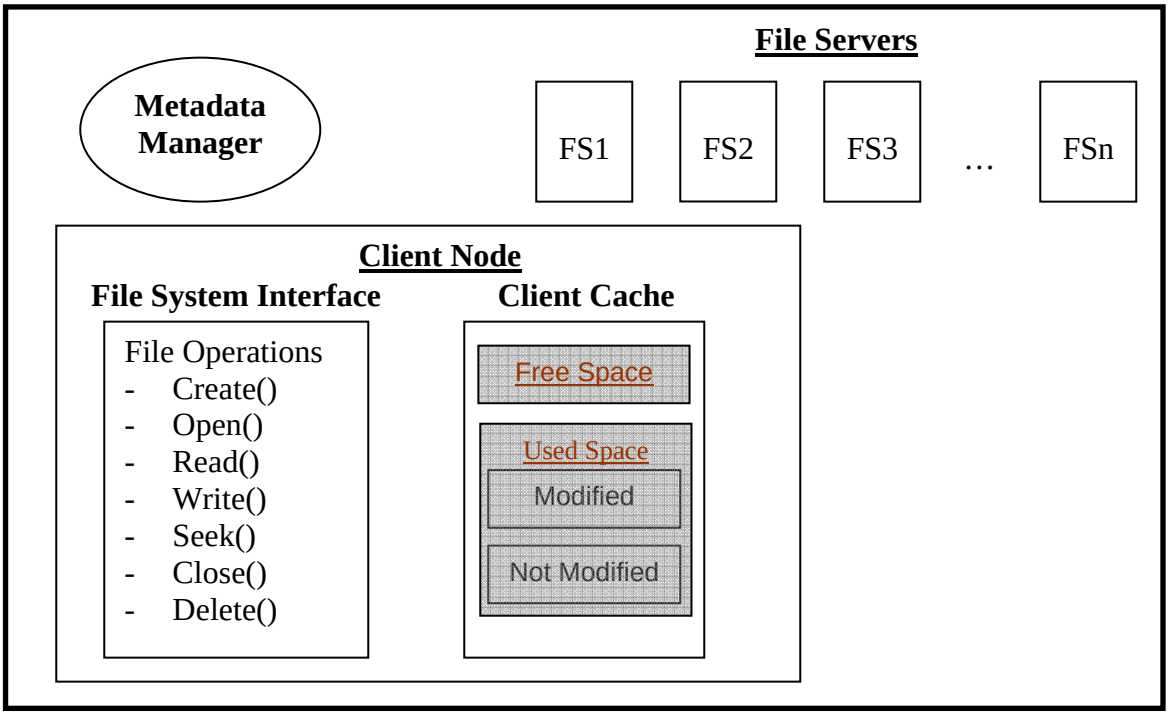
In this project, your will implement a simple Parallel File System (PFS) which provides access to file data for parallel applications. The following figure gives an overview of a simple PFS and its components.



Like most other files systems, the PFS is also designed as client-server architecture with multiple servers which typically run on separate nodes and have disks attached to them. Each file in the PFS system is striped across the disks on the file servers and the application processes interact with the file system using a client library. The PFS also has a Metadata Manager which handles all the metadata associated with files. The Metadata Manager does not take part in the actual read/write operations. The client, file servers and the metadata manager need not run on different machines though running them on different machines generally results in higher performance.

As part of your project, you will be implementing all the four primary components of a simple PFS:

1. Metadata Manager
2. File Servers
3. Client
4. Client Cache

A brief description of all the components is given below.

Metadata Manager

The metadata manager is responsible for the storage and maintaining all the metadata associated with a file. For this project, we will assume a flat file system consisting of only one root directory. For all the files in the directory, you need to maintain the following metadata:

1. Filename
2. File size

3. Time of creation
4. Time of last modification
5. File recipe

Note that a given file in the PFS is striped across various file servers to facilitate parallel access. The specifics of this file distribution are described by what is known as the *file recipe*. You are free to abstract out the components of a file recipe as part of your system design. Note that the *block size* and the *stripe size* will be a fixed value which will be provided to you as part of a configuration file. The *stripe width* which defines how many nodes the file will be split across, will be specified by the user when the file is created. When a client opens a file, the metadata manager returns to the client the file recipe. Thus the client directly communicates to the file servers to access file data.

Token Management Function

The metadata manager also performs the task of a token manager for ensuring consistency in the system. The goal here is to guarantee single-node equivalent POSIX/UNIX consistency for file system operations across the system. Every file system operation acquires an appropriate read or write lock to synchronize with conflicting operations on other nodes before reading or updating any file system data. For this project, you need to implement block-range locking in order to allow parallel applications to write concurrently to different blocks of the same file. The token manager coordinates locks by handing out *lock tokens*. There are two types of lock tokens that are issued by the token manager: *read tokens* and *write tokens*. Multiple nodes can simultaneously acquire read tokens for the same file-block. However, block-range tokens for parallel writes are negotiated as follows. The first node to read/write a file acquires the block-range token for the whole file (zero to infinity). As long as no other nodes require access to the same file, all read and write operations are processed locally. When a second node begins writing to the same file it will need to revoke at least part of the block-range token held by the first node. When the first node receives a revoke request, it checks whether the file is still in use. If the file has since been closed, the first node will give up the whole token, and the second node will then be able to acquire a token covering the whole file. On the other hand, if the second node starts writing to a file before the first node closes the file, the first node will relinquish only part of its block-range token. If the first node is writing sequentially at offset o_1 and the second node at offset o_2, the first node will relinquish its token from o_2 to infinity (if $o_2 > o_1$) or from zero to o_1 (if $o_2 < o_1$).

File Servers

There are a fixed number of file servers in the network which will be also be specified in the configuration file provided to you. Every file in the PFS is striped across a number of these file servers. You can simulate file striping by implementing each stripe using the native file system in the file servers.

PFS Client

The following APIs will be supported by the PFS Client

```
int pfs_create(const char *filename, int stripe_width);
int pfs_open(const char *filename, const char *mode);
ssize_t pfs_read(int filedes, void *buf, ssize_t nbyte,
                 off_t offset, int *cache_hit);
```

```
ssize_t pfs_write(int filedes, const void *buf, size_t nbyte,
                  off_t offset, int *cache_hit);
int pfs_close(int filedes);
int pfs_delete(const char *filename);
int pfs_fstat(int filedes, struct pfs_stat *buf);
// Check the config file for the definition of pfs_stat structure
```

Client Cache

The PFS uses local cache on each client having a default size of 2 MB. The client cache consists of (1) free space and (2) used space. The used space in the client cache can further be divided into blocks that have been modified (i.e. dirty blocks which have not been flushed to the server yet) or not. As part of the cache management scheme, you need to implement the following two threads:

1. Harvesters Thread: The job of the harvester thread is to maintain sufficient amount of free space in the client cache. Whenever the free space falls below a specific lower threshold, the harvester thread frees up memory from the used space based on an LRU policy till the amount of free space reaches an upper threshold.
2. Flusher Thread: The job of the flusher thread is to periodically (every 30 seconds) flush all the dirty blocks (modified blocks) back to the appropriate file servers.

Note that you will have to implement a hashing scheme to map and look up the file blocks onto the cache.

If the client has the appropriate tokens, and a read/write call issued by the client results in a cache hit, the operations should be performed on locally on the client cache. On the other hand, whenever there is a cache miss, the blocks need to be fetched from the file server which will then be cached locally on the client using the available free space. The client returns the information about a cache hit or miss for every read/write call using the Boolean argument `cache_hit` which is passed as a parameter to both the `pfs_read()` and `pfs_write()` function calls. If the client needs to acquire fresh tokens from the token manager for a read/write call, the file data need to be fetched from the file servers. Whenever a client is required to relinquish a part of its token, it needs to flush the dirty blocks to the file servers before relinquishing the tokens to the token manager.

Implementation Notes

You can use pthreads library and socket APIs. Note that all the communication between various nodes in the system needs to be implemented using sockets. Also, note that you do not need to worry about issues related to fault-tolerance and reliability.