Elementary TCP Sockets

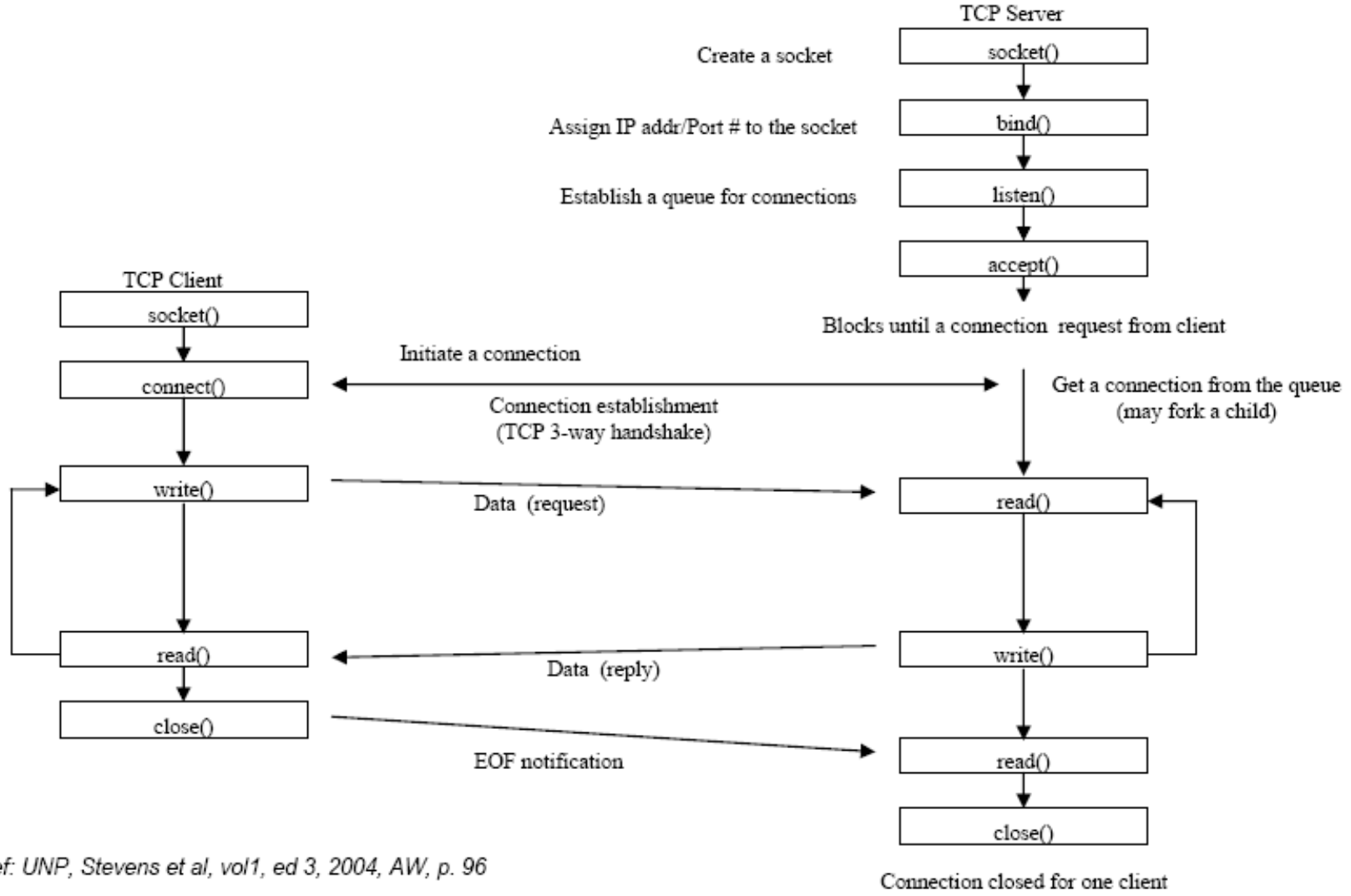
Introduction

- This session describes the elementary socket functions required to write a complete TCP client and server.
  - All the elementary socket functions that we will be using and then develop the TCP client and server sockets.
  - Also describe concurrent servers, a common Unix technique for providing concurrency when numerous clients are connected to the same server at the same time.
  - Each client connection causes the server to fork a new process just for that client.
  - we consider only the one-process-per-client model using fork, but we will consider a different one-thread-per-client model.

Elementary TCP Sockets

- socket function
- connect function
- bind function
- listen function
- accept function
- fork and exec functions
- Concurrent servers
- close function
- getsockname and getpeername functions

Socket Functions



Ref: UNP, Stevens et al, vol1, ed 3, 2004, AW, p. 96

socket Function

- To open a socket for performing network I/O

```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
        returns: nonnegative descriptor if OK, -1 on error
                normally 0 except for raw sockets
```

family	Description
AF_INET	IPv4
AF_INET6	IPv6
AF_LOCAL	Unix domain protocols ~ IPC
AF_ROUTE	Routing sockets ~ appls and kernel
AF_KEY	Key socket

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_RAW	raw socket
SOCK_PACKET	datalink (Linux)

socket Function

- Protocol of sockets for AF_INET or AF_INET6.

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

- Not all combinations of socket family and type are valid. Following figure shows the valid combinations, along with the actual protocols that are valid for each pair.
- Combinations of family and type for the socket function.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP|SCTP	TCP|SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

socket Function

- On success, the socket function returns a small non-negative integer value, similar to a file descriptor.
- We call this a socket descriptor, or a sockfd. To obtain this socket descriptor, all we have specified is a protocol family (IPv4, IPv6, or Unix) and the socket type (stream, datagram, or raw).

AF_xxx Versus PF_xxx

- The "AF_" prefix stands for "address family" and the "PF_" prefix stands for "protocol family."
- The intent was that a single protocol family might support multiple address families and that the PF_ value was used to create the socket and the AF_ value was used in socket address structures.
- But in actuality, a protocol family supporting multiple address families has never been supported and the <sys/socket.h> header defines the PF_ value for a given protocol to be equal to the AF_ value for that protocol.

The connect() Function (1/3)

- Is used by a client to establish a connection with a server via a 3-way handshake

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

int connect(int sockfd,  const struct sockaddr *servaddr,
            socklen_t  addrlen);
                                        Returns: 0 if OK, -1 on error
```

  - *sockfd* is a socket descriptor returned by the socket() function
  - *servaddr* contains the IP address and port number of the server
  - *addrlen* has the length (in bytes) of the server socket address structure

The connect() Function (2/3)

- The client does not have to call before calling connect: the kernel will choose both an ephemeral port and the source IP address if necessary.
- This function returns only when the connection is established or an error occurs
- Some possible errors:
  - If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned
    - The connection-establishment timer expires after 75 seconds (4.4 BSD)
      - The client will resend SYN after 6 seconds later, and again another 24 seconds later. If no response is received after a total of 75 seconds, the error is returned

The connect() Function (3/3)

- **Hard error**: RST received in response to client TCP's SYN (server not running)
  - returns ECONNREFUSED
- **Soft error:** If an ICMP "destination unreachable" is received from an intermediate router.
  - EHOSTUNREACH or ENETUNREACH is returned.
    - Upon receiving the first ICMP message, the client kernel will keep sending SYNs at the same time intervals as mentioned earlier, until after 75 seconds have elapsed (4.4BSD)

Different Error Conditions (1/2)

- We first specify the local host (127.0.0.1), which is running the daytime server
  - solaris % daytimetcpcli
  - 127.0.0.1 Sun Jul 27 22:01:51 2003
- We specify a different machine's IP address
  - solaris % daytimetcpcli 192.6.38.100
  - Sun Jul 27 22:04:59 PDT 2003
- We specify an IP address that is on the local subnet (192.168.1/24) but the host ID (100) is nonexistent.
  - solaris % daytimetcpcli 192.168.1.100
  - connect error: Connection timed out
- We only get the error after the connect times out (around four minutes with Solaris 9). Notice that our err_sys function prints the human-readable string associated with the ETIMEDOUT error.

Different Error Conditions (2/2)

- To specify a host (a local router) that is not running a daytime server. The server responds immediately with an RST.
  - solaris % daytimetcpcli 192.168.1.5
  - connect error: Connection refused

- Specifies an IP address that is not reachable on the Internet. That a router six hops away returns an ICMP host unreachable error. As with the ETIMEDOUT error, in this example, connect returns the EHOSTUNREACH error only after waiting its specified amount of time.
  - solaris % daytimetcpcli 192.3.4.5
  - connect error: No route to host

The bind() Function (1/2)

- Assign a local protocol address to a socket, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

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

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

                                  Returns: 0 if OK, -1 on error
```

  - *sockfd* is a socket descriptor returned by the socket() function
  - *myaddr* is a pointer to a protocol-specific address. With TCP, it has the IP address and port number of the server
  - *addrlen* has the length (in bytes) of the server socket address structure

The bind() Function (2/2)

- IP address/Port number assignment :

Process specifies		Results
IP address	Port	
Wildcard	0	Kernel chooses IP address and port
Wildcard	Nonzero	Kernel chooses IP address, process specifies port
Local IP addr	0	Process specifies IP address, kernel chooses port
Local IP addr	Nonzero	Process specifies IP address and port

  - Wildcard address: INADDR_ANY (IPv4), in6addr_any (IPv6)

```
struct sockaddr_in  servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

  - TCP servers typically bind their well-known port, and clients let the kernel choose an ephemeral port

bind Function (cont.)

- For a host to provide Web servers to multiple organizations:
- Method A: Aliased IP addresses
  - 1. Alias multiple IP addresses to a single interface (ifconfig).
  - 2. Each server process binds to the IP addr for its organization. (one copy of the HTTP server is started for each organization and each copy binds only the IP address for that organization)
  - (Demultiplexing to a given server process is done by kernel.)

- Method B: Wildcard IP address
  - 1. A single server binds to the wildcard IP addr.
  - 2. The server calls getsockname to obtain dest IP from the
  - client.
  - 3. The server handles the client request based on the dest IP.

The listen() Function (1/2)

- Is used by a server to convert an unconnected socket to a passive socket

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

int listen(int sockfd,  int backlog);

                                    Returns: 0 if OK, -1 on error
```

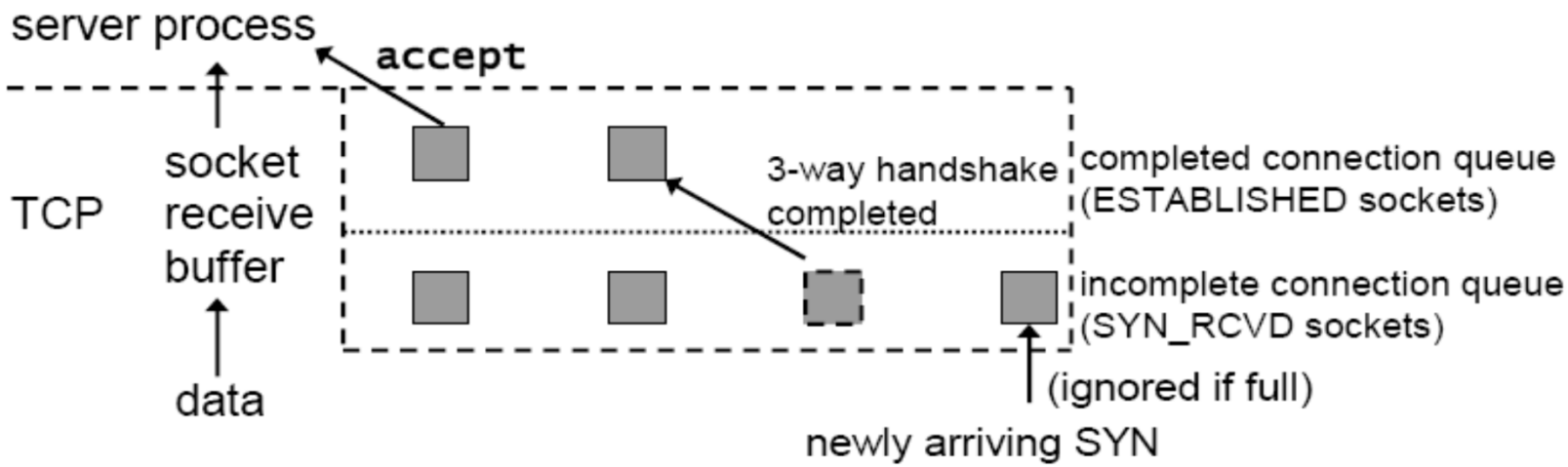
  - *sockfd* is a socket descriptor returned by the socket() function
  - *backlog* specifies the maximum number of connections the kernel should queue for this socket

The listen() Function (2/2)

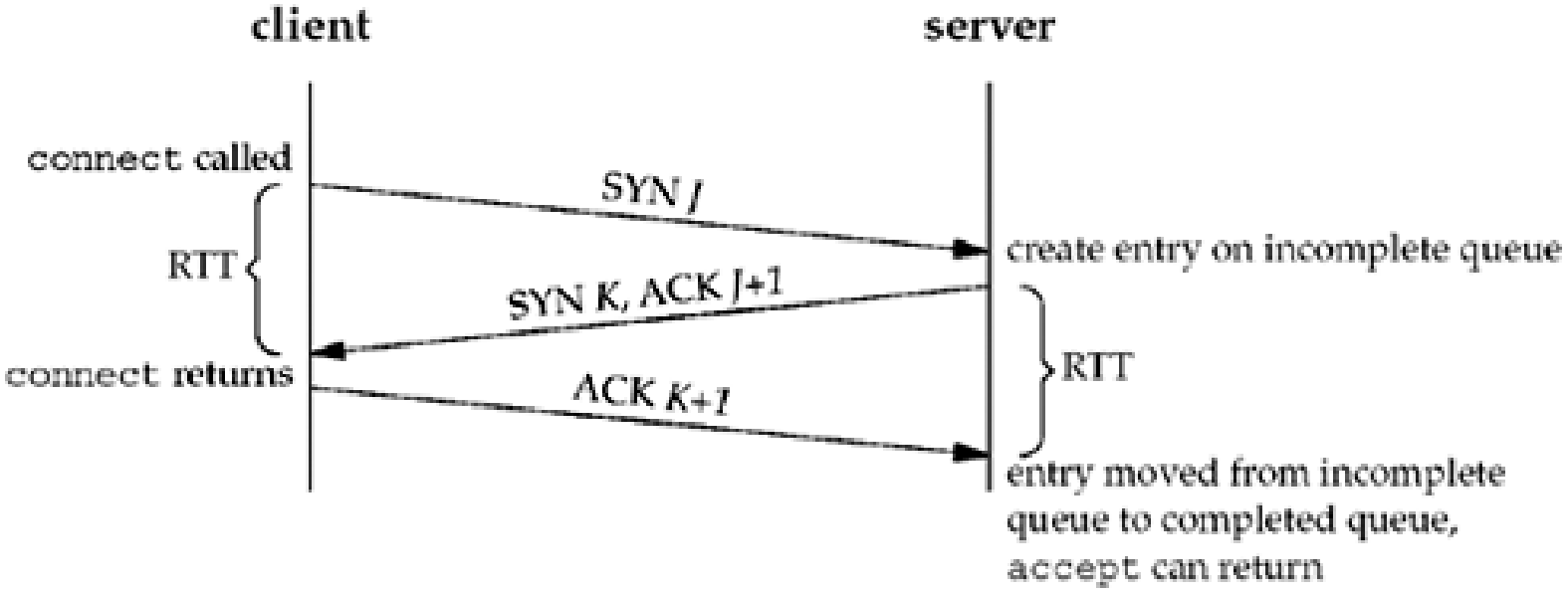
- For a given listening socket, the kernel maintains 2 queues
  - An *incomplete connection queue* *SYN_RCVD*
    - It contains an entry for each SYN received from a client, for which the server is awaiting completion of the TCP 3-way handshake
  - A *completed connection queue* *ESTABLISHED*
    - It contains an entry for each client with whom the TCP 3-way handshake process has completed
- *backlog* is the sum of these two queues
  - *backlog* has not been well-defined so far
  - One may select any number other than 0

About The backlog Parameter

- For a given listening socket, the kernel maintains two queues
- Backlog specifies the max for the sum of both queues



Three-Way Handshake and the Two Queues for listening socket



SYN Flooding

- SYN Flooding: A type of attack aiming at backlog
  - A program sends bogus SYNs at a high rate to a server, filling the incomplete connection queue for one or more TCP ports.
  - The source IP address of each SYN is set to a random number so that the server's SYN/ACK goes nowhere
    - This is called IP spoofing
  - This leaves no room for legitimate SYNs
    - TCP ignores an arriving SYN if the queues are full

The accept() Function (1/3)

- Is called by a server to return a new descriptor, created automatically by the kernel, for the connected socket

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

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

           Returns: non-negative descriptor if OK, -1 on error
```

  - *sockfd* is a socket descriptor returned by the socket() function
  - *cliaddr* contains the IP address and port number of the connected client (a value-result argument)
  - *addrlen* has the length (in bytes) of the returned client socket address structure (a value-result argument)
  - If the completed connection queue is empty, the process is put to sleep

The accept() Function (2/3)

- If accept is successful, its return value is a brand-new descriptor automatically created by the kernel.
- This new descriptor refers to the TCP connection with the client.
- When discussing accept socket, we call the first argument to accept the listening and we call the return value from accept the connected socket.
  - The descriptor created by socket and then used as the first argument to both bind and listen.
- It is important to differentiate between these two sockets.
  - A given server normally creates only one listening socket, which then exists for the lifetime of the server.
  - The kernel creates one connected socket for each client connection that is accepted (i.e., for which the TCP three-way handshake completes).
- When the server is finished serving a given client, the connected socket is closed.

The accept() Function (3/3)

- Both cliaddr and addrlen may be set to the NULL pointer, if the server is not interested in knowing the identity of the client.

Use cliaddr to Get Client's Address

```
int                     listenfd, cliaddr;
socklen_t               len;
struct sockaddr_in      servaddr, cliaddr;
char                    buff[MAXLINE];
…
len = sizeof (cliaddr);
connfd = Accept (listenfd, (SA *) &cliaddr, &len);
printf ("connection from %s, port %d\n",
        Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof (buff)),
        ntohs(cliaddr.sin_port));
…
```

The UNIX fork() Function (1/2)

- It is used in UNIX to create a new process

```
#include <unistd.h>

pid_t fork(void);

    Returns: 0 in child, process ID of child in parent, -1 on error
```

  - *fork()* is called once, but returns twice.
    - Once in the calling process, called the *parent, return the process ID of the newly created process.*
    - return 0, called the child
  - A parent may have more than 1 child process

The UNIX fork() Function (2/2)

- All descriptors open in the parent before fork() are shared with the child after fork() return.
  - The connected socket is then shared between the parent and the child
- Two typical uses of fork() :
  - A process makes a copy of itself so that one copy can handle one operation, and the other copy does something else
    - This is typical for network servers
  - A process want to execute a new program by calling exec() in the child process
    - User commands in UNIX are typically handled this way
- fork() can be used to implement concurrent servers

More on fork()

- child can get it's parent process ID by getppid
- All descriptors open in the parent before fork, e.g. the connected socket, are shared with the child.
- Normally the child then reads and writes the connected socket and the parent closes the connected socket.

exec functions (1/4)

- The differences in the six exec functions are:
  - whether the program file to execute is specified by a filename or a pathname;
  - whether the arguments to the new program are listed one by one or referenced through an array of pointers; and
  - whether the environment of the calling process is passed to the new program or whether a new environment is specified.

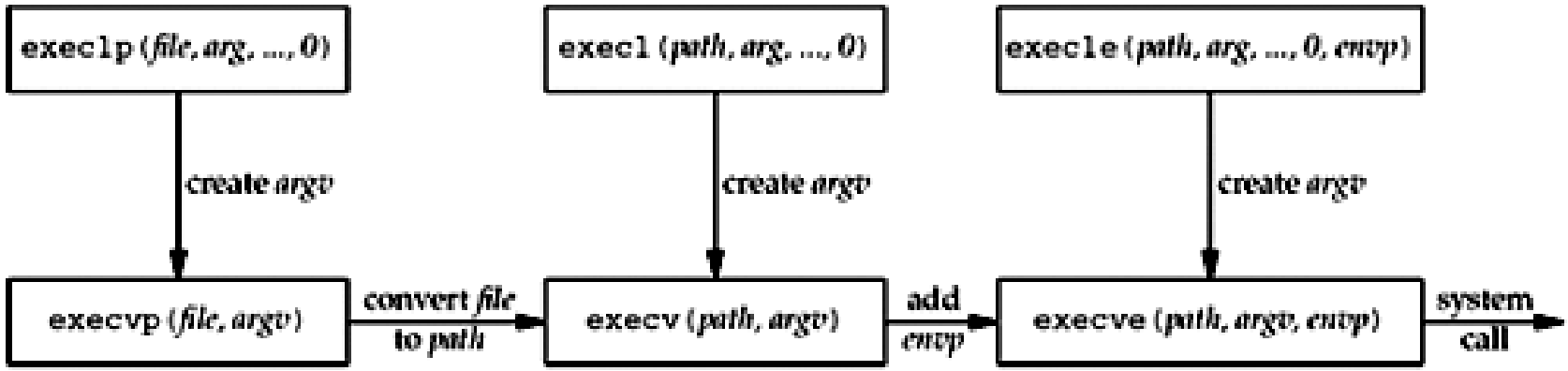
exec functions (2/4)

- #include <unistd.h>
- int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */);
- int execv (const char *pathname, char *const argv[ ]);
- int execle (const char *pathname, const char *arg0, ... /* (char *)0,
- char *const envp[ ] */);
- int execve (const char *pathname, char *const argv[ ], char *const
- envp[ ]);
- int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */);
- int execvp (const char *filename, char *const argv[ ]);

All return: -1 on error, no return on success

exec functions (3/4)

- These functions return to the caller only if an error occurs. Otherwise, control passes to the start of the new program, normally the main function.
- Only execve is a system call within the kernel and the other five are library functions that call execve.



exec functions (4/4)

- Meaning of different letters :
  - l : needs a list of arguments
  - v : needs an argv[] vector (l and v are mutually exclusive)
  - e : needs an envp[] array
  - p : needs the PATH variable to find the executable file
- The three functions in the top row specify each argument string as a separate argument to the exec function, with a null pointer terminating the variable number of arguments.
- The two functions in the left column specify a filename argument. This is converted into a pathname using the current PATH environment variable.
- The four functions in the left two columns do not specify an explicit environment pointer. Instead, the current value of the external variable environ is used for building an environment list that is passed to the new program.

Concurrent Servers

- The concurrent servers is an iterative server. For something as simple as a daytime server.
- But when a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time.
- The simplest way to write a concurrent server under Unix is to fork a child process to handle each client.

Concurrent Servers: Outline

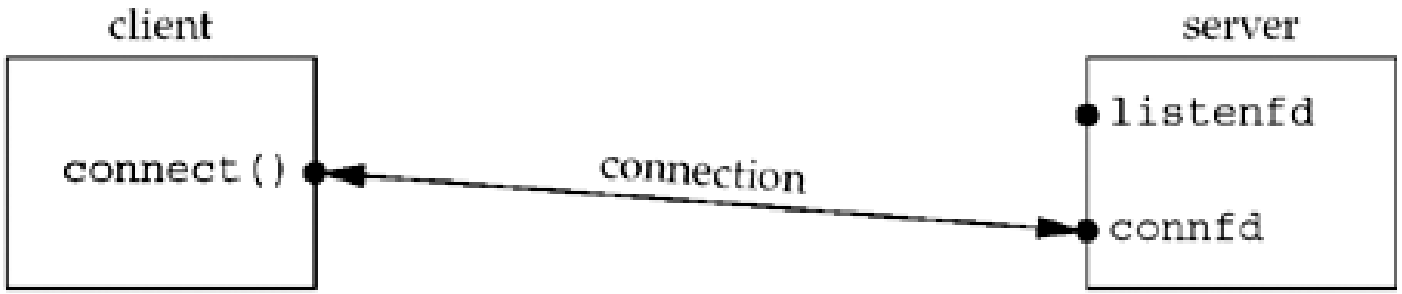
```
pid_t pid;
int listenfd, connfd;
listenfd = Socket (...);
/* fill in socket_in{} with server's well-known port */
Bind (listenfd, ...);
Listen (listenfd, LISTENQ);

for ( ; ; ){
        connfd = Accept (listenfd, ...); /* probably blocks */
        if ( (pid = Fork ( ) ) == 0) {
                Close (listenfd); /* child closes listening socket */
                doit (connfd); /* process the request */
                Close (connfd); /* done with this client */
                exit (0); /* child terminates */
        }
        Close (connfd); /* parent closes connected socket */
}
```

Concurrent Servers: Shared Descriptors(1/4)

- When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket).
- The parent closes the connected socket since the child handles the new client.
- The following figure shows the status of the client and server while the server is blocked in the call to accept and the connection request arrives from the client.

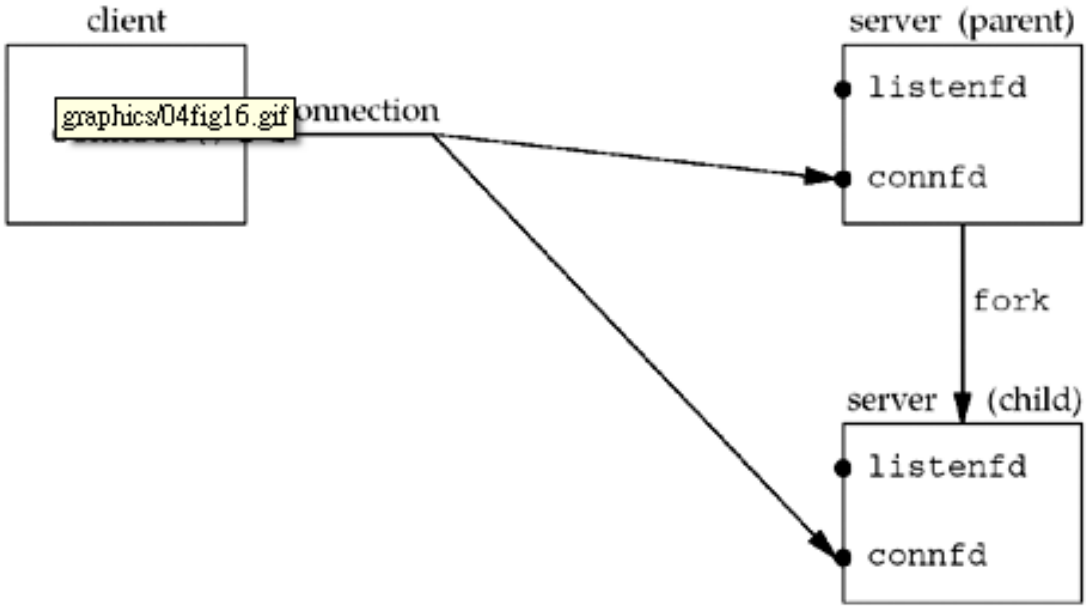
Concurrent Servers: Shared Descriptors(2/4)

- Immediately after accept returns, we have the scenario shown in the following figure.
- The connection is accepted by the kernel and a new socket, connfd, is created.
- This is a connected socket and data can now be read and written across the connection.



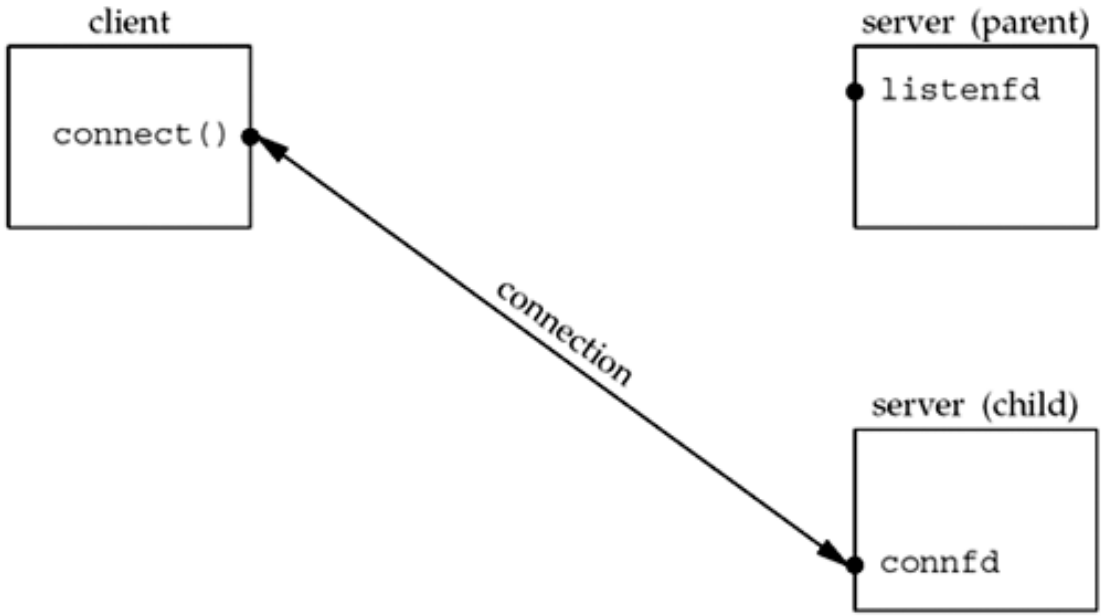
Concurrent Servers: Shared Descriptors(3/4)

- The next step in the concurrent server is to call fork. This figure shows the status after fork returns.
- Both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child.

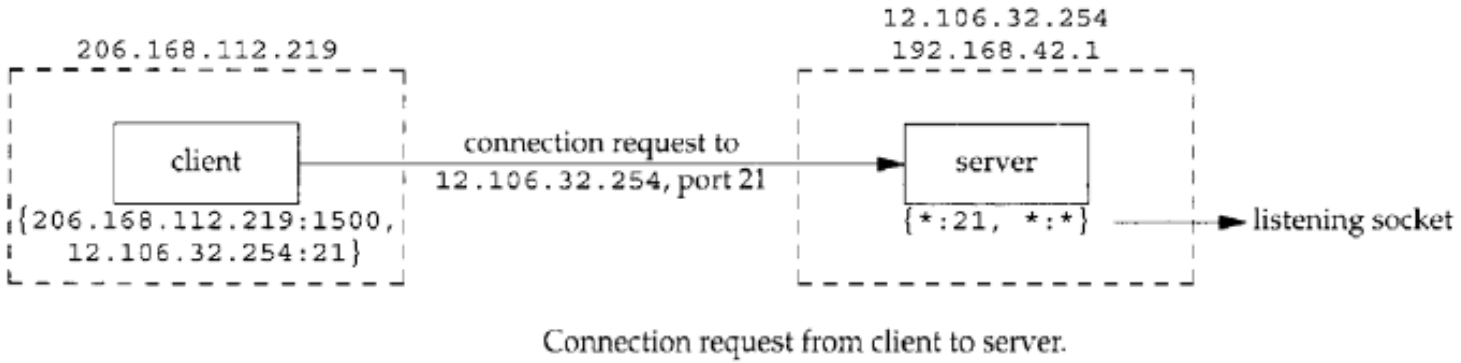


Concurrent Servers: Shared Descriptors(4/4)

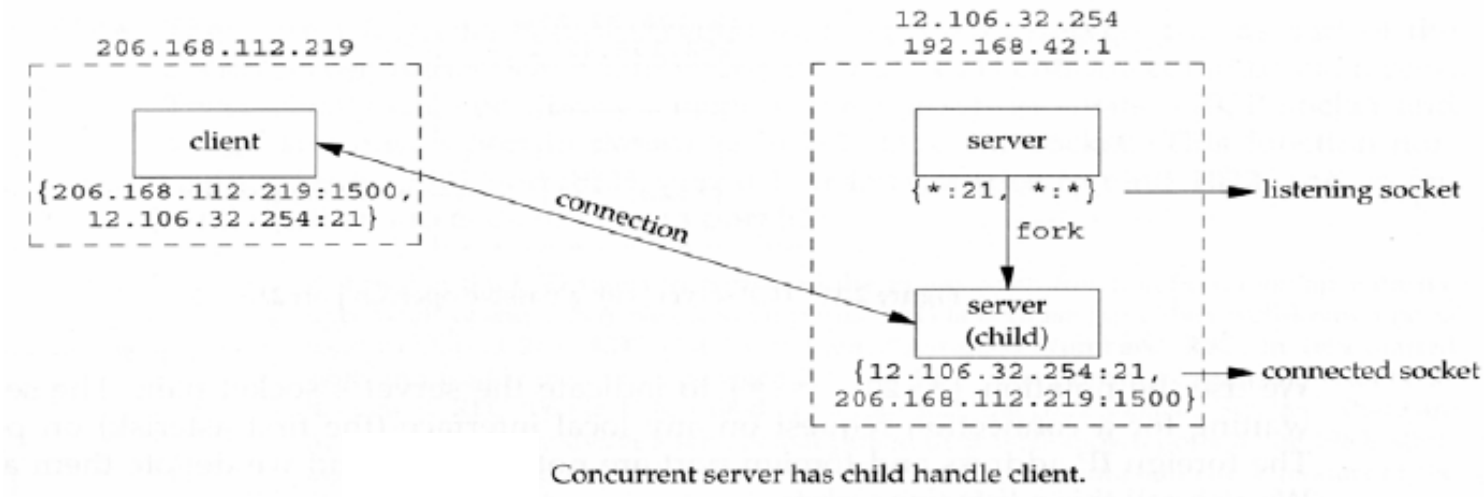
- The next step is for the parent to close the connected socket and the child to close the listening socket.
- This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call accept again on the listening socket, to handle the next client connection.



Server/Client Example (1/2)

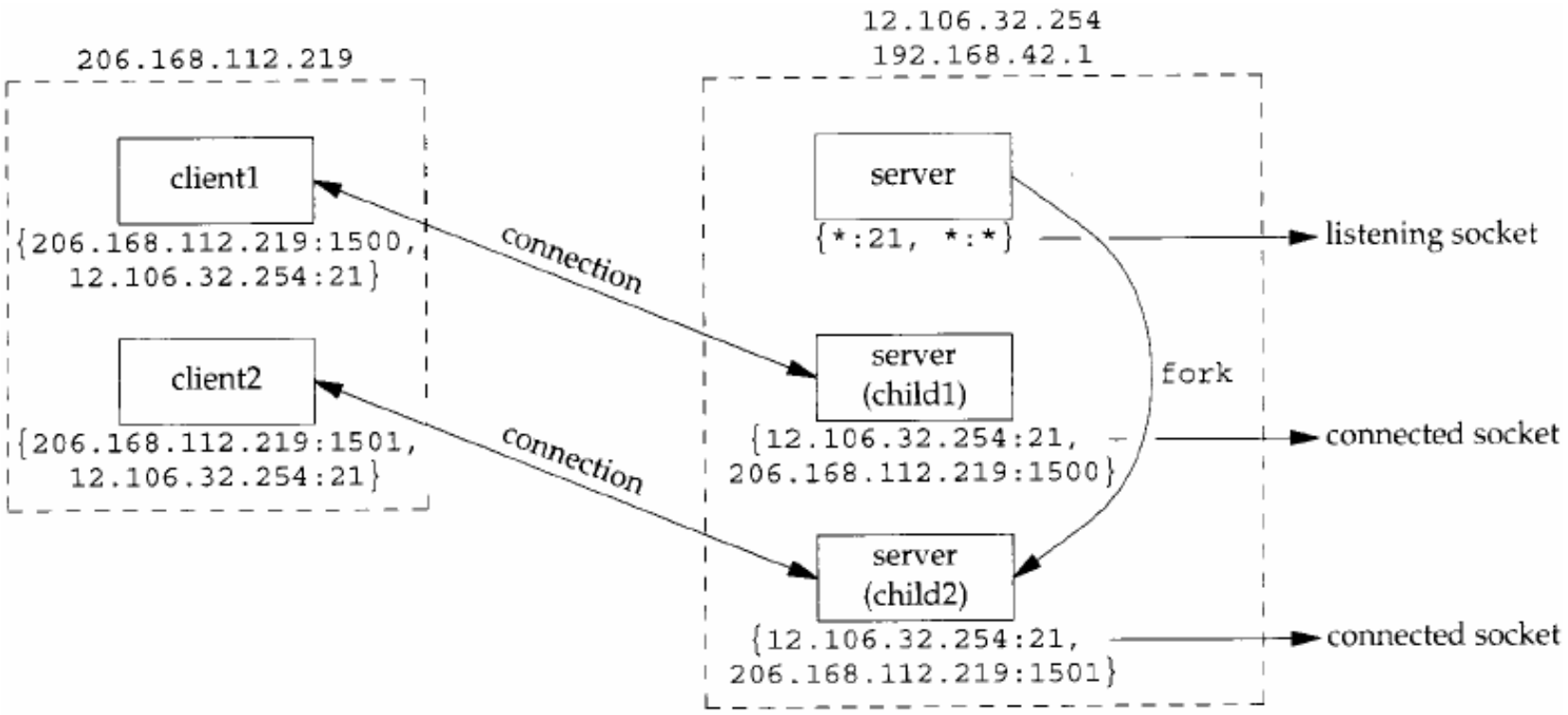


Connection request from client to server.



Concurrent server has child handle client.

Server/Client Example (2/2)



Second client connection with same server.

The UNIX close() Function (1/2)

- Is used to close a socket and terminate a TCP connection

```
#include <unistd.h>

int close(int sockfd);

                                    Returns: 0 if OK, -1 on error
```

  - *sockfd* is a socket descriptor returned by the socket() function

The UNIX close() Function (2/2)

- close() marks socket as closed and returns immediately.
  - *sockfd* is no longer usable
  - TCP continues to try sending unsent data
    - Hardly knows whether it was ever successful
- close() simply decrements the reference count
  - Socket goes away when the reference count becomes 0
- What if the parent does not close the connected socket for the client ?
  - May run out of descriptors eventually
  - No client connection will be terminated
    - Reference count remains at 1

getsockname()/getpeername() (1/3)

- Is used to get the local/foreign protocol address associated with a socket

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

int getsockname(int sockfd,  struct sockaddr *localaddr,
                socklen_t  *addrlen);

int getpeername(int sockfd,  struct sockaddr *peeraddr,
                socklen_t  *addrlen);

                                Returns: 0 if OK, -1 on error
```

  - *sockfd* is a socket descriptor returned by the socket() call
  - All *localaddr/peeraddr/addrlen* are value-result arguments

getsockname()/getpeername() (2/3)

- Reasons for using these two functions :
  - After connect successfully returns in a TCP client that does not call bind, getsockname returns the local IP address and local port number assigned to the connection by the kernel.
  - After calling bind with a port number of 0 (telling the kernel to choose the local port number), getsockname returns the local port number that was assigned.
  - getsockname can be called to obtain the address family of a socket.
  - In a TCP server that binds the wildcard IP address, once a connection is established with a client (accept returns successfully), the server can call getsockname to obtain the local IP address assigned to the connection. The socket descriptor argument in this call must be that of the connected socket, and not the listening socket.

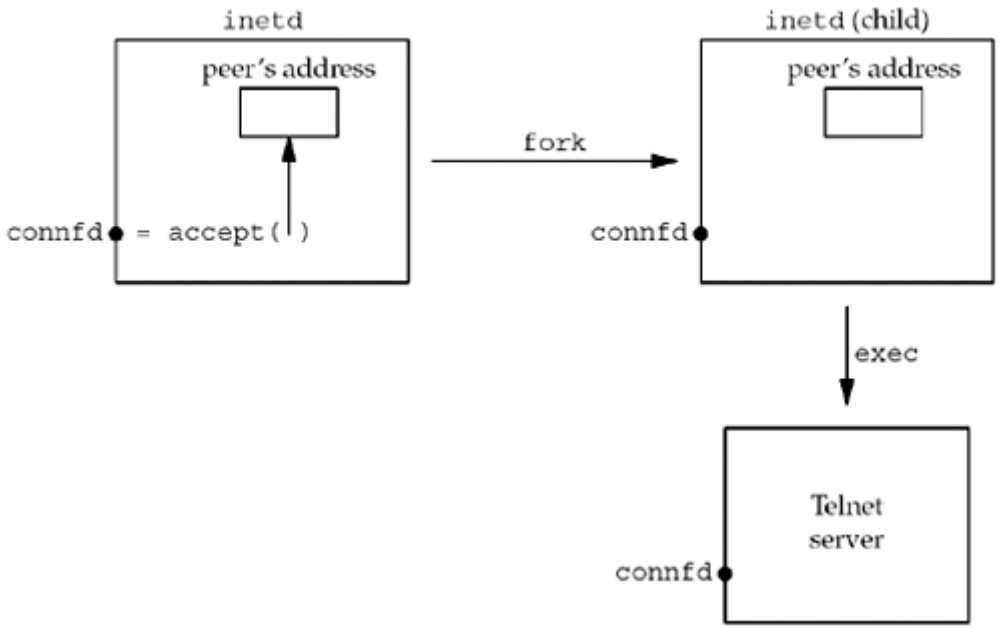
getsockname()/getpeername() (3/3)

- When a server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call getpeername. This is what happens whenever inetd forks and execs a TCP server.

Example of inetd spawning a server

The Telnet server how to know the value of *connfd* after exec:

1. Always setting descriptors 0, 1, 2 to be the connected socket before exec.
2. Pass it as a command-line arg. in exec



Example of inetd Spawning a Server

- inetd calls accept (top left box) and two values are returned: the connected socket descriptor, connfd, is the return value of the function, and the small box we label "peer's address" (an Internet socket address structure) contains the IP address and port number of the client.
- fork is called and a child of inetd is created.
- Since the child starts with a copy of the parent's memory image, the socket address structure is available to the child, as is the connected socket descriptor.
- The child execs the real server, the memory image of the child is replaced with the new program file for the Telnet server (i.e., the socket address structure containing the peer's address is lost), and the connected socket descriptor remains open across the exec.
- One of the first function calls performed by the Telnet server is getpeername to obtain the IP address and port number of the client.

Example

- Obtaining the Address Family of a Socket
- lib/sockfd_to_family.c

```
1       #include "unp.h"
2       int
3       sockfd_to_family(int sockfd)
4       {
5               struct sockaddr_storage ss;
6               socklen_t len;
7               len = sizeof(ss);
8               if (getsockname(sockfd, (SA *) &ss, &len) < 0)
9                       return (-1);
10              return (ss.ss_family);
11      }
```