For my client-side serialization, I decided to break the process into 3 parts: the rpc header, interposed function non-pointer arguments, and pointer arguments. For the rpc header, I decided to create a generic struct that would pass over the opcode used to determine which function to call on the server side as well as the size of the rest of the packet. Sending the size of the remaining packet helped with server-side deserialization. For the non-pointer arguments, I made unique structs for each interposed function to send a specific set of values to the server. Originally, I was passing in the pointer arguments within these structs as well, but quickly realized issues with that implementation and decided to separate the pointers. To do so, I used memcpy to implement pass-by-value for all pointer arguments, such as the pathname, instead of pass-by-reference.

Since I sent the remaining size as part of the rpc_header struct, I would be able to first receive the rpc header on the server-side. Once storing the rpc header, I used a while loop to continuously receive until the server received all the expected bytes. Using a switch-case statement, I could separate the individual functions and deserialize the remainder of the packet uniquely for each function. This method of implementation worked fine since I was able to maintain consistency between what was being sent and received between the client and server. By keeping the order consistent on the client side, deserializing the information received on the server side was much easier and could follow a similar format for each case.

For server-side serialization, I noticed that most calls only required 2 pieces of information: some return value and the errno value. Since this was the case, I decided against using structs to send information back to the client. Instead, I created the return message independently for each case, usually with the return value first and then the errno value. For the few cases where more would need to be returned, I decided to just append extra information to the end of the return message.

Then, for client-side deserialization, I was able to completely customize the implementation for each interposed function because I can determine the ordering, size, and information that the server sends back to the client. Since most functions had a similar format, I was able to use similar deserialization techniques to retrieve the return and errno values. When there was extra information, usually a pointer being passed with pass-by-value, I would use memcpy to safely copy the contents over on the client-side.

Lastly, for my serialization and deserialization of the struct dirtreenode, I decided to implement a recursive algorithm similar to those shown in Recitation 2. Starting at the root node, I would store the name length, name, number of subdirectories for the current node, and then recursively call the serialization/deserialization function on all of the subdirectories using a for-loop. Since the serialized message would have variable length, I used realloc() to increase the size of the message in every recursive iteration. The recursion would eventually backtrack once a node had 0 subdirectories, preventing an infinite recursive loop from occurring. In terms of the deserialization function, I mirrored the preorder traversal implementation and created structs in each recursive step for each node needed. During deserialization, I kept track of a total offset to extract the correct information from the serialized message.