TCP Virtual Connections
- Designed for long-term flows of data between a pair of endpoints
- most traffic on internet is TCP

IP Address
- usually use the IP Domain Name a hierarchical name string
- translated to IP Address by the Domain Name Service (DNS)

Determining IP address and port number
- multiple paths in the network between source and destination
- there are many networks between source and destination
- packets travel with an ISP and then between a set of ISPs
- each ISP is connected to several others and pays for access
- if an ISP might favor certain routes, because they will be cheaper
- in any case backbone switches often have choice of which path to use
- one factor in making the choice is relative congestion
- if favoured route is congested, router might pick another output-port for a certain packet
- the choice is made at each router, it is not globally co-ordinated
- and so, packets can arrive at destination in a different order than they were sent

Client-Server Model
- server is a process that
  - waits to receive network messages from clients
  - processes the message in some way
  - sends a response message back to client
- client is a process that
  - sends requests to server and waits for response
  - configuration
  - many clients, one server
  - server is often client for another server (e.g., browser, web-server, database)

Basic communication-endpoint naming
- Internet Protocol address (IP address)
- Port
- Destination address
- Source address

Simple example
- sending process
  - allocates message buffer for payload
  - copies payload data into buffer
  - issues send
- receiving process
  - issues recv to wait on port

Transport protocols
- UDP
  - send/receive datagrams
  - addressing is just IP address and port number
  - best-effort transport
  - but, if any router queue in network is full, message will be dropped
- TCP
  - send/receive streams
  - addressing using virtual connection between two hosts
  - reliable, in-order delivery of stream packets
  - sending rate adapts to available bandwidth
  - reliability provided by software: receipt acknowledgement and retransmission

TVP Virtual Connections
- Designed for long-term flows of data between a pair of endpoints
- most traffic on internet is TCP — otherwise Internet would not work

Establishing a TCP connection
1. Caller brieves to connect, expecting connection for receive
2. Caller returns connection request by calling connect and blocking in connect()
3. Caller returns connect from connect()
- Connection is now established between caller and callee
- Connection is lost when either side is closed

Setup connection request
- setup socket to send connection request
- listen and accept from client socket
- send/receive data on socket
TCP steps on Server

- setup address connection-listening address
  ```
  struct sockaddr_in conAddr;
  memset(&conAddr, 0, sizeof(struct sockaddr_in));
  conAddr.sin_family = PF_INET;
  conAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  conAddr.sin_port = htons(7891);
  bind (so, (struct sockaddr *)&conAddr, sizeof(struct sockaddr_in));
  ```
- setup socket to listen for connection requests
  ```
  listen (so, maxNumberOfPendingRequestsQueued)
  ```
- block waiting for connection requests to arrive
  ```
  struct sockaddr_in caller;
  memset(&caller, 0, sizeof(struct sockaddr_in));
  callerSo = accept (so, (struct sockaddr *)&caller, &cl_len);
  close (callerSo);
  ```
- send/recv messages to/from caller using callerSo socket
  ```
  while (1) {
    printf("..."
  ```

A naive web server

- 1. wait for request
- 2. process request , read from file
- 3. wait for file read to complete
- 4. may repeat 2 & 3 several times
- 5. prepare reply and send
- 6. goto step 1

What is wrong?

**Summary**

**Caller**

1. Create a socket
2. Specify contact point (binding)
3. Listen for calls
4. Accept call
5. Transfer data
6. Close connection

**Callee**

1. Create socket
2. Specify contact point (binding)
3. Listen for calls
4. Accept call
5. Transfer data
6. Close connection

**Other useful functions**

- `inet_aton()` string to network address
- `inet_ntoa()` network address to string
- `gethostbyname()` lookup host by IP domain name (get hostent)
- `gethostbyaddr()` lookup host by IP address

**A few additional details**

- purpose of bind step at server
  - each machine typically has multiple network interfaces
  - and so it might have multiple IP addresses
  - bind picks the one to be used for this session
  - bind also picks the connection-request port number (e.g., port 80)

- finding out who called
  ```
  struct sockaddr_in conAddr; // caller
  int con_len = sizeof (conAddr);
  int callerSo = accept (so, (struct sockaddr *)&conAddr, &con_len);
  close(callerSo);
  ```

**Complete Example (server)**

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main() {

  struct sockaddr_in lHost;
  memset(&lHost, 0, sizeof(lHost));
  lHost.sin_family = AF_INET;
  lHost.sin_port = htons(7891);
  lHost.sin_addr.s_addr = htonl(INADDR_ANY);

  fd = socket(PF_INET, SOCK_STREAM, 0);
  bind(fd, (struct sockaddr *)&lHost, sizeof(lHost));
  listen (fd, 4);

  while (1) {
    struct sockaddr_in caller;
    int cl_len = sizeof (caller);
    int callerSo = accept (so, (struct sockaddr *)&caller, &cl_len);
    close (callerSo);
  }
}
```

**Complete Example (caller)**

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main() {

  struct sockaddr_in remHost;
  unsigned long callerIP = 0;
  int callerSo = 0;

  callerIP = htonl(INADDR_ANY);
  callerSo = accept (so, (struct sockaddr *)&remHost, sizeof(struct sockaddr_in));
  close(callerSo);
}
```

**BSD Socket API Summary**

- `socket()` creates the socket
- `connect()` initiates a connection
- `bind()` indicates the IP address to use
- `listen()` marks the socket to receive connections
- `read()/recv()` reads data
- `write()/send()` sends data
- `close()` shuts down the connection
- `accept()` waits for incoming connection