



More Network Programming

[More Network Programming]

- Useful API's
 - Select/poll and advanced sockets tidbits
- HTTP push server
 - Request framing and server push concepts
 - Demo
- HTTP push server code
 - Components
 - Flow charts
 - Code walk-through (code is online)

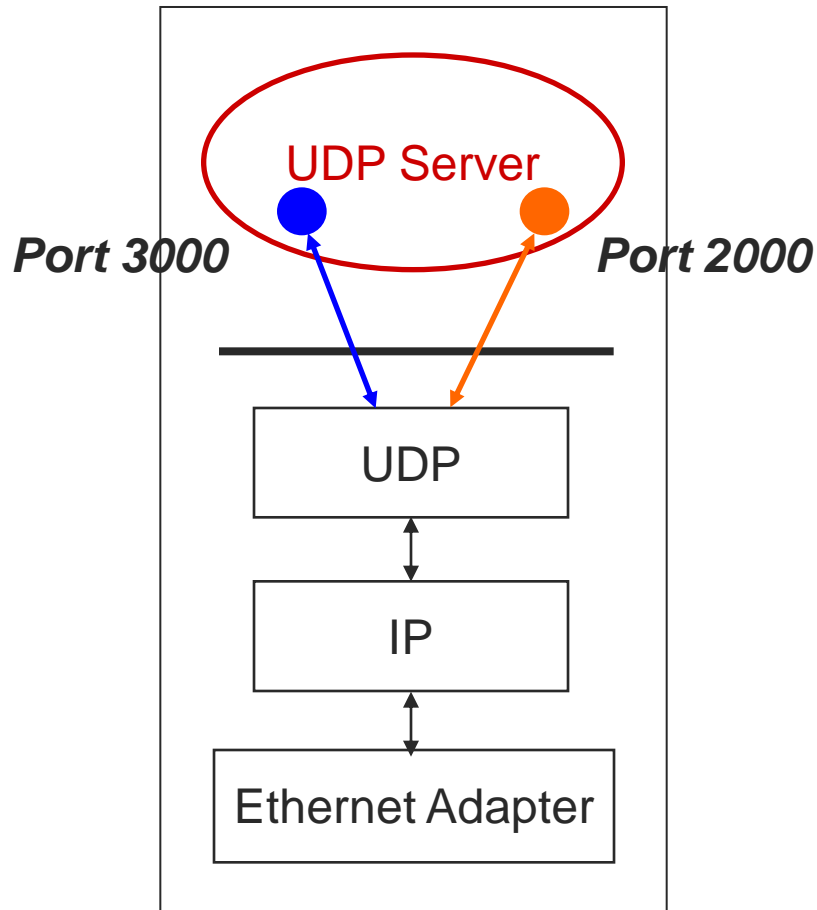


[More Network Programming]

- Useful Application Programming Interfaces
 - Topics
 - More advanced sockets
 - Unix file functionality
 - Multithreaded programming (Posix Threads)
 - Specific APIs
 - select/poll
 - advanced sockets



[A UDP Server]



- How can a UDP server service multiple ports simultaneously?



UDP Server: Servicing Two Ports

```
int s1;          /* socket descriptor 1 */
int s2;          /* socket descriptor 2 */

/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */

while(1) {
    recvfrom(s1, buf, sizeof(buf), ...);
    /* process buf */
    recvfrom(s2, buf, sizeof(buf), ...);
    /* process buf */
}
```

What problems does this code have?



Select and Poll

- Building timeouts with select/poll
 - Similar functions
 - Parameters
 - Set of file descriptors
 - Set of events for each descriptor
 - Timeout length
 - Return value
 - Set of file descriptors
 - Events for each descriptor
- Notes
 - Select is somewhat simpler
 - Poll supports more events



Select and Poll: Prototypes

■ Select

- Wait for readable/writable file descriptors

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

```
int select (int num_fds, fd_set* read_set, fd_set*  
            write_set, fd_set* except_set, struct timeval*  
            timeout);
```

■ Poll

- Poll file descriptors for events

```
#include <poll.h>
```

```
int poll (struct pollfd* pfd, nfds_t nfds, int  
          timeout);
```



Select

```
int select (int num_fds, fd_set* read_set, fd_set*
            write_set, fd_set* except_set, struct timeval*
            timeout);
```

- Wait for readable/writable file descriptors.
- Return:
 - Number of descriptors ready
 - -1 on error, sets `errno`
- Parameters:
 - `num_fds`:
 - number of file descriptors to check, numbered from 0
 - `read_set`, `write_set`, `except_set`:
 - Sets (bit vectors) of file descriptors to check for the specific condition
 - `timeout`:
 - Time to wait for a descriptor to become ready



[File Descriptor Sets]

- Bit vectors

- Often 1024 bits, only first `num_fds` checked
- Macros to create and check sets

```
fds_set myset;  
void FD_ZERO (&myset);      /* clear all bits */  
void FD_SET (n, &myset);    /* set bits n to 1 */  
void FD_CLEAR (n, &myset);  /* clear bit n */  
int FD_ISSET (n, &myset);   /* is bit n set? */
```



[File Descriptor Sets]

- Three conditions to check for
 - Readable
 - Data available for reading
 - Writable
 - Buffer space available for writing
 - Exception
 - Out-of-band data available (TCP)

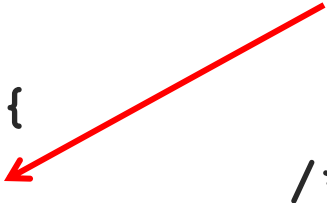


[Timeout]

■ Structure

Number of seconds since
midnight, January 1, 1970 GMT

```
struct timeval {  
    long tv_sec; /* seconds */  
    long tv_usec; /* microseconds */  
};
```



unix will have its own "Y2K" problem one second after 10:14:07pm, Monday January 18, 2038 (will appear to be 3:45:52pm, Friday December 13, 1901)



Select

- High-resolution sleep function
 - All descriptor sets **NULL**
 - Positive **timeout**
- Wait until descriptor(s) become ready
 - At least one descriptor in set
 - **timeout** **NULL**
- Wait until descriptor(s) become ready or timeout occurs
 - At least one descriptor in set
 - Positive **timeout**
- Check descriptors immediately (poll)
 - At least one descriptor in set
 - 0 **timeout**

Which file descriptors are set and what should the timeout value be?



Select: Example

```
fd_set my_read;  
FD_ZERO(&my_read);  
FD_SET(0, &my_read);
```

```
if (select(1, &my_read, NULL, NULL) == 1) {  
    ASSERT(FD_ISSET(0, &my_read));  
    /* data ready on stdin */
```

What went wrong:
after select indicates
data available on a
connection, read
returns no data?



Select: Timeout Example

```
int main(void) {
    struct timeval tv;
    fd_set readfds;

    tv.tv_sec = 2;
    tv.tv_usec = 500000;

    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);

    // don't care about writefds and exceptfds:
    select(1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");

    return 0;
}
```

Wait 2.5 seconds for something to appear on standard input



[Poll]

```
#include <poll.h>
```

```
int poll (struct pollfd* pfd, nfds_t nfds, int  
         timeout);
```

- Poll file descriptors for events.
- Return:
 - Number of descriptors with events
 - -1 on error, sets `errno`
- Parameters:
 - **pfd**:
 - An array of descriptor structures. File descriptors, desired events and returned events
 - **nfds**:
 - Length of the `pfd` array
 - **timeout**:
 - Timeout value in milliseconds



[Descriptors]

■ Structure

```
struct pollfd {  
    int fd;                /* file descriptor */  
    short events;          /* queried event bit mask */  
    short revents;         /* returned event mask */  
};
```

■ Note:

- Any structure with **fd** < 0 is skipped



[Event Flags]

- **POLLIN:**
 - data available for reading
- **POLLOUT:**
 - Buffer space available for writing
- **POLLERR:**
 - Descriptor has error to report
- **POLLHUP:**
 - Descriptor hung up (connection closed)
- **POLLVAL:**
 - Descriptor invalid



[Poll]

- High-resolution sleep function
 - 0 **nfds**
 - Positive **timeout**
- Wait until descriptor(s) become ready
 - **nfds** > 0
 - **timeout INFTIM** or -1
- Wait until descriptor(s) become ready or timeout occurs
 - **nfds** > 0
 - Positive **timeout**
- Check descriptors immediately (poll)
 - **nfds** > 0
 - 0 **timeout**



[Poll: Example]

```
struct pollfd my_pfds[1];
```

```
my_pfds[0].fd = 0;
```

```
my_pfds[0].events = POLLIN;
```

```
if (poll(&my_pfds, 1, INFTIM) == 1) {  
    ASSERT (my_pfds[0].revents & POLLIN);  
    /* data ready on stdin */  
}
```



[Advanced Sockets: `signal`]

```
signal (SIGPIPE, SIG_IGN);
```

- Call at start of main in server
- Allows you to ignore broken pipe signals which are generated when you write to a socket that has already been closed on the other side
- Default handler exits (terminates process)



[Advanced Sockets]

- How come I get "address already in use" from bind()?
 - You have stopped your server, and then restarted it right away
 - The sockets that were used by the first incarnation of the server are still active



Advanced Sockets: setsockopt

```
int yes = 1;
```

```
setsockopt (fd, SOL_SOCKET,  
           SO_REUSEADDR, (char *) &yes, sizeof  
           (yes));
```

- Call just before bind
- Allows bind to succeed despite the existence of existing connections in the requested TCP port
- Connections in limbo (e.g. lost final ACK) will cause bind to fail



[HTTP Request Framing]

- Characteristics

- ASCII-based (human readable)
- Framed by text lines
- First line is command
- Remaining lines are additional data
- Blank line ends request frame

```
GET /surf/too/much.html HTTP/1.0  
Date: 28 February 2000 01:25:53 CST  
Server: www.surfanon.org  
<blank line>
```



HTTP Server Push (Netscape-Specific)

- Idea
 - Connection remains open
 - Server pushes down new data as needed
 - Termination
 - Any time by server
 - Stop loading (or reload) by client
- Components
 - Header indicating multiple parts
 - New part replaces old part
 - New part sent any time
 - Wrappers for each part



HTTP Server Push (Netscape-Specific)

```
HTTP/1.0 200 OK
Content-type: multipart/x-mixed-replace;\
boundary=---never_in_document---
---never_in_document---
```

the data component

```
Content-type: text/html

(actual data)
---never_in_document---
```

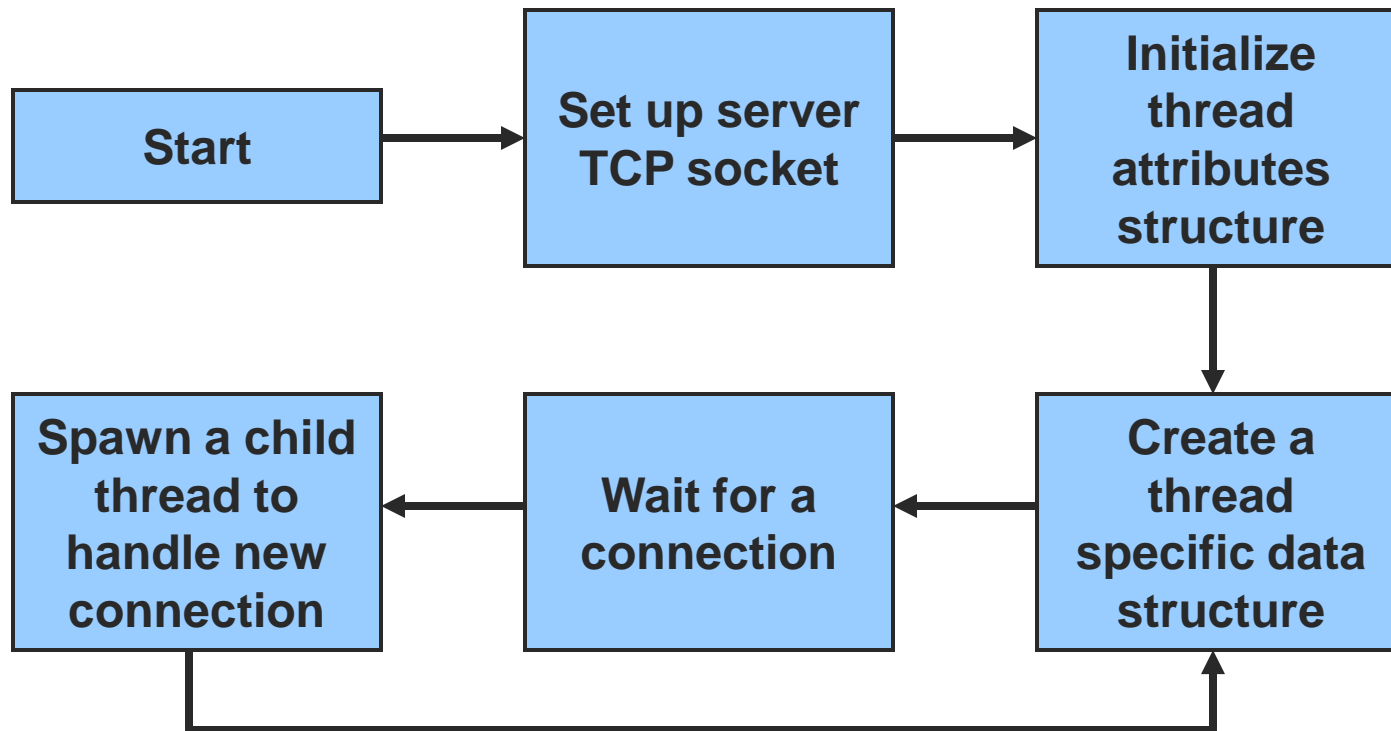


[Example]

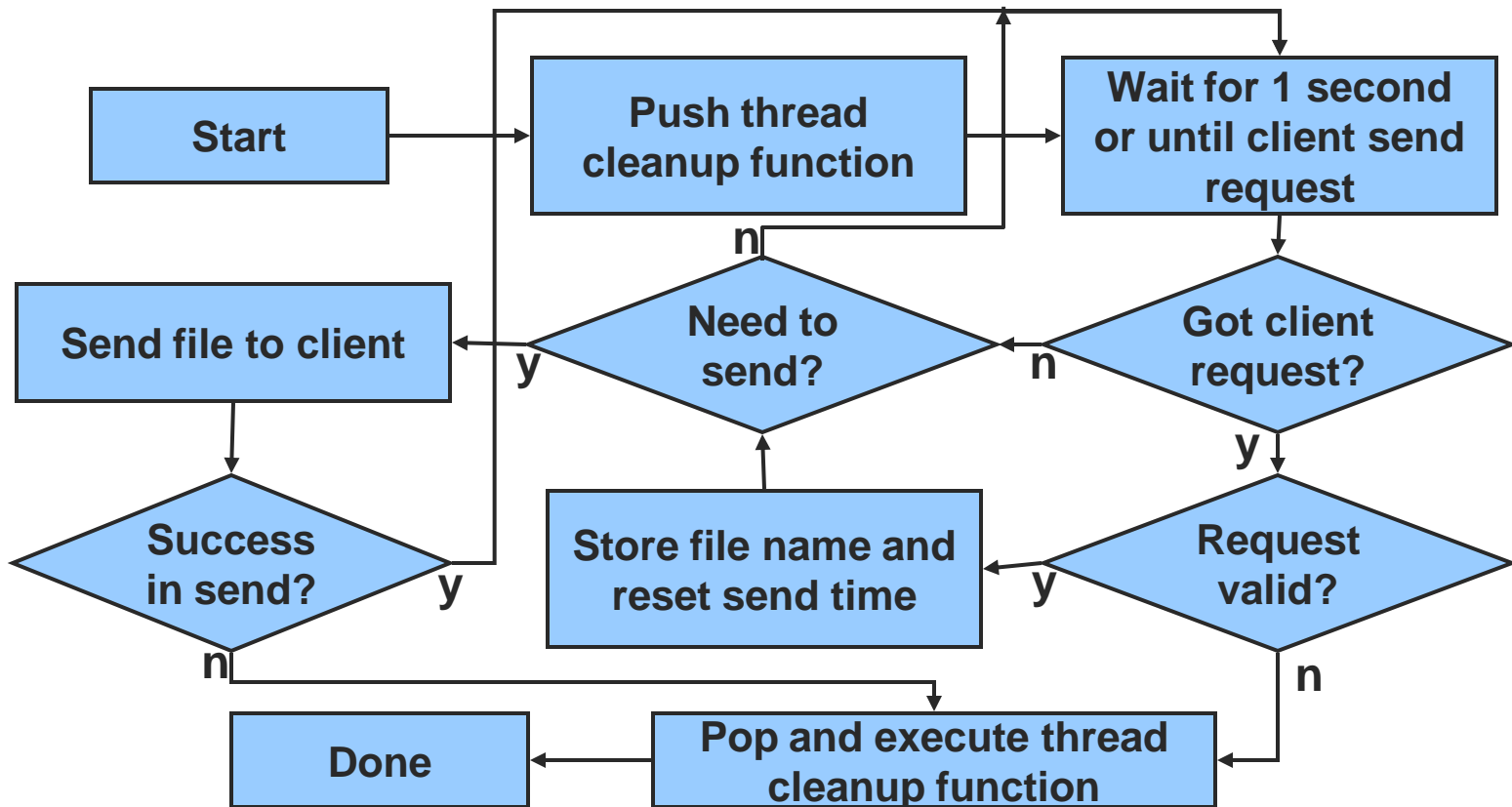
- Push server
 - Client-server connection remains open
 - Server pushes new data
- Use pthreads
- Main thread
 - Accepts new client connections
 - Spawns child thread for each client
- Child threads
 - Parses client requests
 - Constructs response
 - Checks for file modification
 - Pushes file when necessary



Example: Server Thread Flow Chart



Example: Client Thread Flow Chart



set_up_server_socket

```
static int set_up_server_socket (u_short port) {
    int fd;                /* server socket file descriptor */
    int yes = 1;          /* used for setting socket options */
    struct sockaddr_in addr; /* server socket address */
```

```
    /* Create a TCP socket. */
    if ((fd = socket (PF_INET, SOCK_STREAM, 0)) == -1) {
        perror ("set_up_server_socket/socket");
        return -1;
    }
```

```
    /* Allow port reuse with the bind below. */
    if (setsockopt (fd, SOL_SOCKET, SO_REUSEADDR,
                   (char*)&yes, sizeof (yes)) == -1) {
        perror ("set_up_server_socket/setsockopt");
        return -1;
    }
```



set_up_server_socket

```
/* Set up the address. */
bzero (&addr, sizeof (addr));
addr.sin_family      = AF_INET;      /* Internet address */
addr.sin_addr.s_addr = INADDR_ANY; /* fill in local IP address */
addr.sin_port        = htons (port); /* port specified by caller*/
```

```
/* Bind the socket to the port. */
if (bind (fd, (struct sockaddr*)&addr, sizeof (addr)) == -1) {
    perror ("set_up_server_socket/bind");
    return -1;
}
```

```
/* Listen for incoming connections (socket into passive state). */
if (listen (fd, BACKLOG) == -1) {
    perror ("set_up_server_socket/listen");
    return -1;
}
```

```
/* The server socket is now ready. */
return fd;
```

```
}
```



wait_for_connections

```
static void wait_for_connections (int fd){
    pthread_attr_t attr; /* initial thread attributes */
    thread_info_t* info; /* thread-specific connection information */
    int len; /* value-result argument to accept */
    pthread_t thread_id; /* child thread identifier */
```

```
/* Signal a bug for invalid descriptors. */
ASSERT (fd > 0);
```

```
/* Initialize the POSIX threads attribute structure. */
if (pthread_attr_init (&attr) != 0) {
    fputs ("failed to initialize pthread attributes\n", stderr);
    return;
}
```

```
/* The main thread never joins with the children. */
if (pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED)
    != 0) {
    fputs ("failed to set detached state attribute\n", stderr);
    return;
}
```



wait_for_connections

```
/* Use an infinite loop to wait for connections. For each
connection, create a structure with the thread-specific data, then
spawn a child thread and pass it the data. The child is
responsible for deallocating the memory before it terminates. */
while (1) {

    /* Create a thread information structure and initialize
    fields that can be filled in before a client contacts
    the server. */

    if ((info = calloc (1, sizeof (*info))) == NULL) {
        perror ("wait_for_connections/calloc");
        return;
    }
    info->fname      = NULL;
    info->last_sent = (time_t)0;
```



wait_for_connections

```
/* Wait for a client to contact the server. */
len = sizeof (info->addr);
if ((info->fd = accept (fd, (struct sockaddr*)&info->addr,
                      &len)) == -1) {
    perror ("accept");
    return;
}

/* Create a thread to handle the client. */
if (pthread_create (&thread_id, &attr,
                  (void* (*)(void*))client_thread, info) != 0) {
    fputs ("failed to create thread\n", stderr);

    /* The child does not exist, the main thread must clean up. */
    close (info->fd);
    free (info);
    return;
}
}
```



client_thread

```
void client_thread (thread_info_t* info) {  
    /* Check argument. */  
    ASSERT (info != NULL);
```

```
    /* Free the thread info block whenever the thread terminates.  
    Note that pushing this cleanup function races with external  
    termination. If external termination wins, the memory is never  
    released. */  
    pthread_cleanup_push ((void (*)(void*))release_thread_info, info);
```

```
    /* Loop between waiting for a request and sending a new copy of  
    the current file of interest. */  
    while (read_client_request (info) == 0 &&  
          send_file_to_client (info) == 0);
```

```
    /* Defer cancellations to avoid re-entering deallocation routine  
    (release_thread_info) in the middle, then pop (and execute) the  
    deallocation routine.*/  
    pthread_setcanceltype (PTHREAD_CANCEL_DEFERRED, NULL);  
    pthread_cleanup_pop (1);
```

```
}
```



client_has_data

```
static int client_has_data (int fd) {
    fd_set read_set;
    struct timeval timeout;

    /* Check argument. */
    ASSERT (fd > 0);

    /* Set timeout for select. */
    timeout.tv_sec = CHECK_PERIOD;
    timeout.tv_usec = 0;

    /* Set read mask for select. */
    FD_ZERO (&read_set);
    FD_SET (fd, &read_set);

    /* Call select. Possible return values are {-1, 0, 1}. */
    if (select (fd + 1, &read_set, NULL, NULL, &timeout) < 1) {
        /* We can't check errno in a thread--assume nothing bad has happened. */
        return 0;
    }

    /* Select returned 1 file descriptor ready for reading. */
    return 1;
}
```

