Sockets 5: Endianess, Options, Servers

1. Introduction and Addresses
2. Basic System Calls
3. Socket I/O Syscalls
4. IP Addresses and Hostnames
5. **Endianess, Options, Servers**
   - **endianess**
   - **network byte order**
   - **socket options**
   - **server designs**

Endianness

When data such as integers are stored in *multiple bytes*, there are *alternative orders* in which the bytes may be stored in memory.

This is referred to as **endianness** or **byte ordering**.

Byte order is important in network computing because machines using *different architectures* will need to exchange information such as IP addresses.

The two most common byte orderings are:

- **big endian** – most significant byte first (lowest address)
- **little endian** – least significant byte first

(The terms are from *Gulliver's Travels*.)

Endianness (contd.)

Note that because bits are not individually addressable typically, it is always assumed to be low bit "first" (at low end of byte).

The x86 architecture uses a little endian scheme, while big endian is used in Sun's SPARC and IBM's PowerPC.

While each machine will use some **host byte order**, information to be exchanged must be convereted to a *common* **network byte order**.

(Network byte order is big endian!)

Functions are provided to translate integers from host to network byte order and vice versa.

These must be used with IP addresses and port numbers not created via network programming calls.

Endianness (contd.)

Conversion functions:

- **htons** – convert 2 byte host order to network order:
  `uint16_t htons(uint16_t hostshort)`
- **htonl** – convert 4 byte host order to network order:
  `uint32_t htonl(uint32_t hostlong)`
- **ntohs** – convert 2 byte network order to host order:
  `uint16_t ntohs(uint16_t netshort)`
- **ntohl** – convert 4 byte network order to host order:
  `uint32_t ntohl(uint32_t netlong)`

getsockopt, setsockopt

There are various options that can be set on sockets, and two calls allow one to retrieve and set these options:

```
int getsockopt(int s, int level, int optname,
               void *optval, socklen_t *optlen);

int setsockopt(int s, int level, int optname,
               const void *optval, socklen_t optlen);
```

For example, to allow a port to be immediately reused:
(this should be done for servers to allow immediate restart)

```
int i=1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
```

Server Designs

Servers are generally designed to run *indefinitely*, so will contain an *infinite loop!*

Connection-oriented (e.g., TCP) servers will have a separate file descriptor for each connection, and can potentially have multiple active connections (clients) at any time.

There are two basic classes of connection-oriented server:

- **sequential/iterative** – handles a single connection until done, then handles the next, etc.
- **parallel/concurrent** – handles multiple connections all concurrently (simultaneously)

Server Designs (contd.)

There are three main approaches to building parallel/concurrent connection-oriented servers:

1. `fork()` off a *subprocess* to handle each active connection (listening server runs in parent process)
2. use `pthread_create()` to create a new *thread* to handle each active connection (listening server runs in main thread)
3. handle all clients in a single process thread, using `epoll` (or older `poll()` or `select()`) to alternate among all active connections based on whether there is pending I/O

The first two approaches are relatively simple to implement, but cannot handle very large numbers of clients due to their resource requirements.

Modern high capacity servers (e.g., **nginx**) use the third approach.

Server Designs (contd.)

Parallel servers that handle connections in separate processes/threads, must deal with the need to collect these processes/threads when clients terminate.

However, since clients terminate *asynchronously*, it is problematic to collect the processes/threads in the main listening server loop.

It is therefore common to collect subprocess children by either:

1. setting `SIGCHLD` to be *ignored*
2. setting up a handler for `SIGCHLD` that calls `waitpid()`

With client threads, it is common to set the threads as *detached* by using `pthread_detach()`.