



Course	Sockets and MMU	
Module Day 8		3 Hours
	<p>At the end of this module you will be able to know:</p> <ul style="list-style-type: none"> ▪ What is a Socket? ▪ What are the advantages of using Sockets? ▪ Different protocols used in Socket programming? ▪ System Calls related to Sockets ▪ TCP/IP server-client model ▪ Make file 	

Session Plan	
1	Recap of Last class, Introduction to Sockets, advantages, different protocols used in Sockets, System calls, server-client model using TCP/IP, Examples.
2	Make file
3	Revision

	Module Overview
---	-----------------

- ✓ Introduction to Sockets
- ✓ Required Protocols
- ✓ Server Client Communication
- ✓ Server-Client Model
- ✓ Client API's
- ✓ Server API's
- ✓ Make file



Introduction to Sockets

- A Socket is an end point of communication between two systems on a network.
- To be a bit precise, a socket is a combination of IP address and port on one system.
- So on each system a socket exists for a process interacting with the socket on other system over the network.
- If a process wants to communicate with the different network processes, we have to use Sockets.
- Sockets are interfaces that can "plug into" each other over a network .
- Sockets are nothing but files
- Each connection between two processes running at different systems can be uniquely identified through their:
 - ✚ Destination IP Address
 - ✚ Source IP Address
 - ✚ Destination Port number
 - ✚ Source Port Number
 - ✚ Protocol
- IP Addresses are used to identify the **networks**.
- Port numbers are used to identify the **processes**.



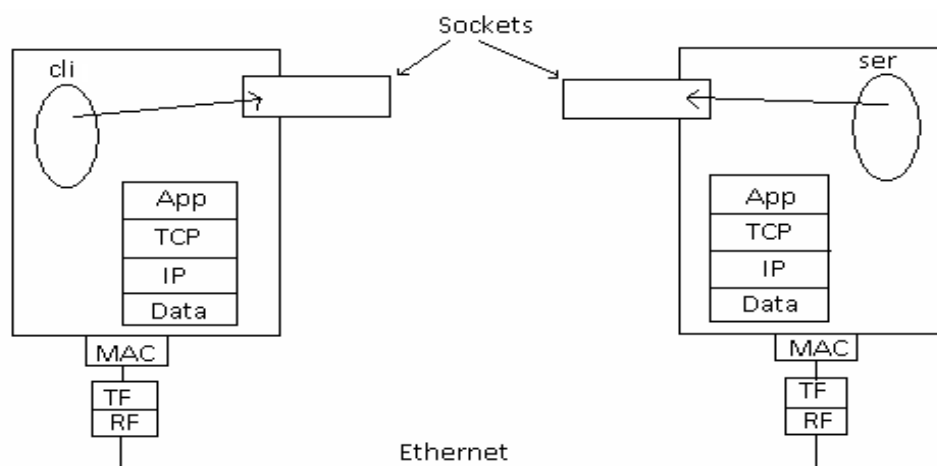
Required Protocols

- Socket programming can be done by using two protocols:
 - USER DATAGRAM PROTOCOL (UDP),
 - TRANSMISSION CONTROL PROTOCOL (TCP).
- Programmer can choose a TCP (connection-oriented) server or a UDP (connectionless) server based on their applications.

UDP	TCP
1. Is a connectionless.	1. Is a connection-oriented.
2. A single socket can send and receive packets from many different computers.	2. A client must connect a socket to a server.
3. Best effort delivery.	3. TCP socket provides bidirectional channel between client and server.
4. Some packets may be lost some packets may arrive out of order.	4. Lost data is re-transmitted.
5. Data is delivered not in-order.	5. Data is delivered in-order.
6. Data is delivered as blocks.	6. Data is delivered as a stream of bytes.



Server-Client Communication



- When client writes the data to socket, the data will be received by the TCP layer

- TCP layer will divide data into packets and will construct a structure called TCP Header, and stores Source & Destination Port numbers along with data (one packet).

Source P N
Dest. P N
Data

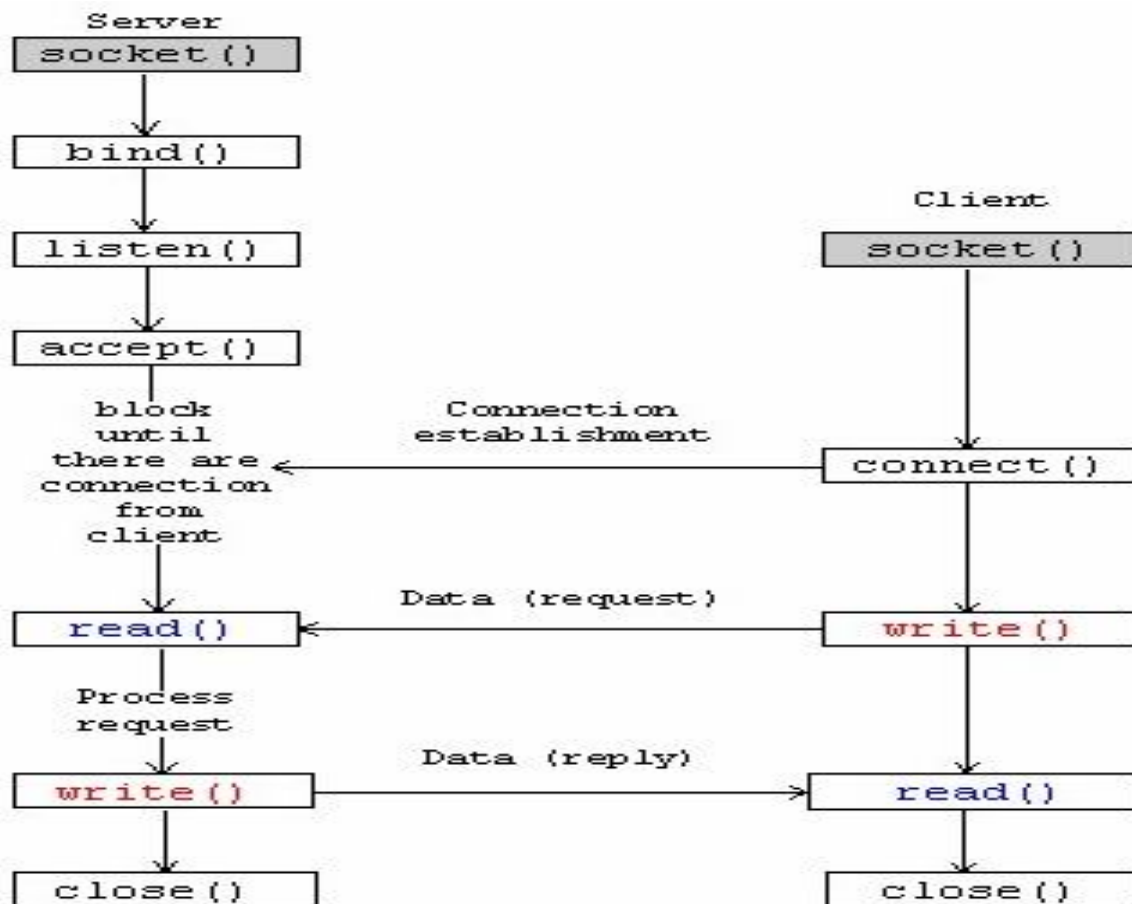
TCP Header

- The TCP Header will sent to IP layer. In IP layer it will add Source and Destination IP addresses to TCP Header, called as IP Header.
- IP Header is given to Media Access Controller (MAC) through Data layer
- MAC will give that data to Transmit FIFO.
- After sending a Packet, TCP will allocate **Round Trip Time (RTT)** as 2ms. Within this time acknowledgement should be received, Otherwise TCP will send the same Packet again until acknowledgement received.



Server-Client Model

The following figure illustrates the example of client/server relationship of the socket APIs for connection-oriented protocol (TCP).



Explanation:-

As shown in the above figure, In Socket programming if server want to communicate with client, then both should be created their own socket. Here are the steps to know how to implement Server-Client Model:

- In Server: It has to create a socket by using `socket()` system call

- In server: Assign valid address (IP address and Port number) to that socket using `bind()` system call. After assigning address, then this socket is visible to all systems in the network.
- In server: Wait for incoming connections from different clients using `listen()` system call.
- In client: create a socket using `socket()` system call.
- In client: send a request to server socket by assigning client address (IP address and Port number) to socket using `connect()` system call.
- In server: accept client request by using `accept()` systemcall. Now connection between server and client is established.
- Now communicate from server to client and vice versa using `send()` and `recv()` system calls.
- After completing communication then end the communication using `close()` system call.]



System Calls related to Server-Client Model

Client API's:

1. `Socket()` ;
2. `Connect()` ;
3. `Send()`
4. `Recv()` ;
5. `Close`

Required Header Files:

```
#include <sys/types.h>
#include <sys/socket.h>
```

Creating a Socket:

In socket programming first thing to do is create a socket. The `socket()` function does this. This function declaration is given below:

```
int socket (int domain, int type, int protocol);
```

Here:

- The domain argument specifies a communication domain.

Name	Purpose
AF_UNIX, AF_LOCAL	Local Communication
AF_INET	IPV4 Internet protocols
AF_INET6	IPV6 Internet protocols
AF_IPX	IPX-Novell Protocols
AF_NETLINK	Kernel user interface device

- The socket has the indicated type, which specifies the communication semantics. Currently defined types are:
 - **SOCK_STREAM**: Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.
 - **SOCK_DGRAM**: Supports data grams (connectionless, unreliable messages of a fixed maximum length).
 - **SOCK_SEQPACKET**: Provides a sequenced, reliable, two-way connection-based data transmission path for data grams of fixed maximum length.
- The **protocol** specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as **0**.
- However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner.

Ex:- /***** WAP to create a socket *****/

Header files

```
int main () {
    int socket_desc;
    socket_desc = socket (AF_INET, SOCK_STREAM, 0);
    if (socket_desc == -1) {
        printf("Could not create socket");    }
    return 0;
```

```
}
```

Sending request to server:- To send a connection request to server, we have to use `connect()` function. This function declaration is given below:

```
int connect (int sockfd, const struct sockaddr *addr,
socklen_t addrlen);
```

Before going to understand the declaration of this function, understand the following points:

- We connect to a remote server on a certain port number. So we need 2 things , IP address and port number to connect.
- To connect to a remote server we need to do a couple of things. First is create a `sockaddr_in` structure with proper values filled in.
- Lets create one for ourselves :

```
struct sockaddr_in server;
```

- Have a look at the structure:

```
struct sockaddr_in {
    short  sin_family; // e.g. AF_INET, AF_INET6
    unsigned short sin_port// e.g. htons(3490)
    struct in_addr sin_addr;//see struct in_addr, below
    char sin_zero[8]; // zero this if you want to
};

struct in_addr {
    unsigned long s_addr; // load with inet_pton()
};
```

- The `sockaddr_in` has a member called `sin_addr` of type `in_addr` which has a `s_addr` which is nothing but a long. It contains the IP address in long format.
- Function `inet_addr` is a very handy function to convert an IP address to a long format.

This is how you do it :

```
server.sin_addr.s_addr=inet_addr("74.125.235.20");
```


- So you need to know the IP address of the remote server you are connecting to.
- The last thing needed is the `connect()` function. It needs a socket and a `sockaddr` structure to connect to.
- ```
struct sockaddr {
 unsigned short sa_family; // address family,
 char sa_data[14]; // 14 bytes of protocol adrs
};
```
- The `connect()` system call connects the socket referred to by the file descriptor `sockfd` to the address specified by `addr`.
- The `addrlen` argument specifies the size of `addr`. The format of the address in `addr` is determined by the address space of the socket `sockfd`.
- **Returns Zero on success, -1 on error.**

Ex:- /\*\*\*\*\* Write a function to send a request from client to server \*\*\*\*\*/

Header files

```
int client_request() {
 struct sockaddr_in server;
 server.sin_addr.s_addr = inet_addr("74.125.235.20");
 server.sin_family = AF_INET;
 server.sin_port = htons(80);
 if (connect (socket_desc , (struct sockaddr *)&server ,
sizeof(server)) < 0) {
 puts("connect error");
 return -1;
 }
 return 0;
}
```

**Note:-** Try connecting to a port different from port 80 and you should not be able to connect which indicates that the port is not open for connection.

### **Sending data to Server:**

- After connected to Server, next we have to send a message.
- Function `send()` will simply send data. It needs the socket descriptor, the data to send and its size.
- This function declaration is given below:
  - `ssize_t send (int sockfd, const void *buf, size_t len, int flags);`
- The `send()` call may be used only when the socket is in a connected state (so that the intended recipient is known). The only difference between `send()` and `write` is the presence of flags. With zero flags argument, `send()` is equivalent to `write`.
- **Return Value:**
  - On Success, returns the number of characters sent,
  - On failure, returns -1.

### **Receiving data from Server:**

- Function `recv()` is used to receive data on a socket. This function declaration is given below:  
`ssize_t recv (int sockfd, void *buf, size_t len, int flags);`
- `recv()` function is used to receive messages from a socket.
- The `recv()` call is normally used only on a connected socket.
- **Return Value:** These calls return the number of bytes received on success and -1 if on error.

### **Closing a Socket:**

- Function `close` is used to close the socket.
- Need to include the `unistd.h` header file for this.

`int close(int fd);`

- `close()` closes a file descriptor, so that it no longer refers to any file and may be reused.
- Return Value:

returns zero on success and -1 on error.

### **Server API's:**

1. `Socket()`;
2. `bind()`;
3. `listen()`;
4. `accept()`;
5. `Send()`
6. `Recv()`;
7. `Close`

### **Required Header Files:**

```
#include <sys/types.h>
#include <sys/socket.h>
```

- We have already discussed about `socket()`, `send()`, `recv()`, and `close()`.  
Now the remaining system calls are:

### **Binding a Socket:**

Function `bind()` can be used to bind a socket to a particular address and port. It needs a `sockaddr_in` structure similar to connect function. This function declaration is given below:

```
int bind(int sockfd, const struct sockaddr *addr,
socklen_t addrlen);
```

When a socket is created with `socket`, it exists in a name space, but has no address assigned to it. `bind()` assigns the address specified to by `addr` to the socket referred to by the file descriptor `sockfd`.

addrlen specifies the size, in bytes, of the address structure pointed to by addr. Traditionally, this operation is called "**assigning a name to a socket**".

**Return Values:** On success, zero is returned. On error, -1 is returned,

### **Listening for Connections:**

After binding a socket to a port the next thing we need to do is listen for connections. For this we need to put the socket in listening mode. Function listen is used to put the socket in listening mode. Syntax of this function is given below:

```
int listen(int sockfd, int backlog);
```

Here:

sockfd: return value of socket() function,  
backlog: How many client requests, it has to wait.

**Return Value :**

Returns zero on success and -1 on error.

**Ex:** `listen(socket_desc , 3);`

listen() marks the socket referred to by sockfd as a passive socket. that is, as a socket that will be used to accept incoming connection requests using accept(). The sockfd argument is a file descriptor that refers to a socket.

### **Accepting incoming Connections:**

Function accept() is used to accept the incoming connections from clients. This function declaration is given below:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Here:

Sockfd: return value socket() function,  
Addr: to store the address of the client.  
Addrlen: address length of the client.

The `accept()` system call is used with connection-based socket types. It extracts the first connection request on the queue of pending connections for the listening socket.

**Return Value:**

On success, these system calls return a nonnegative integer that is a descriptor for the accepted socket and returns -1 on error.



### Example program for Server-Client Model using TCP/IP

**IN SERVER:**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main() {
 int sock, connected, bytes_recieved , true = 1;
 char send_data [1024] , recv_data[1024];
 struct sockaddr_in server_addr,client_addr;
 int sin_size;
 if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
 perror("Socket:");
 exit(1);
 }
```

```

 if(setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&true,sizeof(int
)) == -1) {
 perror("Setsockopt");
 exit(1);
 }
 server_addr.sin_family = AF_INET;
 server_addr.sin_port = htons(5000);
 server_addr.sin_addr.s_addr = INADDR_ANY;
 bzero(&(server_addr.sin_zero),8);
 if(bind(sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
 perror("Unable to bind");
 exit(1);
 }
 if (listen(sock, 5) == -1) {
 perror("Listen");
 exit(1);
 }
 printf("\nTCPServer Waiting for client on port
5000\n");
 fflush(stdout);
 while(1) {
 sin_size = sizeof(struct sockaddr_in);
 connected = accept(sock, (struct sockaddr
*)&client_addr,&sin_size);
 printf("\n I got a connection from (%s , %d)\n",

 inet_ntoa(client_addr.sin_addr),ntohs(client_addr.sin_port)
);

 while (1) {

```

```

 printf("\nENTER THE DATA TO BE SENT TO
THE CLIENT\n");

 printf("\n SEND (q or Q to quit) : ");
 scanf("%s",send_data);
 if (strcmp(send_data , "q") == 0 ||
strcmp(send_data , "Q") == 0) {
 send(connection,
send_data,strlen(send_data), 0);

 close(connection);
 printf("\nsocket is closed by the
server\n");

 close(sock);
 exit(0);
 }
 else

 send(connection,
send_data,strlen(send_data), 0);

 printf("\nWAITING FOR THE DATA FROM THE
CLIENT\n");

 bytes_received =
recv(connection,recv_data,1024,0);

 recv_data[bytes_received] = '\0';
 if (strcmp(recv_data , "q") == 0 ||
strcmp(recv_data , "Q") == 0) {
 close(connection);
 printf("\nsocket has been closed
by the client\n");

 close(sock);
 exit(0);
 }
 else

```

```

 printf("\n RECIEVED DATA = %s \n" ,
recv_data);

 fflush(stdout);
 }
}

return 0;
}

```

### **IN CLIENT:**

```

#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main() {
 int sock, bytes_recieved;
 char send_data[1024],recv_data[1024];
 struct hostent *host;
 struct sockaddr_in server_addr;

 host = gethostbyname("127.0.0.1");
 if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
 perror("\nSocket\n");
 exit(1);
 }

 server_addr.sin_family = AF_INET;
 server_addr.sin_port = htons(5000);

```



```

server_addr.sin_addr = *((struct in_addr *)host-
>h_addr);
bzero(&(server_addr.sin_zero),8);
if (connect(sock, (struct sockaddr *)&server_addr,
sizeof(struct sockaddr)) == -1){
 perror("Connect");
 exit(1);
}
else {
 printf("\nCONNECTED TO THE SERVER\n");
}
while(1) {
 printf("\nWAITING FOR THE INPUT/ DATA FROM THE
SERVER\n");

 bytes_recieved=recv(sock,recv_data,1024,0);
 recv_data[bytes_recieved] = '\0';
 if (strcmp(recv_data , "q") == 0 ||
strcmp(recv_data , "Q") == 0) {
 printf("\nsocket closed by server\n");
 close(sock);
 break;
 }
 else
 printf("\nRecieved data = %s \n" ,
recv_data);

 printf("\nSEND (q or Q to quit) : ");
 scanf("%s",send_data);

 if (strcmp(send_data , "q") != 0 &&
strcmp(send_data , "Q") != 0)
 send(sock,send_data,strlen(send_data), 0);
 else {

```

```

 send(sock,send_data,strlen(send_data), 0);
 printf("\nSOCKET CONNECTION LOST BY
CLIENT\n");

 close(sock);
 break;
 }

 }
 return 0;
}

```

### **Checking Output:**

1. Open two terminals in same directory,
2. Create server executable code and client executable code separately,
3. 1<sup>st</sup> execute server code. Now server is waiting for client requests.
4. Then send a request to server by executing client executable program.
5. Now send data to client from server and vice versa



**Make file**

### **What is make?**

The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.

To use make, you must write a file called the Makefile that describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files. Once a suitable Makefile exists, each time you change some source files, this simple shell command:

make

Suffices to perform all necessary recompilations. The make program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated.

### Why Use a Makefile?

In any project you can have any number of source files. Now imagine having to recompile even the simplest of programs consisting of two source files, `binarytree.c` and `mainprog.c`. You will need to type the following command:

```
gcc -o outprogrname binarytree.c mainprog.c
```

Now, since that `mainprog.c` files takes ages to compile, you don't want to recompile it every time, the same goes for the `binarytree.c` file. Now imagine you had a hundred source files (think big, and then some more).

A Makefile makes this simple by allowing us to specify rules (consisting of dependencies and commands) to build our project.

### A Simple Makefile Example

The following is a very simple, but complete Makefile:

```
outprogrname : binarytree.o mainprog.o
 gcc -o outprogrname binarytree.o mainprog.o
binarytree.o : binarytree.c
 gcc -c binarytree.c
mainprog.o : mainprog.c
 gcc -c mainprog.c
```

Please note that the indented lines begin with a tab character.

In the case where both files needs to be compiled, make will issue three commands - yes this can be wasteful

- namely:

```
gcc -c binaryproc.c
gcc -c binarytree.c
gcc -o outprogrname binarytree.o mainprog.o
```

It is clear that these are the indented lines. You could have executed all of these commands by hand, if you were so incline, but typing `make` is easier. Now going on to

dissect the above, a Makefile consists of a set of rules. Each rule consists of dependencies and commands. The lines with the colons in are dependencies and the lines that are tabbed in are commands.

For each dependency the commands required to bring the target - the file(s) before the colon - up to date follows it on the next line, indented with a tab character. The file(s) to the right of the colon is the files on which the target relies, these are almost always used as input to the commands on the lines following the dependency.

It should be clear that the two rules at the bottom of the given Makefile produces .o files, created from the .c files (the -c flag tells gcc to compile the source file, but not link it). The topmost rule takes the two object files and links them together to create the final executable.

## How Does make Determine Which Commands to Execute?

There are two reasons why make would decide a file needs to be made:

1. The file does not exist.
2. The file is outdated.

The first case is simple, the file does not exist - nuff said. The second case should be reasonably clear as well.

Considering the simple Makefile above, edit binarytree.c. You would now like to compile outprogname. It relies on binarytree.o and mainprog.o. binarytree.o in turn relies on binarytree.c and mainprog.o on mainprog.c.

You've just edited binarytree.c, so as you probably expect, binarytree.o is outdated and as a result so is outprogname. Make then goes ahead and first builds binarytree.o and then outprogname using the commands given.

A file is considered outdated if and only if:

It is listed on the left of a colon

A file on the right of that same colon is newer than that file. File "a" is considered newer than file "b" if its last modified time is more recent than file "b".

## Make variables

Suppose you would like to use another C compiler, say cc. Now you can either replace gcc right through with cc, or you could have been smart and used variables. A make variable, as in any programming language, is simply a place holder for another value. You can create such a variable in a Makefile with a line such as:

```
cc=gcc
```

Actually, variables in make work very much like in bash scripting. So to actually replace the value of cc (which evaluates to gcc) just use `${cc}`.

### Built-in Make Variables

Make has a few built-in variables. Some of them are listed below.

|                     |                                                                                                                                                             |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>\$@</code>    | This will always expand to the current target.                                                                                                              |
| <code>\$&lt;</code> | The name of the first prerequisite. This is the first item listed after the colon.                                                                          |
| <code>\$?</code>    | The names of all the prerequisites that are newer than the target. In the above example this will evaluate to <code>binarytree.o</code> for the first rule. |
| <code>\$^</code>    | The names of all the prerequisites, with spaces between them. No duplicates                                                                                 |
| <code>\$+</code>    | Like <code>\$^</code> , but with duplicates. Items are listed in the order they were specified in the rule.                                                 |