The project aims to abstract network complexities and provide a seamless interface for remote resource access. Utilize Remote Procedure Calls (RPCs) to mimic local services via a network, allowing unmodified programs to interact with remote files as if they were local.

For each function, three pieces of information are sent to the server: the opcode of the function(represented by an int in the range [0,8], each value corresponding to a function call), how long the entire buffer is, and the parameters of the function. The information is stored in a char array in the order of opcode, size, and parameters using memcpy. Then the buffer is sent to the server.
On the server side, the server will first receive 8 bytes of the buffer including the opcode and size, identifying the opcode allows the receiving process to be deterministic, and knowing the exact size will allow the exact number of bytes in the message to be received, and it will not interfere with other packets when handling concurrent clients. The opcode is obtained by casting and dereferencing the first four bytes of the buffer as an integer, and the size is obtained similarly by casting and dereferencing the second four bytes of the buffer as an integer.

Then a while loop is used to receive the rest of the buffer, containing the parameters to the function call. The recv function in the while loop expects MAXMSGLEN every time until the remaining buffer length is less than MAXMSGLEN, then the remainder length is expected in recv. Because the order is preserved when it's transferred through TCP, when the buffer is received on the server side, the data can be deserialized by dereferencing specific parts of the buffer.

Once all the data are deserialized, a function is made on the server side, and the server will serialize the result data in a similar protocol. The server sends three pieces of data back to the client: the size of the entire buffer, errno after the function returns, and the result of the function call.

When the client receives the message, it will first receive four bytes indicating the size of the upcoming buffer, and a while loop is then used to accept the buffer of the exact size in the same procedure as the server side. The server will get the errno by casting and dereferencing the next four bytes of data and update if necessary. Then the client will get the results of the function call by dereferencing the corresponding size of bytes into the appropriate data type of the result variable (such as int for open, ssize_t for write, etc)

For dirtrees specifically, the serialization is done using a preorder tree traversal recursively. Instead of going current->left->right, because one node can have multiple children, the traversal goes like current->subdir1->subdir2->...->subdirn. For each node, three pieces of data are stored, including the name, number of subdirectories, and an array of the subdirectories. Then when deserializing, the preorder traversal is undone, so while the current node still has an unvisited subdirectory (since we know the number of subdirectories each node expects to have), we will traverse down the buffer and update the current node using a recursive approach as well.

To distinguish between local file descriptors and file descriptors created on the server side, a constant offset of 20000 is used for all file descriptors returned by the server. So before sending the procedure to the server, the file descriptor is checked by the client stub library, and local file descriptors will be performed locally using original function calls.

Concurrency is handled by forking a new child every time a new connection is created with a client on the server and terminating the child process when recv indicates the client has closed the connection with the server. This will have a single server process that listens for new clients and then launches additional processes to handle each client separately.