Question 1

1. IP address, port, protocol
2. .
3. Bind() (unconfirmed)
4. Accept() and Connect()
5. DNS
6. .
7. Listen()
8. TRUNC
9. -, lseek, seek\_set
10. lstat

Question 2

1. Unbuffered I/O simply means that don't use any buffer while reading or writing. Generally, when we use System calls like read() and write() they read and write char by char and can cause huge performance degradation
2. Buffered I/O is the technique of temporarily storing the results of an I/O operation in user-space before transmitting it to the kernel (in the case of writes) or before providing it to your process (in the case of reads). By so buffering the data, you can minimize the number of system calls and can block-align I/O operations, which may improve the performance of your application.
3. Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details. A procedure call is also sometimes known as a function call or a subroutine call.

RPC uses the client-server model. The requesting program is a client and the service providing program is the server. Like a regular or local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned. However, the use of lightweight processes or threads that share the same address space allows multiple RPCs to be performed concurrently.

Question 3

1. A file descriptor is a low-level integer "handle" used to identify an opened file (or socket, or whatever) at the kernel level, in Linux and other Unix-like systems. You pass "naked" file descriptors to actual Unix calls, such as [read()](http://linux.die.net/man/2/read), [write()](http://linux.die.net/man/2/write) and so on.

A FILE pointer is a C standard library-level construct, used to represent a file. The FILE wraps the file descriptor, and adds buffering and other features to make I/O easier. You pass FILE pointers to standard C functions such as [fread()](http://linux.die.net/man/3/fread) and [fwrite()](http://linux.die.net/man/3/fwrite).

1. fopen() is a library function while open() is a system call which is why open() gives greater control.
2. fopen() is faster because it has buffered I/O compared to open() which is non buffered. The extra speed gives it synchronization problems.
3. Open() is POSIX defined.
4. Flags: **O\_RDONLY**, **O\_WRONLY**, or **O\_RDWR , O\_APPEND, O\_ASYNC** (Enable signal-driven I/O)

Mode: S\_I**R**USR | S\_IW**USR** | S\_IR**GRP** | S\_IWGRP | S\_IR**OTH**

R = read, W =write , X =execute

Flags define what kind of permissions are defined with the file. Mode defines which user owns the file and what group that user lies in.

1. ssize\_t ret, nr;

while (len != 0 && (ret = write (fd, buf, len){

if (ret == -1){

if (errno == EINTR){

continue;

}

perror("write");

break;

}

len -= ret;

buf += ret;

}

Partial writing refers to that scenario when memory runs out while writing data so only part of data is written.

1. fflush: Flushes (clears) the output buffer of the file handle on the OS.

fsync: Synchronizes all the data variables. Very useful for shared variables.

1. Buffered IO may have inconsistent/incorrect data if there are multiple processes that are sharing the data.

It is possible for the buffer to write data to the file that is outdated. In such cases, character by character (Kernel IO) would be preferred.

1. Data written by a process is first accumulated in a buffer in memory until the buffer is full (or a time limit elapses), at which point the entire buffer is written to the device at once. This reduces the number of real I/O operations involved and makes the system process faster.
2. Connection oriented communication such as TCP is preferred for file transfers because it ensures all packets of the file are received.

If any packet is missed, the protocol would know of it and retransmit ensuring successful transfer.

It becomes more important as the file size grows.

1. Synchronous implies either blocking a thread that would otherwise do other useful things or dedicating a thread to each connection. Either way, this does not scale very well. For simple applications with few or just one active connection, it might be okay. But for any scenario where you need to handle any significant number of concurrent connections, the asynchronous APIs are the only ones that provide adequate performance.
2. recv() is a blocking function so it is placed after the send() function in the code. Every time a message goes from one socket to another, it checks its message buffer for any message received. Before the message buffer can be checked, we would need to establish a connection by defining sockets for both the client and the server. These sockets after instantiation perform a 3-way TCP handshake (SYN, SYN/ACK, ACK). If the request times out for any of these, steps, it restarts its handshake.