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.

Addressing Information

- All addressing information, regardless of direction, must be:
  - Of the correct byte order (discussed shortly) and,
  - 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

- Key features of socket address structures are that:
  - They are of local significance only i.e. they are not communicated between different hosts and,
  - 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.
  - Consequently, a different approach was used as follows:

The *Generic* Socket Address Structure

- The socket definitions use a ***generic*** socket address structure as follows:

```
struct sockaddr

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

};
```

The *Generic* Socket Address Structure

- The socket functions are defined to take a pointer to this <u>generic</u> socket address structure:

- For example in the *bind* function:

  int bind(listenfd, (struct **sockaddr** *) &**servaddr**…….

Notes:

– ***serveraddr*** was previously declared as *struct sockaddr_in*.

– This is then typecast to a pointer of type: struct **sockaddr**, which is the generic socket address structure.

Address Structures and Byte-order

- Having examined the members of the socket address structure, it is important to understand how addressing information is stored in these members:
  - This requires an understanding of ***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 used by a host is known as the ***host byte order.***

Byte-order

- It should be obvious that Client and Server applications will typically extend across host 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:
  - Known as ***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 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.

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 16-bit network value to host byte order

  ntohl( ) – converts 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 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 and 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

Byte Manipulation Functions

- The *family* argument for both functions is AF _INET:
  - We are only concerned with IPv4 addresses.
- The *len* argument is the <u>size</u> 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 they were used when addressing information was required to be passed from the *Application* layer to the *TCP* layer.
- However, there is often a need for addressing information to be passed in the reverse 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 ****addrlen***) ;

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

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

- Notice the second and third arguments to the accept primitive.

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 (through the *cliaddr* pointer)
  - The size of this address (through the *addrlen* pointer)

The ***accept()*** Primtive – 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)
{
……Lines of code omitted
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
…… Lines of code omitted
clntAddrLen = sizeof(cliaddr); //determines length of Client Address Structure

connfd = Accept(listenfd, (struct sockaddr *) &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
…… Lines of code omitted
```