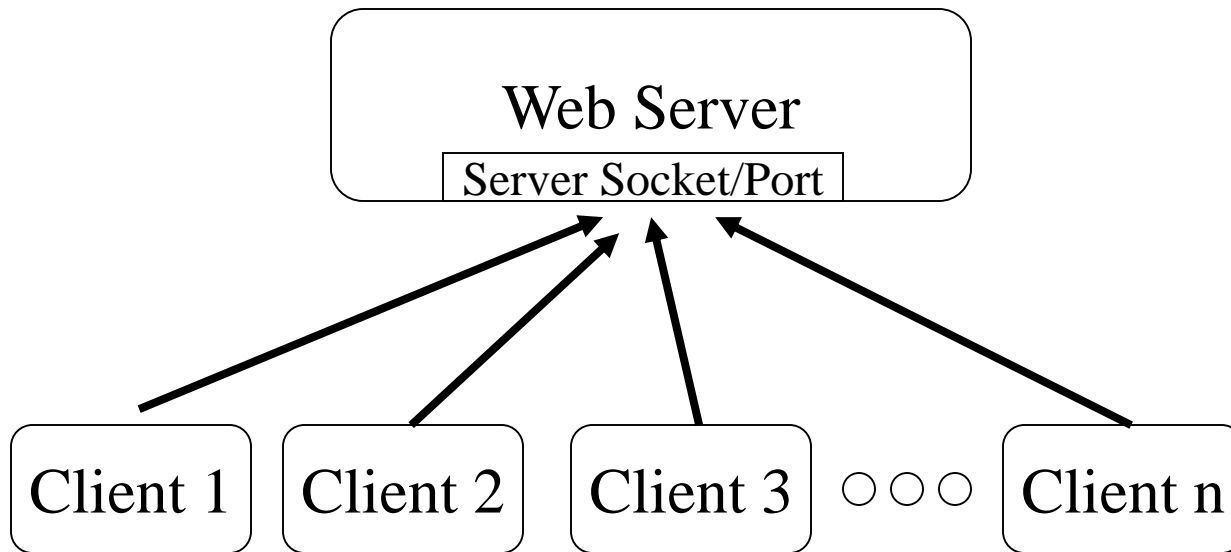


UNIX Networking 3

Benjamin Brewster

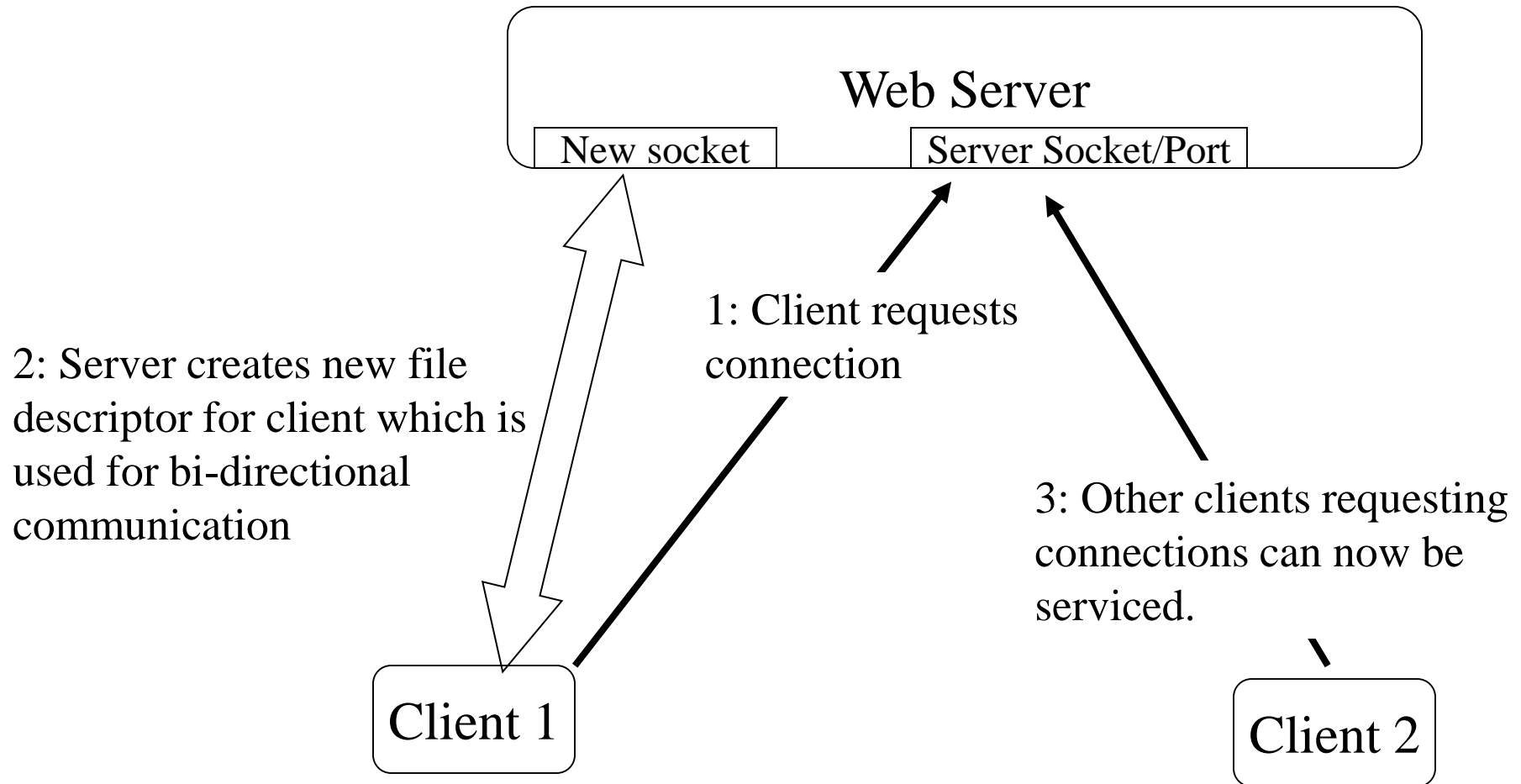
Adapted from Jon Herlocker, OSU

Server Sockets



- A server socket listens on a given port
- Many different clients may be connecting to that port
- Ideally, you would like a separate file descriptor for each client connection

Server Sockets:



Server Sockets

- How to use the socket API to listen for and accept connections
- Lets start by describing a non-concurrent implementation of a server (only one thread of execution)
- Procedure
 - Create network endpoint with `socket()`
 - Bind socket to a port - `bind()`
 - Start listening for connections - `listen()`
 - Loop and accept connections - `accept()`
 - Read and write data to client - `send()`, `recv()`, `read()`, `write()`

Creating the Socket

`socket()` creates an endpoint for network communication

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

For IP, use `AF_INET`

Returns file
descriptor
or -1

For TCP, use `SOCK_STREAM`
For UDP, use `SOCK_DGRAM`

For IP, 0

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    perror("socket");
    exit(1);
}
```

Bind the Socket to a Port

- Ports allow multiple network processes on a machine with a single address
- A server has to choose a port where clients can contact it
- `bind()` associates the chosen port with a socket already created with the `socket()` command

Binding the socket

`bind()` connects a server socket to a port

```
int bind(int sockfd, struct sockaddr *address,  
        size_t add_len);
```

0 on success
or -1 on error

Network address structure
identifying port to listen on

Size of address structure

```
if (bind(sockfd, (struct sockaddr *) &server, sizeof(SERVER)) == -1)  
{  
    perror("bind call failed");  
    exit(1);  
}
```

Setting up an address

- **Server accepting connections:**

```
struct sockaddr_in server;
```

```
server.sin_family = AF_INET;
```

```
server.sin_port = htons(7000);
```

```
server.sin_addr.s_addr = INADDR_ANY;
```


Listening for Connections

- The server will ignore any connection attempts until you tell the `socket()` to start listening
- This is done with `listen()`

Listen

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

Maximum numbers of
connections to queue up

0 on success
or -1 on error

```
if (listen(sockfd, 5) == -1)
{
    perror("listen call failed");
    exit(1);
}
```

Loop and Accept

- Servers generally run continually, waiting for clients to contact them
- Thus a server has an "infinite loop" that continually processes connections from clients
- The `accept()` function takes the next connection off of the listen queue or blocks the process until a connection arrives

Accept

`accept()` returns the next client connection

```
int accept(int sockfd, struct sockaddr *address,  
           size_t &add_len);
```

Returns file
descriptor
or -1

Network address structure
identifying client. This can
be NULL pointer

Pointer to variable containing size
of address structure, can be NULL

```
while (1) {  
    client_sockfd = accept(sockfd, NULL, NULL);  
    if (client_sockfd == -1) {  
        perror("accept call failed");  
        /* Here you may want to continue or exit - depends */  
    }  
}
```

Concurrency Definitions

- Iterative
 - One process
 - Handling one connection at a time
- Concurrent
 - One or more processes
 - Handling multiple connections concurrently

Iterative Servers

- Non-preemptive
 - Client must wait for all previous requests to complete
- Easy to design, implement, and maintain
- Best when
 - request processing time is short
 - No I/O is needed by server

Server Concurrency

- Many clients may connect “at once”
- Want to minimize:
 - Response time
 - Complexity
- Want to maximize
 - Throughput (connections serviced / second)
 - Hardware utilization (%CPU usage)
- Tradeoffs
 - Response time vs. throughput vs. hardware utilization vs. complexity

Concurrent Server

- Concurrency can be provided in two ways
 - Apparent concurrency
 - A single thread of execution, using the `select()` command and non-blocking I/O
 - Real concurrency
 - Multiple threads of execution
 - Could be multiple processes, each with one thread
 - OS preempts thread/process after each quantum



More on
`select` later

Apparent Concurrency

- Only one thread, no preemption, using non-blocking I/O
- Whenever an I/O request would block
 - Switch to another connection
- Up to a certain number of connections:
 - Maximizes CPU utilization
 - Increases throughput
- Less programming/debugging problems than true concurrency
- Code is more complicated
- Works well (only) if requests are short

Real Concurrency Using Threads

- Preemptive
 - Response time not dependent on length of previous client's request
- Harder to design, implement, and maintain
- Up to a certain number of connections
 - Maximizes CPU utilization
 - Maximizes response time
 - Increases throughput
- After too many concurrent connections
 - Everything gets worse -> server eventually hangs
 - Need to put limits on concurrent connections

More Real Concurrency

- Four different methods
 - Create process per client connection
 - A pool of available processes
 - One process, create one thread per connection
 - One process, a pool of available threads

Fork Solution #1

- One process per client connection
 - Fork a new process to handle every connection
 - Advantages:
 - Simple: minimal shared state to manage
 - Disadvantages:
 - Process creation (fork) is slow
 - Context-switching between procs is also slow (minor compared to fork)

Fork Solution #2

- Pool of available processes
 - Maintain a pool of iterative processes to handle connections
 - Advantages:
 - No longer have to fork
 - Have rapid response as long as there is an idle process available
 - Can set the pool size, so that you don't overload the hardware
 - Disadvantages:
 - Still have process context switching

Threads Solution

- Threads allow multiple concurrent execution contexts within a single process
 - Shared address space, shared code, shared data, etc.
- Can implement a web server as a single process with multiple threads
 - Either one thread per connection or a pool of threads
- Advantages:
 - Can sometimes avoid context-switches (user-level threads)
 - Can share data easily
- Disadvantages:
 - Code must be thread-safe (AKA *re-entrant*)
 - Must always worry about inadvertent data-sharing

select ()

- `select ()` is designed for server-like applications that have many communication channels open at once
 - Data or space may become available at any time on any of the channels
 - You want to minimize the delay between when data/space becomes available and your process takes action (calls `read()` or `write()`)
- You give `select` a list of file descriptors, and `select ()` returns when any one of those selectors becomes readable or writable

select()

```
int select(  
    int nfd,          // number of fds of interest  
    fd_set *readfds,  // input fds of interest  
    fd_set *writefds, // output fds of interest  
    fd_set *errorfds, // fds where exception has occurred  
    struct timeval *timeout //when to time out  
)
```


select()

```
int select(  
    int nfd,          // number of fds of interest  
    fd_set *readfds,  // input fds of interest  
    fd_set *writefds, // output fds of interest  
    fd_set *errorfds, // fds where exception has  
    occurred  
    struct timeval *timeout //when to time out  
)
```

- nfd: This should be larger than the number of file descriptors you are interested in
 - FD_SETSIZE - max number of file descriptors
 - Probably more efficient if you pass the exact number of file descriptors you are interested in

File descriptor sets

- The three parameters `readfds`, `writelfds`, and `errorfds` are bit masks
 - Each bit of the number refers to one file descriptor
 - Bit 0 -> file descriptor 0, Bit 1 -> file descriptor 1, etc.
- UNIX provides you with functions to manipulate bit masks for the `select()` system call
 - `FD_ZERO` - set all bits to 0
 - `FD_SET` - set one specific bit to 1
 - `FD_ISSET` - determine if a specific bit is set to 1
 - `FD_CLR` - set one specific bit to 0

```
int fd1, fd2;
fd_set readset;
fd1 = open("file1", O_RDONLY);
fd2 = open("file2", O_RDONLY);

FD_ZERO(&readset);
FD_SET(fd1, &readset);
FD_SET(fd2, &readset);

switch (select(2, &readset, NULL, NULL, NULL))
{
    // do shtuff
}
```

Return value of select()

- -1 upon error
- 0 if time out
 - no events occurred
- Otherwise, the return value is the number of interesting file descriptors:
 - file descriptors where something happened

Waiting on lots of pipes

- What if you have lots of pipes open, and you are waiting for events on all of them?
 - ie you are waiting for data to become available on any of the pipes you have open
- You obviously don't want blocking reads/writes
 - You might end up blocking on fdA when there is data available on fdB
- However, you don't want to spend all of your time looping through your open pipes looking for data
- The answer: `select()`

Where did events occur?

- `select()` reuses the masks that you passed in.
- If a file descriptor became readable, the bit is set in the `readfds`
- If a file descriptor became writable, the bit is set in the `wrfdes`
- If an error or exceptional event occurred, the bit is set in the `errorfds`
- NOTE: it overwrites your mask, so you probably want to keep a copy of the original mask

Where did events occur?

- You need to loop through your file descriptors of interest to determine which ones the event occurred on

```
for (i = 0; i < 3; i++)  
{  
    if (FD_ISSET(i, &readset))  
    {  
        // Say there's data to read on descriptor I...  
        read(i, ...)  
    }  
}
```