Summary

Wednesday, August 26, 2015 12:14 AM

Summary

- 2.1 - Which of your local implementations of standard application clients are fully parameterized? Why is full parameterization needed?
    - FTP, Telnet, SFTP, SSH, SSL, RDP, Printers, HTML, AirPlay, etc, etc
    - Full parameterization is needed to set the destination. Without the destination specified in the parameters then the connection is relying on the server to have a service on the default port 80 which can interpret the protocol.
- 2.2 - Are standard applications like Telnet, ftp, smtp, and NFS (network file system) connectionless or connection-oriented?
    - Since the standard applications listed are involved in the transfer of data and interactive commands I will say that these applications are connection-oriented using TCP rather than UDP which does not perform a checksum and has no reliability.
- 2.3 - What does TCP/IP specify would happen if no server exists when a client request arrives? (Hint: look at icmp.) What happens on your local system?
    - Error would generate stating no host/server exists.
    - No such host exists.
- 2.4 - Write down the data structures and message formats needed for a stateless file server. What happens if two or more clients access the same file? What happens if a client crashes before closing a file?
    - Data Structure:

```
Struct stateLessMessage
{
    bool operation; //Read = 0 Write =1
    char fileName[255]; //file name
    int xferPosition; //position in file which starts the transfer
    int dataSize; //size of the data to transfer
    std::vector<char> bigData; //actual binary data stored in a vector.
    //This will require research - http://www.cplusplus.com/forum/beginner/53018/
}
```

    - Message Format: operation | fileName | xferPosition | dataSize | bigData
    - Two or more clients access the same file -
        - Stateless: Since the interactions with the stateless file server are idempotent, that is identical response for every identical request, there is not an issue if two clients access the same file. The stateless server cannot have any communication that depends on a previous communication.
        - Stateful: This could be problematic as versioning of the file can be corrupted by multiple users.
    - Client Crash - This will leave the session open and eventually could cause the server to run out of system resources, or worse, system hijacking by using endpoint identification from a recent reboot to get into another opened active session.
- 2.5 - Is a server that uses endpoint identification more secure than a server that uses handles? Why or why not?
    - I will say endpoint identification is more secure on the surface due to the fact that you are using the IP address and port to verify the destination of the information. On the other side the issue of a terminated connection that can be hijacked.
    - Handles allow for multiple connections, but also raises a concern as someone can see the handle when it is exposed to the application if they were savvy enough, and this would allow for a false agent posing as a process with this handle with a constant connection to

the application.

- 2.6 - If a handle only contains a table index, the same handle will be reissued whenever a given slot in the table is assigned. Devise a scheme in which a handle can be mapped to a table entry efficiently, but a new value is issued each time the table entry is reissued.
    - Increment the table handle while keeping the number of active handles under a certain parameter, destroying the prior handles that are no longer being used, therefore you can reuse them.
        - Example: 99 handles available, 20 available at a time, you can reset the handle counter at 99 to 0.
- 2.7 - Write down the data structures and message formats needed for a stateful file server. Use the operations open, read, write, and close to access files. Arrange for open to return an integer used to access the file in read and write operations. How do you distinguish duplicate open requests from a client that sends an open, crashes, reboots, and sends an open again?

```
class stateFullFileServer
{
 //Do we need a multidimensional array here or just use the class?!?
 int clientID; //used for client handle
 string fileName; //used for keeping track of what file needed
 int currPos; //position tracker
 bool lastOp; // Read = 0 Write = 1

public:
//Access files by Open, Read, Write, Close
   readFile(clientID, fileName); //will have to flesh these out later
   writeFile(clientID, fileName); //will have to flesh these out later
   closeConnection(clientID); //will have to flesh these out later
   //Open returns int for read & write
   // Open initial read from client will need all "message fields"
   int openConnection(clientID, fileName, currPos, lastOp)
   {
        //Establish the Handle
        clientID = clientID++;
        //What operation is requested
        IF (bool == 0) THEN readFile(clientID, fileName) return 0; //Read
        ELSEIF (bool == 1) THEN writeFile(clientID, fileName) return 1; //Write
        // Close transaction
        closeConnection();
   };
   //follow up requests from same user need no formal introduction, just filename and id
   int getFollowUpRequest(int client, string fileName)
   {
      //How to distiguish between dup requests
      ////Scenario: Open, crash, reboots and sends an open again?
      //find out if we have this client id around - the connection just needs current clientID
      if (stateFullFileServer.client == client) Then stateFullFileServer.client =
stateFullFileServer.client++
   }
```

- 2.8 - In the previous exercise, what happens in your design if two or more clients access the same file? What happens if a client crashes before closing a file?
    - Out of order or duplicate file requests can cause errors or confusion and delay for the recipient. Additionally the position will not be correct on the server.
- 2.9 - Examine the NFS remote file access protocol carefully to identify which operations are

idempotent. What errors can result if messages are lost, duplicated, or delayed?
    - Not sure on this one.

- 2.10 - Does an idempotent protocol necessarily have larger messages than an equivalent protocol that is not idempotent? Why or why not?
    - It would have to because it cannot rely on any assumed information. Each request from the protocol must be complete so that the server will know who is requesting the information, the position, the complete filename itself and everything about the request. There is no stored or assumed information in the idempotent protocol.