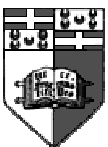




Socket Access Protocol

SERVER SIDE

- A server opens a specific port on the local IP address and listens for a connection.
- A client connects to this port and the connection is established.
- Any data coming on the opened port will be forwarded to the server.
- An internal kernel table is maintained to decide to which port and IP outgoing data has to go.

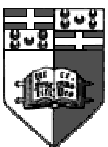




Socket Access Protocol

CLIENT SIDE

- A client requests access to a specific port on the local IP.
- Access is granted if the port is unbound to another process.
- If successful, the client initiates a connection to a well-know port and IP, on which the server is listening.
- Using again an internal table, incoming data will be passed to the client bound to the port.
- Outgoing data will be forwarded to the server port.





Socket Access

- When using sockets apply the following to your programs:

```
#include <sys/socket.h>  
#include <sys/types.h>  
#include <netinet/in.h>
```
- To compile programs using sockets on SunOs systems (ex.babe) use

```
:- gcc -lnsl -lsocket myprog.c
```

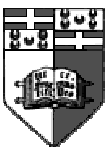
which binds your program to a special socket library
- Some programs such as *ping*, *netstat* and *traceroute* are available in the directory

```
/usr/sbin
```

Ex.

```
/usr/sbin/ping -s <IP>
```

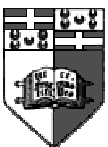
to connect to a remote machine and send packets to and fro.



Unix Socket Access (cont)

- All sockets of all protocols and type use *struct sockaddr* as a base communication structure.
- This structure is a general structure which is then applied type-casted to the specific protocol or type required.

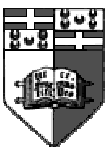
```
struct sockaddr {  
    u_short sa_family; // AF_xxxx  
    char    sa_data[14]; // protocol  
                                // specific  
}
```



Unix Socket Access (cont)

- For internet access, *struct sockaddr_in* is used. It is applied wherever *struct sockaddr* is required.
- Be sure to *bzero()* the structure before using it.

```
struct sockaddr {
    short    sin_family;        // AF_INET
    u_short  sin_port;         // 16 bit port no.
    struct in_addr sin_addr;   // 32 bit IP in
                                // network byte order
    char sin_zero[8]          // set to 0
}
struct in_addr {
    u_long  s_addr; // 32 bit IP in network
                // byte order
}
```

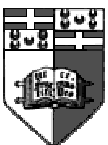




Internet Sockets

- *socket()* opens a new socket and returns its socket descriptor.
- For internet access, *family* is set to `AF_INET`, *type* to `SOCK_STREAM` and protocol to 0 (thus leaving the system to assign the best protocol).
- **N.B.** The *socket()* function does not open any connections or access any ports but creates an endpoint for communication.
- Protocol 0 is internally changed to `IPPROTO_TCP` when `AF_INET` and `SOCK_STREAM` are used.

```
int socket(int family, int type, int protocol);  
        returns socket descriptor or -1 on error
```

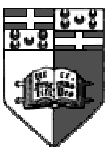




Binding sockets

- A server binds a socket to the local IP address and to a port number it wants to listen on.
- A client can also bind a socket to the local IP address and a port number, but usually we let the system assign an unused port automatically.
- *bind()* completes the local part of the socket tuple.
- *addrlen* should state the size of *myaddr*.
- After binding a socket, any messages received on the bound port will be passed to the binding process.
- Not more than one server should bind itself to a specific port.
- Non-superusers can only bind port numbers 1024-65535.

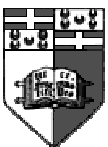
```
int bind(int sockfd, struct sockaddr
          *myaddr, int addrlen);
          returns -1 on error
```





Connecting

- A client tries to connect to a foreign IP address and port by putting the necessary entries in *struct sockaddr_in* and then invoking *connect()*.
- On success, the connection is established and data can flow to and fro.
- The *connect()* function completes the foreign part of the socket tuple.
- If a client calls *connect()* without first calling *bind()* on the socket (unbound) then the system automatically assigns the local IP address and a free unused port to the specified socket. In this case, *connect()* also completes the local part of the socket tuple.
- Automatically assigned port numbers can only be in the range of 1024 to 5000. Why ? (Implementations differ.)

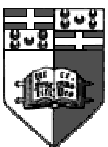




Connecting (cont)

- When trying to connect, the following errors might be reported in *errno*.
 - ETIMEDOUT : timeout on trying to connect
 - ECONNREFUSED: server refused the connection
 - ENETDOWN or EHOSTDOWN: The communication system was unable to connect you.
 - ENETUNREACH or EHOSTUNREACH: Network or host unknown.
 - EISCONN: Socket is already connected.
 - EADDRINUSE: Address already in use.
- *addrlen* should state the size of *myaddr*.
- Internally a connection is retried several times until timeout or success.

```
int connect(int sockfd, struct addr
            *servaddr, int addrlen);
            returns -1 on error
```

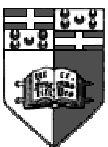




Listening

- After a server binds a specific socket to an IP address and a port, it registers the socket to *listen()* for connections.
- The *backlog* specifies the number of requests for connection that should be queued until the server can handle them.
- Normally backlog is set to the value of 5, the maximum value allowed.
- If the backlog gets full, new connection-requests are simply ignored.

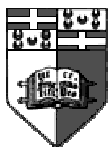
```
int listen(int sockfd, int backlog);  
           returns -1 on error
```



Unix

Connection Acceptance

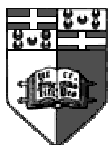
- After a server registers the socket to listen on a connection, it tries to accept a connection.
- *accept()* blocks until a connection request exists on the queue of pending connections.
- *accept()* completes the foreign part of the socket tuple for the server.
- **All** connection-requests are accepted, so it is up to the server to close a connection from an unwanted client.
- The *accept()* system call returns a new socket descriptor to access the new connection.
- The original socket is left open to be able to accept more than one connection.



Unix Connection Acceptance (cont)

- *accept()* fills the *client* structure with the details of the connecting client.
- *addrlen* is a pointer to an integer specifying the size allocated to *client* and on return it will contain the true size used for *client*.

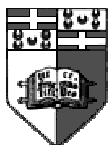
```
int accept(int sockfd, struct addr
           *client, int *addrlen);
returns new socket descriptor
or -1 on error
```





Iterative and Concurrent Servers

- There are two types of servers depending on their behavior after the *accept()* call:
 - **Concurrent:** After *accept()*, a new child is forked which closes the original socket and handles the new connection using the new socket. Meanwhile the parent closes the new socket and calls *accept* again to wait for a new connection.
 - **Iterative:** After *accept()*, the server handles the new connection, closes it and then calls *accept()* again.

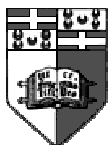




Closing

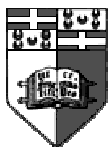
- Calling *close()* will terminate the connection.
- If there is any pending data on the socket, the system will try to send it through.
- Any process (client or server) trying to access, read or write a broken stream will receive the SIGPIPE signal.
- The SIGPIPE signals normally terminates the program. Handling this signal makes your program fault tolerant to an abrupt loss of connection.

```
int close(int sockfd);  
        return -1 on error
```



Unix Reading and Writing data on a socket

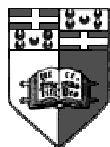
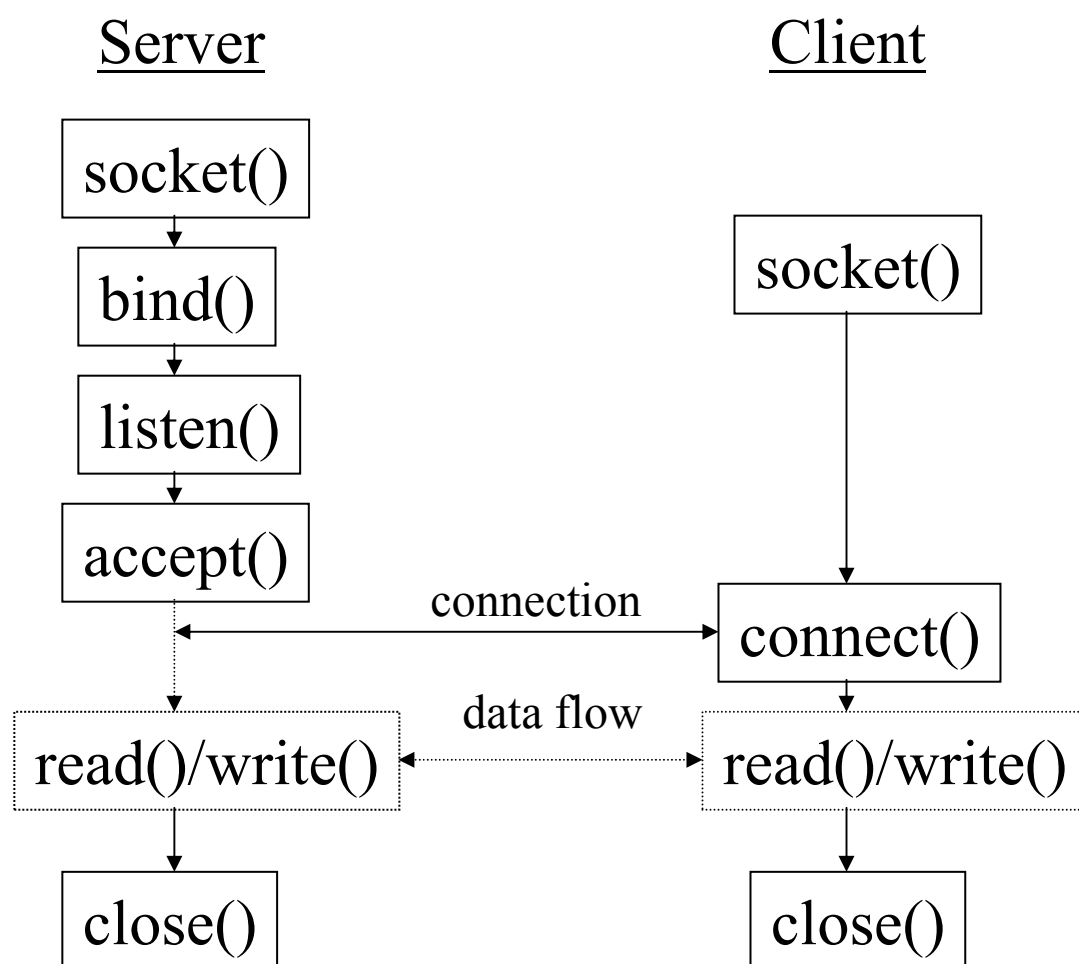
- The usual *read()* and *write()* system calls are used to access sockets.
- The only difference in the system call is that the socket descriptor is used instead of the normal file descriptor.
- All writes on a socket will block until the data will be sent through to the other host, but not until that process reads it.
- All reads from a socket will block until some data is available. *read()* will return the number of bytes read which may be less than requested.
- These blocking calls could lead to a potential deadlock situation unless the communication protocol is well thought out.





Server and Client Typical Design

- The communication protocol between the server and the client is crucial in that it provides synchronization, overall data flow reliability and correctness.





Exercises

- Build a server that listens on a port and a client which connects to it on that port.
- Build an iterative server that accepts a connection, sends through the current time and then closes the connection. Its your time server !!
- Modify the above so that it becomes a concurrent server and each child of the server, upon receiving any character on the socket, sends through the current time on the server side.
- Build a concurrent server that receives a string and executes it. So if it receives 'ls', it will execute 'ls' on the server machine. (This is a mini telnet program without output redirection!!)

