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 used by the Bind Primitive.
- Occasional there is a need to pass information in the reverse direction i.e. TCP to Application layer. This will be looked at another time.
- All addressing information regardless of direction must be of the correct byte order (discussed shortly) and can only be passed by *reference* through a standardised address structure.
- This structure is specified by the Sockets API and is discussed in the next slide.

Socket Address Structures

```
struct in_addr
    {
    in_addr_t s_addr;                /* 32-bit IPv4 address network byte
    ordered */
    } ;

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 number
                                   network byte ordered */
     struct  in_addr  sin_addr;    /* 32-bit IPv4 address
                                            network byte ordered */
     char sin_zero[8];  /* unused */
} ;
```

- Only concerned with three members in the structure:
  - *sin_family*, *sin_addr*, and *sin-port*.
  - The *sin_zero* member pads the structure to at least 16 bytes in size

Socket Address Structures

- Socket address structures are of local significance
  - i.e. they are not communicated between different hosts
- Socket address structures are always passed by reference
- However *socket functions* are designed to deal with socket address structures from many supported protocol families
- These functions do not understand the the generic pointer type (*void *)*
- Consequently a *generic* socket address structure of the following form was created:

The *Generic* Socket Address Structure

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

- The socket functions are defined as taking a pointer to this generic socket address structure
- For example in the *bind* function:

```
int bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
```

where, SA is defined as *struct sockaddr* and serveraddr was declared as (*struct sockaddr_in*).

Byte-order

- Two ways to store a 16-bit integer that is made up of 2 bytes:
  - 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

- The byte ordering used by a given system is known as the *host byte order.* Unfortunately there is no standard between these two byte orderings

Byte-order

- Client and server applications will typically that extend across host systems that use both 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
  - TCP/IP uses *big-endian* byte ordering and is known as *network byte order*
- Certain fields within the socket address structures must be converted from *host byte order* to *network byte order*

Byte Ordering and Manipulation Functions

- Depending on the level of conversion there are two sets of functions that can be used;
  - Byte Ordering functions are the simplest in that they deal with string-to-numeric-to-string conversion
  - Byte Manipulation functions are more complex in that they deal with more complicated string manipulation i.e. from dotted-decimal notation-to-numeric-to-dotted-decimal notation
- There are four byte ordering functions:

  htons( ) – converts host 16-bit value to network byte order

  htonl( ) – converts host 32-bit value to network byte order

  ntohs( ) – converts 16-bit network value to host byte order

  ntohl( ) – converts 32-bit network value to host byte order

Byte Manipulation Functions

- **inet_pton**
  - This function takes an ASCII string (*presentation*) that represents the destination address (in dotted-decimal notation) and converts it to a binary value (*numeric*) for inputting to a socket address structure (i.e. *network byte order*)

- **inet_ntop**
  - This function does the reverse conversion, i.e. from a *numeric* binary value to an ASCII string representation (*presentation) i.e. dotted-decimal notation*

Byte Manipulation Functions

- Both functions work with IPv4 <u>and</u> IPv6 addresses

```
#include <arpa/inet.h>
int inet_pton ( int family, const char *strptr, void *addrptr);
```

  - Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error
  - convert the string pointed to by *strptr*, storing the binary result through the pointer *addrptr*

```
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len) ;
```

  - 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 <u>size</u> of the destination buffer and is passed to prevent the function from overflowing the buffer

The *accept* Function – Capturing Client addresses

- Accept is called by a server to return the next connection from the connection queue:
  - If the queue is empty, the process is put to sleep

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

int accept ( int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen ) ;
```

– Returns: nonnegative Descriptor if OK, -1 on error

The *accept* Function – Capturing 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 (through the *cliaddr* pointer)
  - The size of this address (through the *addrlen* pointer)

The *accept* Function – Capturing Client addresses

- The *cliaddr* and *addrlen* arguments are used to return the *protocol* address of the client process:
  - Before the call to *accept* is made:
    - **addrlen* is set to the size of the client address structure (*cliaddr),*
  - On return this integer value contains the actual number of bytes stored by the kernel in the socket address structure.
- If we are not interested in the address of the client the last two arguments are set to **NULL** pointers.

The *accept* Function – an example

```
int main(int argc, char **argv)
{
.
socklen_t  clntAddrLen;  // new variable to hold length of address structure
struct sockaddr_in servaddr, cliaddr;  //new address structure
char clntName [INET_ADDRSTRLEN];  //buffer to hold the client address
.
clntAddrLen = sizeof(cliaddr); //determines length of Client Address Structure
connfd = Accept(listenfd, (SA *) &cliaddr, &clntAddrLen);  // call to accept
printf("connection from %s, port %d\n", inet_ntop(AF_INET,
&cliaddr.sin_addr, clntName, sizeof(clntName)),
ntohs(cliaddr.sin_port));  //print out client address
.
```