Socket Programming

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

November 3, 2022

https://ratcoinc.github.io/Networks/

WEB PAGE

Dealing with Partial Sends

- The dispatch methods like `write()`, `send()` etc. might not send all the bytes we asked it to
- Due to circumstances beyond our control, the kernel may decide not to send all the data

Dealing with Partial Sends

- The dispatch methods like `write()`, `send()` etc. might not send all the bytes we asked it to
- Due to circumstances beyond our control, the kernel may decide not to send all the data
- The unsent data still resides in our buffer space
- It is now our responsibility to send the remaining data

Dealing with Partial Sends

One can write a small wrapper function[1]:

```
int sendall(int sd, char *buf, int len) {
    int total = 0;        // how many bytes we've sent
    int bytesleft = len; // how many we have left to send

    while(bytesleft > 0) {
        int n = send(sd, buf+total, bytesleft, 0);
        if (n < 0) { break; } // ERROR sendall failed
        total += n;
        bytesleft -= n;
    }

    return total; // return the actual number of bytes sent
}
```

[1] Adapted from: `https://beej.us/guide/bgnet/html/index-wide.html#sendall`

Dealing with Partial Sends

A typical usage[1] of our wrapper method:

```
char buf[1024];
. . .
int len = strlen(buf);
int n = sendall(sd, buf, len);
if (n < len) {
    perror("ERROR in sendall");
    printf("We only sent %d bytes!\n", n);
}
```

[1] Adapted from: https://beej.us/guide/bgnet/html/index-wide.html#sendall

Dealing with Partial Sends

A typical usage[1] of our wrapper method:

```
char buf[1024];
. . .
int len = strlen(buf);
int n = sendall(sd, buf, len);
if (n < len) {
    perror("ERROR in sendall");
    printf("We only sent %d bytes!\n", n);
}
```

How does the receiver know when one packet ends and another begins?

[1] Adapted from: https://beej.us/guide/bgnet/html/index-wide.html#sendall

Dealing with Partial Sends

A typical usage[1] of our wrapper method:

```
char buf[1024];
. . .
int len = strlen(buf);
int n = sendall(sd, buf, len);
if (n < len) {
    perror("ERROR in sendall");
    printf("We only sent %d bytes!\n", n);
}
```

How does the receiver know when one packet ends and another begins?
Data often needs to be ***encapsulated***[2] in case of variable sized packets

[1] Adapted from: https://beej.us/guide/bgnet/html/index-wide.html#sendall

[2] Data Encapsulation: https://beej.us/guide/bgnet/html/index-wide.html#sonofdataencap

Monitoring Multiple Sockets

- By default, `read()`, `recv()` calls ***block*** (a fancy name for *sleep*) the current execution

Monitoring Multiple Sockets

- By default, `read()`, `recv()` calls ***block*** (a fancy name for *sleep*) the current execution
- To monitor multiple sockets (multiple clients) for received data one possibility is to run multiple processes using `fork()` (or multiple threads using `pthread`) each monitoring one socket

Monitoring Multiple Sockets

- By default, `read()`, `recv()` calls ***block*** (a fancy name for *sleep*) the current execution
- To monitor multiple sockets (multiple clients) for received data one possibility is to run multiple processes using `fork()` (or multiple threads using `pthread`) each monitoring one socket
- Multiple processes are harder to coordinate, consume more resources, and sharing data also requires special treatments[1]

[1] `https://stackoverflow.com/questions/33889868/socket-programming-multiple-connections-forking-or-fd-set`

Monitoring Multiple Sockets

- By default, `read()`, `recv()` calls ***block*** (a fancy name for *sleep*) the current execution
- To monitor multiple sockets (multiple clients) for received data one possibility is to run multiple processes using `fork()` (or multiple threads using `pthread`) each monitoring one socket
- Multiple processes are harder to coordinate, consume more resources, and sharing data also requires special treatments[1]
- What about using a non-blocking socket `fcntl(sockfd, F_SETFL, O_NONBLOCK);`[2,3]

[1] https://stackoverflow.com/questions/33889868/socket-programming-multiple-connections-forking-or-fd-set

[2] https://beej.us/guide/bgnet/html/index-wide.html#blocking

[3] https://man7.org/linux/man-pages/man2/fcntl.2.html

Monitoring Multiple Sockets

- By default, `read()`, `recv()` calls ***block*** (a fancy name for *sleep*) the current execution
- To monitor multiple sockets (multiple clients) for received data one possibility is to run multiple processes using `fork()` (or multiple threads using `pthread`) each monitoring one socket
- Multiple processes are harder to coordinate, consume more resources, and sharing data also requires special treatments[1]
- What about using a non-blocking socket `fcntl(sockfd, F_SETFL, O_NONBLOCK);`[2,3]
- Write an infinite loop, poll every socket for data, if no data is available we get -1

[1] https://stackoverflow.com/questions/33889868/socket-programming-multiple-connections-forking-or-fd-set

[2] https://beej.us/guide/bgnet/html/index-wide.html#blocking

[3] https://man7.org/linux/man-pages/man2/fcntl.2.html

Monitoring Multiple Sockets

- By default, `read()`, `recv()` calls ***block*** (a fancy name for *sleep*) the current execution
- To monitor multiple sockets (multiple clients) for received data one possibility is to run multiple processes using `fork()` (or multiple threads using `pthread`) each monitoring one socket
- Multiple processes are harder to coordinate, consume more resources, and sharing data also requires special treatments[1]
- What about using a non-blocking socket `fcntl(sockfd, F_SETFL, O_NONBLOCK);`[2,3]
- Write an infinite loop, poll every socket for data, if no data is available we get -1
- This is a bad idea! Program doing *busy-wait* consumes CPU time

[1] https://stackoverflow.com/questions/33889868/socket-programming-multiple-connections-forking-or-fd-set

[2] https://beej.us/guide/bgnet/html/index-wide.html#blocking

[3] https://man7.org/linux/man-pages/man2/fcntl.2.html

Monitoring Multiple Sockets

- A more elegant solution for monitoring multiple sockets is provided by `poll()`[1] and `select()`[2] APIs
- The OS does all the dirty work and lets us know when a socket is ready for I/O, while our process can sleep, saving system resources

[1] Synchronous I/O Multiplexing: `https://beej.us/guide/bgnet/html/index-wide.html#poll`

[2] Old School, more portable: `https://beej.us/guide/bgnet/html/index-wide.html#select`

Monitoring Multiple Sockets

- A more elegant solution for monitoring multiple sockets is provided by `poll()`[1] and `select()`[2] APIs
- The OS does all the dirty work and lets us know when a socket is ready for I/O, while our process can sleep, saving system resources
- We keep an array of sockets to monitor along with what kind of events we want to monitor for

[1] Synchronous I/O Multiplexing: `https://beej.us/guide/bgnet/html/index-wide.html#poll`

[2] Old School, more portable: `https://beej.us/guide/bgnet/html/index-wide.html#select`

Monitoring Multiple Sockets

- A more elegant solution for monitoring multiple sockets is provided by `poll()`[1] and `select()`[2] APIs
- The OS does all the dirty work and lets us know when a socket is ready for I/O, while our process can sleep, saving system resources
- We keep an array of sockets to monitor along with what kind of events we want to monitor for
- A structure called `pollfd` is used with `poll()` API[3]

```
struct pollfd {     // defined in poll.h
    int fd;         // the socket descriptor to monitor
    short events;   // bitmap of events we want to monitor
    short revents; // returned bitmap of events that occurred
};
```

[1] Synchronous I/O Multiplexing: https://beej.us/guide/bgnet/html/index-wide.html#poll

[2] Old School, more portable: https://beej.us/guide/bgnet/html/index-wide.html#select

[3] https://man7.org/linux/man-pages/man2/poll.2.html

Monitoring Multiple Sockets

- A more elegant solution for monitoring multiple sockets is provided by `poll()`[1] and `select()`[2] APIs
- The OS does all the dirty work and lets us know when a socket is ready for I/O, while our process can sleep, saving system resources
- We keep an array of sockets to monitor along with what kind of events we want to monitor for
- A structure called `pollfd` is used with `poll()` API[3]

```
struct pollfd {    // defined in poll.h
    int fd;        // the socket descriptor to monitor
    short events;  // bitmap of events we want to monitor
    short revents; // returned bitmap of events that occurred
};
```

- Two common events are `POLLIN` (socket is ready to be read) `POLLOUT` (socket is ready for writting)

[1] Synchronous I/O Multiplexing: `https://beej.us/guide/bgnet/html/index-wide.html#poll`

[2] Old School, more portable: `https://beej.us/guide/bgnet/html/index-wide.html#select`

[3] `https://man7.org/linux/man-pages/man2/poll.2.html`

The poll() API

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

waits for one of a given set of file descriptors to become ready for I/O

Return value: On success, returns a nonnegative value denoting the number of file descriptors on which some event (I/O or error) has happened. 0 is returned in case of a time-out. On error, -1 is returned.

Parameters:

`fds`: set of file descriptors to be monitored, negative fds are ignored

`nfds`: number of items in the `fds` array

`timeout`: the number of milliseconds that `poll()` should block waiting until either (1) a fd becomes ready, (2) interrupted by a signal handler, or (3) the timeout expires; a negative timeout waits forever

[1] `https://man7.org/linux/man-pages/man2/poll.2.html`

[2] Note that, a monitored socket also returns 'ready to read' status (`POLLIN`) when a new incoming connection is ready to be accepted

Using `poll()` API

- Create an array of `pollfd`

Using `poll()` API

- Create an array of `pollfd`
- Put the server socket that listens for incoming connections into the list, with `POLLIN` as the monitored event

Using `poll()` API

- Create an array of `pollfd`
- Put the server socket that listens for incoming connections into the list, with `POLLIN` as the monitored event
- Invoke a `poll()` on this list

Using `poll()` API

- Create an array of `pollfd`
- Put the server socket that listens for incoming connections into the list, with `POLLIN` as the monitored event
- Invoke a `poll()` on this list
- Note that `poll()` only returns the number of sockets for which some events have occurred
- Manually scan the entire[1] list and look for non-zero `revents` field

[1] we may terminate early once the specified number of non-zero `revents` field has been observed

Using `poll()` API

- Create an array of `pollfd`
- Put the server socket that listens for incoming connections into the list, with `POLLIN` as the monitored event
- Invoke a `poll()` on this list
- Note that `poll()` only returns the number of sockets for which some events have occurred
- Manually scan the entire[1] list and look for non-zero `revents` field
- How to add a new fd into the list?

[1] we may terminate early once the specified number of non-zero `revents` field has been observed

Using `poll()` API

- Create an array of `pollfd`
- Put the server socket that listens for incoming connections into the list, with `POLLIN` as the monitored event
- Invoke a `poll()` on this list
- Note that `poll()` only returns the number of sockets for which some events have occurred
- Manually scan the entire[1] list and look for non-zero `revents` field
- How to add a new fd into the list?- maintain a counter for number of fds currently present in the list, simply add the new entry in the end and increment the counter

[1] we may terminate early once the specified number of non-zero `revents` field has been observed

Using `poll()` API

- Create an array of `pollfd`
- Put the server socket that listens for incoming connections into the list, with `POLLIN` as the monitored event
- Invoke a `poll()` on this list
- Note that `poll()` only returns the number of sockets for which some events have occurred
- Manually scan the entire[1] list and look for non-zero `revents` field
- How to add a new fd into the list?- maintain a counter for number of fds currently present in the list, simply add the new entry in the end and increment the counter
- What about deleting?

[1] we may terminate early once the specified number of non-zero `revents` field has been observed

Using `poll()` API

- Create an array of `pollfd`
- Put the server socket that listens for incoming connections into the list, with `POLLIN` as the monitored event
- Invoke a `poll()` on this list
- Note that `poll()` only returns the number of sockets for which some events have occurred
- Manually scan the entire[1] list and look for non-zero `revents` field
- How to add a new fd into the list?- maintain a counter for number of fds currently present in the list, simply add the new entry in the end and increment the counter
- What about deleting?- can copy the last element in the array over-top of the one being deleted and decrease the counter; or simply set the `fd` field to a negative number and `poll()` ignores it

[1] we may terminate early once the specified number of non-zero `revents` field has been observed

Using `poll()` API

- Create an array of `pollfd`
- Put the server socket that listens for incoming connections into the list, with `POLLIN` as the monitored event
- Invoke a `poll()` on this list
- Note that `poll()` only returns the number of sockets for which some events have occurred
- Manually scan the entire[1] list and look for non-zero `revents` field
- How to add a new fd into the list?- maintain a counter for number of fds currently present in the list, simply add the new entry in the end and increment the counter
- What about deleting?- can copy the last element in the array over-top of the one being deleted and decrease the counter; or simply set the `fd` field to a negative number and `poll()` ignores it
- List can be dynamically resized with `realloc()`- doubling/halving

[1] we may terminate early once the specified number of non-zero `revents` field has been observed

A Simple Poll Server

Run `server4.c`, then do two or more `telnet` to it
(message from one client is sent to all others)

Sending Data to Multiple Hosts

- **Broadcasting** sends the data to all hosts in the same local network
- For broadcast we need to use UDP (not TCP) and IPv4
- `SO_BROADCAST` needs to be enabled via `setsockopt()`[1]
- The message can be sent to a specific subnet's broadcast address (e.g. 192.168.1.255 for subnet 192.168.1.0/24) or to the global broadcast address 255.255.255.255, aka `INADDR_BROADCAST`
- Avoid broadcast if possible, instead use multicast

[1] `https://beej.us/guide/bgnet/html/index-wide.html#broadcast-packetshello-world`

Sending Data to Multiple Hosts

- **Multicasting** sends the data to a group of hosts in the same local network
- Here `IP_MULTICAST_IF` needs to be enabled via `setsockopt()`
- A multicast group is maintained using `IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP` through `setsockopt()`
- A class D address (224.0.0.0 to 239.255.255.255) is used as a multicast address
- A host can be part of multiple groups[4]

[1] http://www.cs.unc.edu/~jeffay/dirt/FAQ/comp249-001-F99/mcast-socket.html

[2] https://www.ibm.com/docs/en/aix/7.3?topic=sockets-ip-multicasts

[3] https://docs.oracle.com/cd/E26502_01/html/E35299/sockets-137.html

[4] https://stackoverflow.com/questions/9243292/subscribing-to-multiple-multicast-groups-on-one-socket

A Simple Multicast Program

`server5.c`, `client5.c`
run the server in one terminal
run the client in two or more terminals
type strings in server, all client prints the message

[1] Adapted from: `https://web.cs.wpi.edu/~claypool/courses/4514-B99/samples/multicast.c`