

LEC04. ELEMENTARY SOCKETS

Bui Trong Tung, SoICT, HUST

1

Content

- socket()
- UDP Socket APIs
- TCP Socket APIs
- Iterative TCP Server
- Design application protocol

2

socket()

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- Creates an endpoint for communication
- [IN] domain: AF_INET, AF_INET6, or AF_UNSPEC, ...
- [IN] type argument can be:
 - SOCK_STREAM: Provides sequenced, reliable, two-way, connection-based byte streams
 - SOCK_DGRAM: Supports datagrams
 - SOCK_RAW: Provides raw network protocol access
- [IN] protocol is usually 0
- Returns value
 - A new socket descriptor that you can use to do sockety things with
 - If error occurs, return -1 (remember **errno**)

3

bind()

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

- Associate a socket with an IP address and port number
- Where
 - [IN] sockfd : socket descriptor
 - [IN] addr : pointer to a `sockaddr` structure assigned to sockfd
 - [IN] addrlen : specifies the size, in bytes of address structure pointed to by addr
- Return value
 - Returns 0 if no error occurs.
 - Otherwise, return -1 (and **errno** will be set accordingly)

4

shutdown()

```
#include <sys/socket.h>
int shutdown(int socket, int how);
```

- Shut down socket send and receive operations
- Where
 - [IN] `sockfd`: a descriptor identifying a socket.
 - [IN] `how`: `SHUT_RD`, `SHUT_WR`, `SHUT_RDWR`
- Return value
 - Returns 0 if no error occurs.
 - Otherwise, return -1 (and **errno** will be set accordingly)

5

close()

```
#include <unistd.h>
int close(int sockfd);
```

- Close a socket descriptor
- [IN] `sockfd`: a descriptor identifying a socket.
- Return value
 - Returns 0 if no error occurs.
 - Otherwise, return -1 (and **errno** will be set accordingly)
- `close()` vs `shutdown()`
 - `close()` tries to complete this transmission before closing, frees the socket descriptor
 - `shutdown()`: immediately stops receiving and transmitting data, don't releases the socket descriptor

6

Socket options

```
#include <sys/socket.h>
int setsockopt (int sockfd, int level, int optname,
               void *optval, int optlen);
```

- Set the options that control the transferring data on a socket
- Parameters:
 - [IN] `sockfd`: refer to an open socket descriptor
 - [IN] `level`: specifies the protocol level at which the option resides
 - [IN] `optname`: specifies a single option to set
 - [IN] `optval`: points to the setted option value
 - [IN] `optlen`: the size of option value pointed by `optval`
- Return:
 - Returns 0 if no error occurs.
 - Otherwise, return -1 (and **errno** will be set accordingly)

7

Socket options(cont)

```
#include <sys/socket.h>
int getsockopt (int sockfd, int level, int optname,
               void *optval, int *optlen);
```

- Set the options that control the transferring data on a socket
- Parameters:
 - [IN] `sockfd`: refer to an open socket descriptor
 - [IN] `level`: specifies the protocol level at which the option resides
 - [IN] `optname`: specifies a single option to set
 - [OUT] `optval`: points to the setted option value
 - [IN, OUT] `optlen`: the size of option value pointed by `optval`
- Return:
 - Returns 0 if no error occurs.
 - Otherwise, return -1 (and **errno** will be set accordingly)

8

level = SOL_SOCKET

Value name	Type	Description
SO_BROADCAST	int	Configures a socket for sending broadcast data.(Only UDP socket)
SO_DONTROUTE	int	Sets whether outgoing data should be sent on interface the socket is bound to and not a routed on some other interface
SO_KEEPALIVE	int	TCP automatically sends a keep-alive probe to the peer
SO_LINGER	linger	specifies how the close function operates for a connection-oriented protocol
SO_REUSEADDR	int	Allows the socket to be bound to an address that is already in use
SO_RCVTIMEO	timeval	Sets the timeout for blocking receive calls
SO_SNDTIMEO	timeval	Sets the timeout for blocking send calls

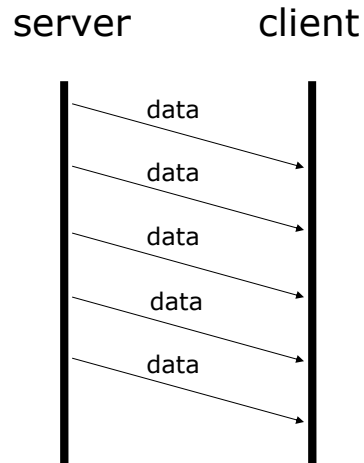
9

UDP SOCKET

10

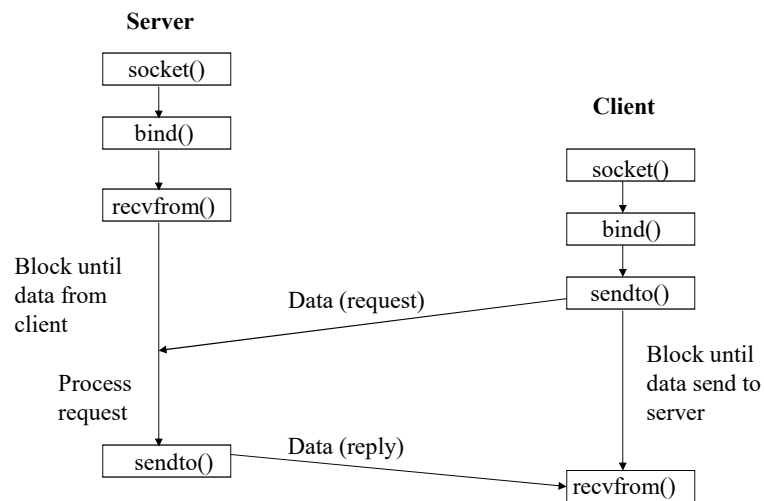
UDP (User Datagram Protocol)

- No reliable
- No flow control
- Familiar example
 - DNS
 - Streaming
- Image
 - Postcard exchange



11

UDP client/server



12

recvfrom()

```
ssize_t recvfrom(int sockfd, void *buf, size_t len,  
int flags, struct sockaddr *from, socklen_t *fromlen );
```

- Received data from a socket
- Parameters:
 - [IN] `sockfd`: the socket file descriptor
 - [OUT] `buf`: the buffer where the message should be stored
 - [IN] `len`: the size of the buffer
 - [IN] `flags`: how to control `recvfrom` function work
 - [OUT] `from`: the address of the sender
 - [OUT] `fromlen`: the size of sender's address
- Return:
 - Success: return the length of the received data in bytes. If the incoming message is too long to fit in the supplied buffer, the excess bytes shall be discarded.
 - Error: -1 and set **errno** to indicate the error.

13

recvfrom() - Flags

- **MSG_PEEK**: Peeks at an incoming message. The data is treated as unread and the next `recvfrom()` or similar function shall still return this data.
- **MSG_OOB**: Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
- **MSG_WAITALL**: On **SOCK_STREAM** sockets this requests that the function block until the full amount of data can be returned, excepting:
 - the connection is terminated
 - **MSG_PEEK** was specified
 - an error is pending for the socket
 - a signal is caught
- Use bitwise OR operator (`|`) to combine more than one flag

14

sendto ()

```
ssize_t sendto(int sockfd, void *buf, size_t len, int flags,  
               struct sockaddr *to, socklen_t *tolen );
```

- Received data from a socket
- Parameters:
 - [IN] `sockfd`: the socket file descriptor
 - [IN] `buf`: points to a buffer containing the message to be sent
 - [IN] `len`: the size of the message
 - [IN] `flags`: how to control `sendto` function work
 - [IN] `to`: the address of the receiver
 - [IN] `tolen`: the length of the `sockaddr` structure pointed to by the `to` argument
- Return:
 - Success: shall return the length of the sent message in bytes
 - Error: -1 and set **errno** to indicate the error.

15

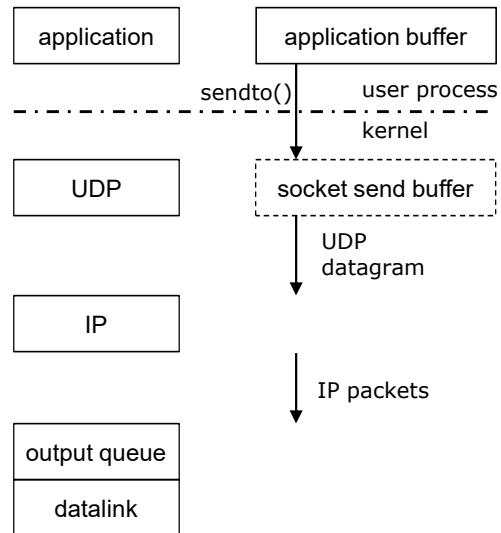
sendto () - Flags

- `MSG_OOB`: Sends out-of-band data on sockets that support out-of-band data.
- `MSG_DONTROUTE`: Don't use a gateway to send out the packet, only send to hosts on directly connected networks
- Use bitwise OR operator (`|`) to combine more than one flag

16

sendto ()

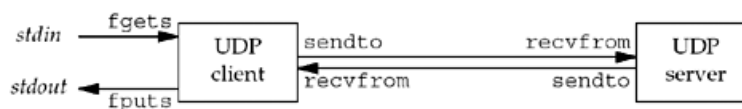
- UDP socket buffer doesn't really exist
- UDP socket buffer has a send buffer size
- If an application writes a datagram larger than the socket send buffer size, `EMSGSIZE` is returned



17

Example

- A simple UDP client and server
 - Server receives data from client
 - Server sends back data to client
 - It present in `udpserv01.c` and `dg_echo.c`



18

Example – UDP Echo Server

```
int sockfd, rcvBytes, sendBytes;
socklen_t len;
char buff[BUFF_SIZE+1];
struct sockaddr_in servaddr, cliaddr;

//Step 1: Construct socket
if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
    perror("Error: ");
    return 0;
}

//Step 2: Bind address to socket
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
if(bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr))){
    perror("Error: ");
    return 0;
}

printf("Server started.");
```

19

Example – UDP Echo Server(cont)

```
//Step 3: Communicate with client
for ( ; ; ) {
    len = sizeof(cliaddr);
    rcvBytes = recvfrom(sockfd, buff, BUFF_SIZE, 0,
        (struct sockaddr *) &cliaddr, &len);
    if(rcvBytes < 0){
        perror("Error: ");
        return 0;
    }
    buff[rcvBytes] = '\0';
    printf("[%s:%d]: %s", inet_ntoa(cliaddr.sin_addr),
        ntohs(cliaddr.sin_port), mesg);

    sendBytes = sendto(sockfd, buff, rcvBytes, 0,
        (struct sockaddr *) &cliaddr, len);
    if(sendBytes < 0){
        perror("Error: ");
        return 0;
    }
}
```

20

Example – UDP Echo Client

```
int sockfd, rcvBytes, sendBytes;
socklen_t len;
char buff[BUFF_SIZE+1];
struct sockaddr_in servaddr;

//Step 1: Construct socket
if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
    perror("Error: ");
    return 0;
}

//Step 2: Define the address of the server
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr = inet_aton(SERV_ADDR, &servaddr.sin_addr);
servaddr.sin_port = htons(SERV_PORT);
```

21

Example – UDP Echo Client(cont)

```
//Step 3: Communicate with server
printf("Send to server: ");
gets_s(buff, BUFF_SIZE);

len = sizeof(servaddr);
sendBytes = sendto(sockfd, buff, strlen(buff), 0,
                  (struct sockaddr *) &servaddr, len);
if(sendBytes < 0){
    perror("Error: ");
    return 0;
}

rcvBytes = recvfrom(sockfd, buff, BUFF_SIZE, 0,
                   (struct sockaddr *) &servaddr, &len);
if(rcvBytes < 0){
    perror("Error: ");
    return 0;
}
buff[rcvBytes] = '\0';
printf("Reply from server: %s", buff);
```

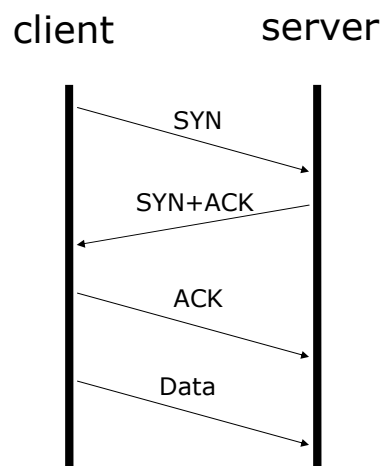
22

TCP SOCKET

23

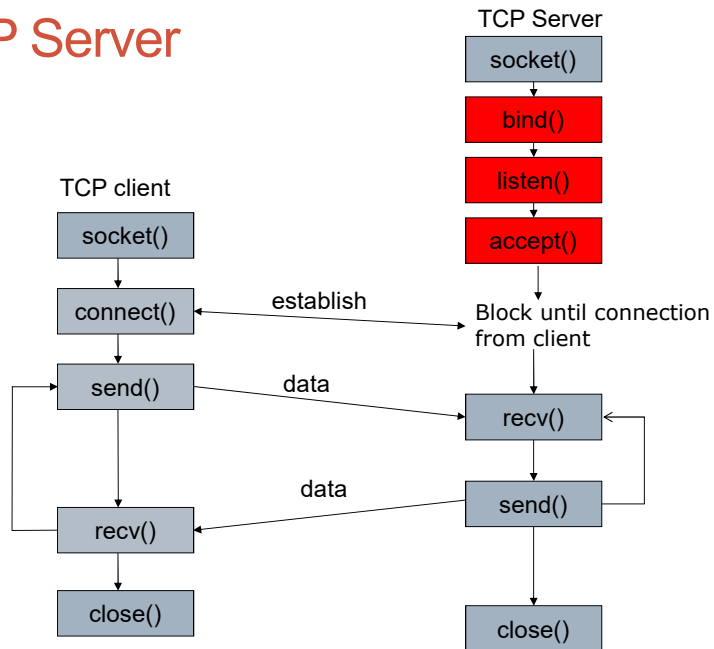
TCP (Transmission Control Protocol)

- Provide reliable communication
- Data rate control
- Example
 - Mail
 - WEB
 - Image



24

TCP Server



25

TCP server side

1. Create a socket – `socket()`.
 2. Bind the socket – `bind()`.
 3. Listen on the socket – `listen()`.
 4. Accept a connection – `accept()`.
 5. Send and receive data – `recv()`, `send()`.
 6. Disconnect connection– `close()`
 7. Close LISTENING socket
- } repeatedly

26

listen()

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

- Establish a socket to LISTENING for incoming connection.
- Parameters:
 - [IN] `sockfd`: a descriptor identifying a bound, unconnected socket
 - [IN] `backlog`: the queue length for *completely* established sockets waiting to be accepted
- Return value
 - On success, 0 is returned
 - On error, -1 is returned, and `errno` is set appropriately

27

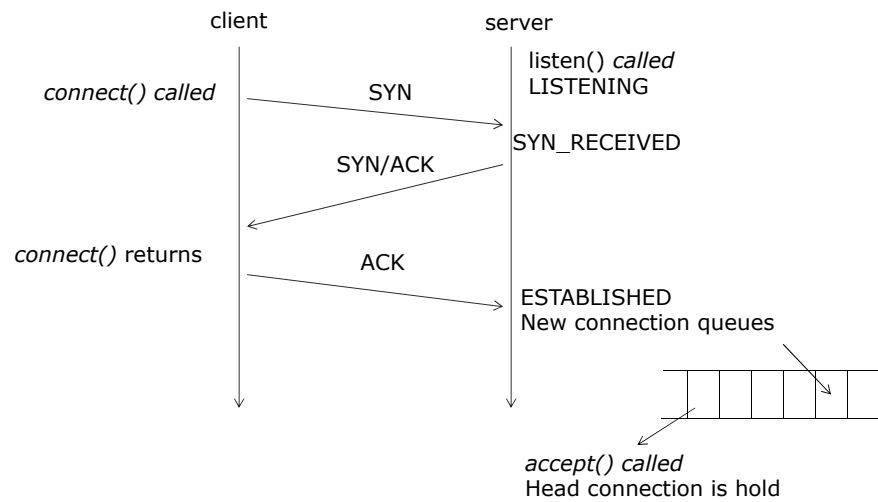
accept()

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- Accept an incoming connection on a LISTENING socket
- Parameters:
 - [IN] `sockfd`: A descriptor identifying a socket which is listening for connections after a `listen()`.
 - [OUT] `addr`: pointer to a `sockaddr` structure filled in with the address of the peer socket
 - [IN, OUT] `addrlen`: the caller must initialize it to contain the size (in bytes) of the structure pointed to by `addr`; on return it will contain the actual size of the peer address.
- Return value
 - Newly connected socket descriptor if no errors
 - -1 if has errors

28

Process connections



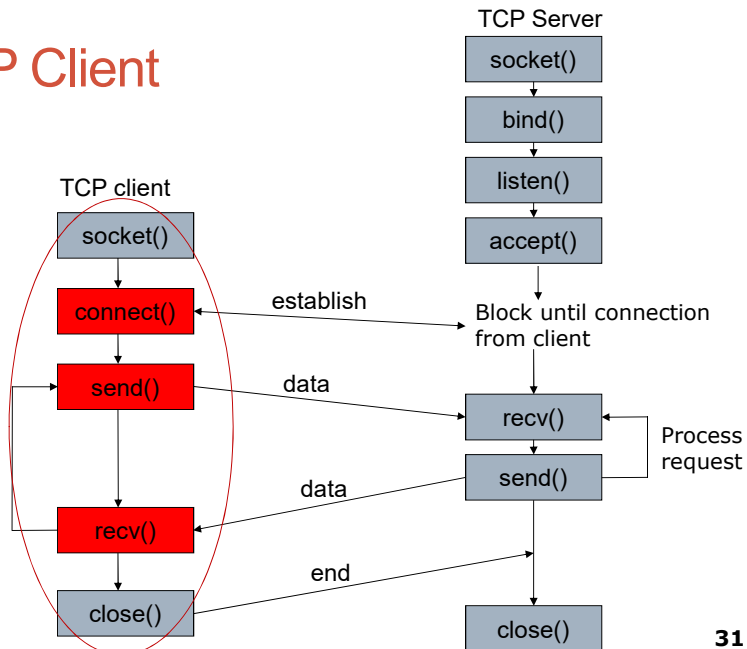
29

Socket Mode

- Types of server sockets
 - *Iterating server*: Only one socket is opened at a time.
 - *Forking server*: After an accept, a child process is forked off to handle the connection.
 - *Concurrent single server*: use select to simultaneously wait on all open sockets, and waking up the process only when new data arrives

30

TCP Client



31

TCP client side

- The typical TCP client's communication involves four basic steps:
 - Create a TCP socket using `socket()`.
 - Establish a connection to the server using `connect()`.
 - Communicate using `send()` and `recv()`.
 - Close the connection with `close()`.
- Why "clients" doesn't need `bind()` ?

32

connect()

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

- Connect a socket to a server
- Parameters:
 - [IN] sockfd: A descriptor identifying an unconnected socket.
 - [IN] serv_addr: The address of the server to which the socket is to be connected.
 - [IN] addrlen: The length of the name.
- Return value
 - If no error occurs, returns 0.
 - Otherwise, it returns -1

33

send()

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t len,
             int flags);
```

- Send data on a connected socket
- Parameter:
 - [IN] sockfd: a descriptor identifying a connected socket.
 - [IN] buf: points to the buffer containing the message to send.
 - [IN] len: specifies the length of the message
 - [IN] flags: specifies the type of message transmission, usually 0
- Return value:
 - If no error occurs, send() returns the total number of characters sent
 - Otherwise, return -1

34

send () - Flags

- MSG_OOB: Send as “out of band” data. The receiver will receive the signal SIGURG and it can then receive this data without first receiving all the rest of the normal data in the queue.
- MSG_DONTROUTE :Don't send this data over a router, just keep it local.
- MSG_DONTWAIT: If **send()** would block because outbound traffic is clogged, have it return EAGAIN. This is like a “enable non-blocking just for this send.”
- MSG_NOSIGNAL: If you **send()** to a remote host which is no longer **recv()**, you'll typically get the signal SIGPIPE. Adding this flag prevents that signal from being raised.

35

send () – Data size is greater buffer's

```
char sendBuff[2048];
int  dataLength, nLeft, idx;

// Fill sendbuff with 2048 bytes of data
nLeft = dataLength;
idx = 0;

while (nLeft > 0){
    // Assume s is a valid, connected stream socket
    ret = send(s, &sendBuff[idx], nLeft, 0);
    if (ret == -1)
    {
        // Error handler
    }
    nLeft -= ret;
    idx += ret;
}
```

36

recv()

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

- Receive data on a socket
- Parameter:
 - [IN] `sockfd`: a descriptor identifying a connected socket.
 - [IN, OUT] `buf`: points to a buffer where the message should be stored
 - [IN] `len`: specifies the length in bytes of the buffer
 - [IN] `flags`: specifies the type of message reception, usually 0
- Return value:
 - If no error occurs, returns the length of received message in bytes
 - If peer has performed an orderly shutdown, return 0
 - Otherwise, return -1

37

recv() - Flags

- `MSG_PEEK`: Peeks at an incoming message. The data is treated as unread and the next `recvfrom()` or similar function shall still return this data.
- `MSG_OOB`: Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
- `MSG_WAITALL`: On `SOCK_STREAM` sockets this requests that the function block until the full amount of data can be returned, excepting:
 - the connection is terminated
 - `MSG_PEEK` was specified
 - an error is pending for the socket
 - a signal is caught
- Use bitwise OR operator (`|`) to combine more than one flag

38

Examp – TCP Echo Server

```
int listenfd, connfd;
char buff[BUFF_SIZE+1];
struct sockaddr_in servAddr;

//Step 1: Construct socket
listenfd = socket(AF_INET, SOCK_STREAM, 0);

//Step 2: Bind address to socket
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
if(bind(listenfd, (struct sockaddr *) &servAddr,
        sizeof(servAddr))) {
    perror("Error: ");
    return 0;
}
```

39

Examp – TCP Echo Server(cont)

```
//Step 3: Listen request from client
if(listen(listenfd, 10)){
    perror("Error! Cannot listen.");
    return 0;
}
printf("Server started!");

//Step 4: Communicate with client
sockaddr_in clientAddr;
int rcvBytes, sendBytes, clientAddrLen = sizeof(clientAddr);
while(1){
    //accept request
    connfd = accept(listenfd, (sockaddr *) & clientAddr,
                    &clientAddrLen);
```

40

Examp – TCP Echo Server(cont)

```
//receive message from client
rcvBytes = recv(connfd, buff, BUFF_SIZE, 0);
if(rcvBytes < 0){
    perror("Error :");
}
else{
    buff[rcvBytes] = '\0';
    printf("Receive from client[%s:%d] %s\n",
        inet_ntoa(clientAddr.sin_addr),
        ntohs(clientAddr.sin_port), buff);
    //Echo to client
    sendBytes = send(connfd, buff, strlen(buff), 0);
    if(sendBytes < 0)
        perror("Error: ",);
}
closesocket(connfd);
} //end while
```

41

Examp – TCP Echo Client

```
int clientfd;
char buff[BUFF_SIZE+1];
struct sockaddr_in servaddr;

//Step 1: Construct socket
clientfd = socket(AF_INET, SOCK_STREAM, 0);

//Step 2: Specify server's address
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(SERV_ADDR);
servaddr.sin_port = htons(SERV_PORT);

//Step 4: Connect server
if(connect(clientfd, (sockaddr *) &servaddr,
    sizeof(servaddr))) {
    perror("Error: ");
    return 0;
}
```

42

Examp – TCP Echo Client(cont)

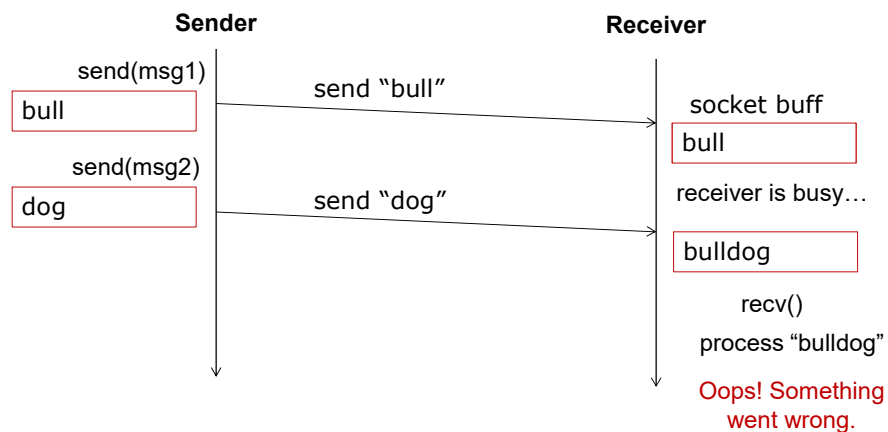
```
//Step 5: Communicate with server
char buff[BUFF_SIZE];
int ret;
//Send message
printf("Send to server: ");
gets_s(buff, BUFF_SIZE);
ret = send(clientfd, buff, strlen(buff), 0);
if(ret < 0){
    perror("Error: ");
    return 0;
}
//Receive echo message
ret = recv(clientfd, buff, BUFF_SIZE, 0);
if(ret < 0){
    perror("Error: ");
    return 0;
}
printf("Receive from server: %s\n", buff);
close(clientfd);
return 0;
```

43

Byte stream problem

TCP does not operate on *packets* of data.

TCP operates on *streams* of data.

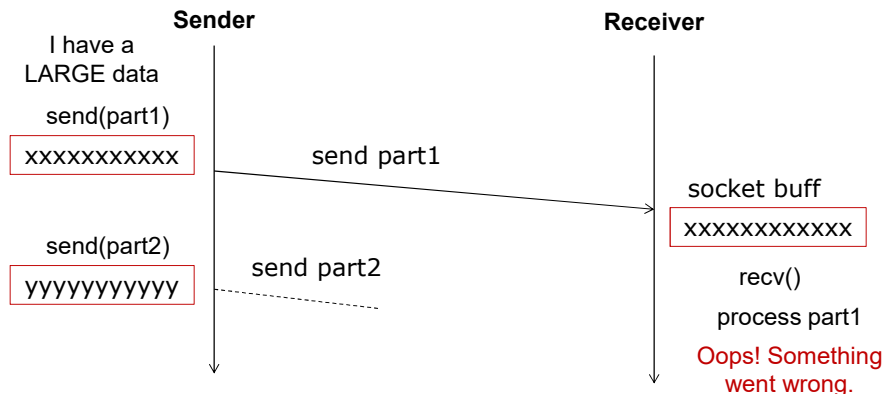


44

Byte stream problem(cont)

TCP does not operate on *packets* of data.

TCP operates on *streams* of data.

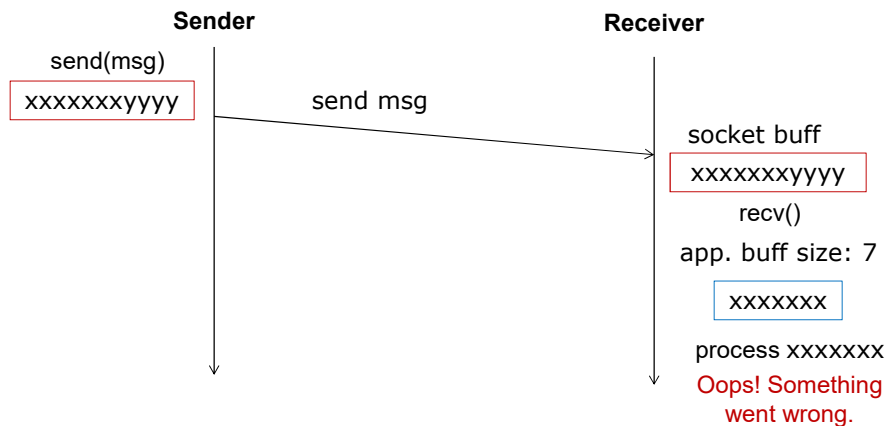


45

Byte stream problem(cont)

TCP does not operate on *packets* of data.

TCP operates on *streams* of data.

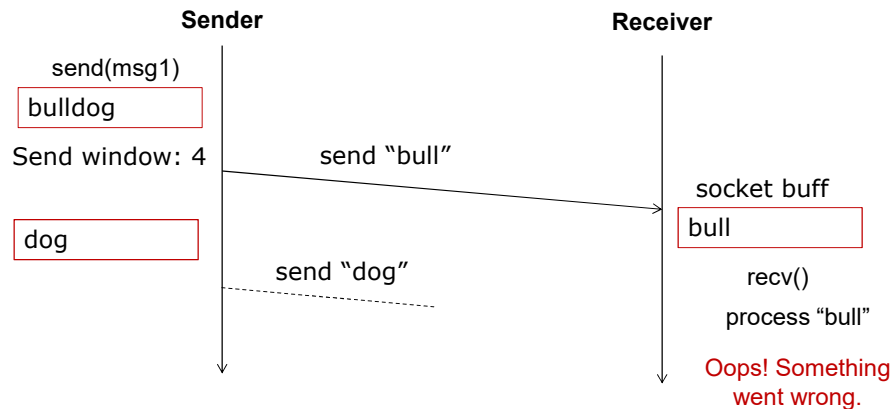


46

Byte stream problem(cont)

TCP does not operate on *packets* of data.

TCP operates on *streams* of data.



47

Byte stream problem(cont)

- Receiver doesn't know the size of message that sender has sent
- Solution 1: Fixed-length message
 - What length? How to pad?
- Solution 2: Delimiters
 - But message also can contain delimiters
 - Complex!
- Solution 3: Length prefixing
 - Send message with its length
 - Receiver:
 - `recv(..., n, MSG_WAITALL)` returns the length of the message
 - Receives data (next slide)



48

Byte stream problem(cont)

```
char    recvBuff[BUFF_SIZE], *data;
int     ret, nLeft;
nLeft = msgLength; //length of the data needs to be received
data = (char *) malloc(msgLength);
memset(data, 0, msgLength)
idx = 0;

while (nLeft > 0)
{
    ret = recv(s, &recvBuff, BUFF_SIZE, 0);
    if (ret == -1){
        // Error handler
        break;
    }
    idx += ret;
    memcpy(data + idx, recvBuff, ret)
    nLeft -= ret;
}
```

49

connect () with UDP

- If server isn't running, the client blocks forever in the call to `recvfrom()` → asynchronous error
- Use `connect()` for a UDP socket
 - But it's different from calling `connect()` on a TCP socket
 - Calling `connect()` on a UDP socket doesn't create a connection
 - The kernel just checks for any immediate errors and returns immediately to the calling process
- We do not use `sendto()`, but `write()` or `send()` instead
- We do not need to use `recvfrom()` to learn the sender of a datagram, but `read()`, `recv()` instead
- Asynchronous errors are returned to the process for connected UDP sockets

50

Example

```
int n;
char sendline[MAXLINE], recvline[MAXLINE + 1];
struct sockaddr_in servaddr;
connect(sockfd, (struct sockaddr *) &servaddr, servlen);
while (fgets(sendline, MAXLINE, fp) != NULL) {
    send(sockfd, sendline, strlen(sendline));
    n = recv(sockfd, recvline, MAXLINE);
    recvline[n] = 0; /* null terminate */
    printf("%s", recvline);
}
```

51

APPLICATION PROTOCOL DESIGN

52

Protocol

- Set of rules:
 - Message format
 - Message sequence
 - Process message
- Goals
 - Everyone must know
 - Everyone must agree
 - Unambiguous
 - Complete

53

Example: POP session

```
C: <client connects to service port 110>
S: +OK POP3 server ready <1896.6971@mailgate.dobbs.org>
C: USER bob
S: +OK bob
C: PASS redqueen
S: +OK bob's maildrop has 2 messages (320 octets)
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: QUIT
S: +OK dewey POP3 server signing off (maildrop empty)
C: <client hangs u>
```

54

Example: FTP authentication

```
> ftp 202.191.56.65
C: Connected to 202.91.56.65
S: 220 Servers identifying string
User: tungbt (C: USER tungbt)
S: 331 Password required for tungbt
Password: (C: PASS)
S: 530 Login incorrect
C: ls
S: 530 Please login with USER and PASS
C: USER tungbt
S: 331 Password required for tungbt
Password: (C: PASS)
S: 230 User tungbt logged in
```

55

Steps in design

1. Define services
2. Choose application model(client/server, P2P,...)
3. Establish the design goals
4. Design the message structure: format, fields, types of messages, encoding, ...
5. Protocol processing
6. Interaction with environment (DNS, DHCP...)

56

Design Goals

- Do we need reliable exchanges?
- How many types of parties are involved? Can they all communicate to each other?
- Is the authentication of parties needed
- How important is the authentication of parties?
- Is the transferred data confidential? What degree of authorization is needed?
- Do we need complex error handling?

57

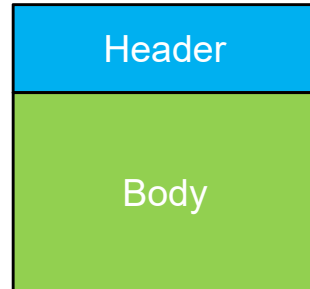
Design Issues

- Is it to be stateful vs stateless?
- Is the transport protocol reliable or unreliable?
- Are replies needed?
 - How to respond to lost replies?
- Is it to be broadcast, multicast or unicast?
 - Broadcast, multicast: must use UDP Socket
- Are there multiple connections?
 - How to synchronize?
- How many types of parties are involved? Can they all communicate to each other?
- Session management
- Security: authentication, authorization, confidential...

58

Designing the Message

- Header: contains structured fields describing the actual data in the message, such as
 - message type
 - command
 - body size
 - recipient information
 - sequence information
 - retransmission count...
- Body: the actual data to be transmitted:
 - the command parameters
 - the data payload



The simplest formats:

- Type – Length – Value(TLV)
- Type – Value

59

Control Messages

- Define the stages of the dialogue between the parties
 - Control the dialogue between the parties
- Address various communication aspects:
- communication initiation or ending
 - describe the communication stage (e.g. authentication, status request, data transfer)
 - coordination (e.g. receipt confirmation, retry requests)
 - resource changes (e.g. requests for new communication channels)
- Usual format:

Command	Parameter
---------	-----------

 - Command: SHOULD has fix length or use delimiter
 - Example: USER, PASS, PWD (FTP),

60

Data transfer

- Messages that carry data over the network
- They are usually sent as a responses to specific commands
- Data is usually fragmented in multiple messages
- Header describe:
 - the type of the binary data format
 - clues for the layout of the structured data (when the structure is flexible/dynamic)
 - data size, offset or sequence information
 - type of the data block: last / intermediary
 -

61

Message Format

Byte oriented

- The first part of the message is typically a byte to distinguish between message types.
- Further bytes in the message would contain message content according to a pre-defined format
- Advantages: compactness
- Disadvantages: harder to process, debug or test
- Example: DHCP, DNS

62


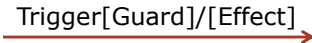

Data Format

Text-oriented

- A message is a sequence of one or more lines
- The start of the first line of the message is typically a word that represents the message type.
- The rest of the first line and successive lines contain the data.
- Advantage:
 - easy to understand, monitor
 - flexible
 - easy to test
- Disadvantage
 - may make the messages unjustifiably large
 - may become complex
- Example: HTTP, FTP, email protocols

63

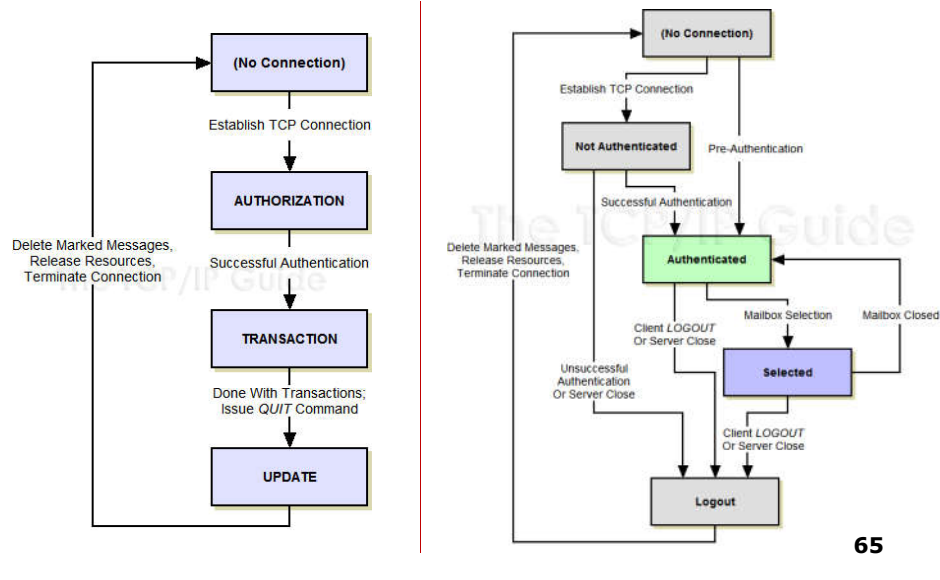
Protocol Processing

- Describe the sequences of messages, at each and all the stages in the of each communication scenario, for all parties in the system
- Finite State Machine is mandatory:
 - State: 
 - Transaction: 
 - Choose: 
- And/ Or use state Table

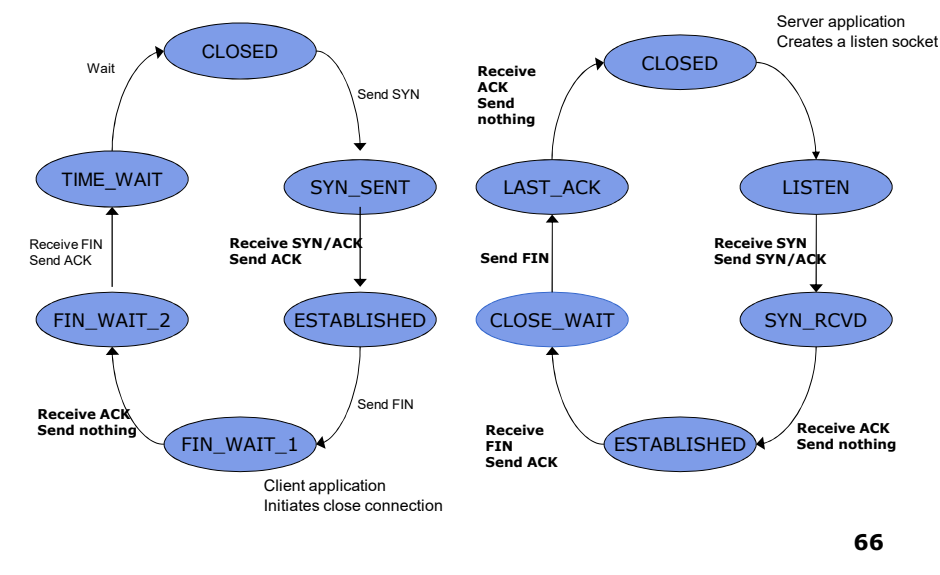
Current state	Transcation		Next state
	Receive	Send	

64

Example: POP3 and IMAP4 session

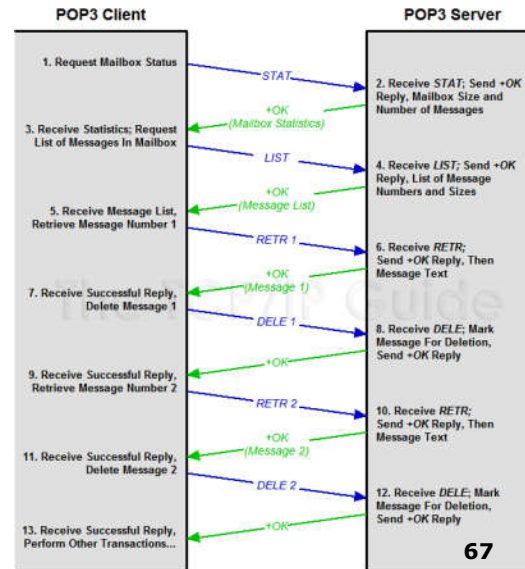


Example: TCP connection



Message Transaction Diagram

- Represents the sequence of message transaction
- Example: POP3



Implementing an Application Protocol

- Type of message
 - Use integer: `enum msg_type {...}`
 - Use string
- Data structure
 - Use struct. Example:

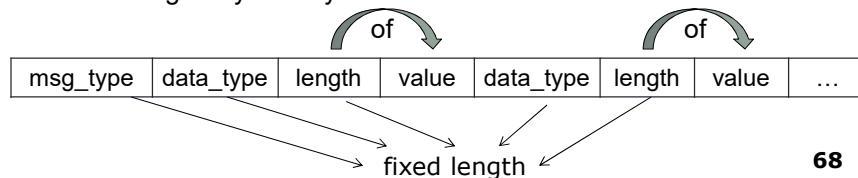
```
struct message{
    char msg_type[4];
    char data_type[8];
    int value;
}
```

or

```
struct message{
    msg_type type;
    struct msg_payload payload;
};
```

```
struct msg_payload{
    int id;
    char fullname[30];
    int age;
    //...
}
```

- Use string or byte array



Implementing an Application Protocol

- Message handler(pseudo code)

```
//handle message
switch (msg_type){
    case MSG_TYPE1:
        {
            //...
        }
    case MSG_TYPE2:
        {
            //...
            if(data_type == DATA_TYPE1)
                //...
        }
    //...
}
```

69