Network Interface

• The network protocol stack is a part of the OS
• Need an API to interface applications to the protocol stack.
  – What’s an API? Do we need standardization?
• The socket interface is the most prominent API. It was developed for BSD. It has since become the de-facto standard.

Socket Interface. What is it?

• Gives a file system abstraction to the capabilities of the network.
• Each protocol offers a set of services. The socket API provides the right abstraction to access these services
• The API defines function calls to create, close, read and write to/from a socket.

Creating a socket

int socket(int domain, int type, int protocol)
The call returns a integer identifier called a handle

Protocol Family
– PF_INET or PF_UNIX
Communication semantics
– SOCK_STREAM or SOCK_DGRAM
Usually UNSPEC

What do you need for socket communication?

• Basically 4 parameters
  – Source Identifier (IP address)
  – Source Port
  – Destination Identifier
  – Destination Port

In the socket API, this information is communicated by binding the socket.

Binding a socket

int bind (int socket, struct sockaddr *address, int addr_len)
• This call is executed by the server.
• It binds the socket to the specified address. The address parameter specifies the local component of the address, e.g. IP address and UDP/TCP port

Listen

int listen (int socket, int backlog)
• This server side call specifies the number of pending connections on the given socket.
• When the server is processing a connection, “backlog” number of connections may be pending in a queue.
Passive Open

int accept(int socket, struct sockaddr *address, int *addr_len)

- This call is executed by the server.
- The call does not return until a remote client has established a connection.
- When it completes, it returns a new socket handle corresponding to the just-established connection

Active Open

int connect(int socket, struct sockaddr *address, int *addr_len)

- This call is executed by the client. *address contains the remote address.
- The call attempts to connect the socket to a server. It does not return until a connection has been established.
- When the call completes, the socket “socket” is connected and ready for communication.

Summary

- **Client:**
  - int socket(int domain, int type, int protocol)
  - int connect(int socket, struct sockaddr *address, int addr_len)

- **Server:**
  - int socket(int domain, int type, int protocol)
  - int bind(int socket, struct sockaddr *address, int addr_len)
  - int listen(int socket, int backlog)
  - int accept(int socket, struct sockaddr *address, int *addr_len)

Message Passing

- int send(int socket, char *message, int msg_len, int flags)
- int recv(int socket, char *buffer, int buf_len, int flags)

Protocol Implementation

- One alternative is; each layer of the protocol stack exposes an API to the higher layer
- Nice clean design, similar to the socket API interface.
- In practice, this is not done due to inefficiencies inherent in the design

Process Model

- Process (thread) is an abstraction provided by the O.S.
- O.S. manages resource (address space, CPU cycles) allocation to processes.
- Context switch – occurs when the O.S. stops one process from executing and starts another
- From the network stack perspective, there are two possible models:
  - process-per-protocol
  - process-per-message
Process per protocol
- Each protocol layer in the stack implemented as a different process
- Messages pass from one process to the other as they move up/down the stack
- A context switch is required when a message moves through each layer of the protocol stack
  - Inefficient

Process per message
- Associates processes with messages. Protocols implemented as procedures
- At each level in the stack, the procedure implementing the corresponding protocol is called
- More efficient: Procedure calls are less expensive than context switches.

Message Buffers
- Each layer of protocol stack adds removes its own header. This forces copying of the message at each layer.
  - Very expensive operation
- Most stack implementations define a message abstraction shared by all layers of the stack.
- The abstraction provides copy free mechanisms for adding/stripping headers, fragmentation and reassembly

Common Support Routines
- Event Manager.
  - Schedules events to be called at a future time.
  - Supports functions to add an event and remove an event from the event list.
  - E.g. timeout handling for reliable transmission
- ID Mapper
  - Provides bindings between identifiers
    - E.g. mapping a TCP port address to a structure that contains the owning process, data transmitted etc
  - Mapper supports multiple maps.
  - Provides functions to insert/delete bindings and query for existence of keys.