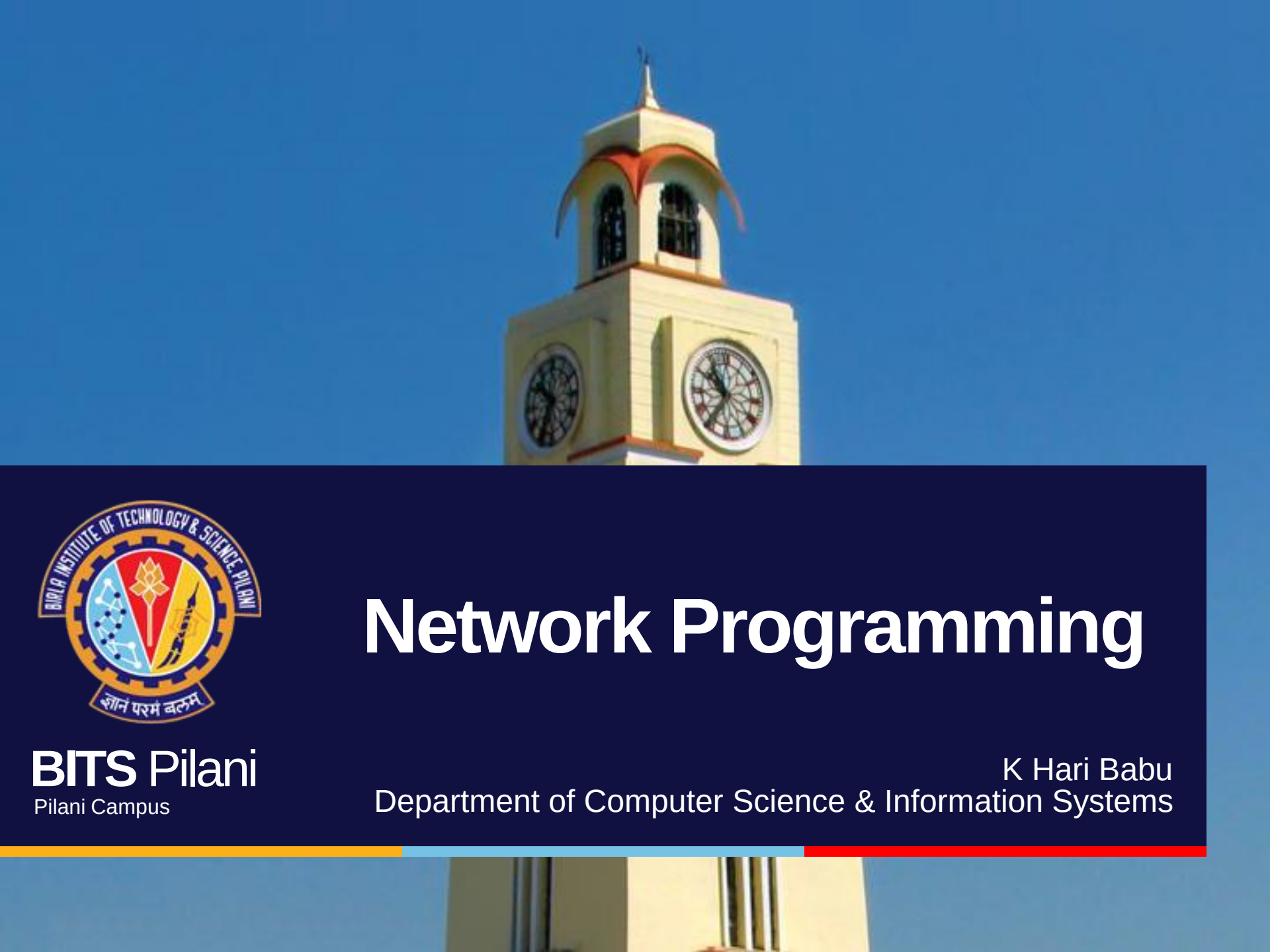


Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

BITS Pilani

Pilani Campus

Outline

Outline

- Socket Options
- UDP Sockets
  - Client
  - Server
    - Connect() call
- TCP or UDP?
- Domain Name Conversion
  - get hostbyname()
  - getaddrinfo()

innovate achieve lead

BITS Pilani

Pilani Campus

Socket Options

Socket Options

innovate achieve lead

- Socket options affect various features of the operation of a socket.

```
1  #include <sys/socket.h>
2  int getsockopt(int  sockfd , int  level , int  optname , void * optval ,
3                 socklen_t * optlen );
4  int setsockopt(int  sockfd , int  level , int  optname , const void * optval ,
5                 socklen_t  optlen );
6  //Both return 0 on success, or -1 on error
```

  - *sockfd* - socket descriptor
  - *level* - system layer that interprets the option
    - general socket (SOL_SOCKET),
    - protocol specific (IPPROTO_IP, IPROTO_TCP, etc.)
  - *optname* - name of the option
  - *optval* - value of the option
  - *optlen* - length of the option value

Socket Options (2)

- Two types of options
  - Flag - binary option that enables or disables a feature
  - Value - can be integer, char, struct timeval, and more
- Levels of socket options
  - General SOL_SOCKET
    - SO_BROADCAST, SO_DONTROUTE, SO_KEEPALIVE, SO_LINGER, SO_RCVBUF, SO_SNDBUF
  - IP, ICMP, IPV6, ICMPV6
    - IP_HDRINCL, IP_TOS, IP_TTL, IP_MULTICAST_TTL
  - TCP
    - TCP_MAXSEG, TCP_NODELAY

Generic Socket Options (1)

- SO_BROADCAST
  - Enabling a process to send broadcast packets
  - UDP type datagram sockets only, no connection-oriented sockets
  - works if supported by network (e.g. ethernet)
- SO_DONTROUTE
  - Bypassing normal routing mechanisms
  - Used by routing daemons, such as gated or routed
- SO_KEEPALIVE

SO_KEEPALIVE

- Normally used by servers, but it is not part of the standard
- When the connection is idle and the peer host crashes or becomes unreachable, setting this option sends out 9 keepalive probes after 2 hours of inactivity
- Probes are sent 75 seconds apart, and can only be changed as a system parameter
  - Peer responds with the expected ACK
    - Another probe will be sent after another 2 hours
  - Peer responds with an RST
    - Socket is closed and error set to ECONNRESET
  - Peer does not respond
    - Another probe is sent after 75 seconds
    - If there is no response to all the probes, socket is closed with error set to ETIMEDOUT or EHOSTUNREACH

SO_LINGER

```
1  struct linger {
2     int l_onoff;  /* 0=off, nonzero=on */
3     int l_linger; /*linger tme in secs */
4  }
```

- Only TCP close() is affected by SO_LINGER

no linger - graceful shut

- **Case 1**: linger->l_onoff is zero (linger->l_linger has no meaning)… This is the default.

  Close will return immediately. On close(), the underlying stack attempts to **gracefully shutdown** the connection after ensuring **all unsent data is sent**. In the case of connection-oriented protocols such as TCP, the stack also ensures that **sent data is acknowledged** by the peer. The stack will perform the above-mentioned graceful shutdown in the **background** (after the call to close() returns), regardless of whether the socket is blocking or non-blocking.

linger 0 time - abort

- **Case 2**: linger->l_onoff is non-zero and linger->l_linger is zero:

  A close() returns immediately. The underlying stack **discards any unsent data**, and, in the case of connection-oriented protocols such as TCP, **sends a RST** (reset) to the peer (this is termed a hard or abortive close). All subsequent attempts by the peer's application to read()/recv() data will result in an ECONNRESET.

linger some time

- **Case 3**: linger->l_onoff is non-zero and linger->l_linger is non-zero:

  A close() will either block (if a blocking socket) or fail with EWOULDBLOCK (if non-blocking) until a graceful shutdown completes or the time specified in linger->l_linger elapses (time-out). Upon time-out the stack behaves as in case 2 above.

Generic Socket Option (5)

- SO_RCVBUF
  - Each socket has a receiver buffer
  - Changes the receiver buffer size
  - The timing of setting this option is important
- SO_SNDBUF
  - TCP socket has a send buffer but UDP socket does not
  - For TCP, SO_SNDBUF changes the send buffer size
  - For UDP, SO_SNDBUF limits the maximum UDP datagram size

Generic Socket Options (6)

- SO_RCVLOWAT
  - Each socket has a receive low-water mark
  - Amount of data that must be in the socket receive buffer before `select()` returns "readable"
- SO_SNDLOWAT
  - Each socket has a send low-water mark
  - Amount of space that is available in the socket send buffer before `select()` returns "writable"

Generic Socket Options (7)

- SO_REUSEADDR
  - Allows a listening server to start and bind its well-known port, even if previously established connections exist that use the same port as their local port
    - when a server restarts while previous child is still alive
  - Allows a new server to be started on the same port as an existing server that is bound to the wildcard address, as long as each instance binds to a different local IP address
    - multiple servers can reside on the same machine as long as each server has a different local ip address
  - Security concerns of SO_REUSEADDR
    - some systems do not allow specific binding to happen AFTER wildcard binding
    - avoid rogue server to attach to existing services

IPv4 Socket Options

- IP_HDRINCL
  - Set for a raw IP socket
  - The program can build its own IP header for all the datagrams sent on the raw socket
- Raw sockets
  - Read and write ICMP and IGMP packets
  - Read and write IPv4 datagrams with an IPv4 protocol field not processed by the kernel
    - OSPF (89)
  - Build its own IP header using IP_HDRINCL socket option

TCP Socket Options

- TCP_MAXSEG
  - Fetch and set the TCP Maximum Segment Size
- TCP_NODELAY
  - If set, disables TCP Nagle algorithm
  - TCP Nagle algorithm states that if a given connection has outstanding data (sent but not acknowledged), then no small packets will be sent on the connection
  - Reduces the number of small packets

innovate achieve lead

BITS Pilani

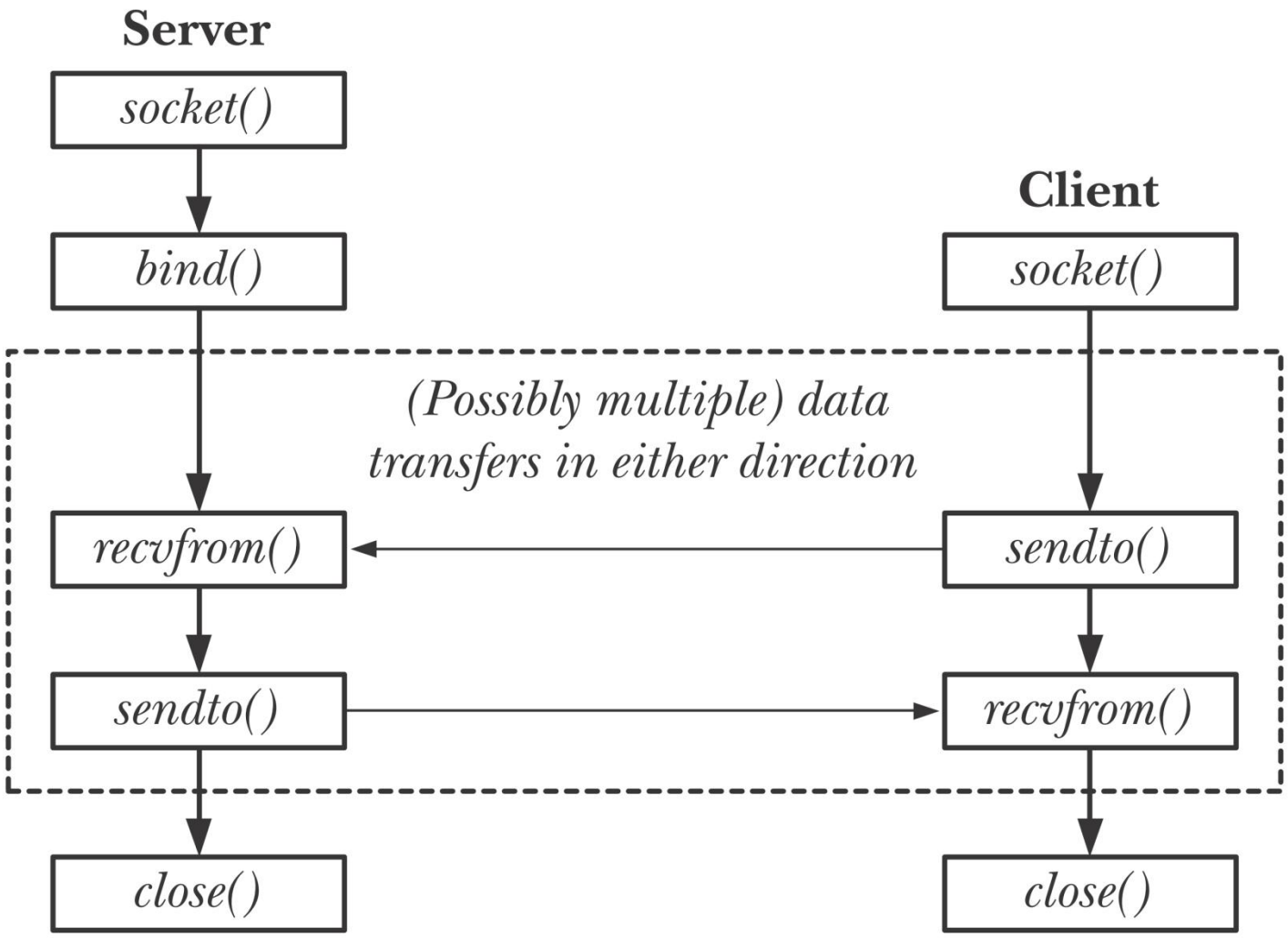
Pilani Campus

UDP Sockets

UDP Sockets

- If TCP communication is compared to making a telephone call, UDP communication is compared to posting a letter.
  - Connectionless
  - Unreliable
  - Datagram protocol
  - Popular applications
    - DNS(the Domain Name System)
    - NFS(the Network File System)
    - SNMP(Simple Network Management Protocol)
    - Multimedia
- Just as with the postal system, when multiple datagrams (letters) are sent from one address to another, there is no guarantee that they will arrive in the order they were sent, or even arrive at all.

UDP Client Server



Exchanging Messages

- The *recvfrom*() and *sendto*() system calls receive and send datagrams (*one at a time*) on a datagram socket.

```
1  #include <sys/socket.h>
2  ssize_t recvfrom(int  sockfd , void * buffer , size_t  length ,
3      int  flags ,struct sockaddr * src_addr , socklen_t * addrlen );
4  //Returns number of bytes received, 0 on EOF, or -1 on error
5  ssize_t sendto(int  sockfd , const void * buffer , size_t  length ,
6      int  flags , const struct sockaddr * dest_addr , socklen_t  addrlen );
7  //Returns number of bytes sent, or -1 on error
```

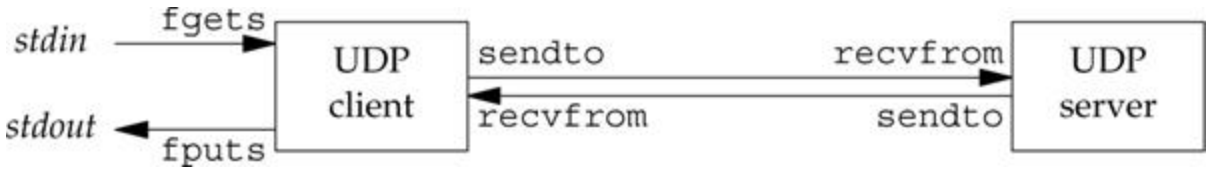
  - The return value and the first three arguments to these system calls are the same as for read() and write().
  - Final two arguments of *recvfrom*() and *sendto()* are similar to the final two arguments in *accept*() and *connect*().

Recfrom() & sendto()

- Both functions return the length of data that was written or read.
- Writing a datagram of length 0 is acceptable.
  - IP+UDP header + no app layer data.
- recvfrom() can return 0.
  - This is not same as read() returning 0.
  - 0 is the length of datagram.
- Regardless of the value specified for length, *recvfrom*() retrieves exactly one message from a datagram socket.
  - If the size of that message exceeds length bytes, the message is silently truncated to length bytes.
  - Using rcvmsg(), MSG_TRUNC flag can be used to detect this.

UDP Client-Server

- We will use echo protocol.



- Server
  - Create UDP socket & bind at well-known port number.
  - Read datagram and echo back to sender
- Client
  - Create UDP Socket
  - Read from *stdin* and send datagram to server
  - Read from UDP socket and write to *stdout*.

UDP Server

innovate achieve lead

```
1  int
2  main(int argc, char **argv)
3  {
4      int     sockfd;
5      struct sockaddr_in servaddr, cliaddr;
6      sockfd = socket(AF_INET, SOCK_DGRAM, 0);
7      bzero(&servaddr, sizeof(servaddr));
8      servaddr.sin_family = AF_INET;
9      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
10     servaddr.sin_port = htons(SERV_PORT);
11     bind(sockfd, (SA *) &servaddr, sizeof(servaddr));
12     dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
13 }
```

- UDP socket is created by specifying the second arg as SOCk_DGRAM.

UDP Server

innovate achieve lead

```
1   #define    SA   struct sockaddr
2   void dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
3    {
4       int     n;
5       socklen_t len;
6       char    mesg[MAXLINE];
7       for ( ; ; ) {
8           len = clilen;
9           n = recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
10          sendto(sockfd, mesg, n, 0, pcliaddr, len);
11      }
12   }
```

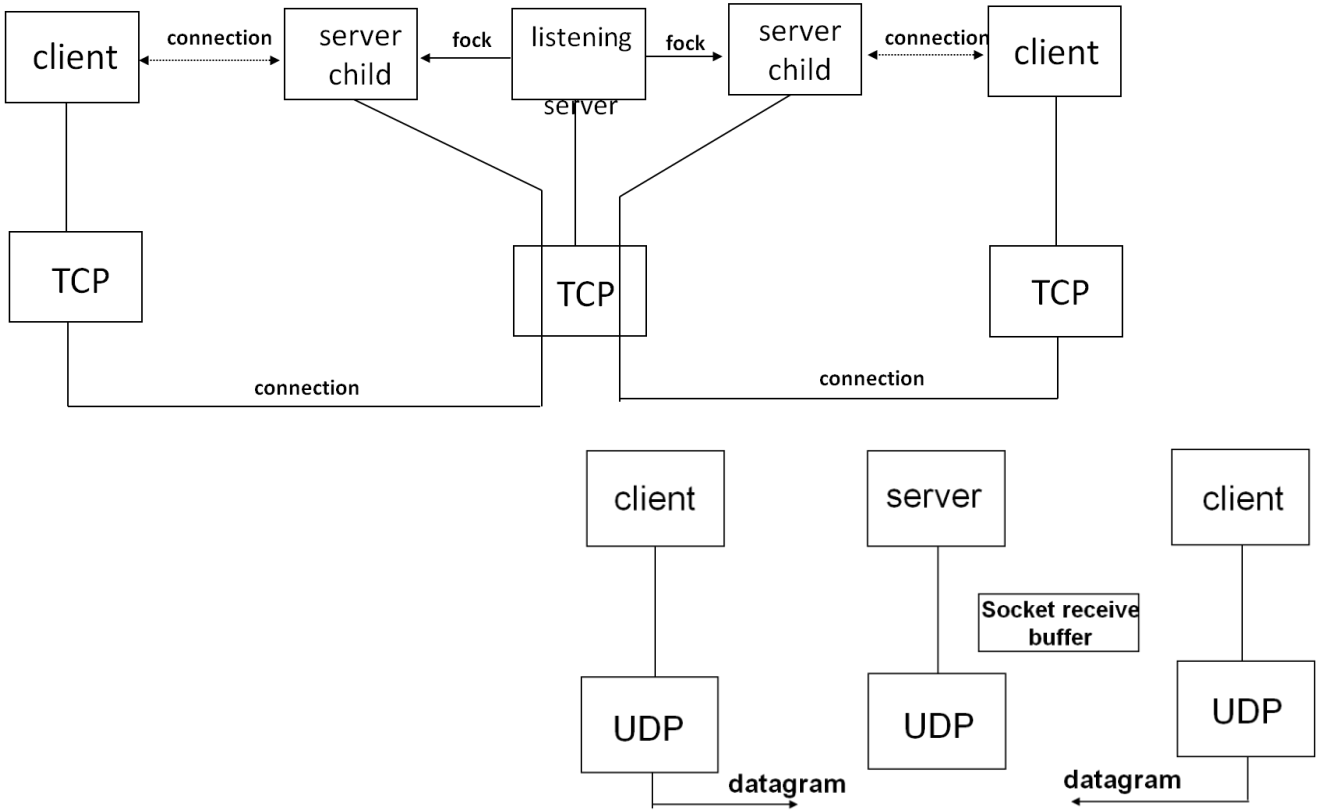
- This function never terminates.
- Since UDP is connectionless protocol, there is nothing like an EOF as in TCP.

UDP Server

innovate achieve lead

- What we discussed is an iterative server.
  - Client requests are handled one after another. There is no call to fork().
  - In general most TCP servers are concurrent, and most UDP servers are iterative.
- UDP socket has a receive buffer. Incoming datagrams are queued there.
  - When server calls recvfrom(), a datagram is returned in FIFO order.
  - Buffer has a limited size (default 40000bytes).

TCP Server vs UDP Server



- In TCP server, there is connected socket for each client and there is separate buffer in each socket.
- In UDP server, datagrams from all clients are placed in a single buffer. No separate socket for each client.

UDP Client

```
1  int main(int argc, char **argv)
2  {
3      int sockfd;
4     struct sockaddr_in servaddr;
5     if (argc != 2)
6       err_quit( “usage : udpcli <Ipaddress>”);
7     bzero(&servaddr, sizeof(servaddr);
8     servaddr.sin_family = AF_INET;
9     servaddr.sin_port = htons(SERV_PORT);
10    inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
11    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
12    dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr);
13    exit(0);
14 }
```

- When client uses *sendto()* at that kernel allocates an ephemeral port to the socket local endpoint.

UDP Client

```
1   void  dg_cli(FILE *fp, int sockfd, const SA *pservaddr, soklen_t servlen)
2   {
3     int n;
4     char sendline[MAXLINE], recvline[MAXLINE+1];
5     while(fgets(sendline, MAXLINE, fp) != NULL)  {
6         sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
7       n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
8         recvline[n] = 0;     /* null terminate */
9         fputs(recvline,stdout);
10    }
11  }
```

- In *recvfrom()* last two arguments are specified as NULL. This tells the kernel that we are not interested in knowing who sent the reply.

Lost Datagrams

- If the clients datagram is lost to the server, server will wait indefinitely on *recvfrom()*.
- If the server's datagram to client is lost, then client will wait indefinitely on *recvfrom()*.
- Solution:
  - Put a timeout . But this will not tell whether client's datagram is lost or server's reply is lost.
    - Use timers
  - Use application level reliability support.
    - Sequence numbers
    - Acknowledgements
    - Timeout & retransmission

UDP Sockets & Asynchronous Errors

- Client *sendto()* returns successfully after writing a datagram to UDP socket send buffer.
- But due to some reason say *"Server Not Running"*, the server host will send ICMP *"port unreachable"* error.
  - This error or "*network unreachable"* or "*host unreachable"* are asynchronous errors.
- The basic rule is that an asynchronous error is not returned for a UDP socket unless the socket has been connected.
- Reason:
  - An unconnected Udp socket can be used to send datagrams to multiple destinations. If error is received, then Udp can return only errno but not which destination.
  - Therefore only connected UDP socket receives asynch errors.

Connected UDP Sockets

- Calling connect() on a datagram socket causes the kernel to record a particular address as this socket's peer.
  - This does not result in anything like a TCP connection: there is no three-way handshake. Instead, the kernel just records the IP address and port number of the peer.
- With a connected UDP socket, three changes:
  - We can no long specify the destination IP address and port for an output operation. That is, we do not use *sendto()* but use write or send instead.
  - We do not use recvfrom but read or recv instead.
  - Asynchronous errors are returned to the process for a connected UDP socket.
- Only datagrams sent by the peer socket may be read on the socket.

Using connect() Multiple Times

- A process with a connected UDP socket can call connect again for that socket for one of two reasons:
  - To specify a new IP address and port
  - To unconnect the socket
- connect can be called only one time for a TCP socket. But in UDP, it can be called multiple times with different peer addresses.
- To unconnect a UDP socket, we call connect but set the family member of the socket address structure (sin_family for IPv4 or sin6_family for IPv6) to AF_UNSPEC.

UDP datagram size & IP fragmentation

- IP fragmentation occurs transparently to higher protocol layers, but nevertheless is generally considered undesirable.
  - IP doesn't perform retransmission
  - A datagram can be reassembled at the destination only if all fragments arrive
  - The entire datagram is unusable if any fragment is lost or contains transmission errors.
- In some cases, this can lead to significant rates of data loss (for UDP) or degraded transfer rates (for TCP).
- Modern TCP implementations employ algorithms (path MTU discovery) to determine the MTU of a path between hosts.

UDP datagram size & IP fragmentation

- A UDP-based application generally doesn't know the MTU of the path between the source and destination hosts.
- UDP-based applications that aim to avoid IP fragmentation typically adopt a conservative approach.
  - The transmitted IP datagram is less than the IPv4 minimum reassembly buffer size of 576 bytes.
  - This value is likely to be lower than the path MTU.
  - 576- 20(IPv4) -8 (UDP header) = 548 bytes.
- In practice, many UDP-based applications opt for a still lower limit of 512 bytes for their datagrams.

innovate achieve lead

BITS Pilani
Pilani Campus

TCP vs UDP?

When to use UDP instead of TCP?

innovate achieve lead

- Advantages of UDP:
  - UDP supports broadcasting and multicasting
  - UDP has no connection setup or teardown
    - For a two packet request-reply, we need 8 extra packets to be transmitted in TCP
    - UDP: RTT+SPT, TCP: 2 *RTT + SPT
- Features of TCP not provided by UDP:
  - Positive acknowledgments, retransmission of lost packets, duplicate detection, and sequencing of packets reordered by the network
    - Seq nos, estimate RTO
  - Windowed flow control
  - Slow start and congestion avoidance
    - to determine the current network capacity and to handle periods of congestion

When to use UDP instead of TCP?

innovate achieve lead

- Recommendations:
  - UDP must be used for broadcast and multicast applications
    - Error control or reliability be added if reqd at appl layer
  - UDP can be used for simple request-reply applications, but error detection must be built into the application
    - Acknowledgements, timeouts, retransmissions
  - UDP should not be used for bulk data transfer
    - Bulk transfer requires flow control along with error control which is like replicating TCP at appl layer
- Certain types of applications (e.g., streaming video and audio transmission) can function acceptably without the reliability provided by TCP.
  - Delays by TCP retires may be worse than losing a part of the stream.
  - Use UDP and adopt application-specific recovery strategies due to occasional packet loss.

innovate achieve lead

BITS Pilani
Pilani Campus

Name and Address Conversions (T1: 11, R1: 59.8)

DNS

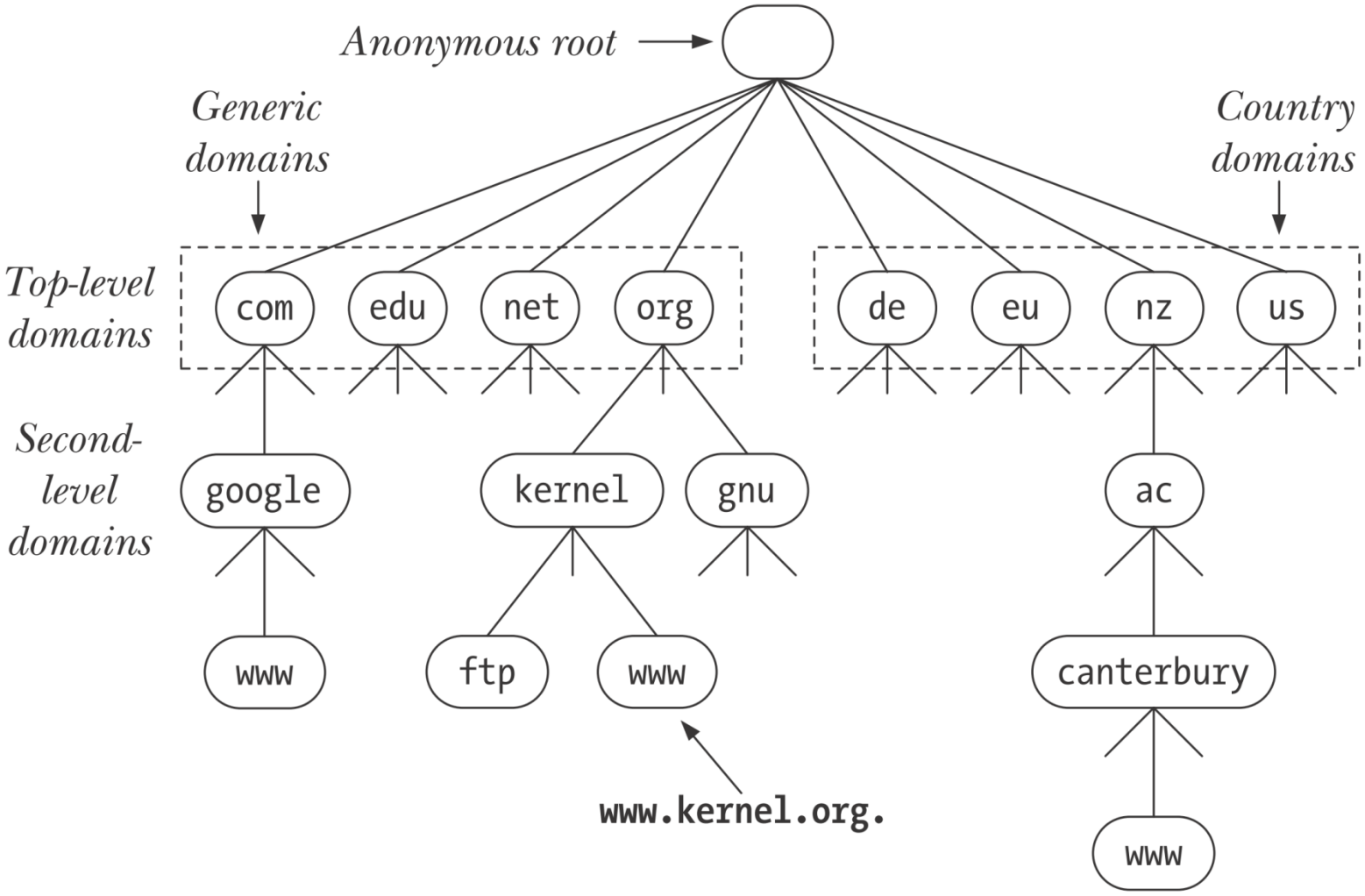
- Two functions which we use to convert a domain name to IP address:
  - gethostbyname() - obsolete
  - getaddrinfo() - supports both IPv6 and IPv4
- The DNS is used primarily to map between hostnames and IP addresses.
  - A hostname can be either a simple name, such as *solaris* or *freebsd*, or a fully qualified domain name (FQDN), such as *solaris.unpbook.com.*
- Before DNS, mappings between hostnames and IP addresses were defined in a manually maintained local file, /etc/hosts

```
1  # IP-address     canonical hostname      [aliases]
2  127.0.0.1       localhost
```

DNS

- The /etc/hosts scheme scales poorly.
- DNS was devised to address this problem.
  - Hostnames are organized into a hierarchical namespace.
    - Node in the DNS hierarchy has a label (name), which may be up to 63 characters.
    - At the root of the hierarchy is an unnamed node, the “anonymous root.”
  - A node’s domain name consists of all of the names from that node up to the root concatenated together, with each name separated by a period (.)
  - No single organization or system manages the entire hierarchy.
    - Instead, there is a hierarchy of DNS servers, each of which manages a branch (a zone) of the tree.
    - For adding a host, admin has to add it to local name server.
  - DNS servers employ caching techniques to avoid unnecessary communication for frequently queried domain names.

DNS


Anonymous root
Generic domains
Country domains
Top-level domains
com
edu
net
org
de
eu
nz
us
Second-level domains
google
kernel
gnu
ac
www
ftp
www
canterbury
www.kernel.org.
www

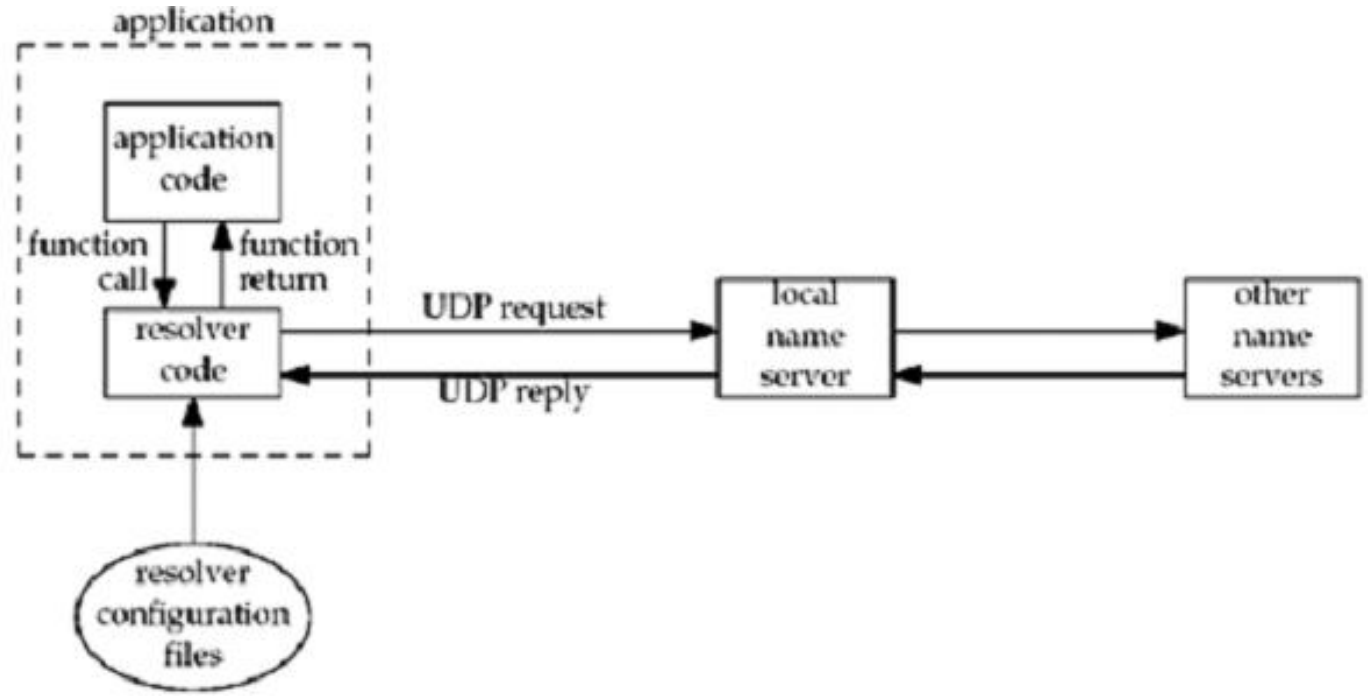
DNS Lookup

- Not every name server knows about every other name server.
- Name server must know the IP address of root servers.
- Root servers know the name and location for all second-level domains.

Recursive and Iterative Lookups

- DNS resolution requests fall into two categories: recursive and iterative.
  - In a recursive request, the requester asks the server to handle the entire task of resolution.
- When an application on the local host calls *getaddrinfo*(), that function makes a recursive request to the local DNS server.
- If the local DNS server does not itself have the information to perform the resolution, it resolves the domain name iteratively.

Resolvers and Name Servers



Resolver is part of the application

The /etc/services File

- Well-known port numbers are centrally registered by IANA.
  - Each of these ports has a corresponding service name.
    - Because service numbers are centrally managed and are less volatile than IP addresses, an equivalent of the DNS server is usually not necessary. Instead, the port numbers and service names are recorded in the file /etc/services .
  - The getaddrinfo() and getnameinfo() functions use the information in this file to convert service names to port numbers and vice versa.

The /etc/services File


```
1   # Service name  port/protocol  [aliases]
2   echo            7/tcp          Echo     # echo service
3   echo            7/udp          Echo
4   ssh             22/tcp                  # Secure Shell
5   ssh             22/udp
6   telnet          23/tcp                  # Telnet
7   telnet          23/udp
8   smtp            25/tcp                  # Simple Mail Transfer Protocol
9   smtp            25/udp
10  domain          53/tcp                  # Domain Name Server
11  domain          53/udp
12  http            80/tcp                  # Hypertext Transfer Protocol
13  http            80/udp
14  ntp             123/tcp                 # Network Time Protocol
15  ntp             123/udp
16  login           513/tcp                 # rlogin(1)
17  who             513/udp                 # rwho(1)
18  shell           514/tcp                 # rsh(1)
19  syslog          514/udp                 # syslog
```

Host and Service Conversion

- The getaddrinfo() function converts host and service names to IP addresses and port numbers.
  - successor to the obsolete gethostbyname() and getservbyname() functions
- Given a host name and a service name, getaddrinfo() returns a list of socket address structures, each of which contains an IP address and port number.

```
1  #include <sys/socket.h>
2  #include <netdb.h>
3  int getaddrinfo(const char * host , const char * service ,
4          const struct addrinfo * hints , struct addrinfo ** result );
5  //Returns 0 on success, or nonzero on error
```

The getaddrinfo() Function

- As input, getaddrinfo() takes the arguments host, service, and hints.
  - Host:
    - It can be hostname or numeric address string 172.24.2.19
  - Service:
    - This contains either service name or port number.
  - Hints:
    - The hints argument points to an addrinfo structure that specifies further criteria for selecting the socket address structures returned via result.

```
1  #include <sys/socket.h>
2  #include <netdb.h>
3  int getaddrinfo(const char * host , const char * service ,
4          const struct addrinfo * hints , struct addrinfo ** result );
5  //Returns 0 on success, or nonzero on error
```

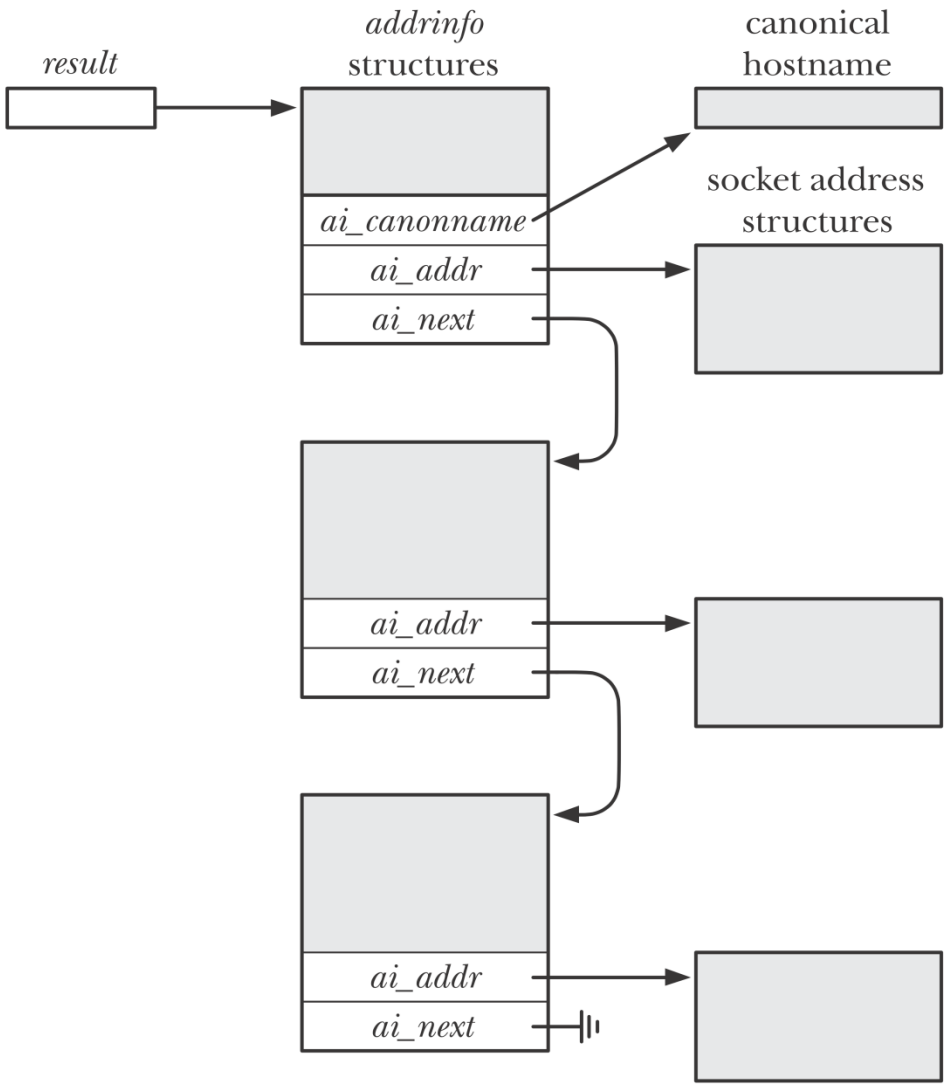
Addrinfo structure

innovate achieve lead

- As output, getaddrinfo() dynamically allocates a linked list of addrinfo structures and sets result pointing to the beginning of this list.
- Each of these addrinfo structures includes a pointer to a socket address structure corresponding to host and service.

```
1  struct addrinfo {
2      int     ai_flags;          /* Input flags (AI_* constants) */
3      int     ai_family;         /* Address family */
4      int     ai_socktype;       /* Type: SOCK_STREAM, SOCK_DGRAM */
5      int     ai_protocol;       /* Socket protocol */
6      size_t ai_addrlen;         /* Size of structure pointed to by ai_addr */
7      char  *ai_canonname;       /* Canonical name of host */
8      struct sockaddr *ai_addr;  /* Pointer to socket address structure */
9      struct addrinfo *ai_next;  /* Next structure in linked list */
10 };
```

Result


result
addrinfo structures
canonical hostname
socket address structures
ai_canonname
ai_addr
ai_next
ai_addr
ai_next
ai_addr
ai_next

The *hints* Argument

- Hints is either a null pointer or a pointer to an addrinfo structure.
  - the caller fills in this structure with hints about the types of information the caller wants returned.
- The members of the hints structure that can be set by the caller are:
  - ai_flags (zero or more AI_XXX values OR'ed together)
  - ai_family (an AF_xxx value)
  - ai_socktype (a SOCK_xxx value)
  - ai_protocol
- For example,
  - if the specified service is provided for both TCP and UDP, set ai_socktype member of the hints structure to SOCK_DGRAM. Then only information returned will be for datagram sockets.

The *hints* Argument

- AI_PASSIVE
  - The caller will use the socket for a passive open.
- AI_CANONNAME
  - Tells the function to return the canonical name of the host.
- AI_NUMERICHOST
  - Prevents any kind of name-to-address mapping; the hostname argument must be an address string.
- AI_NUMERICSERV
  - Prevents any kind of name-to-service mapping; the service argument must be a decimal port number string.
- AI_V4MAPPED
  - If specified along with an ai_family of AF_INET6, then returns IPv4-mapped IPv6 addresses corresponding to A records if there are no available AAAA records.

The *hints* Argument

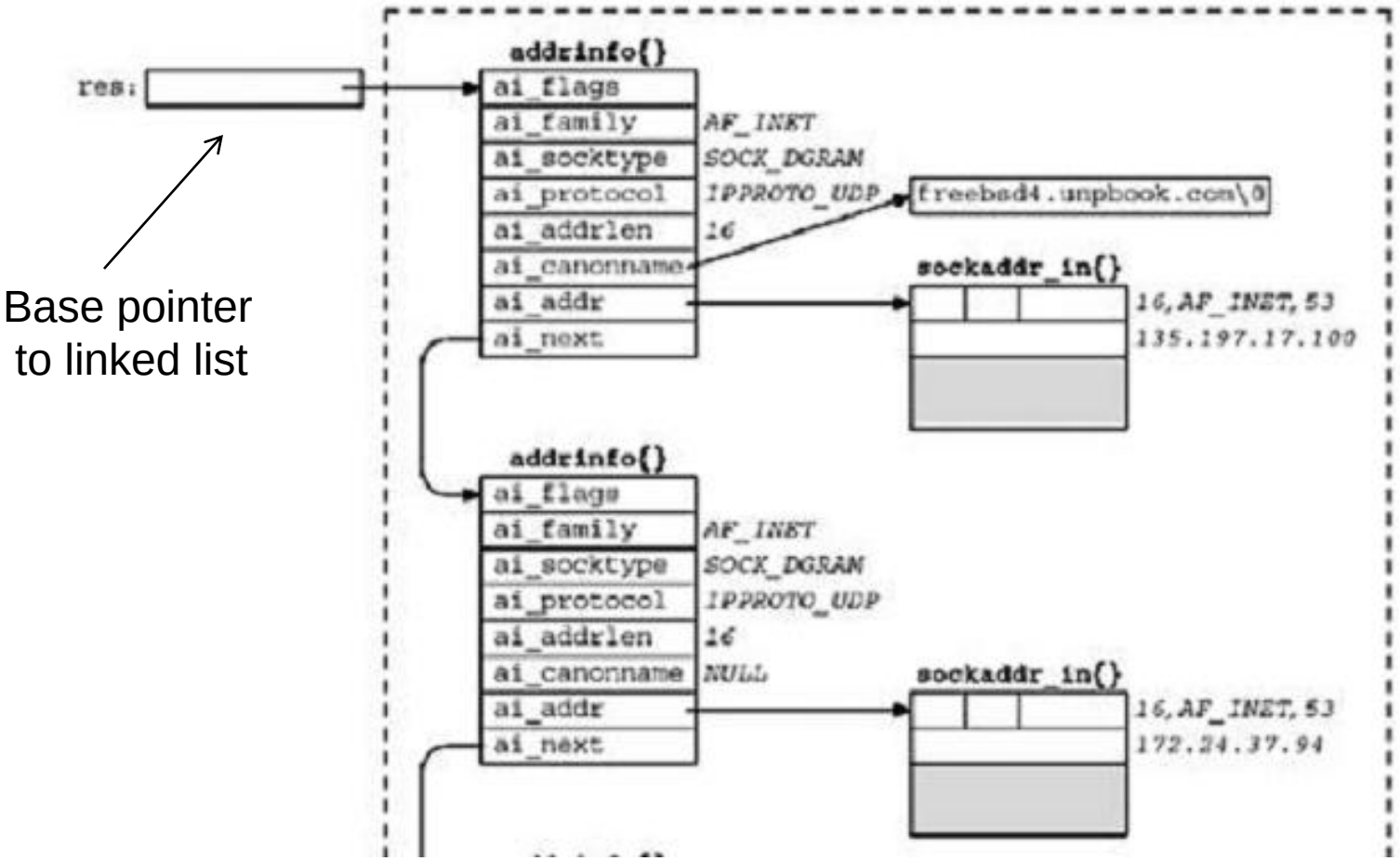
- Returns linked list of addrinfo structures, linked through the ai_next pointer.
- There are two ways that multiple structures can be returned:
  - Multiple ips per hostname; one sockaddr structure for each ip
  - Service is provided for multiple socket types; SOCK_STREAM or SOCK_DGRAM
- For example, if no hints are provided and if the domain service is looked up for a host with two IP addresses, four addrinfo structures are returned:
    - One for the first IP address and a socket type of SOCK_STREAM
    - One for the first IP address and a socket type of SOCK_DGRAM
    - One for the second IP address and a socket type of SOCK_STREAM
    - One for the second IP address and a socket type of SOCK_DGRAM

Example

- Consider the following code

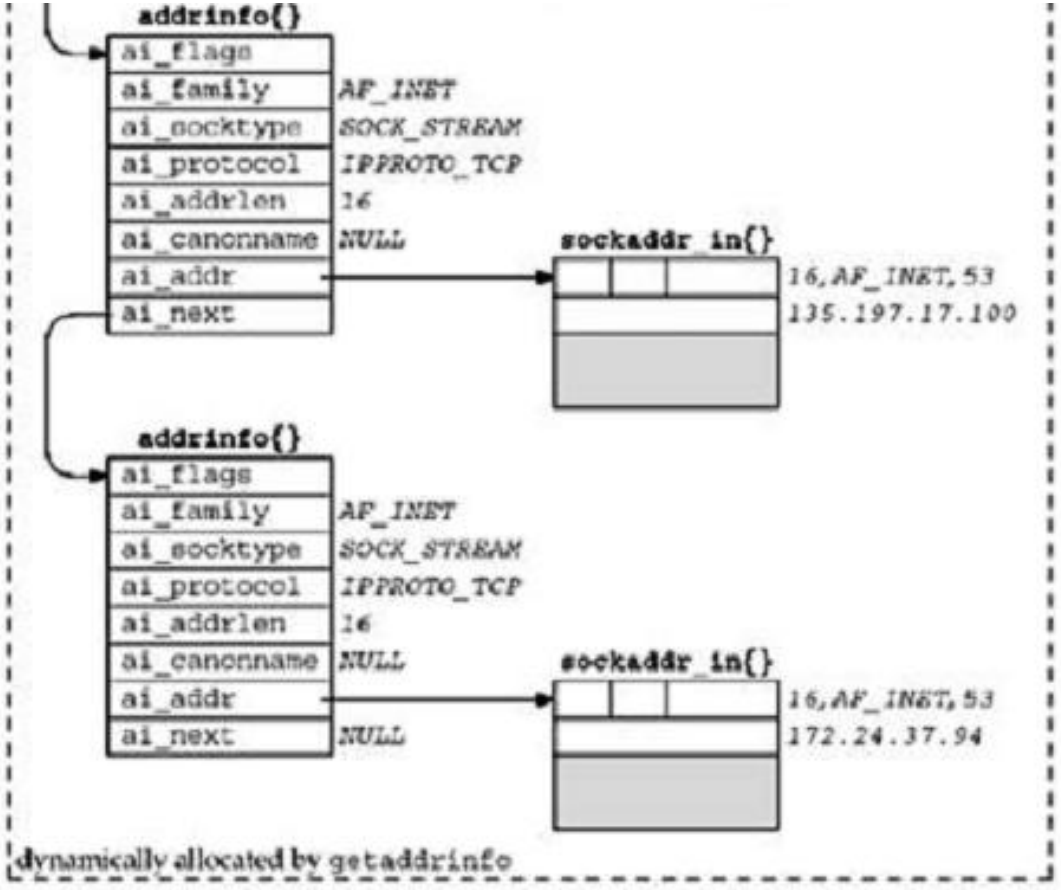
```
1  struct addrinfo hints, *res;
2  bzero(&hints, sizeof(hints) ) ;
3  hints.ai_flags = AI_CANONNAME;
4  hints.ai_family = AF_INET;
5  getaddrinfo("freebsd4", "domain", &hints, &res);
```

Result



Base pointer to linked list

Result


addrinfo{}
ai_flags
ai_family
ai_socktype
ai_protocol
ai_addrlen
ai_canonname
ai_addr
ai_next
AF_INET
SOCK_STREAM
IPPROTO_TCP
16
NULL
sockaddr_in{}
16, AF_INET, 53
135.197.17.100
addrinfo{}
ai_flags
ai_family
ai_socktype
ai_protocol
ai_addrlen
ai_canonname
ai_addr
ai_next
AF_INET
SOCK_STREAM
IPPROTO_TCP
16
NULL
NULL
sockaddr_in{}
16, AF_INET, 53
172.24.37.94
dynamically allocated by getaddrinfo

Usage

- Sockaddr structure in addrinfo structures is ready for
  - a call to socket
  - then either a call to connect or sendto (for a client), or bind (for a server).
- The arguments to socket are the members ai_family, ai_socktype, and ai_protocol.
- The second and third arguments to either connect or bind are ai_addr, and ai_addrlen

On client side

innovate achieve lead

```
1    int tcp_connect (const char *host, const char *serv)
2    {
3      int sockfd, n;
4      struct addrinfo hints, *res, *ressave;
5     bzero(&hints, sizeof (struct addrinfo));
6     hints.ai_family = AF_UNSPEC;
7     hints.ai_socktype = SOCK_STREAM;
8     if ( (n = getaddrinfo (host, serv, &hints, &res)) != 0)
9         err_quit("tcp_connect error for %s, %s: %s",
10                host, serv, gai_strerror (n));
11    ressave = res;
12    do {
13        sockfd = socket (res->ai_family, res->ai_socktype, res->ai_protocol);
14        if (sockfd < 0)
15            continue;              /*ignore this one */
16        if (connect (sockfd, res->ai_addr, res->ai_addrlen) == 0)
17            break;                 /* success */
18        Close(sockfd);            /* ignore this one */
19   } while ( (res = res->ai_next) != NULL);
20    if (res == NULL)               /* errno set from final connect() */
21    err_sys ("tcp_connect error for %s, %s", host, serv);
22    freeaddrinfo (ressave);
23    return (sockfd);
24   }
```

Server side usage

innovate achieve lead

```
int tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
  {  struct addrinfo hints, *res, *ressave;
     bzero(&hints, sizeof (struct addrinfo)) ;
     hints.ai_flags = AI_PASSIVE;
     hints.ai_family = AF_UNSPEC;
     hints.ai_socktype = SOCK_STREAM;
     if ( (n = getaddrinfo (host, serv, &hints, &res)) != 0)
         err_quit("tcp_listen error for %s, %s: %s",
                  host, serv, gai_strerror(n)) ;
     ressave = res;
 do {listenfd =socket(res->ai_family, res->ai_socktype, res->ai_protocol);
         if (listenfd < 0)continue;            /* error, try next one */
         setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof (on) ) ;
         if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
             break;                /* success */
         close (listenfd);         /* bind error, close and try next one */
      } while ( (res = res->ai_next) != NULL);
      if (res == NULL)             /* errno from final socket () or bind () */
          err_sys ("tcp_listen error for %s, %s", host, serv);
      listen (listenfd, LISTENQ);
      if (addrlenp)
          *addrlenp = res->ai_addrlen;     /* return size of protocol address */
      freeaddrinfo (ressave);
      return (listenfd);
}
```

```
listenfd = tcp_listen (NULL, argv[1], NULL);
```

Acknowledgements

Q&A

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You