Overview of Socket Primitives using the Lab Exercises

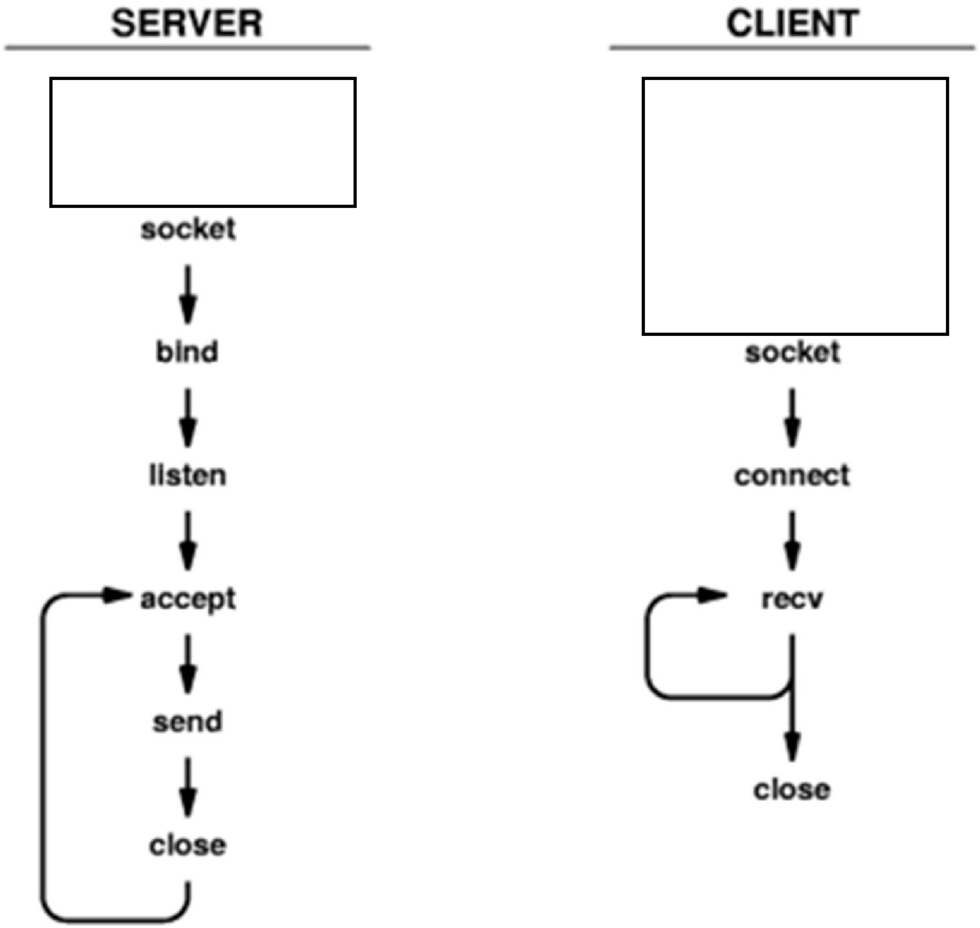
- Refer to the previous class slides for text on each of the Socket Primitives.
- Your previous lab session involved writing a well known *Networked Application* known as *Daytime.*
- The operation of this application is as follows:
  - The Client application makes a connection request to the server using the CONNECT primitive,
  - The Server application accepts the connection using the ACCEPT primitive,
  - The Server application retrieves the Date and Time from its local OS and returns it in a formatted string using the SEND primitive

Overview of Socket Primitives through the Lab Exercises

- The Server application then closes the connection using the CLOSE primitive,
- The Client application retrieves the data from the connection using the RECV primitive, displays it on the local *stdout* and then closes the connection using the CLOSE primitive,
- The Client application kills itself off using the built-in c-function *exit(0);*

◆ The following slide highlights the sequence of primitive calls.

An Example Client-Server Interaction Using *Sockets*



Challenge for this week's lab

- Having examined the **Daytime** *Client* and *Server* applications in some detail the next lab will build upon this understanding.
- You are required to change your *Daytime Client* and *Server* applications to perform as an **Echo** *Client* and *Server* applications.
- The instruction sheet for the lab will outline the task and give some pointers for solving the problem.

Challenge for this week's lab

- The essence of this application is as follows:
  - The Client application will take a string as a command-line argument,
  - This string is sent to the server across the open connection,
  - The Server application will read the string and return it to the Client application exactly as it came in.
- To complete this task it is necessary to understand the operation of the ***recv()*** primitive.

The **recv()** primitive

- Recall its use in the Daytime Client application:

```
while ((numBytes = recv(sock, recvbuffer, BUFSIZE - 1, 0)) > 0)
        {
        recvbuffer[numBytes] = '\0';
        fputs(recvbuffer, stdout);
        }
```

- The following slide outlines some key points about this primitive.

The **recv()** primitive

- The while loop is necessary as the data may not arrive in a single call to *recv()*.
- **numBytes** is the return value from **recv():**
  - It represents the number of bytes read from the socket.
  - It returns one of three values:

    <1 represents an error condition,

    0 represents a closed connection, and,

    >1 represents an open connection with potentially more data to be received.

Challenge for this week’s lab

- If the recv()n primitive is used as above in the **Echo** *Client* and *Server* applications it will cause problems.
- Either or both of the applications will remain inside the loop.
  - You will need to determine how to solve this problem.

Addressing

- A key aspect of *Networked Applications* such as *Daytime* is **Addressing**.
- In order for the Client and Server applications to communicate with each other some form of explicit addressing <u>is</u> required.
- The *Destination* (recall this term from the *Five-component Communications Model*) Server application requires an unambiguous address in order for the *Source* Client application to initiate a connection request.

Addressing

- Recall that the IP layer facilitates transporting datagrams/packets between hosts across an internetwork i.e. *host-to-host* .
- However, given that the data encapsulated inside these datagrams/packets is typically destined for an **Application** on the Destination host:
  - And possibly one of many Applications residing in the Application layer,
  - A finer granularity of addressing is required.
- Here Transport layer addressing plays a vital role.

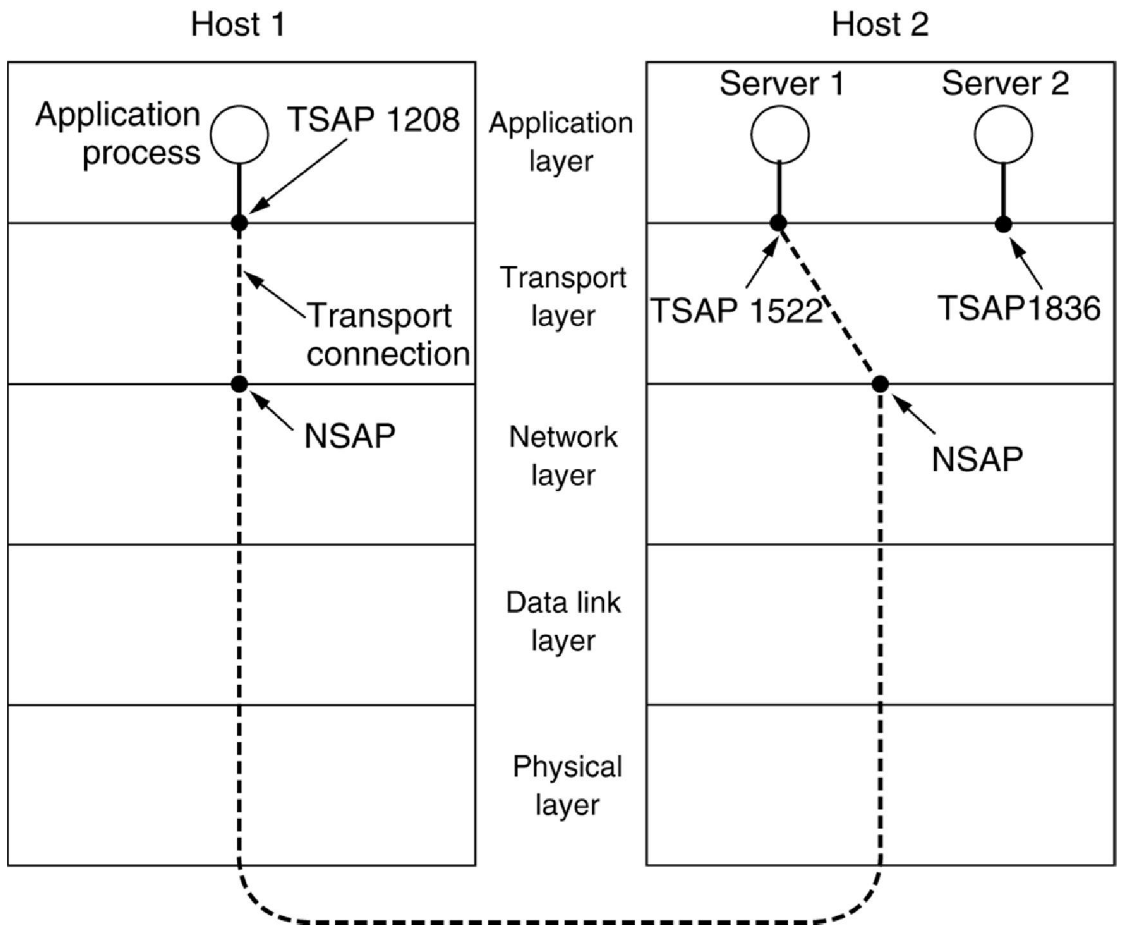
Transport Addressing

- Recall that Transport protocols provide services to the Application layer.
- To ensure unambiguous addressing of individual applications, the Transport layer provides its own addressing schema separate to the IP layer:
  - These are generally known as Transport Service Access Points (TSAPs),
  - These TSAPs uniquely identify entities in the Transport layer known as *end points,*
  - In TCP parlance these end points are known as *ports.*

Transport Addressing

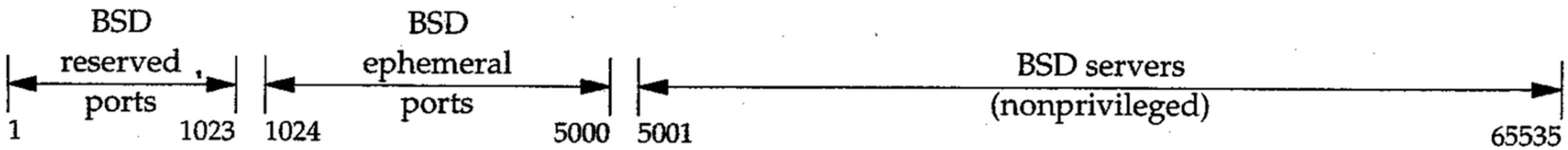
- ***Port*** numbers are used by:
  - Server applications to advertise their services and to Listen for Connection Requests,
  - Client applications to uniquely identify a Server application when making a Connection Request.
- The *network layer* also defines *end points.* These are known as Network Service Access Points (NSAPs):
  - IP addresses are examples of NSAPs.
- The following slide illustrates the relationship between the NSAPs, TSAPs and transport connections

TSAPs, NSAPs and transport connections



The *Port Number* Range

- TCP Port numbers are sixteen bits long.
- This creates a port number address space comprising approx. 65K addresses (16 bits implies 2^16 addresses) as follows:



The *Port Number* Range

- *Reserved* addresses are for well known applications such as HTTP (port 80), FTP (ports 20 and 21), Telnet (port 23) etc.
  – These can only be allocated by users with SU privileges.
- *Ephemeral* addresses are allocated by TCP to **Client** applications:
  – It is not immediately obvious that Client applications require a port number,
  – However, there needs to be a return address for data from the Server application.

The *Port Number* Range

- *Non-privileged* addresses are for any other applications:
  - This is the range that will be used in the lab exercises.
- It is important to note that the Ephemeral and Non-privileged ranges differ on different OS's:
  - Your home host may use different ranges depending on the OS used.

Examining Addressing Details

- The operation of *Port Numbers* from a TCP perspective can be viewed using the *netstat* utility:
  - Simply type the command *netstat –ntap* at the command-line prompt
- This command reveals details on connections that exist within the host OS.

Examining Addressing Details

- An explanation of the flags:
  - ‘n’ reveals IP addresses in dotted-decimal notation,
  - ‘t’ filters on TCP addresses only,
  - ‘a’ shows all connections,
  - ‘p’ reveals the application associated with each connection.
- The following slide shows a sample output when the **netstat** command is used.

Examining Addressing Details

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
Tcp	0	0	0.0.0.0:1022	0.0.0.0:*	LISTEN	12937/webserver
Tcp	0	0	147.252.30.9:1022	147.252.234.34:4136	ESTABLISHED	13268/webserver
Tcp	0	0	147.252.234.34:4136	147.252.30.9:1022	ESTABLISHED	13267/httpclient

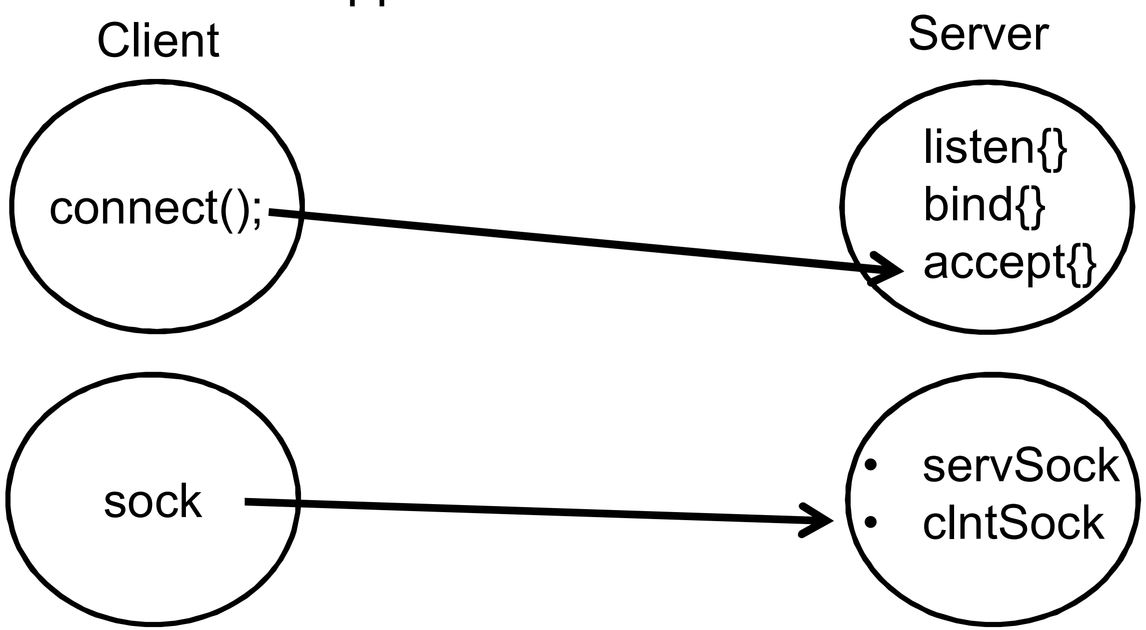
- This shows:
  - A server with *listening* and *connected* sockets on port 1022,
  - A client on an *ephemeral* port 4136.

Socket Pairs

- Note the columns Local Address and Remote Address:
  - These refer to the TSAPs at each end of a connection,
  - These differ in order depending on which end of the connection you are viewing it from.
- This combination of Local Address and Remote Address is known as a ***Socket Pair***:
  - {147.252.30.9:1022, 147.252.234.34:4136} is how the connection is seen from the server's TCP perspective.

Socket Pairs

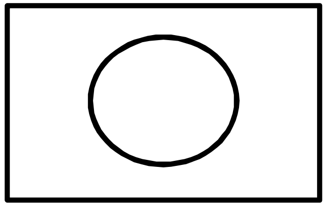
- The following slide reveals how connections are established from a Socket Pair perspective:
  - It assumes the following primitive calls have been made resulting in the creation of socket identifiers within the Client and Server applications:



Sockets before and after Connection Establishment

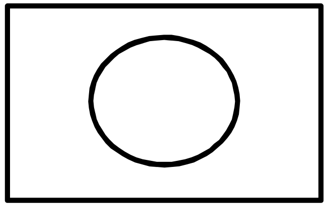
Before Connection Establishment

198.69.10.2



Client Socket: {198.69.10.2 .1500}

206.62.226.35

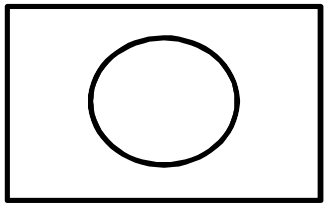


Server Listening Socket: {*.21, *.*}

Client Connection Request to: 206.62.226.35, Port 21 →

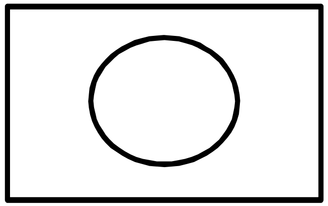
After Connection Establishment

198.69.10.2



Client Connected Socket:
{198.69.10.2 .1500, 206.62.226.35.21}

206.62.226.35



Server Listening Socket: {*.21, *.*}
Server Connected Socket:
{206.62.226.35.21, 198.69.10.2.1500}