPv6 client calls connect with the IPv4-mapped IPv6 address in the IPv6 socket address structure.**

IPV6 Client – IPV4 Server

- **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.**

IPV6 Client – IPV4 Server



IPV4 Client - IPV6 Server

- **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 (using the IN6_IS_ADDR_V4MAPPED macro described in Section 12.4), the server never knows that it is communicating with an IPv4 client.**

- **The dual-protocol stack handles this detail. Similarly, the IPv4 client has no idea that it is communicating with an IPv6 server.**

IPV6 Address Testing Macros

IPv6 Address Macro, Function, Option

- IPv6 address testing macros:
  - IN6_IS_ADDR_* (e.g. V4MAPPED)
- Protocol independent socket address functions:
  - sock_* (e.g. cmp_addr) (section 3.8)
- IPv6_ADDRFORM socket option:
  - change a socket type between IPv4 and IPv6, by setsockopt function with IPv6_ADDRFORM option

IPV6 Address Testing Macros

- There are a small class of IPV6 Applications, that must know whether they are communicating with IPV4 peers.
- These applications need to know whether the peer is using IPV4 mapped IPV6 address
- There are twelve macros that help in this requirement

IPV6 Address Testing Macros

```
int IN6_IS_ADDR_V4MAPPED(const struct in6_addr *aptr);

int IN6_IS_ADDR_V4COMPAT(const struct in6_addr *aptr);
```

Source Code Portability

IPV4 Applications can be converted to IPV6 with minor modifications

But we need to check if the receipient host has support for IPV6

We may do this using #ifdefs in the code.. But code will have too many of these #ifdefs

Better way is to make the program protocol independent

Source Code Portability

The first step is… remove all gethostbyname and gethostbyaddr and use

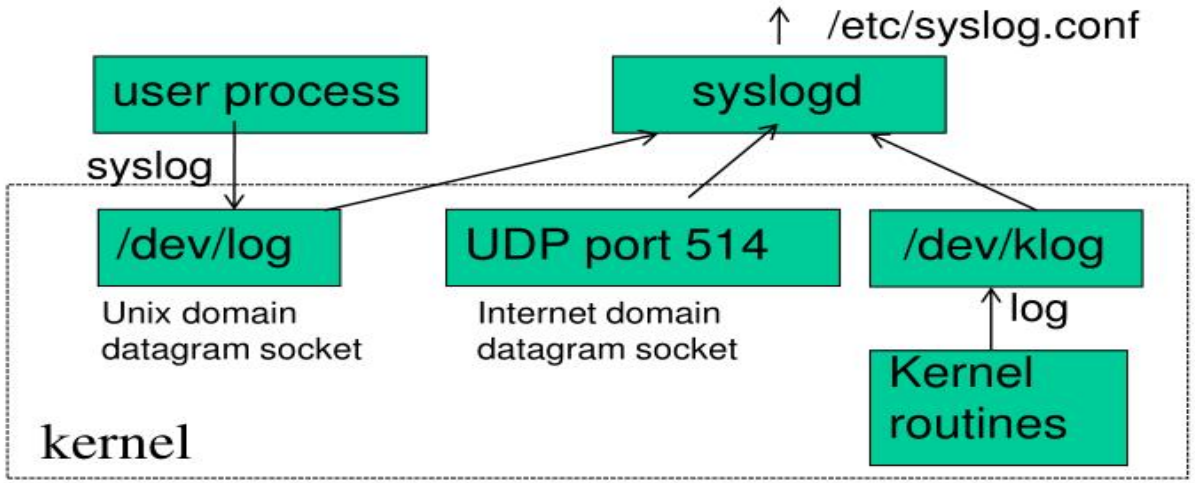
getaddrinfo and getnameinfo functions

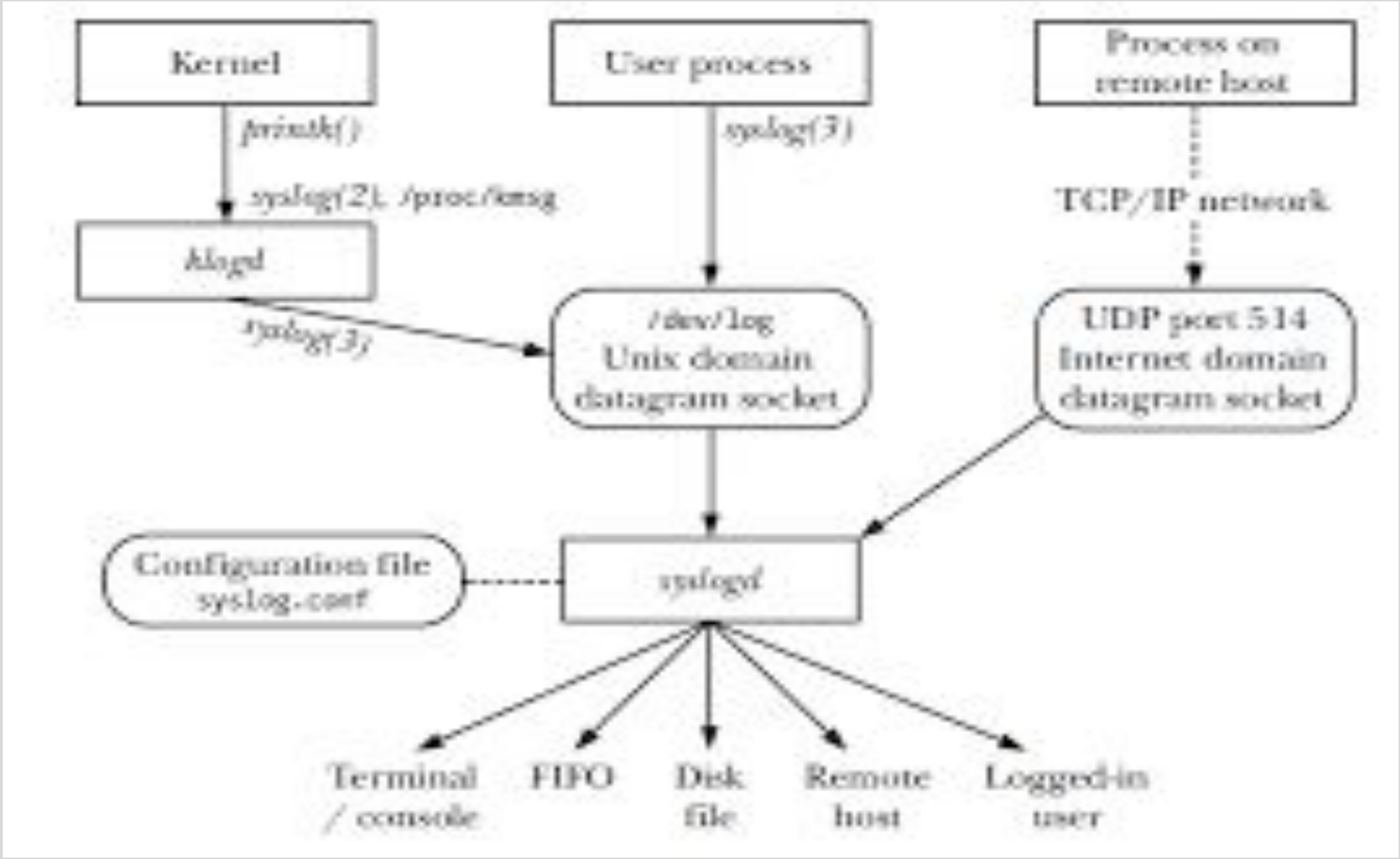
This helps deal with sockaddr as opaque objects.

Syslogd Deamon

Syslogd Daemon

- Different ways to communicate with syslogd
  - Unix domain socket
  - UDP socket (on port 514)
  - A file opened for accepting messages from kernel




Kernel
User process
Process on remote host
printk()
syslog(3)
syslog(2), /proc/kmsg
TCP/IP network
klogd
syslog(3)
/dev/log
Unix domain
datagram socket
UDP port 514
Internet domain
datagram socket
Configuration file
syslog.conf
syslogd
Terminal
/ console
FIFO
Disk
file
Remote
host
Logged-in
user

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);

Message Priority Level

12. 3 `syslog` function

- Log message have a level between 0 and 7.

level	value	description
LOG_EMERG	0	system is unusable (highest priority)
LOG_ALERT	1	action must be taken immediately
LOG_CRIT	2	critical conditions
LOG_ERR	3	error conditions
LOG_WARNING	4	warning conditions
LOG_NOTICE	5	normal but significant condition (default)
LOG_INFO	6	informational
LOG_DEBUG	7	debug-level message (lowest priority)

Figure 12.1 *level* of log message.

Use of syslog() function

```
static void
err_doit (int errnoflag, int level, const char *fmt, va_list ap)
{
        int             errno_save, n;
        char    buf[MAXLINE];

        errno_save = errno;
        vsprintf(buf, fmt, ap); n = strlen(buf);
        if (errnoflag)
                snprintf(buf+n, sizeof(buf)-n, ": %s", strerror(errno_save));
        strcat(buf, "\n");

        if (daemon_proc) {
                syslog (level, buf);   /* send to syslogd daemon for logging */
        } else {
                fflush(stdout);         /* print out to the screen */
                fputs(buf, stdout);
                fflush(stdout);
        }
        return;
}
```

Daemon Init

```
#define MAXFD   64
extern int      daemon_proc;    /* defined in error.c */
void daemon_init (const char *pname, int facility)
{
        int             i;
        pid_t       pid;
        if ( (pid = Fork()) != 0)
                exit(0);                        /* parent terminates */
        /* 1st child continues */
        setsid();                               /* become session leader */
        Signal(SIGHUP, SIG_IGN);
        if ( (pid = Fork()) != 0)
                exit(0);                        /* 1st child terminates */
        /* 2nd child continues */
        daemon_proc = 1;                        /* for our err_XXX() functions */
        chdir("/");                             /* change working directory */
        umask(0);                               /* clear our file mode creation mask */
        for (i = 0; i < MAXFD; i++)
                close(i);
        openlog(pname, LOG_PID, facility);
}
```

Daytime tcp server as a daemon

```
int main(int argc, char **argv)
{
        int   listenfd, connfd;   socklen_t  addrlen, len;
        struct sockaddr *cliaddr; char  buff[MAXLINE];    time_t  ticks;

        daemon_init (argv[0], 0);
        if (argc == 2)
                listenfd = Tcp_listen(NULL, argv[1], &addrlen);
        else if (argc == 3)
                listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
        else
                err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");
        cliaddr = Malloc(addrlen);
        for ( ; ; ) {
                len = addrlen;
                connfd = Accept(listenfd, cliaddr, &len);
                err_msg("connection from %s", Sock_ntop(cliaddr, len));
                ticks = time(NULL);
                snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
                Write(connfd, buff, strlen(buff));
                Close(connfd);
        }
}
```

`inetd` daemon

internet superserver

There are a lot of internet servers that will be running on a typical Unix system

- FTP, telnet, rlogin, TFTP, daytime, time, echo Servers

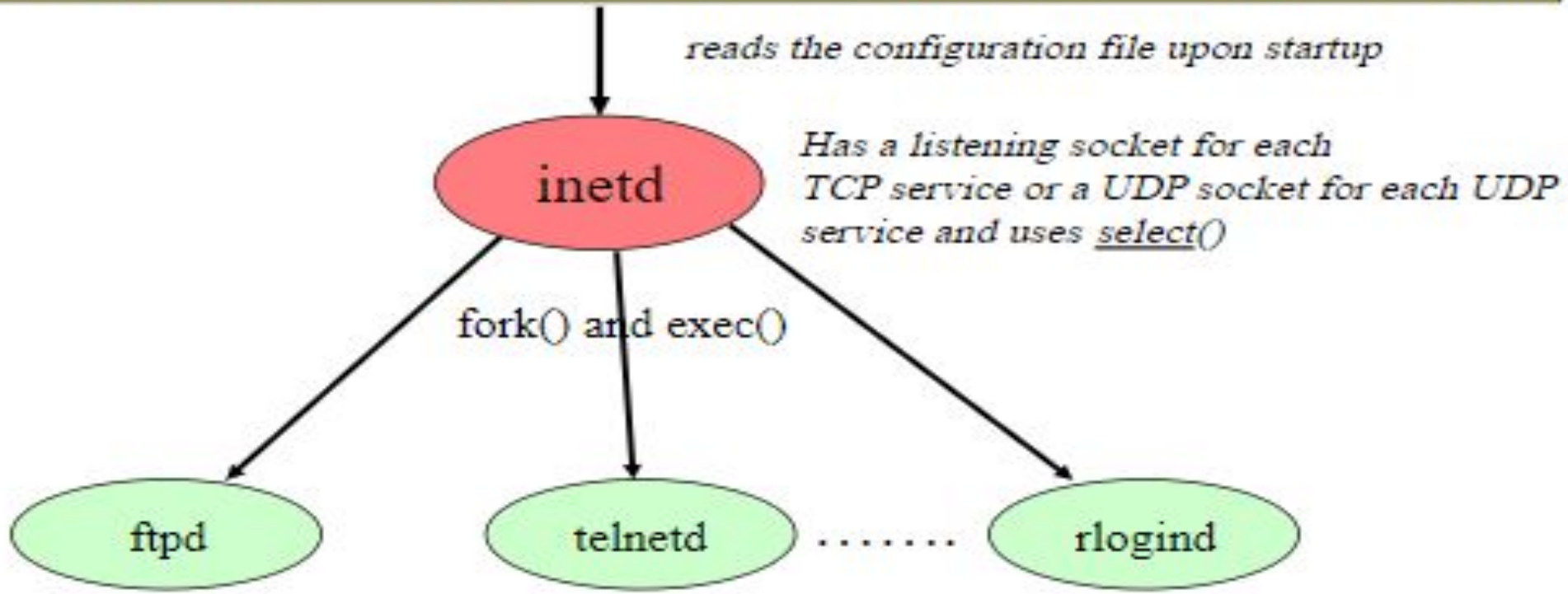
In earlier systems, each service had a process (daemon) associated with.

- each process does the *same startup tasks*
  creating sockets, binding listening….
- each process took *an entry in the process table*
- these processes are *asleep most of the time*waiting for requests.

After 4.3 BSD Unix system, a single server `inetd` daemon is used to wait for requests for all of these services and then create the corresponding server process on demand.

/etc/inetd.conf file

ftp	stream	tcp	nowait	root	/usr/sbin/in.ftpd	in.ftpd
telnet	stream	tcp	nowait	root	/usr/sbin/in.telnetd	in.telnetd
name	dgram	udp	wait	root	/usr/sbin/in.tnamed	in.tnamed
shell	stream	tcp	nowait	root	/usr/sbin/in.rshd	in.rshd
login	stream	tcp	nowait	root	/usr/sbin/in.rlogind	in.rlogind
tftp	dgram	udp	wait	root	/usr/sbin/in.tftpd	in.tftpd -s /tftpboot



steps performed by `inetd`