RPC Design

Serialization Protocol for Client-Server Communication

1. Opcode Design:

Every request begins with a 4-byte opcode, which uniquely identifies the requested operation (e.g., open, read, write, stat). This universal format ensures consistency and minimizes space usage compared to string-based RPCs.

Upon receiving a request, the opcode determines the structure of the rest of the message. For example:

- Open (Opcode = 0): Requires a pathname, flags, and mode.
- Read (Opcode = 1): Requires a file descriptor and byte count.
- Write (Opcode = 2): Requires a file descriptor, byte count, and data.

This approach allows efficient parsing and extensibility for future operations.

2. Length-Prefix for Strings and Continuous Memory

When transmitting variable-length data, such as strings or dynamically allocated structures, we prefix the data with its length (4 bytes). For example, in open() requests:

```
| Opcode (4) | Pathname Length (4) | Pathname (n) | Flags (4) | Mode (4) |
```

This ensures that the receiver can correctly extract the string without relying on null terminators, making the protocol robust for binary data.

3. Handling Large Read/Write Operations

Since read/write operations may exceed the maximum message size (4096 bytes), we implement chunked transmission: The client splits large reads/writes into multiple requests, ensuring that no message exceeds MAX_MSG_LEN. This guarantees smooth handling of large files without hitting message size limits.

4. Serializing the dirtree Structure

To serialize a hierarchical directory tree (dirtreenode), I use a recursive depth-first encoding:

Store node name length (4 bytes), followed by the node name, number of subdirectories (4 bytes), and then recursively serialize each subdirectory. Example serialization format:

```
| Name Length (4) | Name (n) | Num Subdirs (4) | Subdir 1 | Subdir 2 | ...
```

5. File Descriptor (FD) Offsetting

Since the client and server maintain separate FD namespaces, we offset server-generated FDs using a fixed offset (FD_OFFSET = 5000):

6. Freeing dirtree After Retrieval

After a getdirtree() request, the client receives and reconstructs the tree. Since this data is dynamically allocated, we explicitly free the structure using freedirtree(), ensuring:

7. No memory leaks on the client.

My design implements proper cleanup of recursively allocated nodes. On the server, we also free the tree after serialization to prevent unnecessary memory consumption.