BITS Pilani
Pilani Campus

# Network Programming

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

BITS Pilani
Pilani Campus

# Outline

# Outline

- Name conversion
  - getaddrinfo()
  - getnameinfo()
  - gethostbyname()
- Advanced I/O
  - recv(), send()
  - readv(), writev()
  - recvmsg(), sendmsg()
- Framing & Encoding
  - HTTP

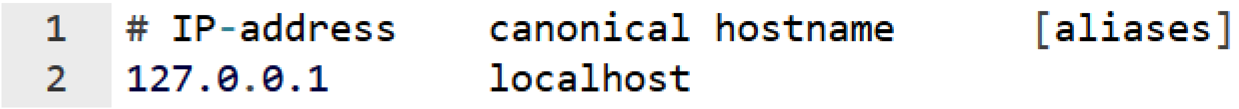
innovate achieve lead

BITS Pilani
Pilani Campus

# Name and Address Conversions (T1: ch11)

# 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

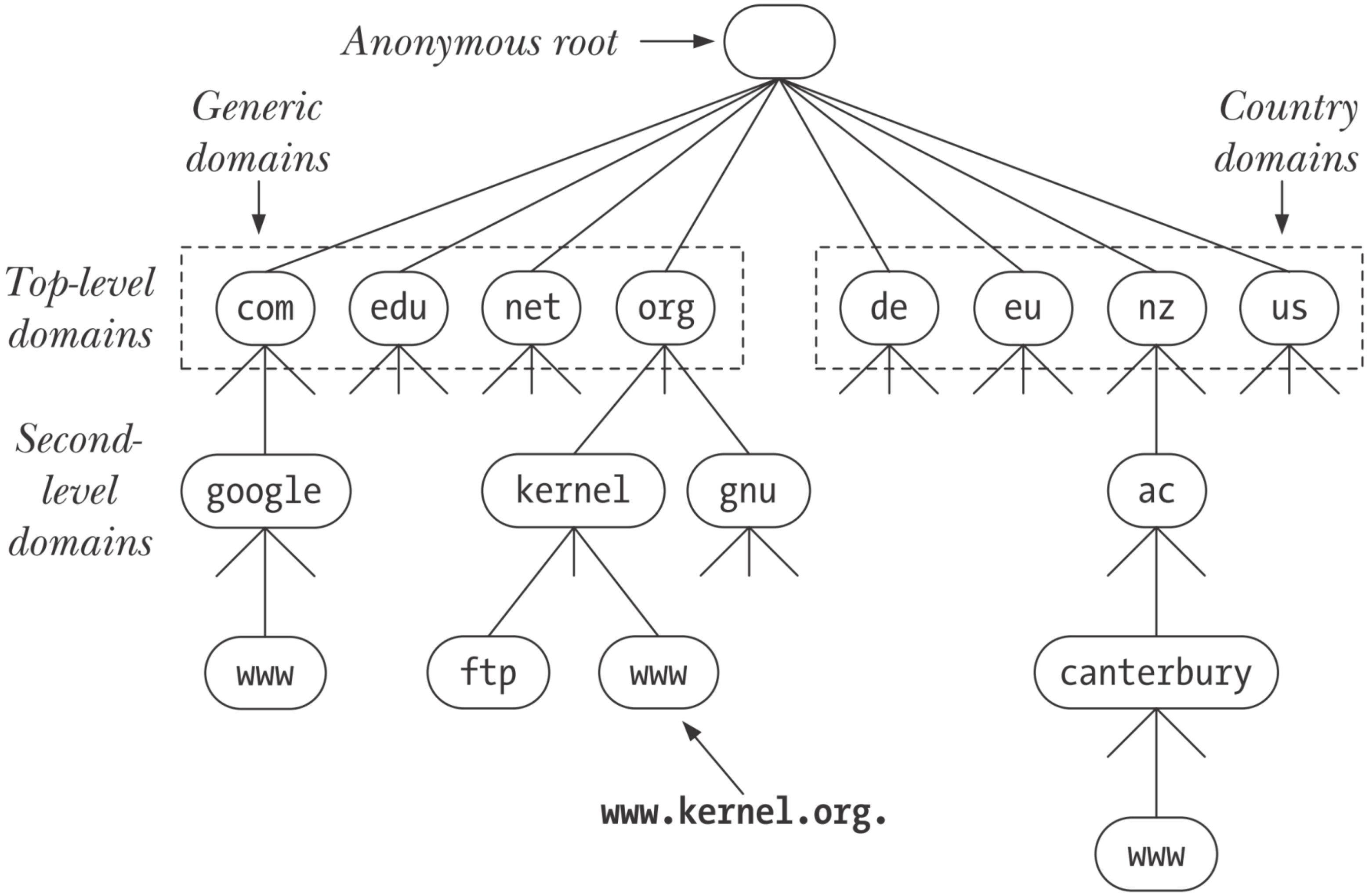


```
# IP-address    canonical hostname      [aliases]
127.0.0.1       localhost
```

# 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
www.kernel.org.

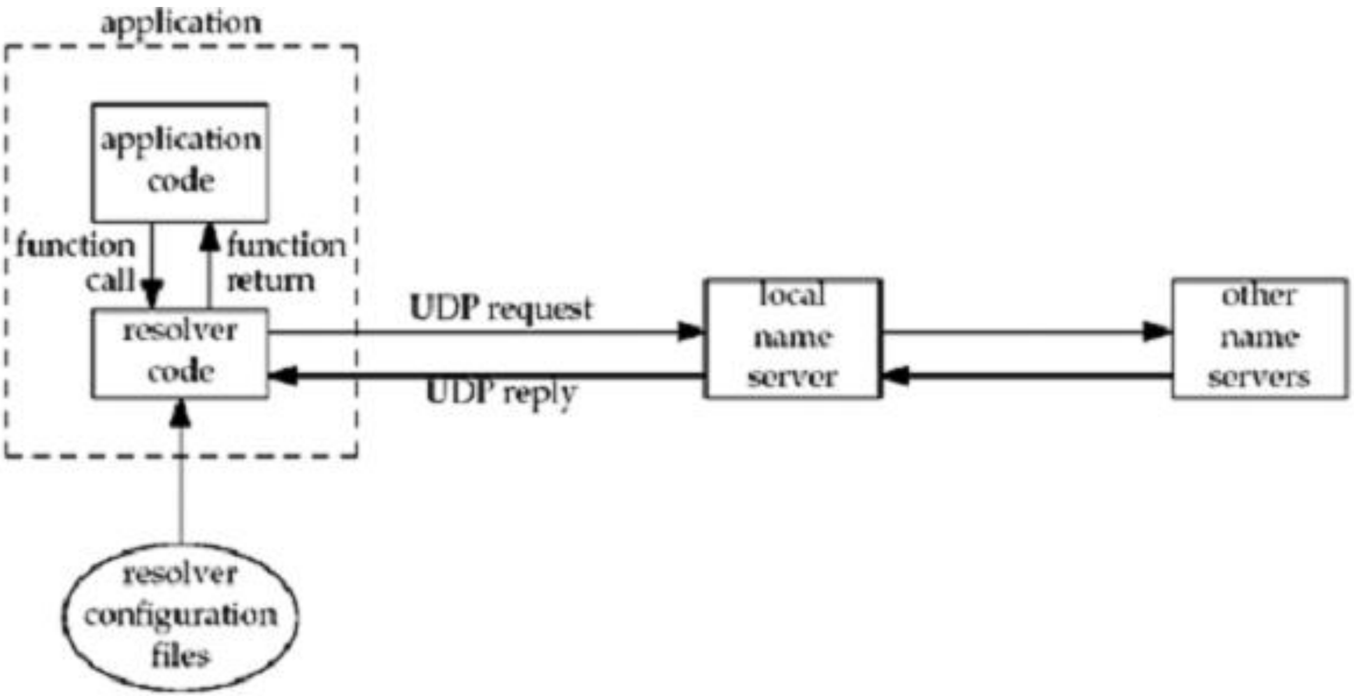
# DNS Lookup

- 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

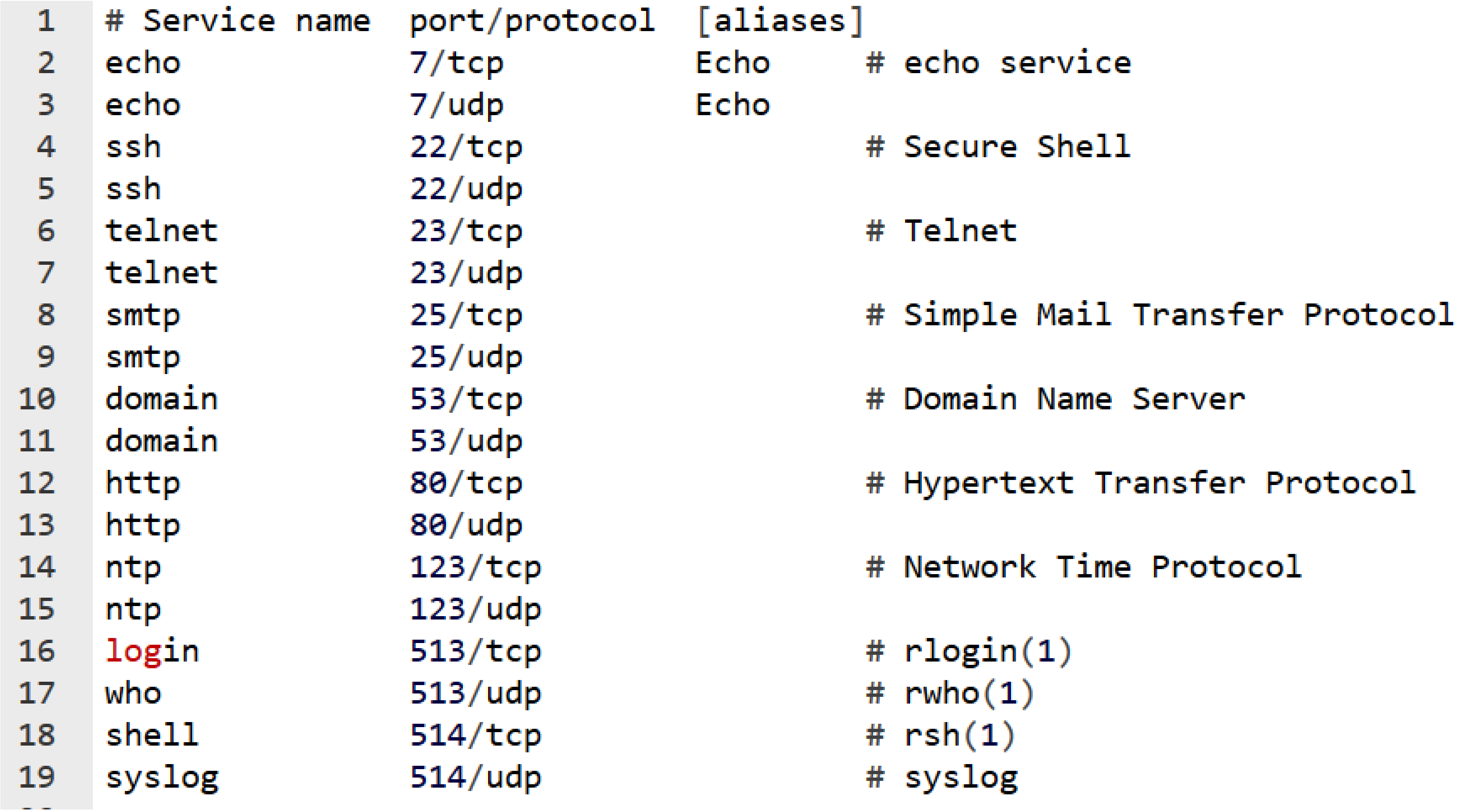


Resolver is part of the application

# 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.

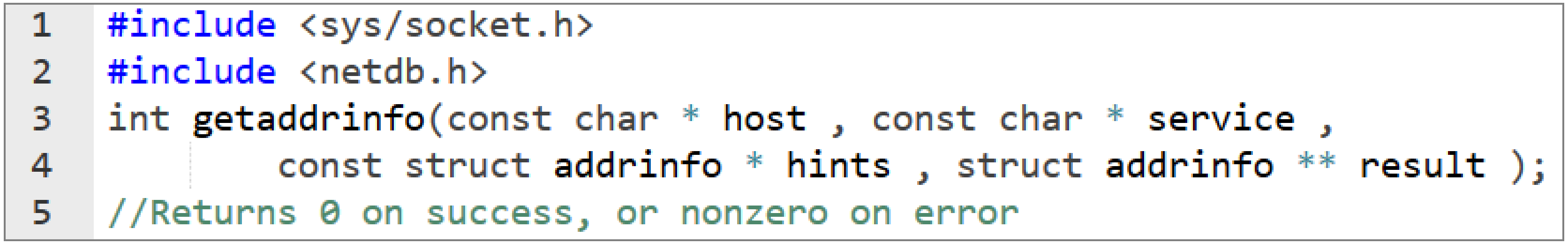
# The /etc/services File



```
# Service name  port/protocol  [aliases]
echo            7/tcp          Echo      # echo service
echo            7/udp          Echo
ssh             22/tcp                   # Secure Shell
ssh             22/udp
telnet          23/tcp                   # Telnet
telnet          23/udp
smtp            25/tcp                   # Simple Mail Transfer Protocol
smtp            25/udp
domain          53/tcp                   # Domain Name Server
domain          53/udp
http            80/tcp                   # Hypertext Transfer Protocol
http            80/udp
ntp             123/tcp                  # Network Time Protocol
ntp             123/udp
login           513/tcp                  # rlogin(1)
who             513/udp                  # rwho(1)
shell           514/tcp                  # rsh(1)
syslog          514/udp                  # syslog
```

# Host and Service Conversion

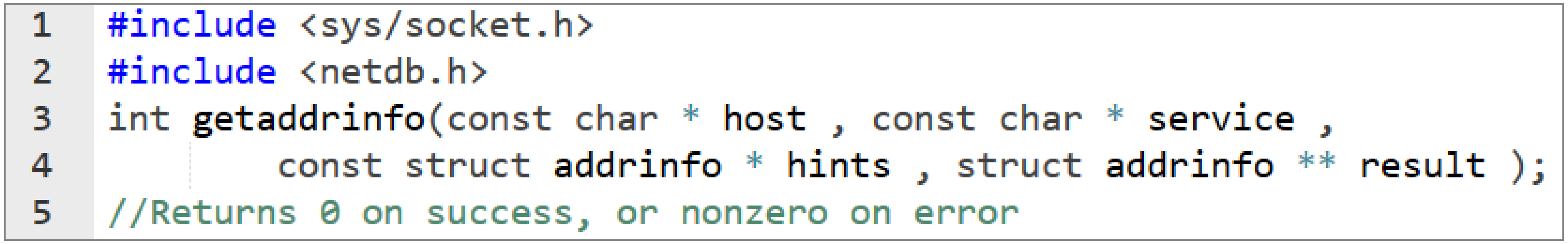
- The getaddrinfo() function converts host and service names to IP addresses and port numbers.
    - successor to the obsolete gethostbyname() and getservbyname() functions
- Given a host name and a service name, getaddrinfo() returns a list of socket address structures, each of which contains an IP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



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

# The getaddrinfo() Function

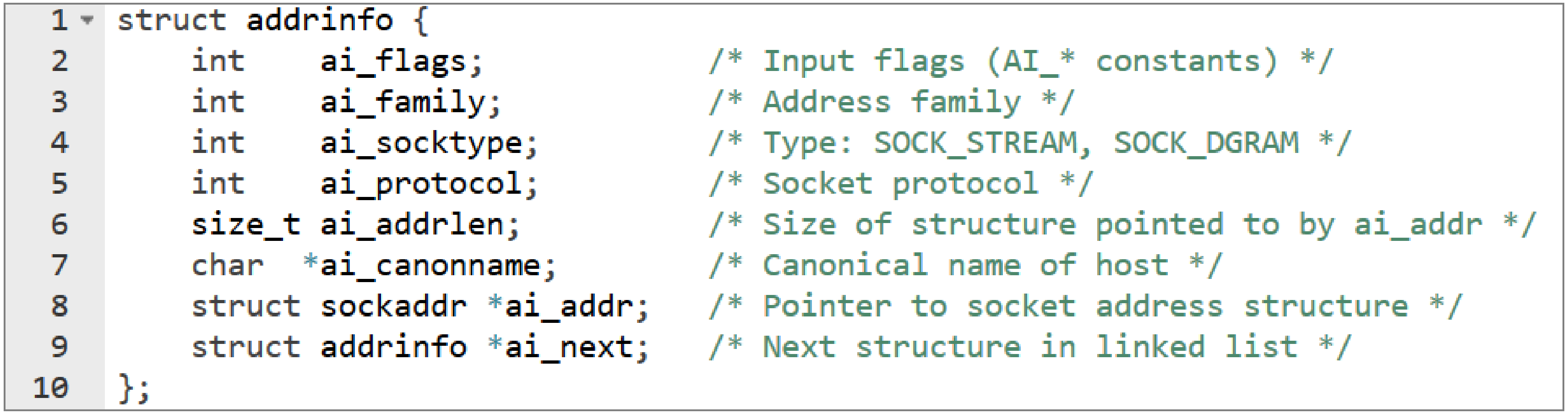
- As input, getaddrinfo() takes the arguments host, service, and hints.
  - Host:
    - It can be hostname or numeric address string 172.24.2.19
  - Service:
    - This contains either service name or port number.
  - Hints:
    - The hints argument points to an addrinfo structure that specifies further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



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

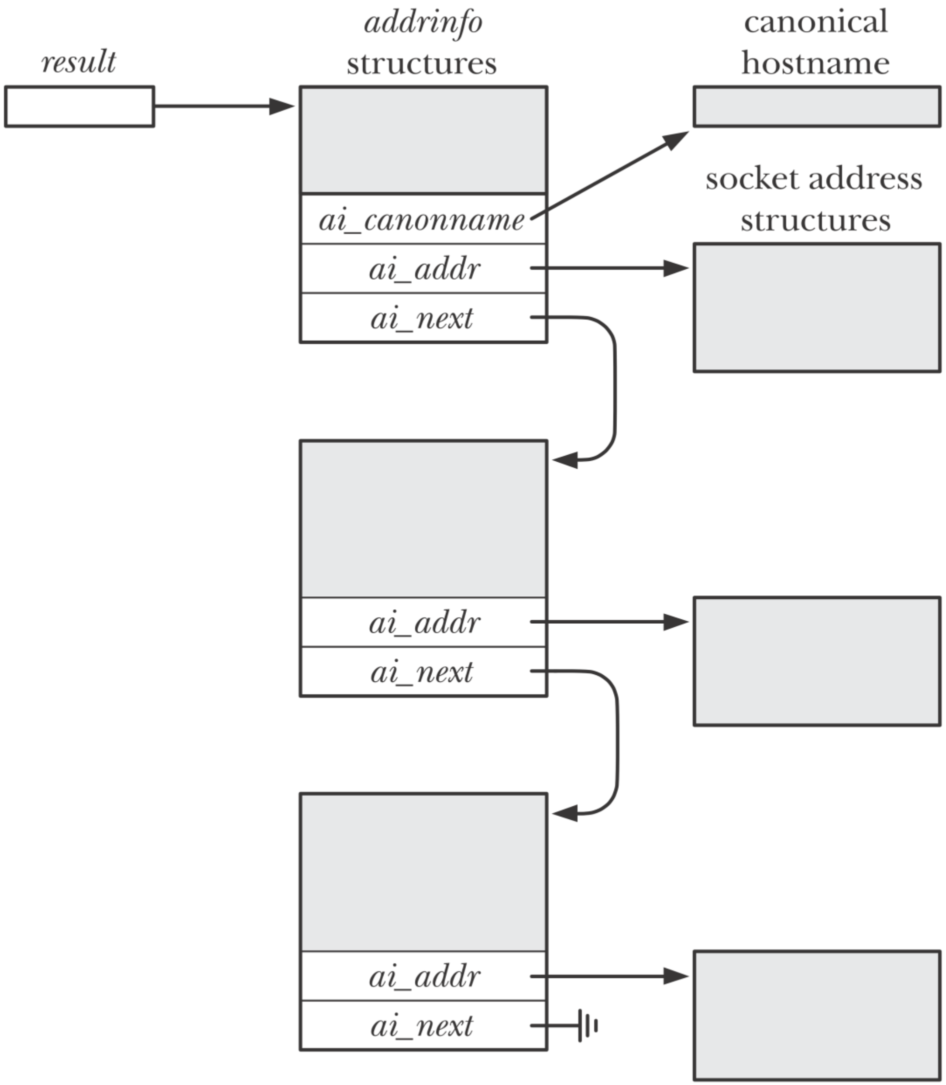
# Addrinfo structure

- 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.



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

# 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

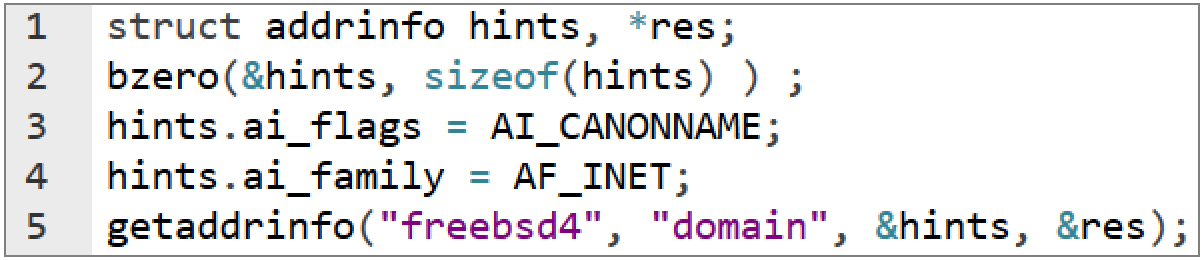
- AI_PASSIVE
  - Returnsocket address structures suitable for a passive open.
  - If *host* is null, then IP will be INADDR_ANY or IN6ADDR_ANY_INIT.
- AI_CANONNAME
  - Tells the function to return the canonical name of the host.
- AI_NUMERICHOST
  - the hostname argument must be a numeric address string.
  - Prevents name resolution.
- AI_NUMERICSERV
  - the service argument must be a decimal port number string.
  - Prevents any kind of name-to-service resolution;
- 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

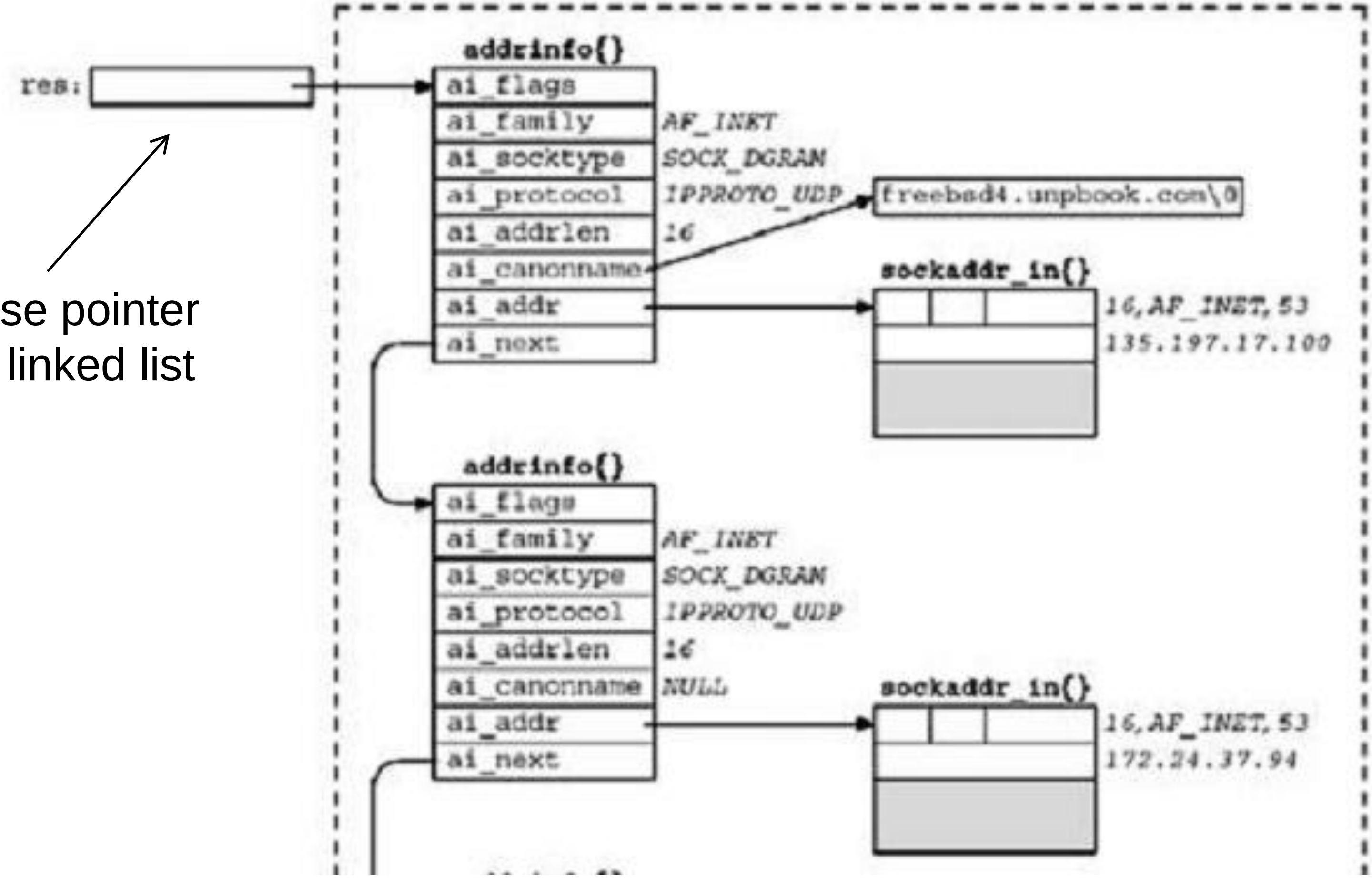
# Example

- Consider the following code



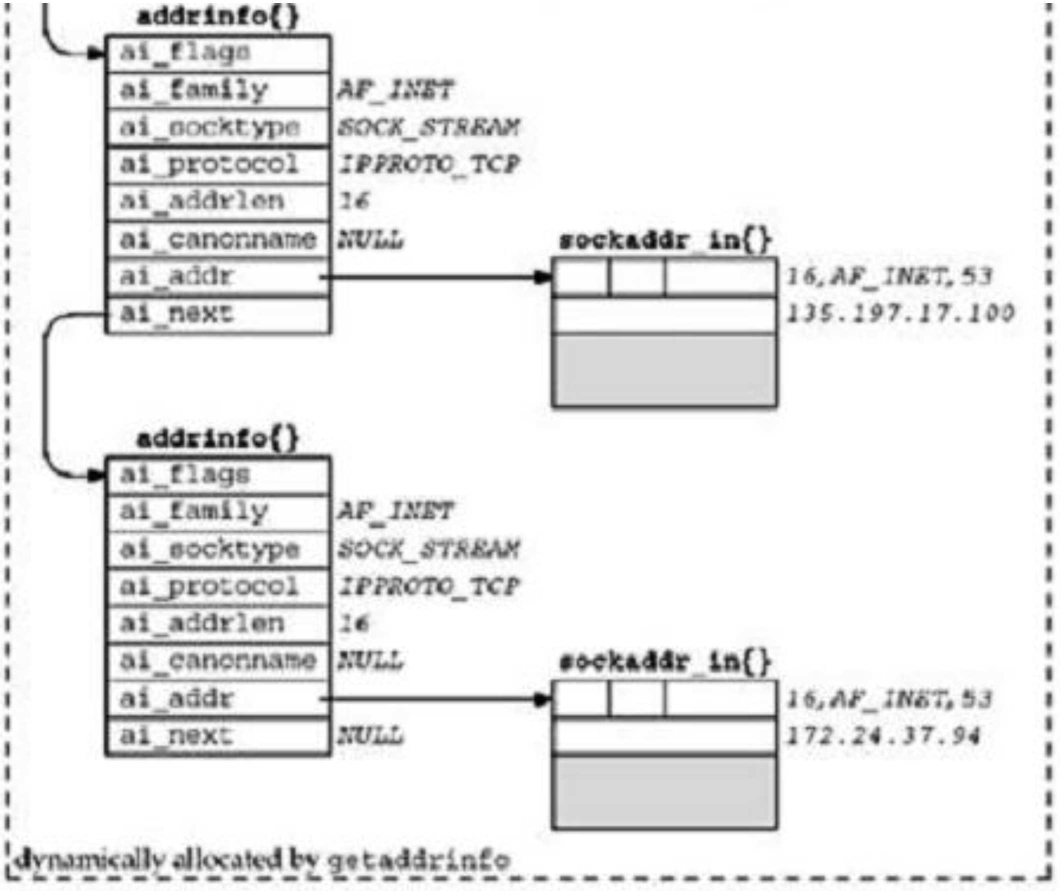
```
struct addrinfo hints, *res;
bzero(&hints, sizeof(hints) ) ;
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_INET;
getaddrinfo("freebsd4", "domain", &hints, &res);
```

# Result



Base pointer
to linked list

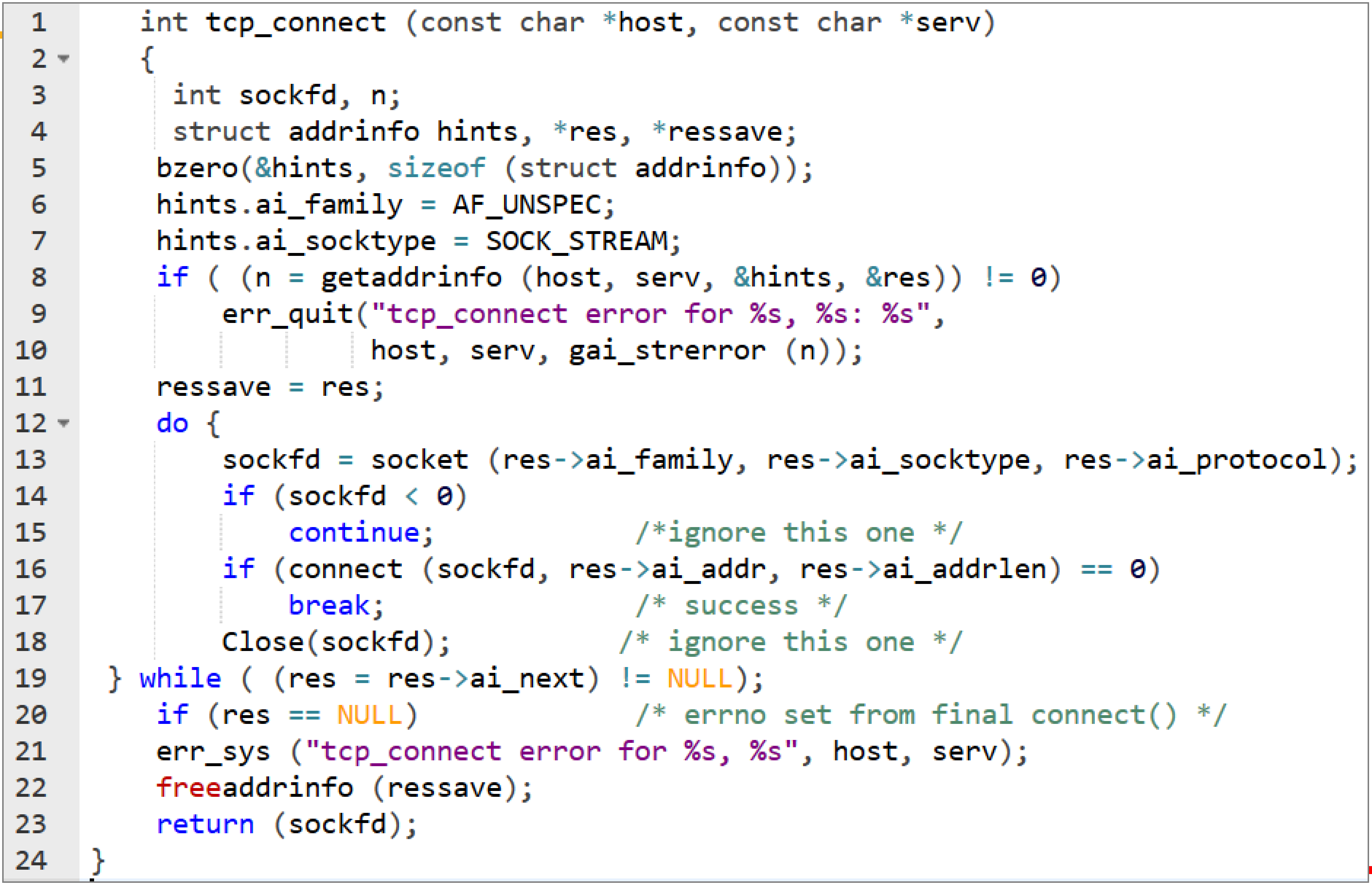
# Result


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

# Usage

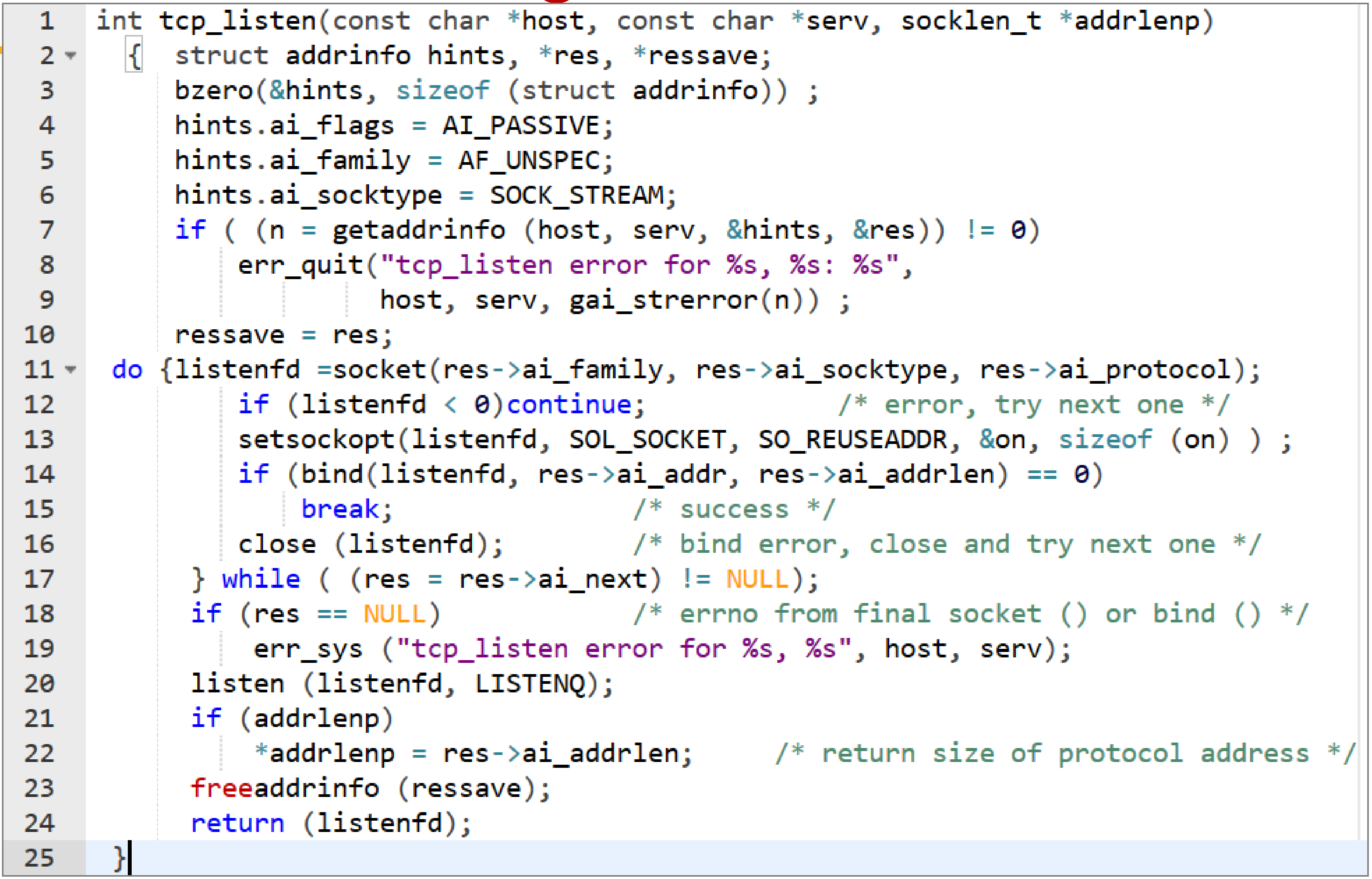
- Sockaddr structure in addrinfo structures is ready for
  - a call to socket
  - then either a call to connect or sendto (for a client), or bind (for a server).
- The arguments to socket are the members *ai_family, ai_socktype, and ai_protocol.*
- The second and third arguments to either connect or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

# On client side

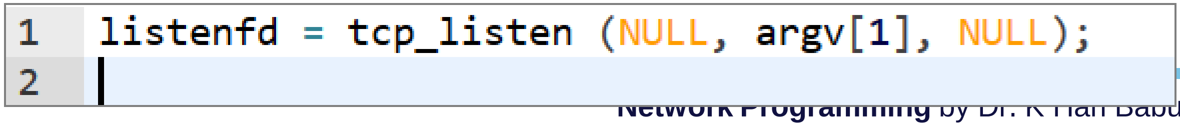


```
  int tcp_connect (const char *host, const char *serv)
  {
   int sockfd, n;
   struct addrinfo hints, *res, *ressave;
  bzero(&hints, sizeof (struct addrinfo));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  if ( (n = getaddrinfo (host, serv, &hints, &res)) != 0)
      err_quit("tcp_connect error for %s, %s: %s",
              host, serv, gai_strerror (n));
  ressave = res;
  do {
      sockfd = socket (res->ai_family, res->ai_socktype, res->ai_protocol);
      if (sockfd < 0)
          continue;            /*ignore this one */
      if (connect (sockfd, res->ai_addr, res->ai_addrlen) == 0)
          break;               /* success */
      Close(sockfd);          /* ignore this one */
 } while ( (res = res->ai_next) != NULL);
  if (res == NULL)             /* errno set from final connect() */
  err_sys ("tcp_connect error for %s, %s", host, serv);
  freeaddrinfo (ressave);
  return (sockfd);
}
```

# Server side usage



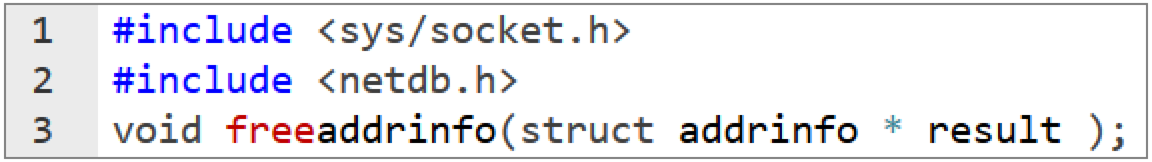
```
int tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
  {  struct addrinfo hints, *res, *ressave;
     bzero(&hints, sizeof (struct addrinfo)) ;
     hints.ai_flags = AI_PASSIVE;
     hints.ai_family = AF_UNSPEC;
     hints.ai_socktype = SOCK_STREAM;
     if ( (n = getaddrinfo (host, 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);
```

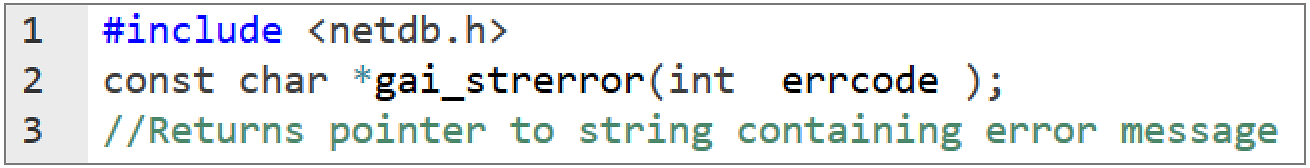
# Freeing *addrinfo* Lists

- The *getaddrinfo()* function dynamically allocates memory for all of the structures referred to by *result*.



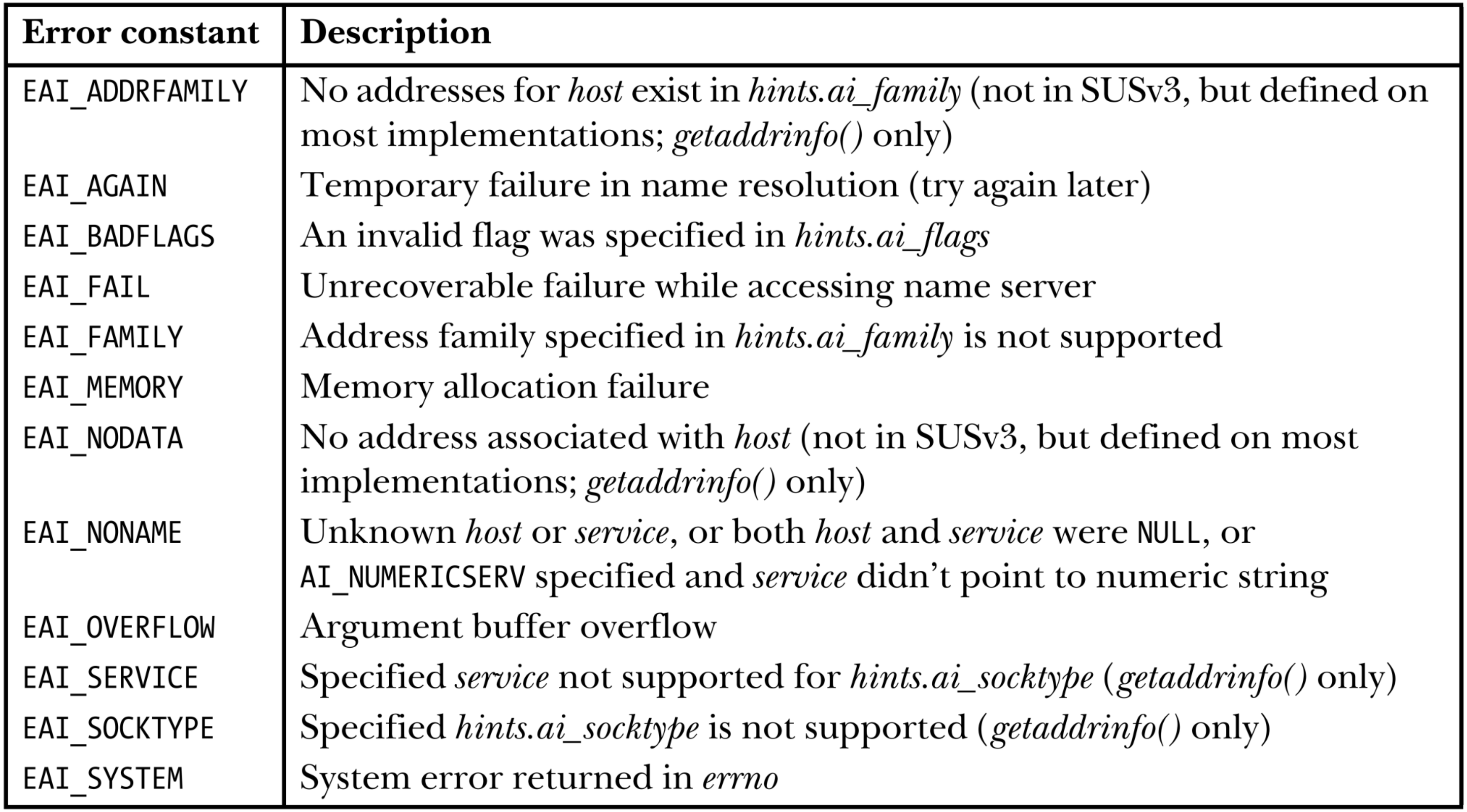
```
#include <sys/socket.h>
#include <netdb.h>
void freeaddrinfo(struct addrinfo * result );
```

- Diagnosing errors
  - ○ Errors returned by getaddrinfo() are not  stored in errno.
  - ○ They have to be looked up using



```
#include <netdb.h>
const char *gai_strerror(int  errcode );
//Returns pointer to string containing error message
```

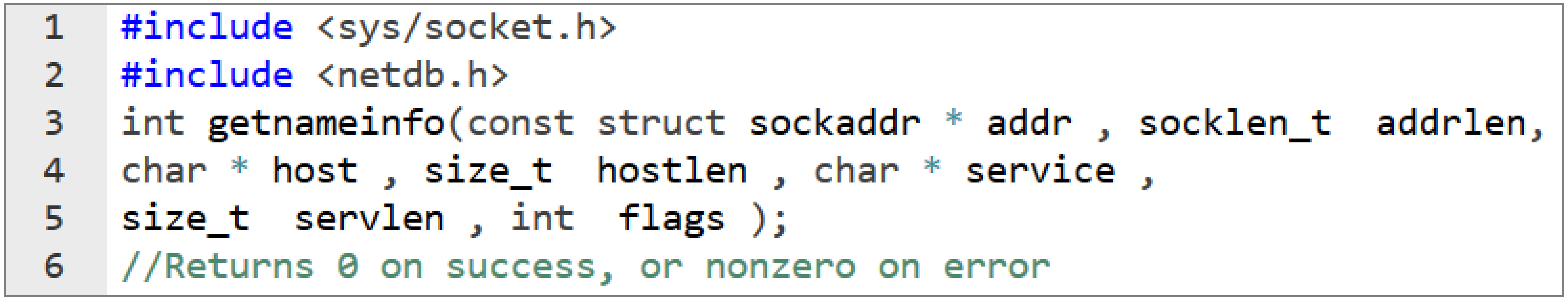
# Error Codes



| Error constant | Description |
|---|---|
| EAI_ADDRFAMILY | No addresses for *host* exist in *hints.ai_family* (not in SUSv3, but defined on most implementations; *getaddrinfo()* only) |
| EAI_AGAIN | Temporary failure in name resolution (try again later) |
| EAI_BADFLAGS | An invalid flag was specified in *hints.ai_flags* |
| EAI_FAIL | Unrecoverable failure while accessing name server |
| EAI_FAMILY | Address family specified in *hints.ai_family* is not supported |
| EAI_MEMORY | Memory allocation failure |
| EAI_NODATA | No address associated with *host* (not in SUSv3, but defined on most implementations; *getaddrinfo()* only) |
| EAI_NONAME | Unknown *host* or *service*, or both *host* and *service* were NULL, or AI_NUMERICSERV specified and *service* didn't point to numeric string |
| EAI_OVERFLOW | Argument buffer overflow |
| EAI_SERVICE | Specified *service* not supported for *hints.ai_socktype* (*getaddrinfo()* only) |
| EAI_SOCKTYPE | Specified *hints.ai_socktype* is not supported (*getaddrinfo()* only) |
| EAI_SYSTEM | System error returned in *errno* |

# getnameinfo()

- This function is the converse of *getaddrinfo*(). Takes socket address structure and returns host, and service.

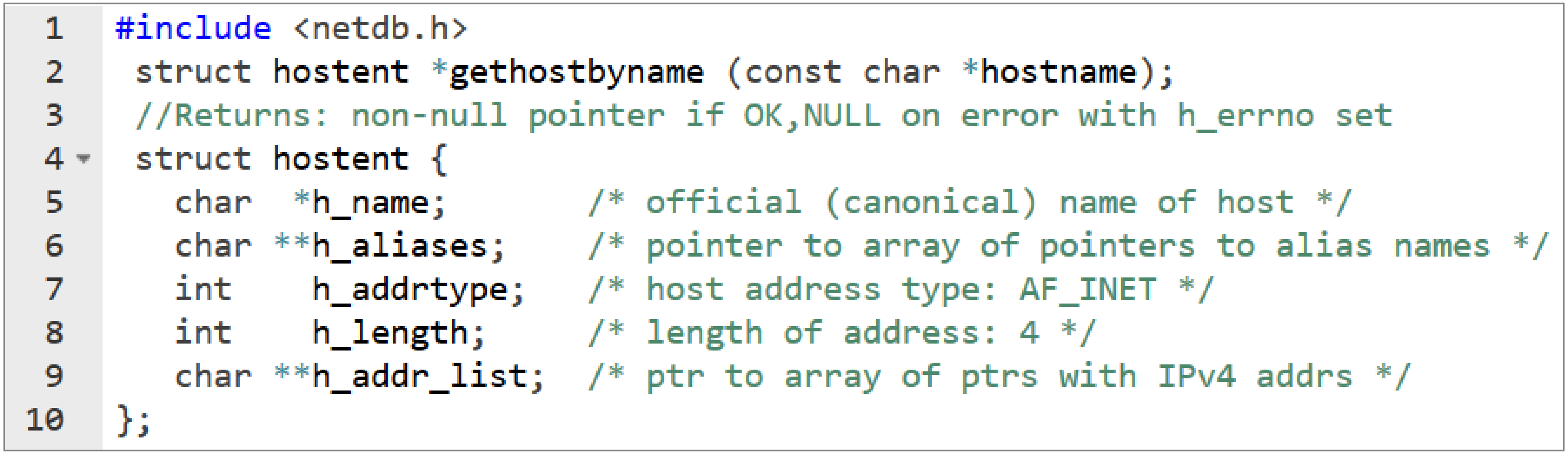


```
#include <sys/socket.h>
#include <netdb.h>
int getnameinfo(const struct sockaddr * addr , socklen_t  addrlen,
char * host , size_t  hostlen , char * service ,
size_t  servlen , int  flags );
//Returns 0 on success, or nonzero on error
```

- NI_DGRAM
  - Default is stream socket service. This make datagram service.
- NI_NAMEREQD
  - Default: if no name found, return numeric ip and port. If this is on, error is returned.

# gethostbyname()

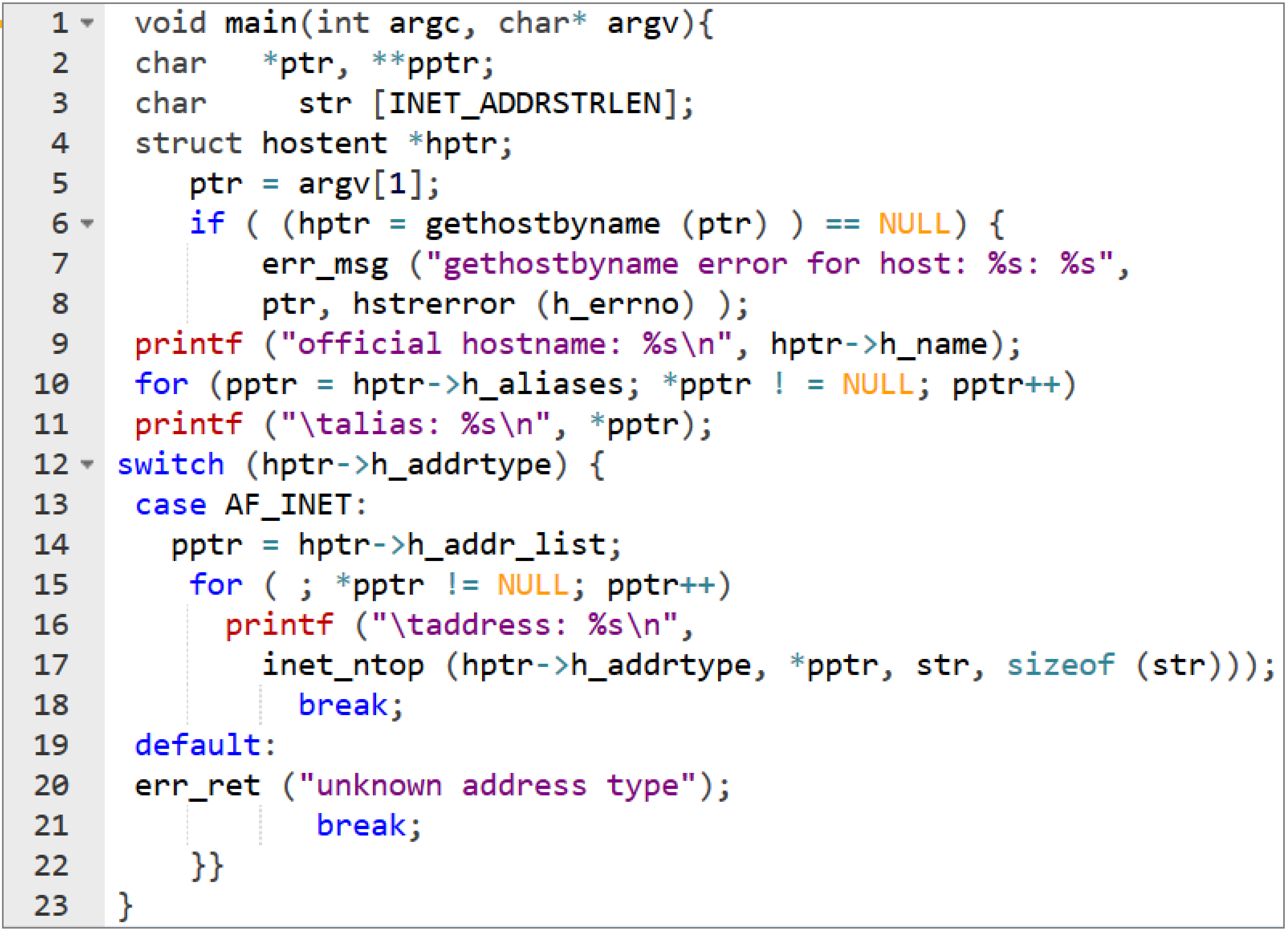
- Returns only A type (IPv4) records



```
#include <netdb.h>
 struct hostent *gethostbyname (const char *hostname);
 //Returns: non-null pointer if OK,NULL on error with h_errno set
 struct hostent {
   char  *h_name;       /* official (canonical) name of host */
   char **h_aliases;    /* pointer to array of pointers to alias names */
   int    h_addrtype;   /* host address type: AF_INET */
   int    h_length;     /* length of address: 4 */
   char **h_addr_list;  /* ptr to array of ptrs with IPv4 addrs */
};
```

- can return multiple IP addresses pointed by h_addr_list.

# gethostbyname() example



```
 void main(int argc, char* argv){
 char   *ptr, **pptr;
 char     str [INET_ADDRSTRLEN];
 struct hostent *hptr;
    ptr = argv[1];
    if ( (hptr = gethostbyname (ptr) ) == NULL) {
        err_msg ("gethostbyname error for host: %s: %s",
        ptr, hstrerror (h_errno) );
 printf ("official hostname: %s\n", hptr->h_name);
 for (pptr = hptr->h_aliases; *pptr ! = NULL; pptr++)
 printf ("\talias: %s\n", *pptr);
switch (hptr->h_addrtype) {
 case AF_INET:
   pptr = hptr->h_addr_list;
    for ( ; *pptr != NULL; pptr++)
      printf ("\taddress: %s\n",
        inet_ntop (hptr->h_addrtype, *pptr, str, sizeof (str)));
          break;
 default:
 err_ret ("unknown address type");
           break;
    }}
}
```

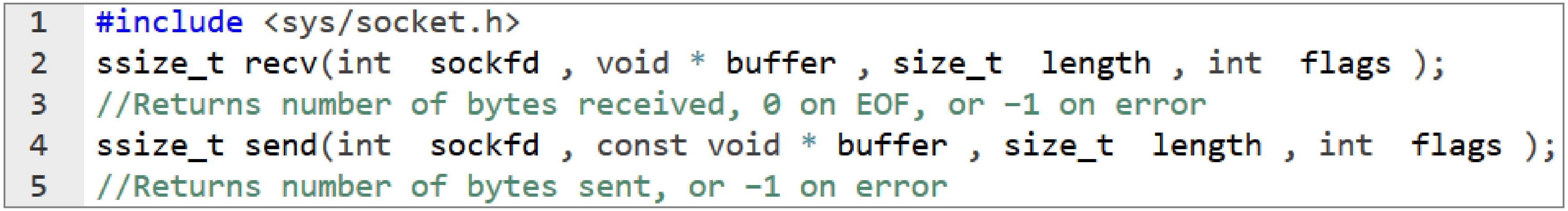
innovate achieve lead

BITS Pilani
Pilani Campus

# Advanced I/O Functions (T1: ch 14)

# recv() and send()

- The recv() and send() system calls perform I/O on connected sockets (TCP or connected UDP sockets).
- Socket-Specific I/O System Calls:
  - They provide socket-specific functionality not available with read(0 and write().



```
#include <sys/socket.h>
ssize_t recv(int  sockfd , void * buffer , size_t  length , int  flags );
//Returns number of bytes received, 0 on EOF, or -1 on error
ssize_t send(int  sockfd , const void * buffer , size_t  length , int  flags );
//Returns number of bytes sent, or -1 on error
```

  - Same as read() and write() except for *flags*.
  - Return values are same as read() and write().

# recv() flags

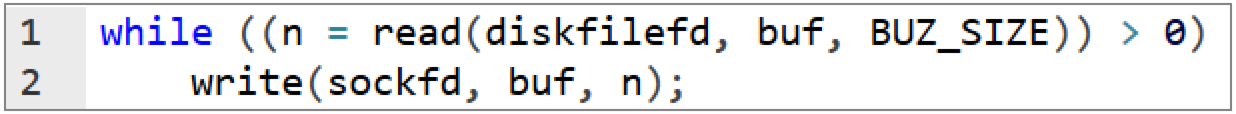
- MSG_DONTWAIT:
  - perform a non-blocking recv().
  - can be done using fcntl() call but that will make sock fd non-blocking. Here only this operation is non-blocking.
- MSG_OOB:
  - receive out-of-band data on the socket.
- MSG_PEEK:
  - retrieve a copy of the requested bytes from the socket buffer.
  - Data is not removed from the socket buffer.
  - Used for knowing the no of bytes available on the buffer.
- MSG_WAITALL
  - Blocks until *length* bytes are read from socket buffer.
  - May get interrupted by signals.

# send() flags

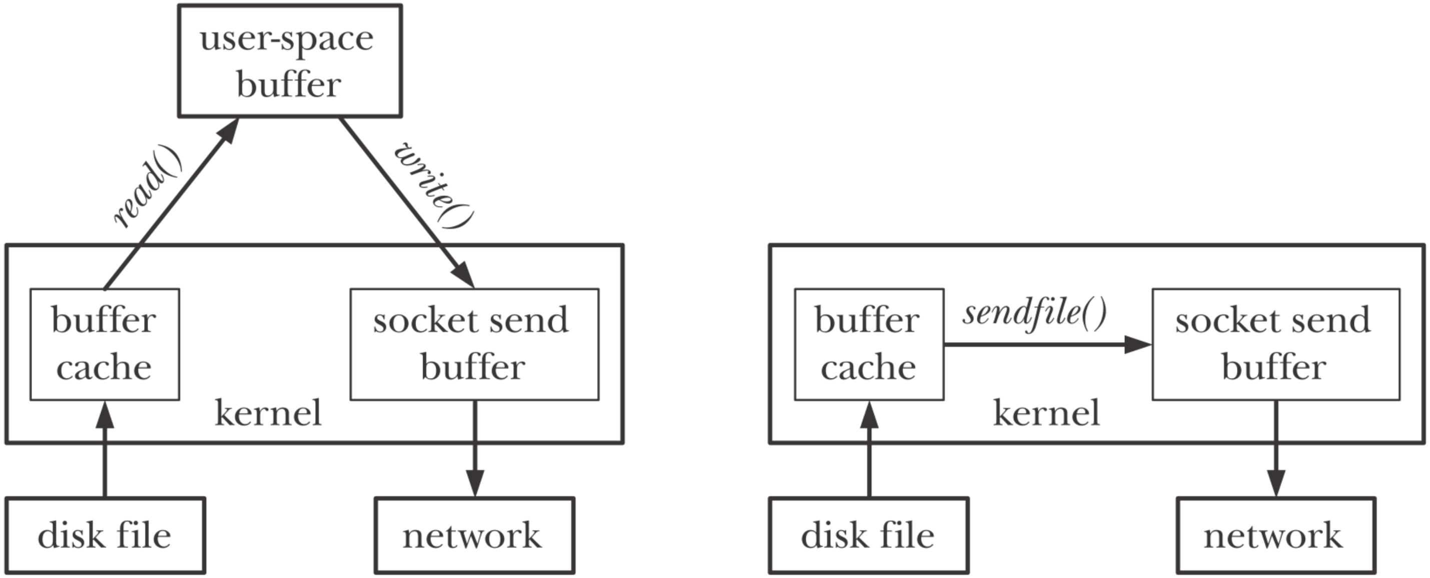
- MSG_DONTWAIT
  - Perform a non-blocking send.
- MSG_MORE
  - Data written using send() or sendto() calls with this flag is packaged into a single datagram until a send() without this flag.
- MSG_NOSIGNAL
  - Do not generate SIGPIPE signal. Return only EPIPE error.
- MSG_OOB
  - Write out of band data on TCP.

# sendfile() sys call (R1: 61.4)

- Transferring large file in web servers requires repeated calls to read() and write().
    - This is inefficient.



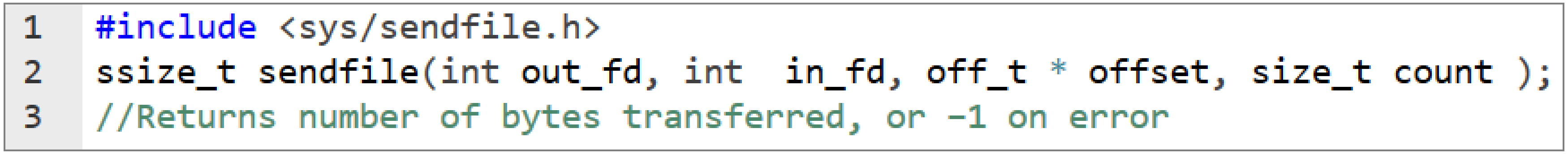
```
while ((n = read(diskfilefd, buf, BUZ_SIZE)) > 0)
    write(sockfd, buf, n);
```



a) *read() + write()*  b) *sendfile()*

# sendfile() sys call (R1: 61.4)

- The sendfile() sys call is designed to eliminate copying file data into user space.
  - File contents are directly transferred to the socket without going through user space.
  - This is referred as a *zero-copy transfer.*



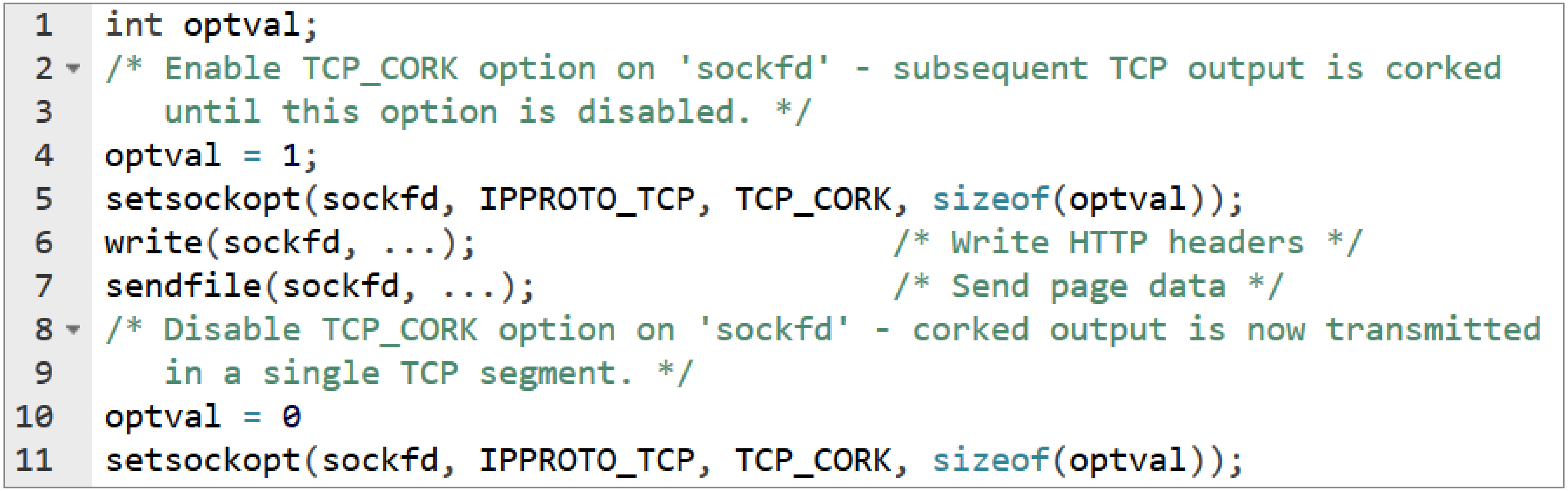
```c
#include <sys/sendfile.h>
ssize_t sendfile(int out_fd, int  in_fd, off_t * offset, size_t count );
//Returns number of bytes transferred, or -1 on error
```

  - *out_fd:* is the socket fd.
  - *In_fd:* is regular file fd.
  - *off_t:* is the offset. This is a value-result argument.
  - *count* is the number of bytes to be transferred.
  - *sendfile* doesn't change the file offset for *in_fd*.

# TCP_CORK socket option

- When a file is requested, web server would first send the HTTP headers and followed by the file contents.
  - Normally this will result in 2 TCP segments.
  - Leads to inefficient use of network bandwidth.
- TCP_CORK option if enabled buffers data in TCP until either
  - upper limit on the size of a segment is reached,
  - the TCP_CORK option is disabled,
  - the socket is closed, or
  - a maximum of 200 milliseconds passes from the time that the first corked byte is written.
- This can be achieved by putting both into single buffer or using *writev()*. But these can't be used with *sendfile*().

# TCP_CORK

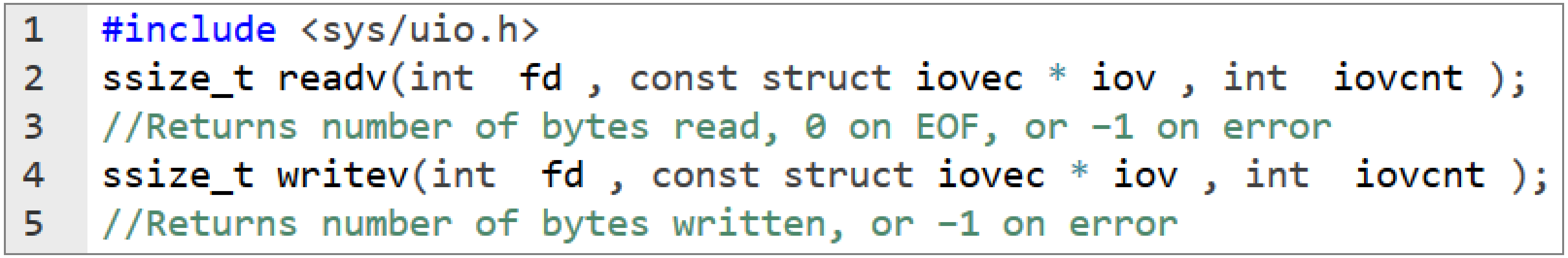


```
int optval;
/* Enable TCP_CORK option on 'sockfd' - subsequent TCP output is corked
   until this option is disabled. */
optval = 1;
setsockopt(sockfd, IPPROTO_TCP, TCP_CORK, sizeof(optval));
write(sockfd, ...);                      /* Write HTTP headers */
sendfile(sockfd, ...);                   /* Send page data */
/* Disable TCP_CORK option on 'sockfd' - corked output is now transmitted
   in a single TCP segment. */
optval = 0
setsockopt(sockfd, IPPROTO_TCP, TCP_CORK, sizeof(optval));
```

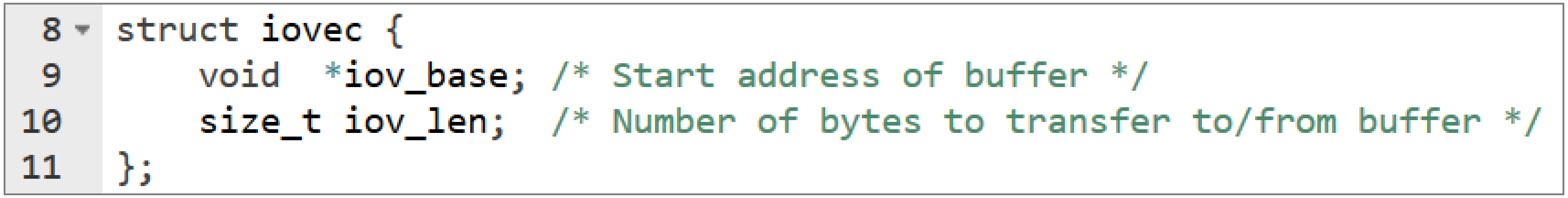
- There is also UDP_CORK option that buffers multiple data outputs into a single datagram.

# readv() and writev()

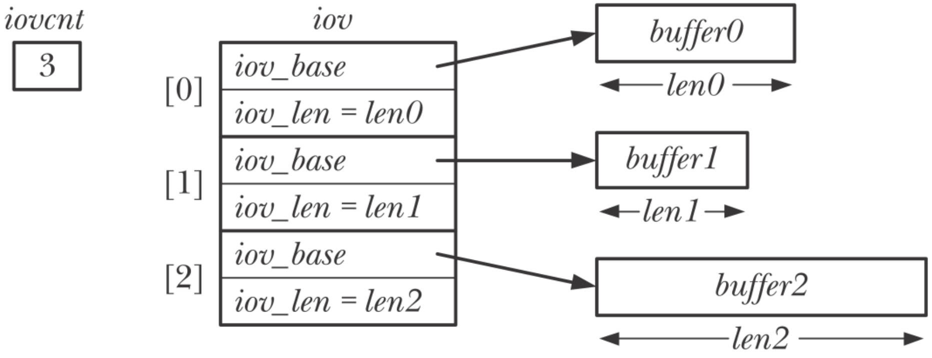
- The readv() and writev() system calls perform scatter-gather I/O.
- *iov* points to an array of buffers, each in *iovec* structure.



```
#include <sys/uio.h>
ssize_t readv(int  fd , const struct iovec * iov , int  iovcnt );
//Returns number of bytes read, 0 on EOF, or -1 on error
ssize_t writev(int  fd , const struct iovec * iov , int  iovcnt );
//Returns number of bytes written, or -1 on error
```



```
struct iovec {
     void  *iov_base; /* Start address of buffer */
     size_t iov_len;  /* Number of bytes to transfer to/from buffer */
};
```

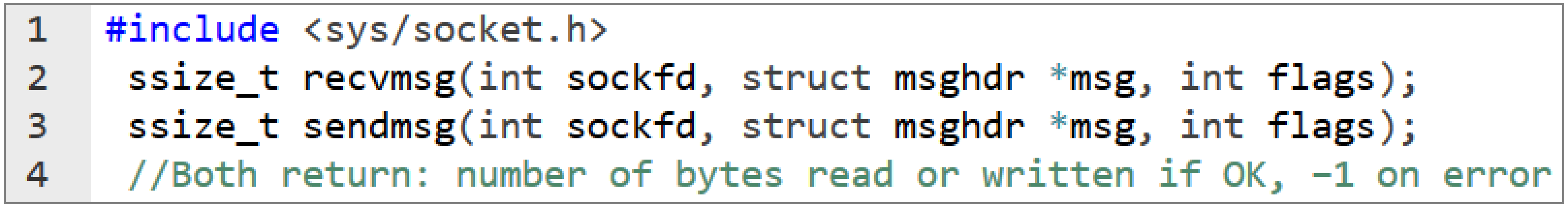


# readv() and writev()

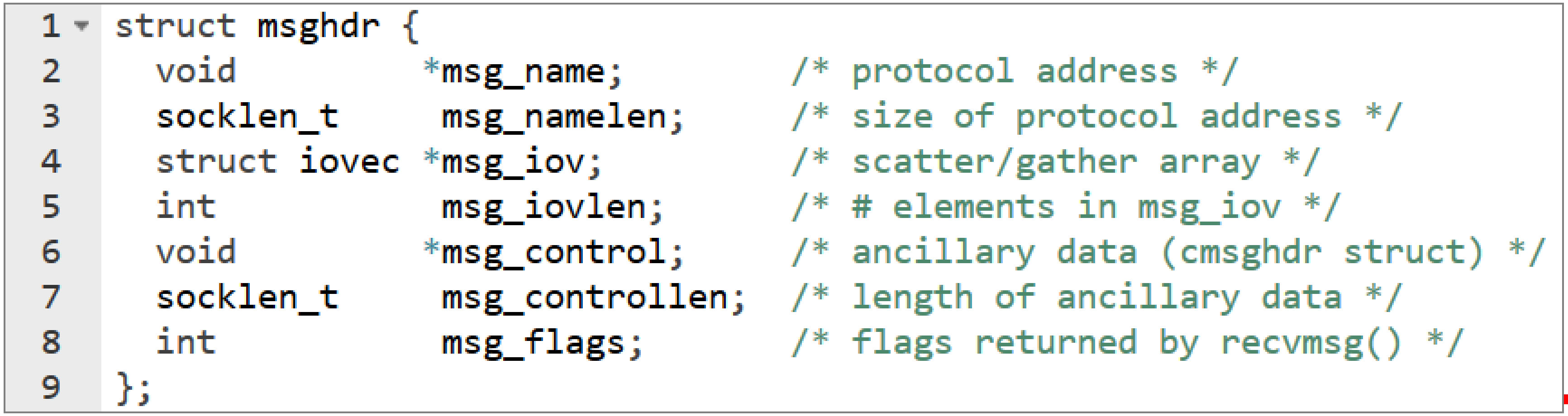
- The readv() system call performs scatter input:
  - Reads from the file and puts the data into the buffer starting at iov[0]. Once the first buffer is full, it goes to another.
- readv() completes atomically:
  - Kernel performs a single data transfer.
  - Assured that all the bytes read are contiguous in the file. File offset can’t be changed by other process.
- The writev() call performs gather output:
  - Starting from the first buffer, writes the data contiguously into the file.
  - Partial write is possible.
- writev() completes atomically.
- readv() and writev() are used for convenience and speed.
  - Reduce number of sys calls.

# sendmsg() & recvmsg() sys calls

- The sendmsg() and recvmsg() system calls are the most general purpose of the socket I/O system calls.
  - The sendmsg() system call can do everything that is done by write(), send(), and sendto();
  - the recvmsg() system call can do everything that is done by read(), recv(), and recvfrom().



```
#include <sys/socket.h>
 ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
 ssize_t sendmsg(int sockfd, struct msghdr *msg, int flags);
 //Both return: number of bytes read or written if OK, -1 on error
```



```
struct msghdr {
  void         *msg_name;        /* protocol address */
  socklen_t     msg_namelen;     /* size of protocol address */
  struct iovec *msg_iov;         /* scatter/gather array */
  int           msg_iovlen;      /* # elements in msg_iov */
  void         *msg_control;     /* ancillary data (cmsghdr struct) */
  socklen_t     msg_controllen;  /* length of ancillary data */
  int           msg_flags;       /* flags returned by recvmsg() */
};
```

# sendmsg() & recvmsg() sys calls

- Can be used to send or receive ancillary data (control information).
- Flags are

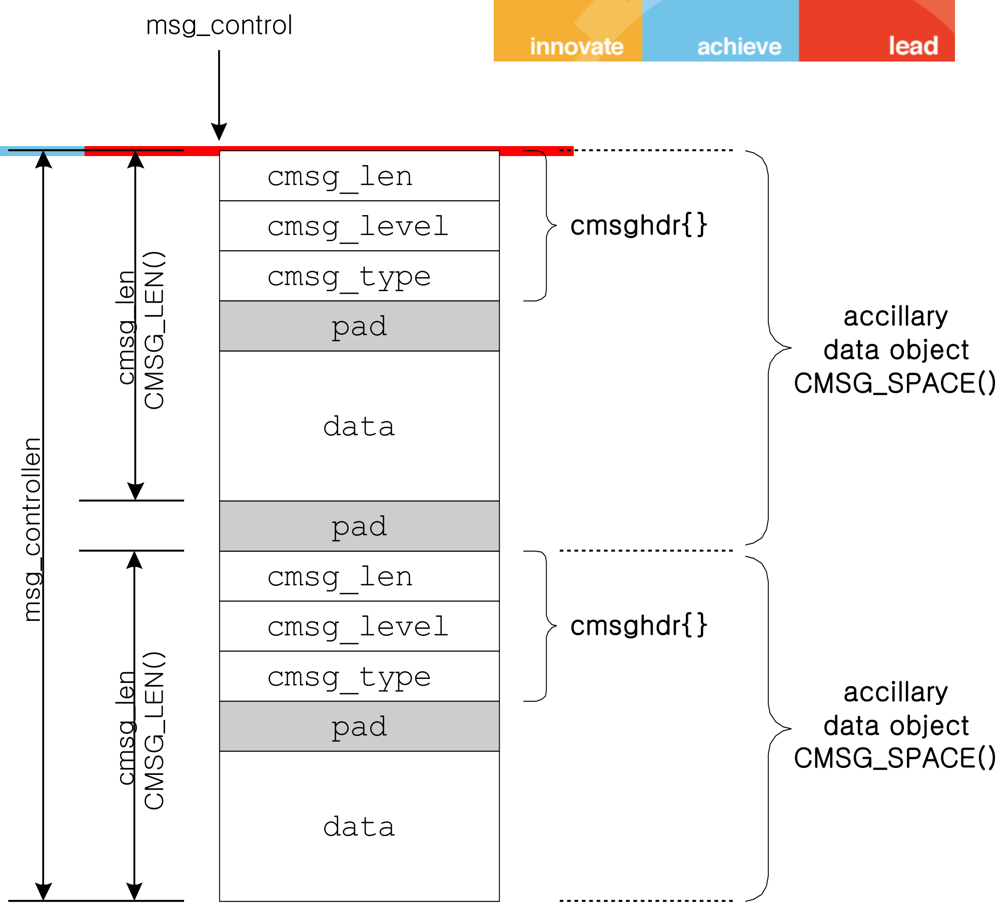
| Flag | **Examined by:**<br>Send flags<br>Sendto flags<br>Sendmsg flags | **Examined by:**<br>recv flags<br>recvfrom flags<br>recvmsg flags | **Returned by:**<br>Recvmsg<br>msg_flags |
|---|---|---|---|
| MSG_DONTROUTE | ● | | |
| MSG_DONTWAIT | ● | ● | |
| MSG_PEEK | | ● | |
| MSG_WAITALL | | ● | |
| MSG_EOR | ● | | ● |
| MSG_OOB | ● | ● | ● |
| MSG_BCAST | | | ● |
| MSG_MCAST | | | ● |
| MSG_TRUNC | | | ● |
| MSG_CTRUNC | | | ● |

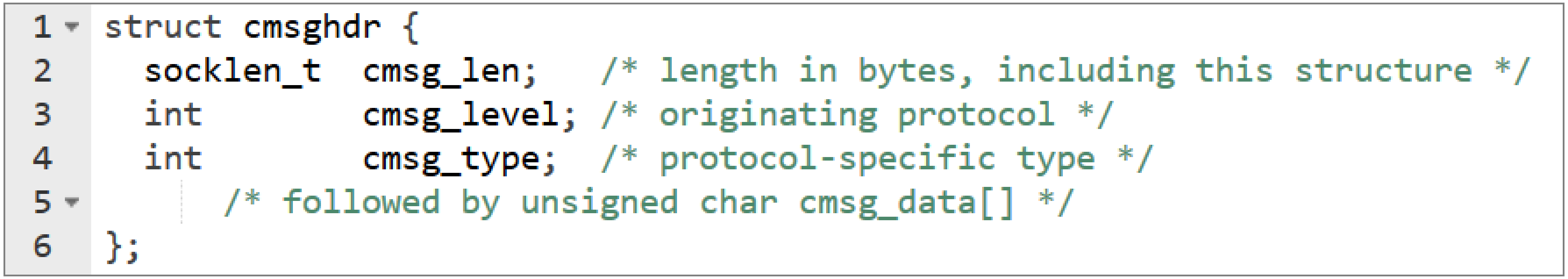
# Flags returned by rcvmsg()

- MSG_BCAST
  - is returned if the datagram was received as as a broadcast.
- MSG_MCAST
  - is returned if the datagram was received as a link-layer multicast.
- MSG_TRUNC
  - is returned if the datagram was truncated
- MSG_CTRUNC
  - is returned if the ancillary data was truncated
- MSG_EOR
  - is turned on if the returned data ends a logical record.
- MSG_OOB
  - This flag is never returned for TCP out-of-band data. This flag is returned by other protocol suites (e.g., the OSI protocols).
- MSG_NOTIFICATON
  - This flag is returned for SCTP receivers to indicate that the message read is an event notification, not a data message.

# Ancillary Data

- Ancillary data can be sent and received using the msg_control and msg_controllen members of the msghdr structure.
  - Another term for ancillary data is control information.





```
struct cmsghdr {
  socklen_t  cmsg_len;   /* length in bytes, including this structure */
  int        cmsg_level; /* originating protocol */
  int        cmsg_type;  /* protocol-specific type */
      /* followed by unsigned char cmsg_data[] */
};
```

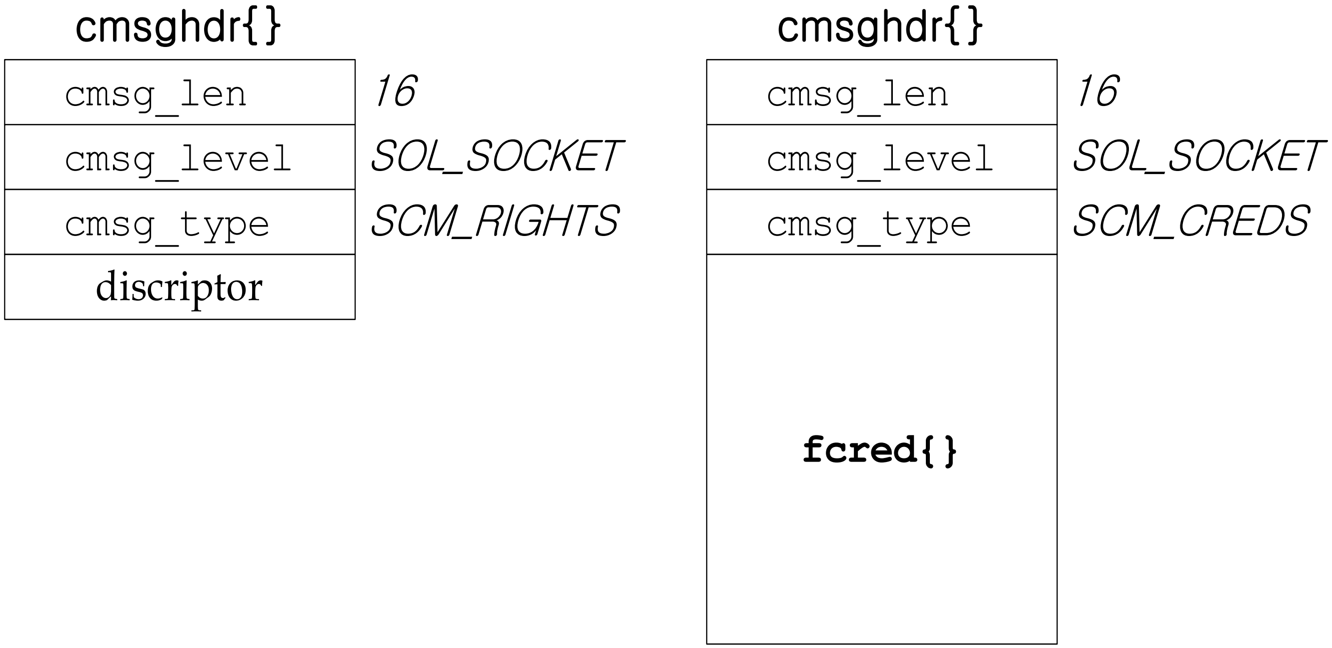
# Ancillary Data

- Ancillary data is domain specific.

| Protocol | cmsg_level | Cmsg_type | Description |
|---|---|---|---|
| IPv4 | IPPROTO_IP | IP_RECVDSTADDR<br>IP_RECVIF | receive destination address with UDP datagram<br>receive interface index with UDP datagram |
| IPv6 | IPPROTO_IPV6 | IPV6_DSTOPTS<br>IPV6_HOPLIMIT<br>IPV6_HOPOPTS<br>IPV6_NEXTHOP<br>IPV6_PKTINFO<br>IPV6_RTHDR | specify / receive destination options<br>specify / receive hop limit<br>specify / receive hop-by-hop options<br>specify next-hop address<br>specify / receive packet information<br>specify / receive routing header |
| Unix domain | SOL_SOCKET | SCM_RIGHTS<br>SCM_CREDS | send / receive descriptors<br>send / receive user credentials |

# Ancillary Data

- File descriptors and process credentials can be passed between unrelated processes using ancillary data.



cmsghdr{}

| | |
|---|---|
| cmsg_len | *16* |
| cmsg_level | *SOL_SOCKET* |
| cmsg_type | *SCM_RIGHTS* |
| discriptor | |

cmsghdr{}

| | |
|---|---|
| cmsg_len | *16* |
| cmsg_level | *SOL_SOCKET* |
| cmsg_type | *SCM_CREDS* |
| **fcred{}** | |

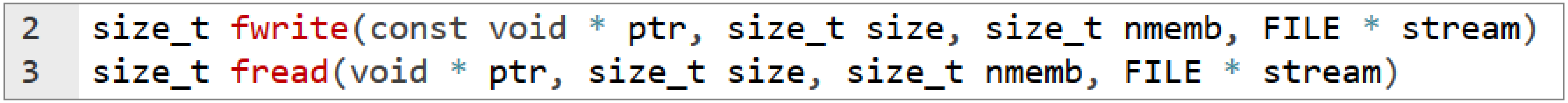
# Framing, Encoding & Decoding

# Encoding & Framing

- Wire format: format of message in the network.
  - Application protocol specifies the wire format for the messages to be exchanged between sender and receiver.
  - Can be: binary or text represented.
- Encoding:
  - Sender has to fill in the fields of the message ( bits or byes or multiple bytes) considering the network byte order.
- Framing
  - How to identify boundaries between messages

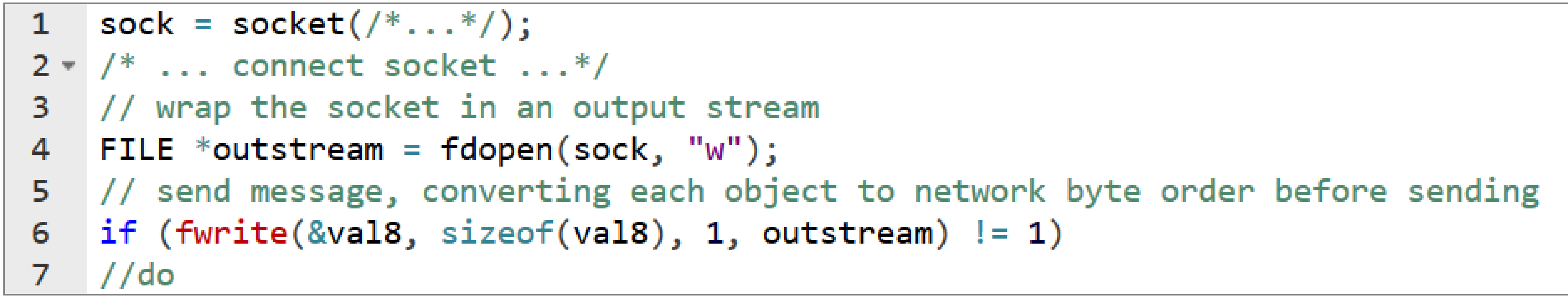
# Wrapping TCP Sockets in Streams

- Wrapping TCP Sockets in Streams
  - Using fdopen
  - fread() and fwrite() functions read/write a number of objects from/to the stream.
  - They return no of objects written or read.



```
size_t fwrite(const void * ptr, size_t size, size_t nmemb, FILE * stream)
size_t fread(void * ptr, size_t size, size_t nmemb, FILE * stream)
```

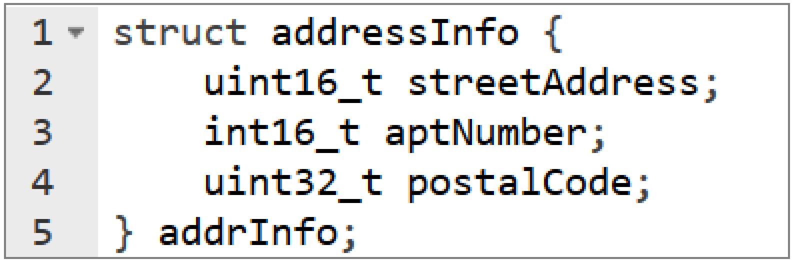
- FILE-streams can only be used with TCP sockets



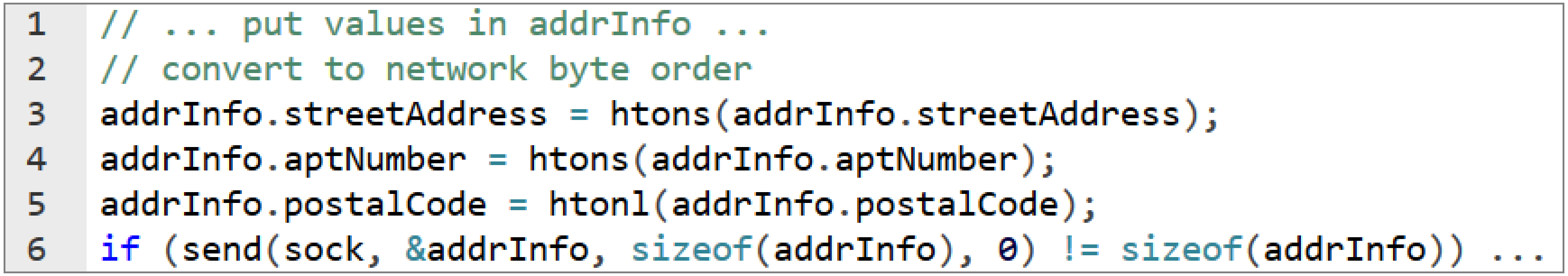
```
sock = socket(/*...*/);
/* ... connect socket ...*/
// wrap the socket in an output stream
FILE *outstream = fdopen(sock, "w");
// send message, converting each object to network byte order before sending
if (fwrite(&val8, sizeof(val8), 1, outstream) != 1)
//do
```

# Wrapping TCP Sockets in Streams

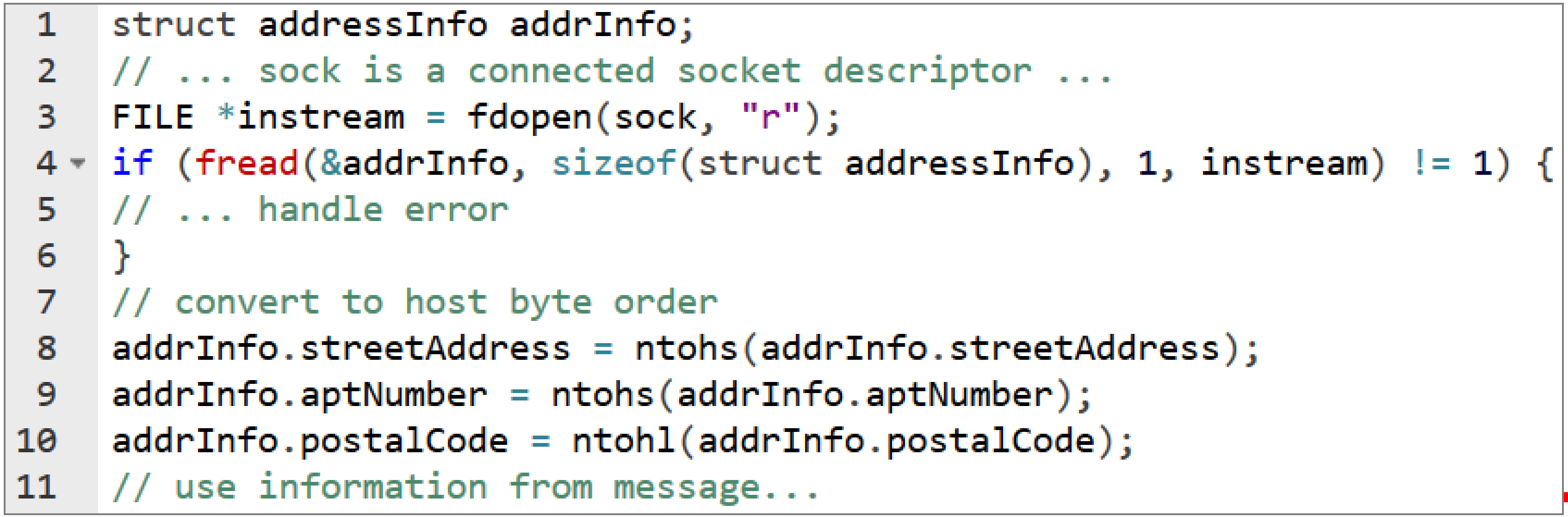
- If we have to exchange data



```c
struct addressInfo {
    uint16_t streetAddress;
    int16_t aptNumber;
    uint32_t postalCode;
} addrInfo;
```



```c
// ... put values in addrInfo ...
// convert to network byte order
addrInfo.streetAddress = htons(addrInfo.streetAddress);
addrInfo.aptNumber = htons(addrInfo.aptNumber);
addrInfo.postalCode = htonl(addrInfo.postalCode);
if (send(sock, &addrInfo, sizeof(addrInfo), 0) != sizeof(addrInfo)) ...
```



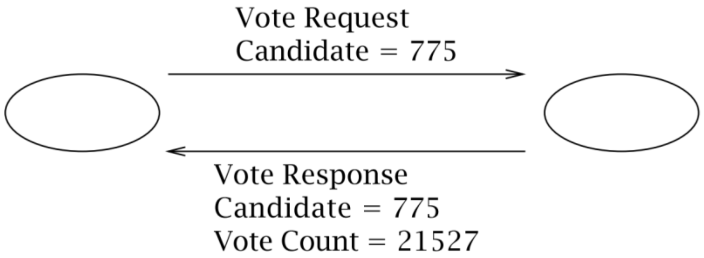
```c
struct addressInfo addrInfo;
// ... sock is a connected socket descriptor ...
FILE *instream = fdopen(sock, "r");
if (fread(&addrInfo, sizeof(struct addressInfo), 1, instream) != 1) {
// ... handle error
}
// convert to host byte order
addrInfo.streetAddress = ntohs(addrInfo.streetAddress);
addrInfo.aptNumber = ntohs(addrInfo.aptNumber);
addrInfo.postalCode = ntohl(addrInfo.postalCode);
// use information from message...
```

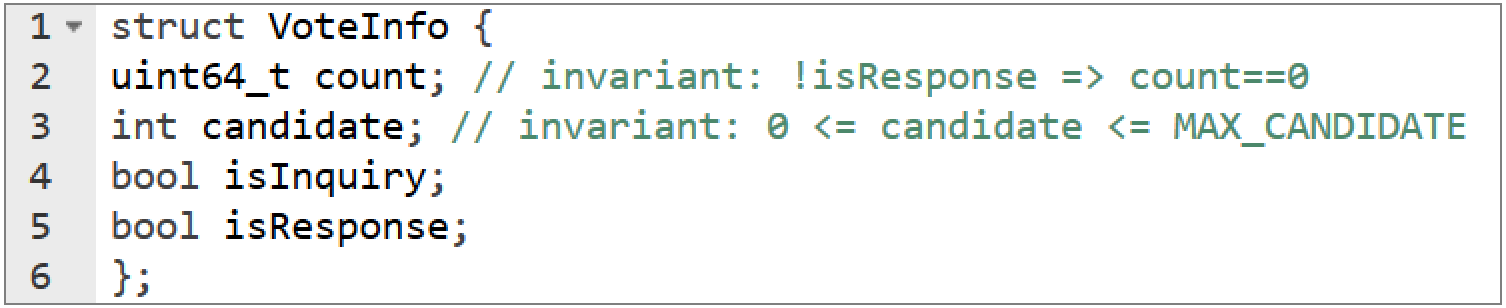
# Wrapping TCP Sockets in Streams

- Using recv() will not help because  there is no guarantee that all the bytes of the message will be returned in a single call.
  - We can use MSG_WAITALL

# Case

- Client sends a request message to the server:
  - inquiry request: how many votes are polled for a candidate.
  - Vote request: vote for a candidate.
- For both the requests server responds with
  - Response flag
  - Candidate id
  - Vote count

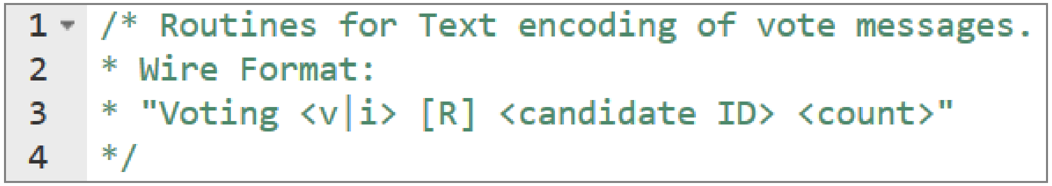




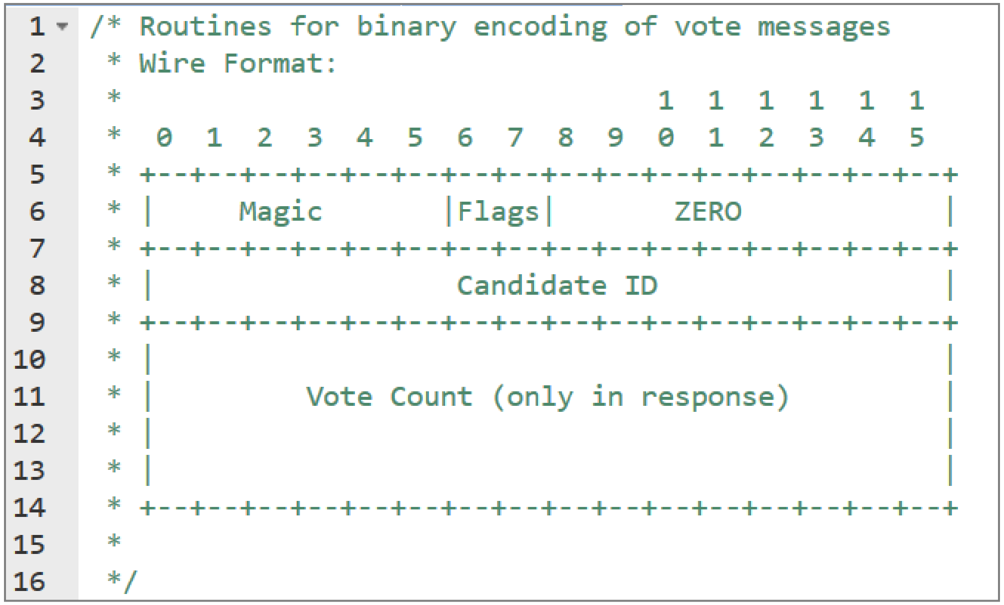
```
struct VoteInfo {
uint64_t count; // invariant: !isResponse => count==0
int candidate; // invariant: 0 <= candidate <= MAX_CANDIDATE
bool isInquiry;
bool isResponse;
};
```

# Encoding & decoding

- Application uses VoteInfo structure for its internal bookkeeping.
- Encoding: convert VoteInfo structure to wire format.
- Decoding: convert back to VoteInfo structure

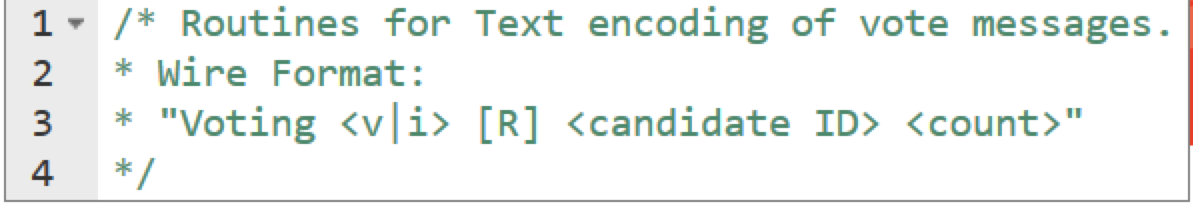


```
/* Routines for Text encoding of vote messages.
* Wire Format:
* "Voting <v|i> [R] <candidate ID> <count>"
*/
```

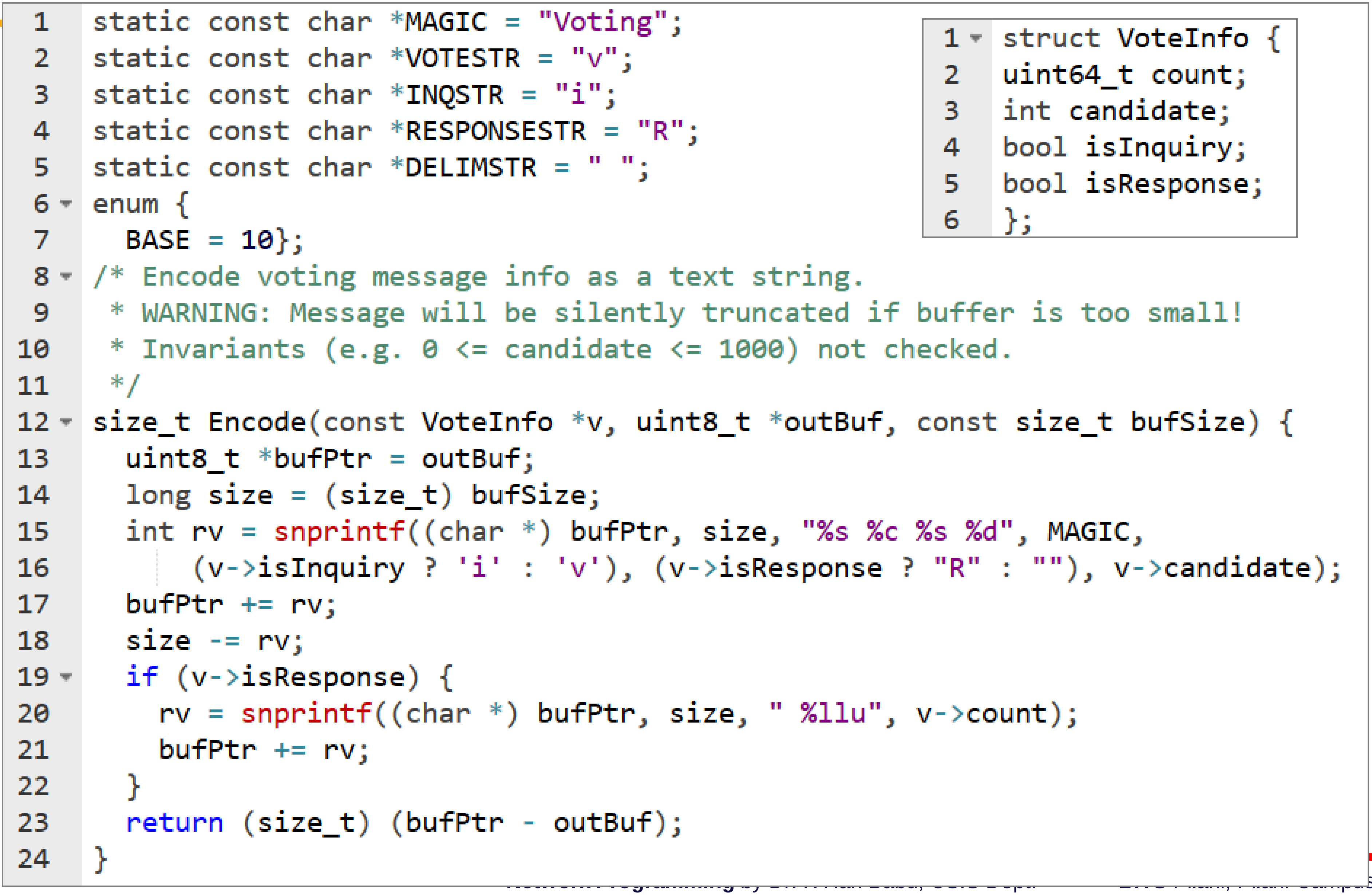


```
/* Routines for binary encoding of vote messages
 * Wire Format:
 *                                  1  1  1  1  1  1
 *  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 * |     Magic       |Flags|       ZERO            |
 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 * |                  Candidate ID                 |
 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 * |                                               |
 * |         Vote Count (only in response)         |
 * |                                               |
 * |                                               |
 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 *
 */
```

# Encoding



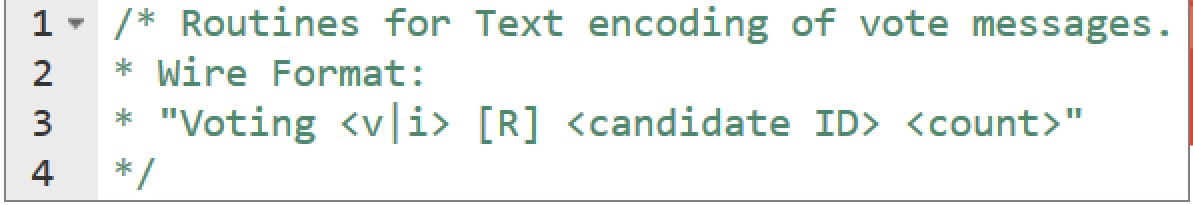
```
/* Routines for Text encoding of vote messages.
* Wire Format:
* "Voting <v|i> [R] <candidate ID> <count>"
*/
```



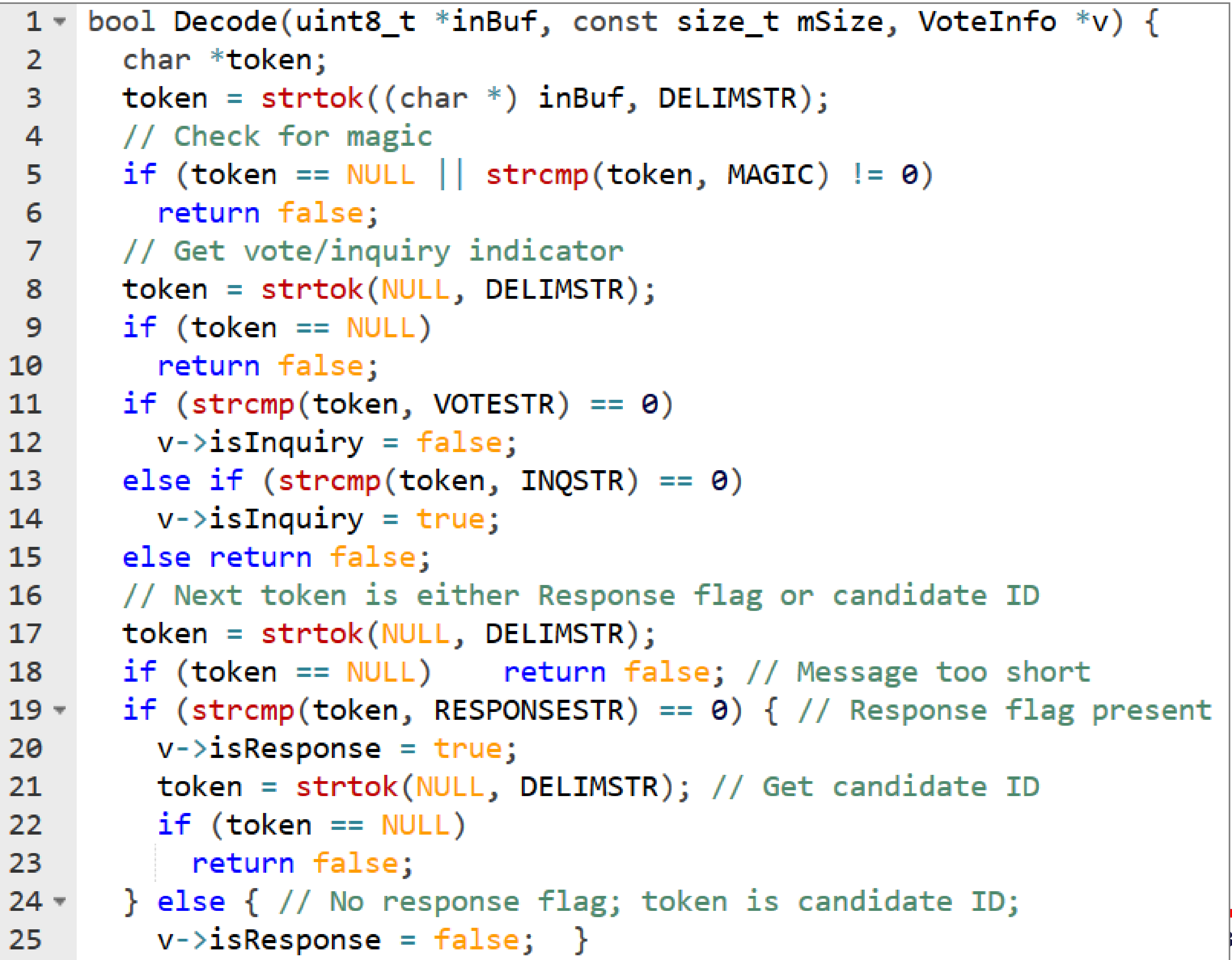
```
struct VoteInfo {
uint64_t count;
int candidate;
bool isInquiry;
bool isResponse;
};
```

```
static const char *MAGIC = "Voting";
static const char *VOTESTR = "v";
static const char *INQSTR = "i";
static const char *RESPONSESTR = "R";
static const char *DELIMSTR = " ";
enum {
  BASE = 10};
/* Encode voting message info as a text string.
 * WARNING: Message will be silently truncated if buffer is too small!
 * Invariants (e.g. 0 <= candidate <= 1000) not checked.
 */
size_t Encode(const VoteInfo *v, uint8_t *outBuf, const size_t bufSize) {
  uint8_t *bufPtr = outBuf;
  long size = (size_t) bufSize;
  int rv = snprintf((char *) bufPtr, size, "%s %c %s %d", MAGIC,
      (v->isInquiry ? 'i' : 'v'), (v->isResponse ? "R" : ""), v->candidate);
  bufPtr += rv;
  size -= rv;
  if (v->isResponse) {
    rv = snprintf((char *) bufPtr, size, " %llu", v->count);
    bufPtr += rv;
  }
  return (size_t) (bufPtr - outBuf);
}
```

# Decoding

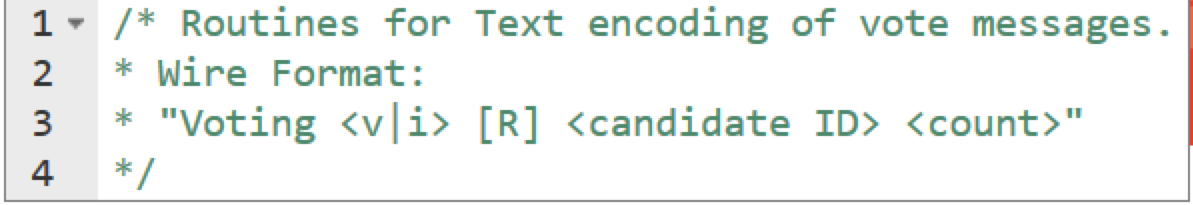


```
/* Routines for Text encoding of vote messages.
* Wire Format:
* "Voting <v|i> [R] <candidate ID> <count>"
*/
```

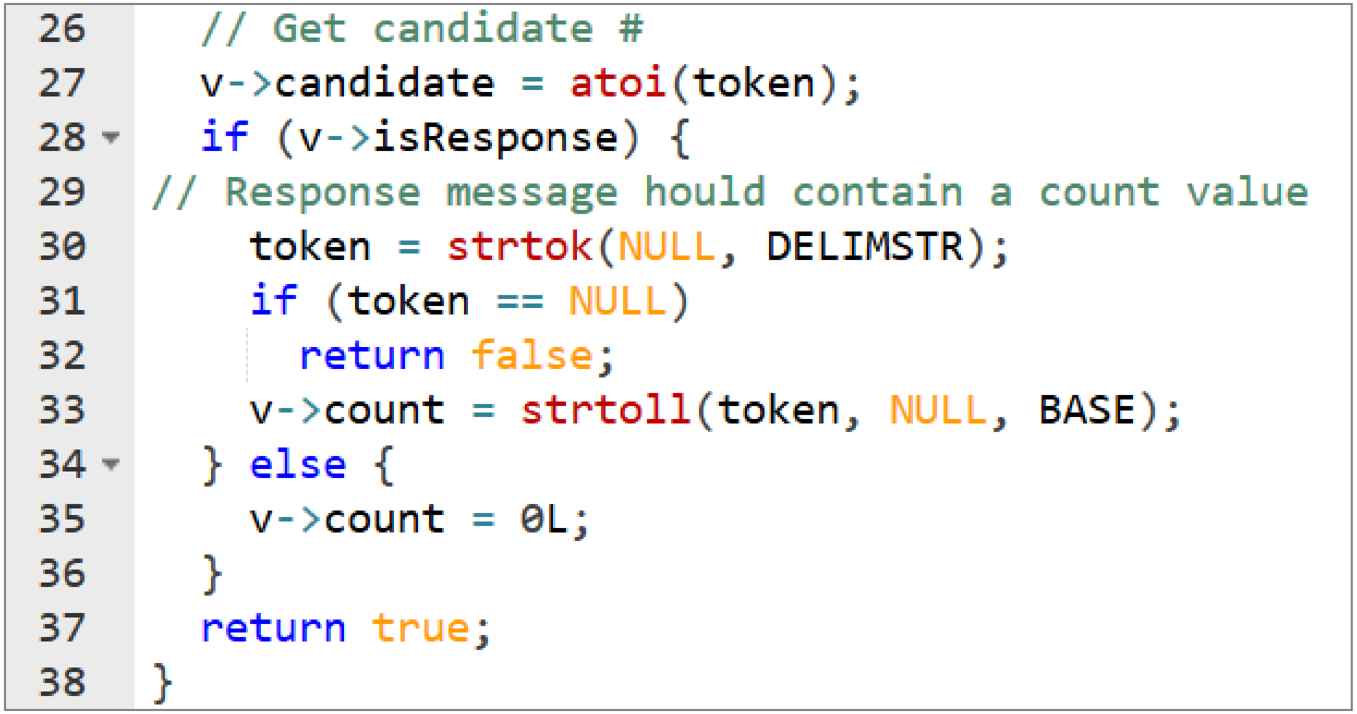


```
bool Decode(uint8_t *inBuf, const size_t mSize, VoteInfo *v) {
  char *token;
  token = strtok((char *) inBuf, DELIMSTR);
  // Check for magic
  if (token == NULL || strcmp(token, MAGIC) != 0)
    return false;
  // Get vote/inquiry indicator
  token = strtok(NULL, DELIMSTR);
  if (token == NULL)
    return false;
  if (strcmp(token, VOTESTR) == 0)
    v->isInquiry = false;
  else if (strcmp(token, INQSTR) == 0)
    v->isInquiry = true;
  else return false;
  // Next token is either Response flag or candidate ID
  token = strtok(NULL, DELIMSTR);
  if (token == NULL)    return false; // Message too short
  if (strcmp(token, RESPONSESTR) == 0) { // Response flag present
    v->isResponse = true;
    token = strtok(NULL, DELIMSTR); // Get candidate ID
    if (token == NULL)
      return false;
  } else { // No response flag; token is candidate ID;
    v->isResponse = false;  }
```

# Decoding



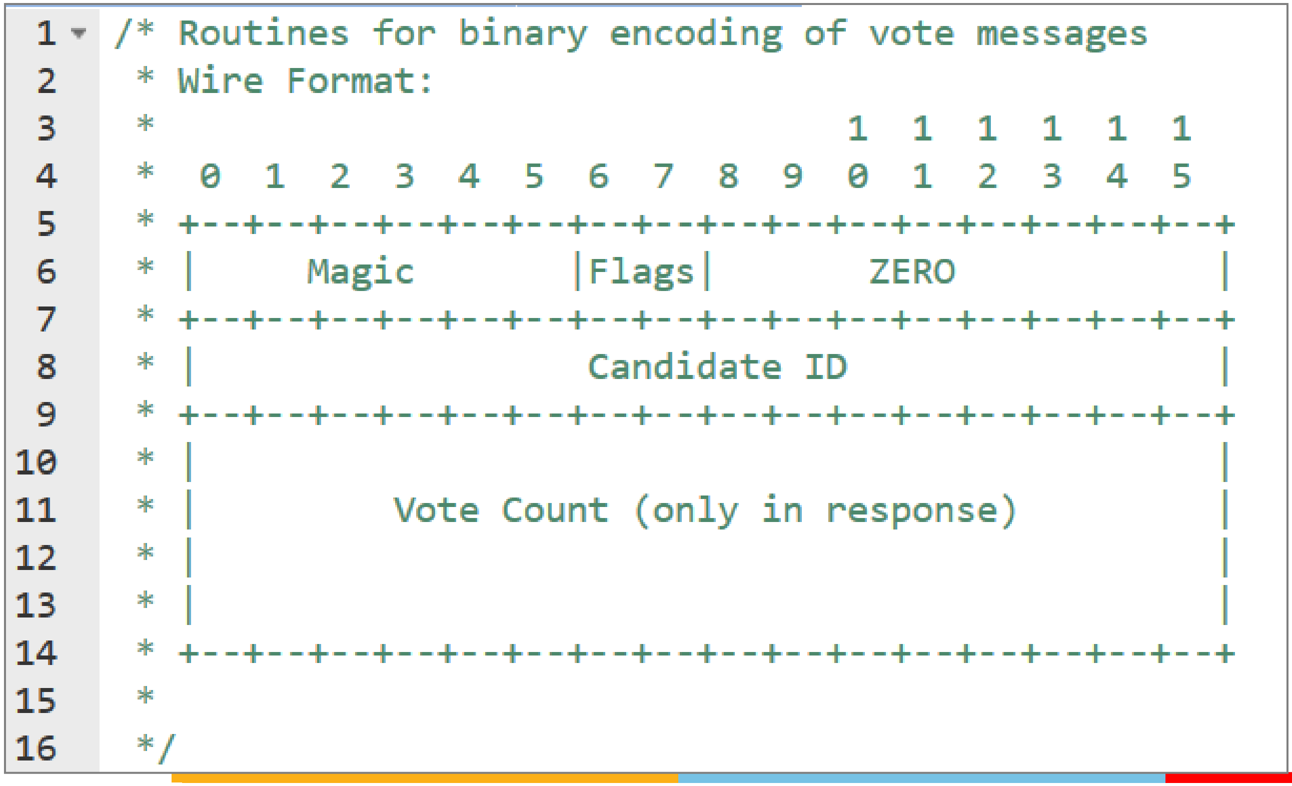
```
/* Routines for Text encoding of vote messages.
* Wire Format:
* "Voting <v|i> [R] <candidate ID> <count>"
*/
```



```
  // Get candidate #
  v->candidate = atoi(token);
  if (v->isResponse) {
// Response message hould contain a count value
    token = strtok(NULL, DELIMSTR);
    if (token == NULL)
      return false;
    v->count = strtoll(token, NULL, BASE);
  } else {
    v->count = 0L;
  }
  return true;
}
```

# Encoding a Binary Wire Format

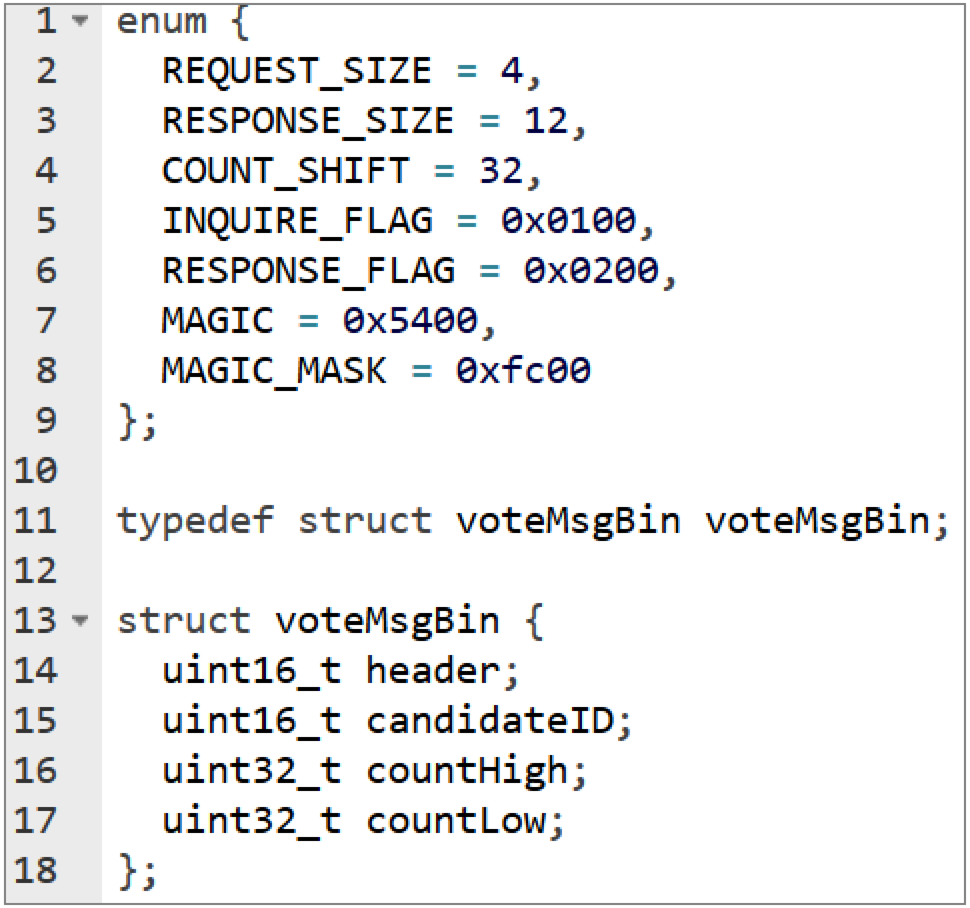
- Text based formats vary in length.
- Binary formats always have fixed size.
- Magic value 010101 helps in ensuring that we are receiving the right message.



```
/* Routines for binary encoding of vote messages
 * Wire Format:
 *                                1  1  1  1  1  1
 *  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 * |     Magic       |Flags|       ZERO            |
 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 * |                  Candidate ID                 |
 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 * |                                               |
 * |         Vote Count (only in response)         |
 * |                                               |
 * |                                               |
 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 *
 */
```

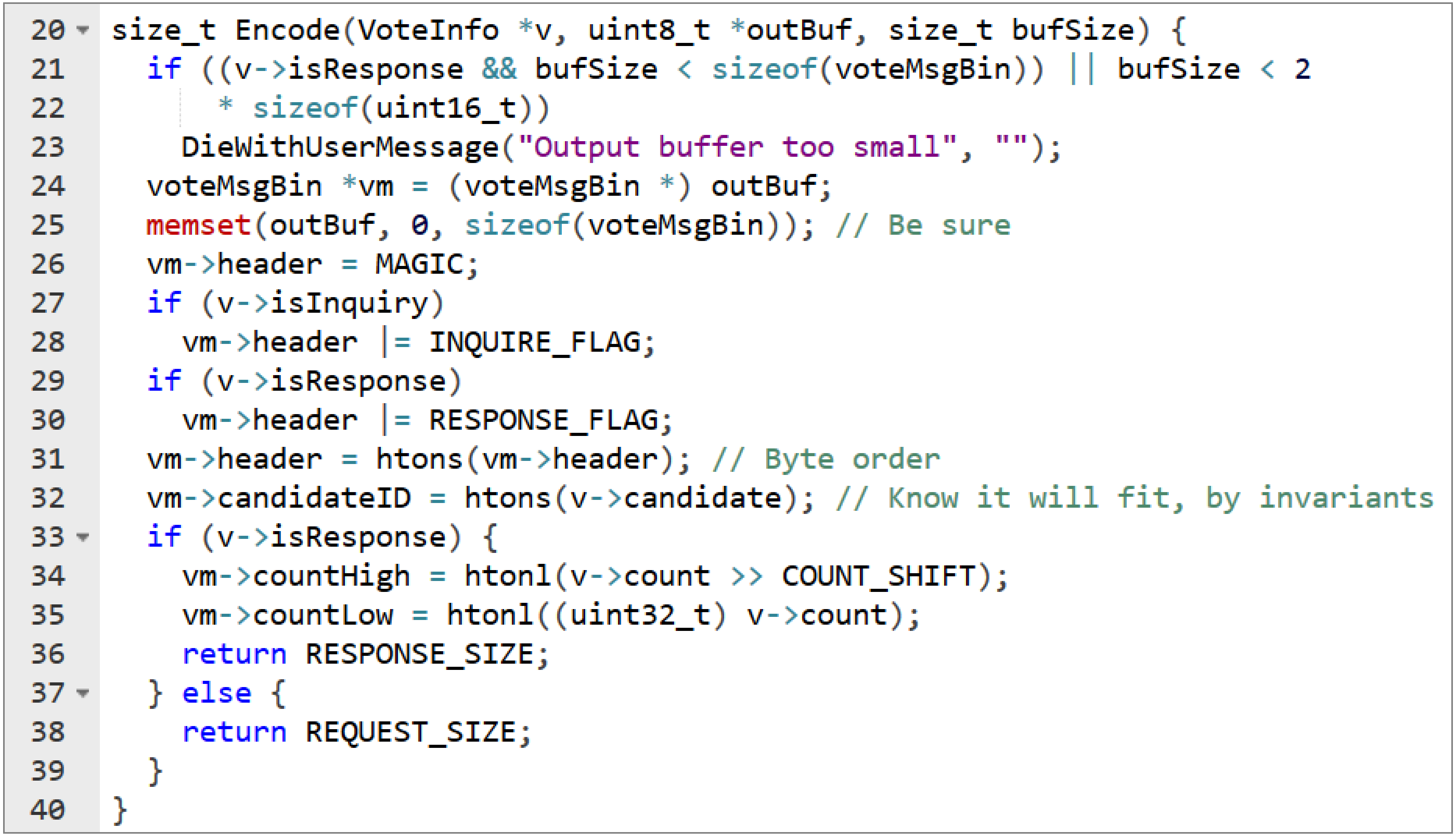
# Encoding a Binary Wire Format

- Generally declare an equivalent structure for the wire format.



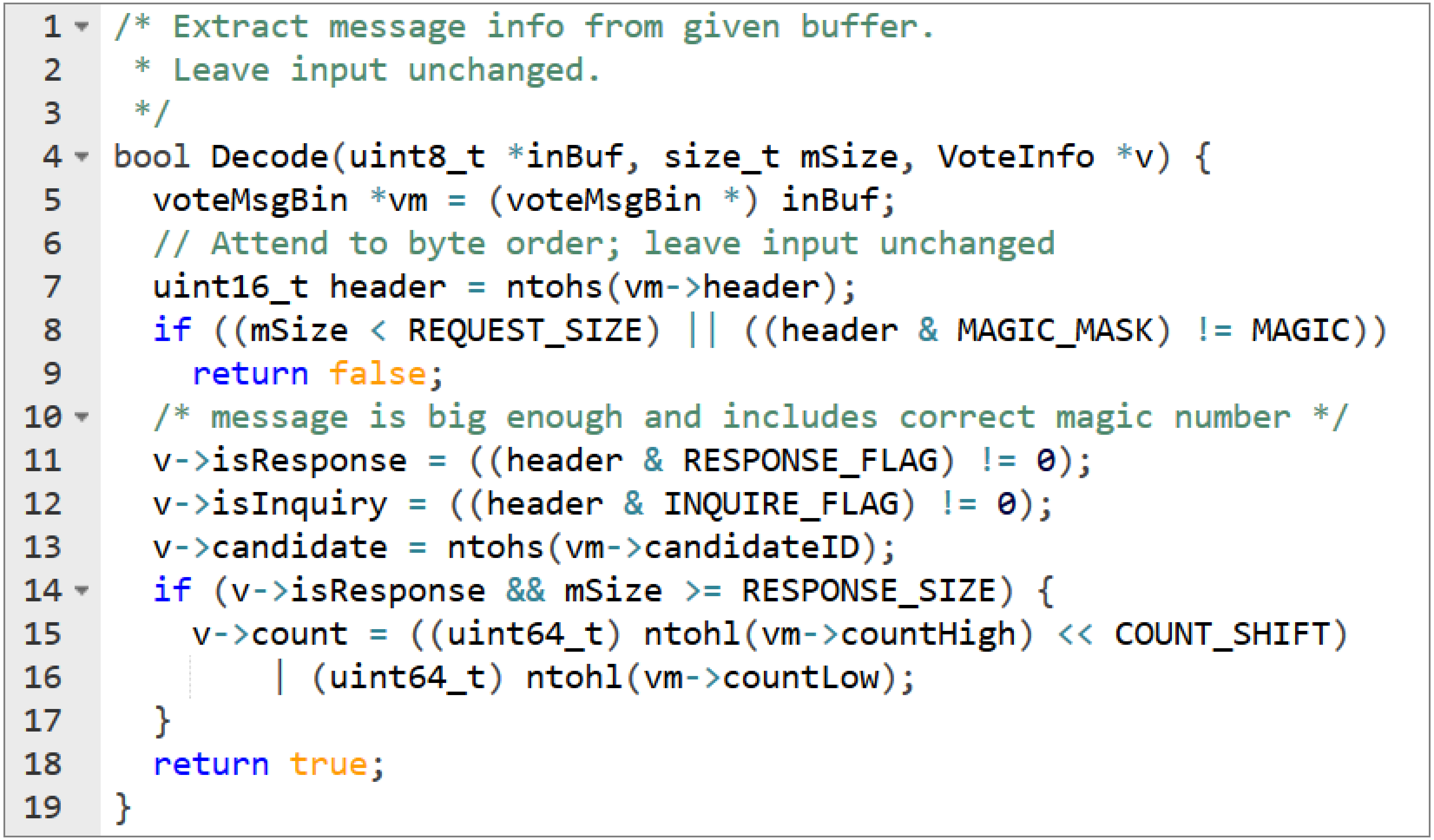
```
enum {
  REQUEST_SIZE = 4,
  RESPONSE_SIZE = 12,
  COUNT_SHIFT = 32,
  INQUIRE_FLAG = 0x0100,
  RESPONSE_FLAG = 0x0200,
  MAGIC = 0x5400,
  MAGIC_MASK = 0xfc00
};

typedef struct voteMsgBin voteMsgBin;

struct voteMsgBin {
  uint16_t header;
  uint16_t candidateID;
  uint32_t countHigh;
  uint32_t countLow;
};
```

# Encoding a Binary Wire Format



```
size_t Encode(VoteInfo *v, uint8_t *outBuf, size_t bufSize) {
  if ((v->isResponse && bufSize < sizeof(voteMsgBin)) || bufSize < 2
      * sizeof(uint16_t))
    DieWithUserMessage("Output buffer too small", "");
  voteMsgBin *vm = (voteMsgBin *) outBuf;
  memset(outBuf, 0, sizeof(voteMsgBin)); // Be sure
  vm->header = MAGIC;
  if (v->isInquiry)
    vm->header |= INQUIRE_FLAG;
  if (v->isResponse)
    vm->header |= RESPONSE_FLAG;
  vm->header = htons(vm->header); // Byte order
  vm->candidateID = htons(v->candidate); // Know it will fit, by invariants
  if (v->isResponse) {
    vm->countHigh = htonl(v->count >> COUNT_SHIFT);
    vm->countLow = htonl((uint32_t) v->count);
    return RESPONSE_SIZE;
  } else {
    return REQUEST_SIZE;
  }
}
```

# Decoding a Binary Wire Format



```
/* Extract message info from given buffer.
 * Leave input unchanged.
 */
bool Decode(uint8_t *inBuf, size_t mSize, VoteInfo *v) {
  voteMsgBin *vm = (voteMsgBin *) inBuf;
  // Attend to byte order; leave input unchanged
  uint16_t header = ntohs(vm->header);
  if ((mSize < REQUEST_SIZE) || ((header & MAGIC_MASK) != MAGIC))
    return false;
  /* message is big enough and includes correct magic number */
  v->isResponse = ((header & RESPONSE_FLAG) != 0);
  v->isInquiry = ((header & INQUIRE_FLAG) != 0);
  v->candidate = ntohs(vm->candidateID);
  if (v->isResponse && mSize >= RESPONSE_SIZE) {
    v->count = ((uint64_t) ntohl(vm->countHigh) << COUNT_SHIFT)
        | (uint64_t) ntohl(vm->countLow);
  }
  return true;
}
```

# Framing

- Framing refers to the general problem of enabling the receiver to locate the boundaries of a message.
    - the application protocol must specify how the receiver of a message can determine when it has received all of the message.
        - Whether information is encoded as text, as multibyte binary numbers, or as some combination of the two
    - This is trivial in UDP.
    - But TCP doesn't preserve the boundaries.
        - If the message length is fixed, then we can wait until we read all of the bytes. But if the length is varying then?
- Two techniques:
    - Delimiter based: The end of the message is indicated by a unique marker.
    - Explicit length: The variable-length field or message is preceded by a length field that tells how many bytes it contains.

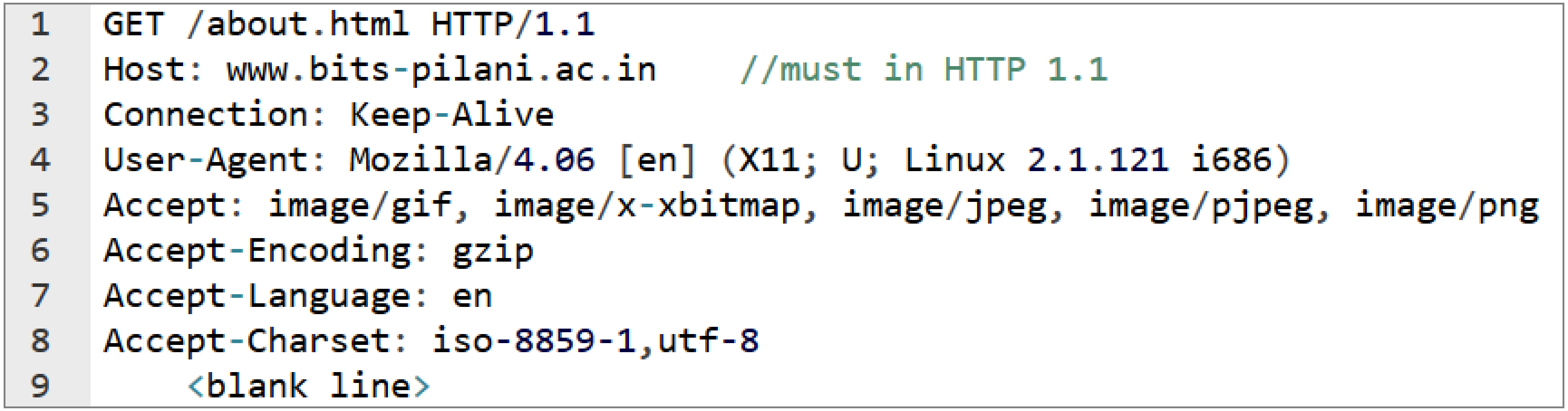
BITS Pilani

Pilani Campus

# Framing in HTTP

# HTTP Request

- HTTP is a request-response stateless protocol. HTTP/1.1 supports persistent connections.
- HTTP Request:
  - GET request  doesn't include anything in the body.

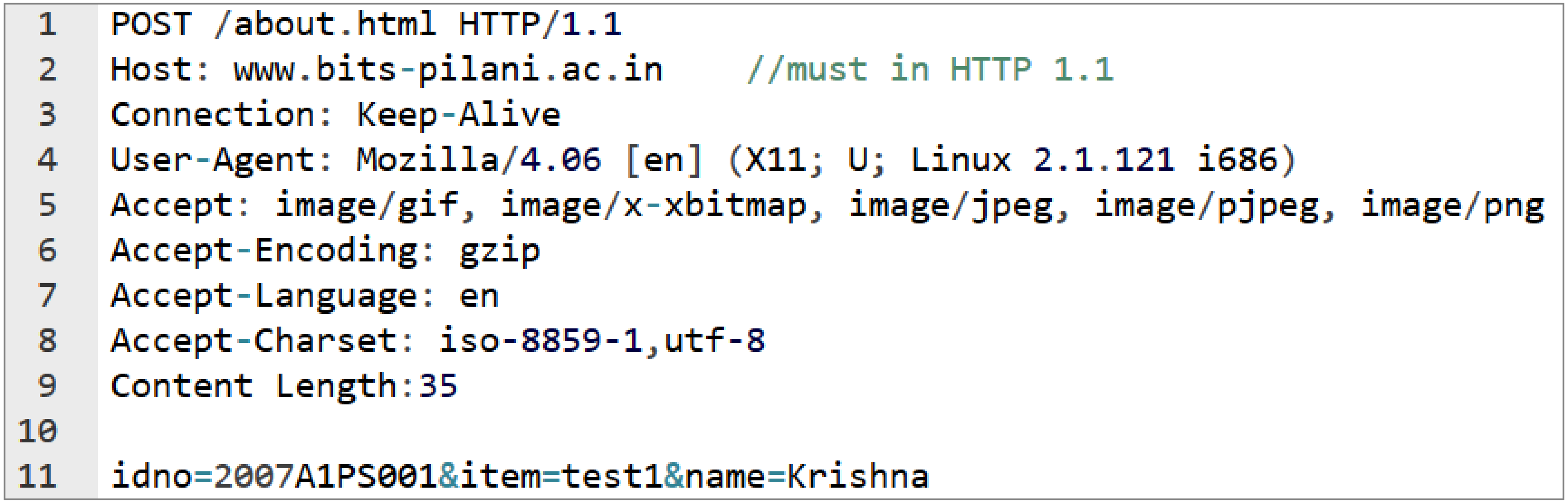


```
GET /about.html HTTP/1.1
Host: www.bits-pilani.ac.in    //must in HTTP 1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.06 [en] (X11; U; Linux 2.1.121 i686)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,utf-8
    <blank line>
```

  - Each line is terminated by \r\n.
  - Headers are terminated by an \n.
- Framing: a blank line

# HTTP Request

- POST request  includes content in the body.
  - Length of the content is specified in the header.

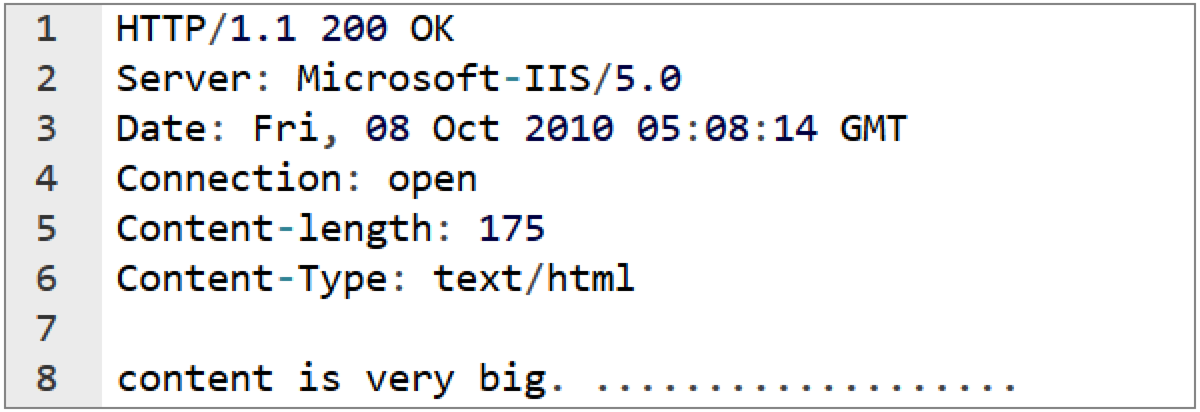


```
POST /about.html HTTP/1.1
Host: www.bits-pilani.ac.in    //must in HTTP 1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.06 [en] (X11; U; Linux 2.1.121 i686)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,utf-8
Content Length:35

idno=2007A1PS001&item=test1&name=Krishna
```

  - Each line is terminated by \r\n.
  - Headers are terminated by an \n.
- Framing: a blank line is end of headers.
  - Body’s boundary is identified by explicit-length method

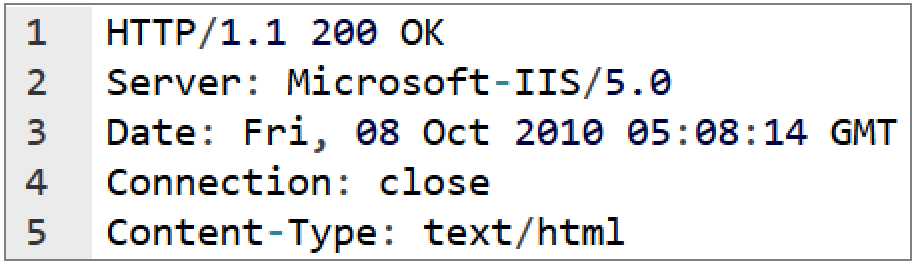
# HTTP Response

- Response header contains content-length field.



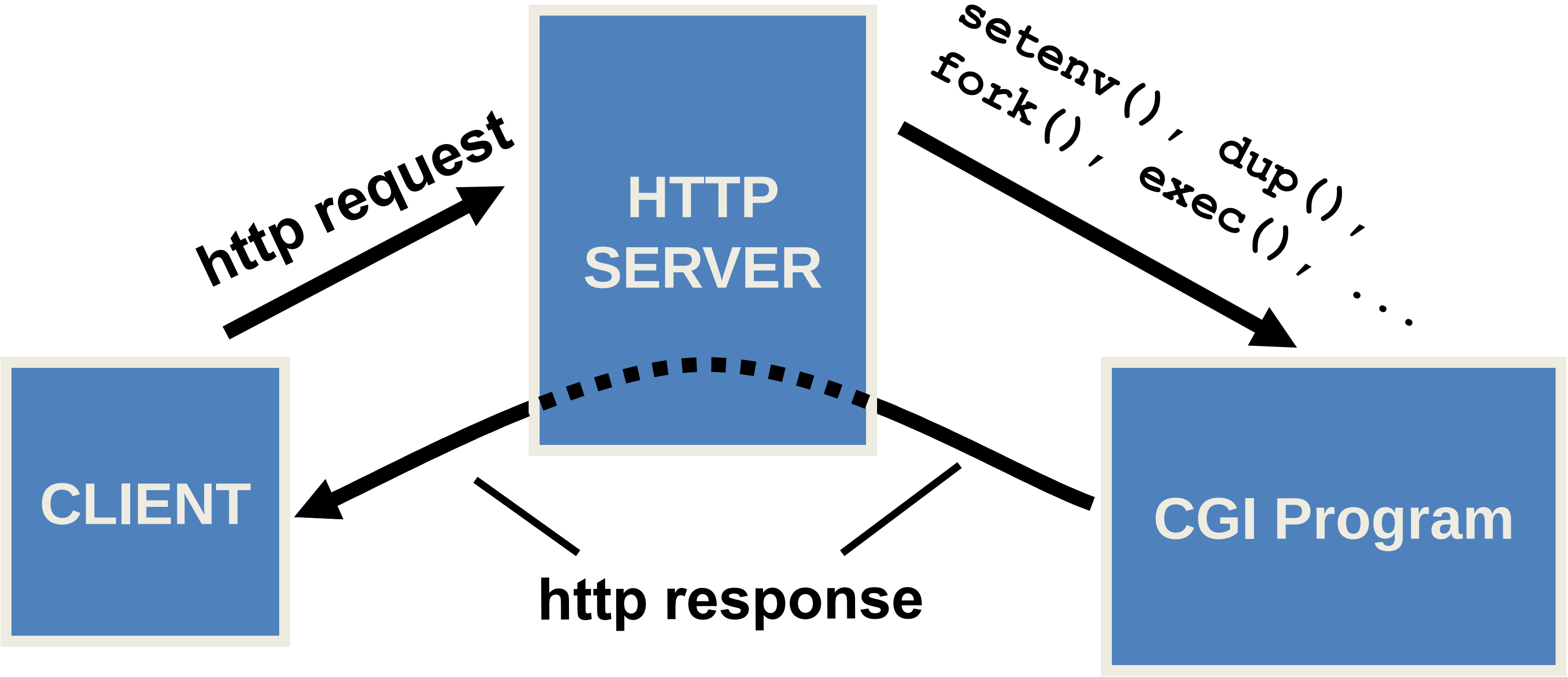
```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Fri, 08 Oct 2010 05:08:14 GMT
Connection: open
Content-length: 175
Content-Type: text/html

content is very big. ...................
```

- In dynamic output cases



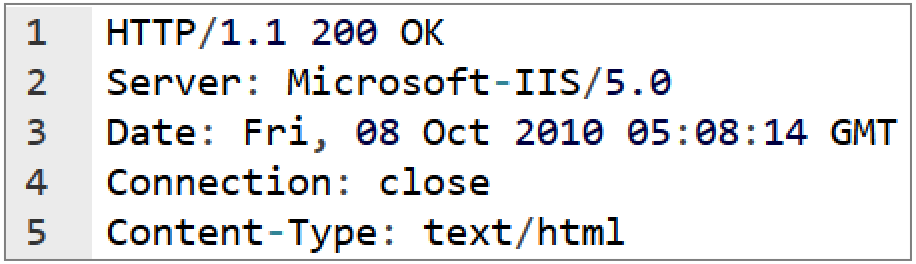
```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Fri, 08 Oct 2010 05:08:14 GMT
Connection: close
Content-Type: text/html
```

# CGI Programming



# HTTP & Dynamic Outputs

- It may  not be convenient or even possible for  a  server to know  the length of  an item before sending.
- Servers use the Common Gateway Interface  (CGI) mechanism to create dynamic documents.
- To provide for dynamic Web pages, the HTTP standard specifies that if  the server does not know the length of  an item a priori,  the server can inform the browser that it will close the connection after transmitting the item



```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Fri, 08 Oct 2010 05:08:14 GMT
Connection: close
Content-Type: text/html
```

# HTTP & Dynamic Outputs

- Closing connection results in poor performance.
- If server doesn’t know the output length a priori, it can also use chunked transfer encoding.
- The sender breaks the message body into chunks of arbitrary length, and each chunk is sent with its length prepended;
- It marks the end of the message with a zero-length chunk.
- The sender uses the *Transfer-Encoding: chunked* header to signal the use of chunking.
- This mechanism allows the sender to buffer small pieces of the message, instead of the entire message, without adding much complexity or overhead.

# Acknowledgements

# Q&A

innovate achieve lead

BITS Pilani
Pilani Campus

# Thank You