Network Programming

COSC 1176/1179

Lecture 10

Raw sockets

& Unix domain sockets
Lecture Overview

During this lecture, we will learn

- Unix domain sockets
- IPv4 and IPv6 headers
- Raw socket construction
- Uses of raw sockets
UNIX DOMAIN
SOCKETS
Uses of Unix domain sockets

Used for two-way client/server communication between processes on a **single host**.

All communication is using the normal socket API.

An alternative to other inter process communication (IPC) methods, such as:

- **Unnamed pipes**
  - `pipe( )`
- **Named pipes (FIFO’s)**
  - `mknod( )`
- **Message queues (/dev/mqueue)**
  - `msgget( )`
- **Shared memory**
  - `shmget( )`
Advantages

Often twice as fast as a TCP socket when both peers are on the same host.

- No costs/delays of network infrastructure.
- No checksums are calculated.
- No protocol header encapsulation/decapsulation.
- No routing is performed.
- No ACK's.
- No TCP flow control.
- Packets are not broken down into MTU-sized datagrams.
Advantages

Domain sockets use the file space as the address name space.

- UNIX file permissions can be used to control communication access via umask().

- It is possible to get the clients user ID and group ID on some systems. We will not cover this. Some libc implementations have this functionality, some do not.
Disadvantages

Limited to communication on a single host.

Unix domain sockets do not support the transmission of out-of-band data – the MSG OOB flag for send( ) and recv( ).

No much of a loss here. It is quite rare if you needed to deal with OOB data.
Does anybody actually use it?

It is a very common (preferred) method used for communication between processes (IPC)

Xorg uses it for local machine communication between client/server.

Preferred for systems services
- dbus
- hal
- etc..
Does anybody actually use it?

---

Active sockets on typical Linux box---

IPv4 + IPv6  Unix domain
What you need to know

New protocol and address families.

```c
#define PF_LOCAL 1
#define PF_UNIX PF_LOCAL
...
#define AF_LOCAL PF_LOCAL
#define AF_UNIX PF_UNIX
```

Historically AF_UNIX was used for the UNIX domain protocol. POSIX renamed the protocol to “local IPC” and changed AF_UNIX to AF_LOCAL.

Most people still use the term “UNIX domain” so AF_UNIX is still around.
What you need to know

New socket address structure.

```c
#define UNIX_PATH_MAX 108

struct sockaddr_un {
    uint_8 sun_len;            /* length */
    sa_family_t sun_family;    /* AF_LOCAL */
    char sun_path[UNIX_PATH_MAX]; /* path */
};
```

Where is the port number?
Path should be null-terminated C string.
Unix domain sockets may not appear as “files”.
Creating a UNIX domain socket

```c
int sockfd = socket (domain, type, protocol);
if (sockfd == -1)
    ...
```

Domain can be AF_UNIX or AF_LOCAL

Type can be SOCK_STREAM or SOCK_DGRAM

Protocol has to be 0. Let the kernel deal with it
Binding a UNIX domain socket

```c
struct sockaddr_un un;

memset(&un, 0, sizeof(un));
un.sun_family = AF_LOCAL;
sprintf(un.sun_path, "/tmp/%d.socket", getpid());

unlink(un.sun_path);  /* expect ENOENT error */
// set umask() here to change socket permissions

int ret = bind(sockfd, (struct sockaddr*)&un,
                sizeof(un));
listen(sockfd, 8);
```
Binding a UNIX domain socket

Do not use a relative path for your socket. Always use a full path name (starting `/').

Sockets are normally named after their process name or process id.

It is common to add a `.socket` suffix to your file name.

Make sure to call `unlink()` before you call `bind()`.

If the socket already exists, `bind()` will fail and set `errno` to `EINVAL`. 
What to do next?

Once you have created your UNIX domain socket you can communicate just like you would with a normal file descriptor or network socket using:

**UNIX I/O function calls:**
- `read( )` & `write( )`

**Socket I/O function calls:**
- `recv( )`, `recvfrom( )` & `recvmsg( )`
- `send( )`, `sendto( )`, `sendmsg( )`
Useful functions - socketpair( )

```c
int socketpair( int domain,
               int type,
               int protocol,
               int sockfds[2] );
```

Similar to the UNIX pipe( ) function.

Two descriptors are returned in the sockfds array. Each descriptor is an unnamed UNIX domain socket.

There is no need for any socket/bind/listen/connect calls, they are already connected to one another.
Useful functions - sending a FD

You can send a file/socket descriptor to another process

Requires:
- Unix domain sockets
- `sendmsg()`, `recvmsg()`
- specially formatted messages (`msghdr`)

Descriptors can be for files, sockets, pipe, etc

Once message is sent, the file/socket/etc is shared. Normal for sender to close descriptor after sending.
RAW SOCKETS
Why Raw sockets?

Provide access to other protocols, e.g.
- ICMP #1  control messages
- IGMP #2  group management protocol
- OSPF #89 routing protocol

So servers, routers, monitors etc can be user processes

Provides access to IP headers

Permits you to build your own IP headers!

Provides access to “raw” (low level) packets
Uses of raw sockets

Normal uses:
- Firewalls
- Network tools like ping and tracert
- Monitor programs like wireshark and etherape

Also potential for rogue applications to create malformed IP packets (packet injection)

Packet Sniffing

That's why only superuser/root can create raw sockets
Raw socket creation

soc = socket(AF_INET, SOCK_RAW, ???);
if(soc < 0)
{
    err("socket");
    return EXIT_FAILURE;
}

What can be specified as the protocol? Is the default value of zero acceptable? Will the socket function always succeed?
Raw socket creation

IPPROTO_IP = 0,  No/any protocol
IPPROTO_ICMP = 1,  Internet Control Message Protocol.
IPPROTO_IGMP = 2,  Internet Control Message Protocol.
IPPROTO_TCP = 6,  Transmission Control Protocol.
IPPROTO_UDP = 17,  User Datagram Protocol.
IPPROTO_RAW = 255,  Raw IP packets.

You must be the superuser (root) to use raw sockets!
chk = setsockopt(soc, IPPROTO_IP, IP_HDRINCL, 
&val, sizeof(val));

if(chk < 0)
{
    err("setsockopt");
    return EXIT_FAILURE;
}

If HDRINCL is set:
- kernel does not automatically create IP header
- first byte of data = start of IP header
- kernel always recreates the IP header checksum

If not set:
- kernel creates normal IP headers for you
- first byte of data = start of payload
Binding and Connecting

chk = inet_pton(AF_INET, "127.0.0.1", &local.sin_addr);
if(chk <= 0)
{
    msg("%d, %d", chk, errno);
    err("inet_pton");
    return EXIT_FAILURE;
}

Do we do a bind( )?
- rare but can be done
- what does it imply to the kernel?

Can we do a connect( )?
- also rare (and also can be done on a raw socket)
- the connected address corresponds to what address (on the peer)?
while(TRUE)
{
    memset(&buf, '\0', sizeof(buf));
    memset(&rmo, '\0', sizeof(SA_IN));
    len = sizeof(rmo);

    chk = recvfrom(soc, &buf, sizeof(buf), 0,
                   (SA_PTR)&rmo, &len);

    while(idx < chk)
    {
        fprintf(stdout, "%c", buf[idx++]);
    }

    fprintf(stdout, "\n");
    idx = 0;
}
Reading from raw sockets

What will you receive?

Packets are filtered for you:
- dest address == bind() address
- src address == connect() address
- protocol == socket() protocol

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Filtered</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP</td>
<td>✗</td>
</tr>
<tr>
<td>TCP</td>
<td>✗</td>
</tr>
</tbody>
</table>
| ICMP     | ✓        | except some requests handled by kernel
| IGMP     | ✓        | after kernel has processed them
| other    | ✓        | things kernel doesn't process - always

Fragments will be assembled into whole IP datagram.
Received data includes IP header!
Reading from raw sockets

Want even more?

Unix & Linux also provide lower level “datalink sockets”.

Uses “protocol” type SOCK_PACKET.

Operates similarly to SOCK_RAW but includes all UDP and TCP packets.

Expect no filtering – you receive ALL packets.
Uses of raw sockets

DATA

TCP/UDP

DATA

IP

TCP/UDP

DATA

FRAME

IP

TCP/UDP

DATA

FRAME
### IPv4 Header

<table>
<thead>
<tr>
<th>bit offset</th>
<th>0-3</th>
<th>4-7</th>
<th>8-15</th>
<th>16-18</th>
<th>19-31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Version</td>
<td>Header length</td>
<td>Differentiated Services</td>
<td></td>
<td>Total Length</td>
</tr>
<tr>
<td>32</td>
<td>Identification</td>
<td></td>
<td>Flags</td>
<td></td>
<td>Fragment Offset</td>
</tr>
<tr>
<td>64</td>
<td>Time to Live</td>
<td>Protocol</td>
<td></td>
<td></td>
<td>Header Checksum</td>
</tr>
<tr>
<td>96</td>
<td></td>
<td></td>
<td>Source Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td>Destination Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td></td>
<td></td>
<td>Options</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160 or 192+</td>
<td></td>
<td></td>
<td></td>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

What's inside the “protocol” field?

# IPv6 Header

<table>
<thead>
<tr>
<th>Octet Offset</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Offset</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Bit Offset</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Bit Offset</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Bit Offset</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Bit Offset</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>Bit Offset</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>Bit Offset</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>Bit Offset</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

- 0: Version
- 1: Traffic Class
- 2: Flow Label
- 4: Payload Length
- 8: Next Header
- C: Hop Limit
- 10: Source Address
- 14: Destination Address

**What happened to the checksums?**
**Where do you put the “protocol”?**
**How does a router know the header length?**