Our project implements a distributed file-caching proxy system that sits between the client and the remote file server. The proxy communicates with the server via a simple Java RMI protocol, using methods like getFileMetadata, fetchFileChunk, pushFileChunk, and deleteFile. Essentially, whenever a client wants to open a file, the proxy checks the server for the latest metadata (like file size and last-modified time) to ensure it's working with the most up-to-date version. To guarantee that each client sees a consistent snapshot, we create session-specific copies when files are opened for reading. This way, even if another client writes to the file later, the first client continues working on its own copy.

To manage the limited cache space on the proxy, we implemented an LRU (eviction policy within our CacheManager class. This module keeps track of every cached file's last access time using a LinkedHashMap, and also monitors which files are actively in use so they aren't evicted. When the total cache size exceeds our set limit, the system automatically evicts the least recently used files that aren't being accessed by any client. On the transfer side, we fetch and push files in fixed-size chunks (50KB) to help balance memory usage and network performance, even though in high latency environments this might mean more remote calls.

Other design decisions include using Java RMI to simplify our distributed communication, and employing Java's ReentrantReadWriteLocks to allow concurrent read access while serializing writes. We also standardized error codes to mimic traditional C file operations, which makes our proxy's behavior more predictable. Overall, our design strikes a balance between ensuring fresh and consistent data for clients and optimizing performance under limited cache resources.