08. Address Structures. Accept Call

Addressing Information

Very often there is a need to exchange addressing information between the Application layer and the TCP layer.

Both the client and server need to do this after the socket has been created.

- The client needs to pass the contact details for the server before the Connect Primitive is called.
- The server needs to inform the TCP of the address that it wants to listen on. This is user by the Bind Primitive.

Occasionally there is a need to pass information in the reverse direction (TCP to Application layer).

All addressing information, regardless of direction, must be of the correct byte order and only passed by reference through a standardised address structure. This structure is specified by the Sockets API.

Socket Address Structures

```
struct in_addr {
    in_addr_t s_addr;           // 32-bit IPv4 address netwo
};

struct sockaddr_in {
    uint8_t sin_len;            // length of structure (16)
    sa_family_t sin_family;     // AF_INET
    in-port_t sin_port;         // 16-bit TCP or UDP port nu
                                // network byte ordered
    struct in_addr sin_addr;    // 32-bit IPv4 address
                                // network byte ordered
```

```
    char sin_zero[8];            // unused
};
```

Only concerned with: `sin_family`, `sin_addr` and `sin_port`.

Key features of socket address structures

- They are of local significance only, they are not communicated between different hosts.
- Socket address structures are always passed by reference.

When used with the socket primitives, the structures are adaptable as they can be used with many protocol families, not just TCP/IP. Ordinarily, this adaptability would be provided for by the use of the generic pointer type (`void *`) in the socket primitive definitions. However, the socket primitives pre-date the generic pointer type.

The Generic Socket Address Structure

The socket definitions use a **generic** socket address structure:

```
struct sockaddr {
    uint8_t sa_len;
    sa_family_t sa_family;          // address family: AF_xxx
    char sa_date[14];               // protocol-specific addr
};
```

The socket functions are defined to take a pointer to this generic socket address structure.

```
int bind(listenfd, (struct sockaddr *) &servaddr ...)
/* serveraddr was previously declared as struct sockaddr_in.
     This is then typecast to a pointer of type: struct socka
   socket address structure */
```

Address Structures and Byte-order

To understand how addressing information is stores in the member of the socket address structure, we also need to understand **byte-ordering**.

Fundamentally, hosts store 16-bit integers (2 bytes) in one of two ways:

- Store the **low-order** byte at the starting address, known as **little-endian** byte order
- Store the **high-order** byte at the starting address, known as **big-endian** byte order

There is no requirement for all hosts to use the same byte-order. The order user by a host is known as the **host byte order**.

Byte-order

Client and Server applications will typically extend across systems that use different formats. Consequently, programmers of networked applications must deal with the byte-ordering differences as follows:

- TCP uses 16-bit port number and a 32-bit IPv4 addresses
- Both end-protocol stacks must agree on the order of these bytes to ensure any addressing information exchanged is in the correct order

TCP/IP has defined its own byte order, the **network byte order** (similar to big-endian).

Byte Ordering and Manipulation Functions

The addressing information stored in members of a socket address structure must be converted from **host byte order** to **network byte order**.

Depending on the level of conversion, there are two sets of functions that can be used:

- **Byte Ordering Functions**: are the simplest, they deal with string-to-numeric-to-string conversion.
- **Byte Manipulation Functions**: more complex, they deal with more complicated string manipulation (from dotted-decimal-notation-to-numeric-to-dotted-decimal-notation).

Byte Ordering Functions

There are four byte ordering functions to consider:

- `htons()` — converts host 16-bit value to network byte order
- `htonl()` — converts host 32-bit value to network byte order
- `ntohs()` — converts host 16-bit network value to host byte order
- `ntohl()` — converts host 32-bit network value to host byte order

Byte Manipulation Functions

There are two byte manipulation functions to consider:

- `inet_pton()` — this function takes an ASCII string (presentation) that represents the destination address (in dotted-decimal notation) and converts it to binary value (numeric) for inputting to a socket address structure (network byte order).
- `inet_ntop()` — this function does the reverse conversion, from a numeric binary value to an ASCII string representation (presentation) in dotted decimal notation.

Both functions work with IPv4 and IPv6 addresses:

```
#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);
/* Converts the string pointed to by strptr, storing the bina
     the pointer addptr.
     Returns:
            - 1 if OK
            - 0 if input not a valid presentation format
            - -1 on error
*/

const char *inet_ntop(int family, const void *addrptr, char *
/* Returns:
            - pointer to result if OK
            - NULL on error
*/
```

The family argument for both functions is `AF_INET`, as we are only concerned with IPv4 addresses.

The len argument is the size of the destination buffer. It is passed to prevent the function from overflowing the buffer.

When is Byte Ordering considered?

Some of these Byte Ordering and Manipulation functions were used in the *daytime* and *echo* applications. Specifically when addressing information was required to be passed from the Application layer to the TCP layer.

There is often a need for addressing information to be passed in the reversed direction, from the TCP layer to the Application layer.

One such requirement is when there is a need to capture addressing information relating to client applications contacting servers.

Capturing Client addresses using accept()

Accept is called by a server to return the next connection from the connection queue:

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *a
/* Returns:
            - nonnegative Descriptor if OK
            - -1 on error
*/
```

If the queue is empty, the process is put to sleep.

The accept() Primitive. Client addresses

Accept returns up to three values:

- An integer return code that is either a new socket descriptor or an error indication
- The protocol address of the client process (`cliaddr` pointer)
- The size of this address (`addrlen` pointer)

The `cliaddr` and `addrlen` arguments are used to return the protocol address of the client process.

- Before the call to accept, `*addrlen` is set to the size of the client address structure (`cliaddr`)
- On return, it contains the actual number of bytes stored by the kernel in the socket address structure
- If we are not interested in the address on the client, the last two arguments are set to NULL pointers