Assignment 8:

Using TCP/IP Sockets
Recitation for ECES 338
April 5, 1999

In general, the code surrounding TCP/IP sockets can be somewhat confusing, because it involves a lot of complex system calls. Once a connection (or more appropriately, *socket*) has been established, it acts just like a file, so the more recognizable `read()` and `write()` system calls can be used in order to actually move data between machines. For the purposes of this lecture, I am going to assume that you have a "functional" knowledge of the Internet Protocol (IP). Basically, you should have some general knowledge of how addressing works, and how the many protocols interact with one another.

The Monkey Book gives a very detailed description of Sockets. Unfortunately, their detail can be somewhat overwhelming, because they attempt to cover both UNIX domain as well as Berkeley-style sockets simultaneously. UNIX domain sockets are similar to named pipes – the transmission of data occurs through an actual file on the filesystem. Thus, these sockets are only useful for intra-machine communication. Berkeley sockets, on the other hand, use TCP/IP as the transmission medium, and are useful for sending information between computers. Furthermore, these sockets form the foundation of what is known today as the "Internet", so they are what I'll focus on here.

Basically, today's lecture is going to focus on connection-oriented sockets, as discussed on page 278 – 300 of *The Monkey Book*. Furthermore, I know that you all like examples, so you should pay particular attention to Programs 10.6 and 10.7 (pages 297 and 298, respectively).

- `int socket (int family, int type, int protocol)`
  - This system call creates a new socket instance. Both clients and servers need to make use of this system call. This system call returns an open socket descriptor (like a file descriptor), that is configured as specified by the parameters. The first parameter specifies the protocol family, which can be either `PF_UNIX` or `PF_INET` (Internet Protocol). The second argument specifies a sub-type for the connection family. For a connection-oriented session, specify `SOCK_STREAM`. The third parameter seems to be depreciated, and is always set to zero.

- `int bind (int socket, const struct sockaddr *name, int namelen)`
  - The connection-based paradigm implies a client/server relationship. As such, only servers use this system call. In essence, this call associates an address and port with a given socket. Thus, if your environment affords you with multiple addresses and/or ports, this call ratchets a specific pair to your socket. The first parameter is a socket descriptor, as established by a previous call to `socket()`. The second parameter is a pointer to a pretty heinous structure. The third

parameter is the size of the structure passed as the second parameter. For internet-style connections (`AF_INET`), a pointer to the following `sockaddr` structure should be supplied:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

- The `sin_family` member should be set to the `AF_INET` constant. The `sin_addr` member represents the actual address that should be bound to. Again, there is a lot of depreciated information lying around here, but the value needs to be stored in the `s_addr` member of the `in_addr` structure. Furthermore, the value stored must be in the correct endian order, so a special system call needs to be used. Finally, the `sin_port` member specifies the port that should be used with the given socket. Again, the byte order matters, so a special system call must be used. Consider the following example:

  Example:

  ```
  struct sockaddr_in serv_sa;
  serv_sa.sin_family = AF_INET;
  serv_sa.sin_addr.s_addr = htonl (INADDR_ANY);
  serv_sa.sin_port = htons (7777);
  bind (sd, (struct sockaddr *) &serv_sa, sizeof (serv_sa))
  ```

  This example binds the socket (established in a previous call, and stored in `sd`) to port 7777, on all IP addresses in the system. Thus, for a machine that has four IP addresses, this socket would be present on each address. Finally, some reference must be made to the special "byte-ordering" functions mentioned earlier. The function `htons()` manipulates the endianess of 16-bit wide integers, while `htonl()` works with 32-bit wide (long) integers.

- `int listen (int socket, int backlog)`
  - Again, this system call is server-only. It causes the calling process to start listening for connections on the address/port pair specified by `bind()`. This system call actually establishes a queue, so that multiple incoming connections can be handled in the order received. The first parameter is a socket descriptor, the second specifies the size of the queue (but may have a system-imposed maximum). According to the Solaris 2.5.1 manual page, there is currently no limit imposed upon the `listen()` queue size.

- `int accept (int socket, struct sockaddr *addr, int *addrlen)`
  - This function call will block (by default) the server process, while it waits for an incoming connection. When it completes, a connection will have been established, and it will be up to the server code to handle the client's request(s). While not mandatory, it is strongly recommended that you use a loop with `accept()`, that forks off a new child to handle each incoming request. The

purpose of this loop is to create a server that can simultaneously handle multiple client requests. The first parameter is the socket that has had the `bind()` and listen() calls already applied to it. The second parameter is a generic `sockaddr` pointer, in which `accept()` will return information for the client whose connection attempt that it accepted. The third parameter represents the length of the aforementioned `sockaddr` structure. Finally, `accept()` returns a new socket descriptor, to be used with the newly established client connection. Consider the following example:

Example:

```
struct sockaddr_in cli_sa;
size_returned = sizeof (cli_sa);
accept (sd, (struct sockaddr *) &cli_sa, &size_returned);
```

Basically, this example statically allocates an area of memory (the `cli_sa` variable), and calls `accept()`. Once a connection has been accepted, all of the relevant client information will be stored in memory, and can be referenced through the `cli_sa` variable.

- `int connect (int socket, struct sockaddr *name, int namelength)`
  - In order to establish a connection to a *server*, the *client* must use the `connect()` system call. The parameters required by this function are similar to those of the `bind()` system call. Basically, for a given socket, an IP address and port need to be specified. The `connect()` system call will then connect the remote end of the local socket to the server.