Assignment 3

Network file system

Packet structure

Packet structure

Field	Type	Size	Description
`type`	`MessageT`	4 bytes	Type of message
`id`	`uint32_t`	4 bytes	Unique ID a request
`is_response`	`uint8_t`	4 bytes	Request/Response (optional)
`*_req`	`union ...`	...	Specific struct for each type of request
`payloadLength`	`unsigned int`	4 bytes	Length of the following buffer
`payload`	`void *`	8 bytes	Payload buffer

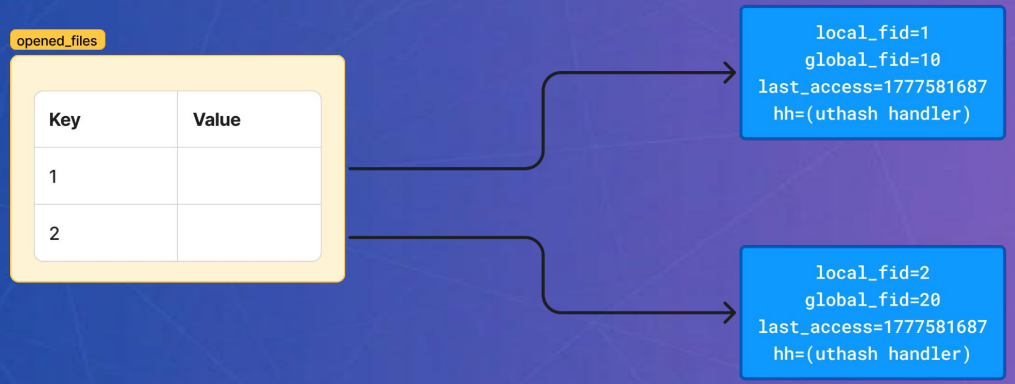
Packet types

PacketType enum

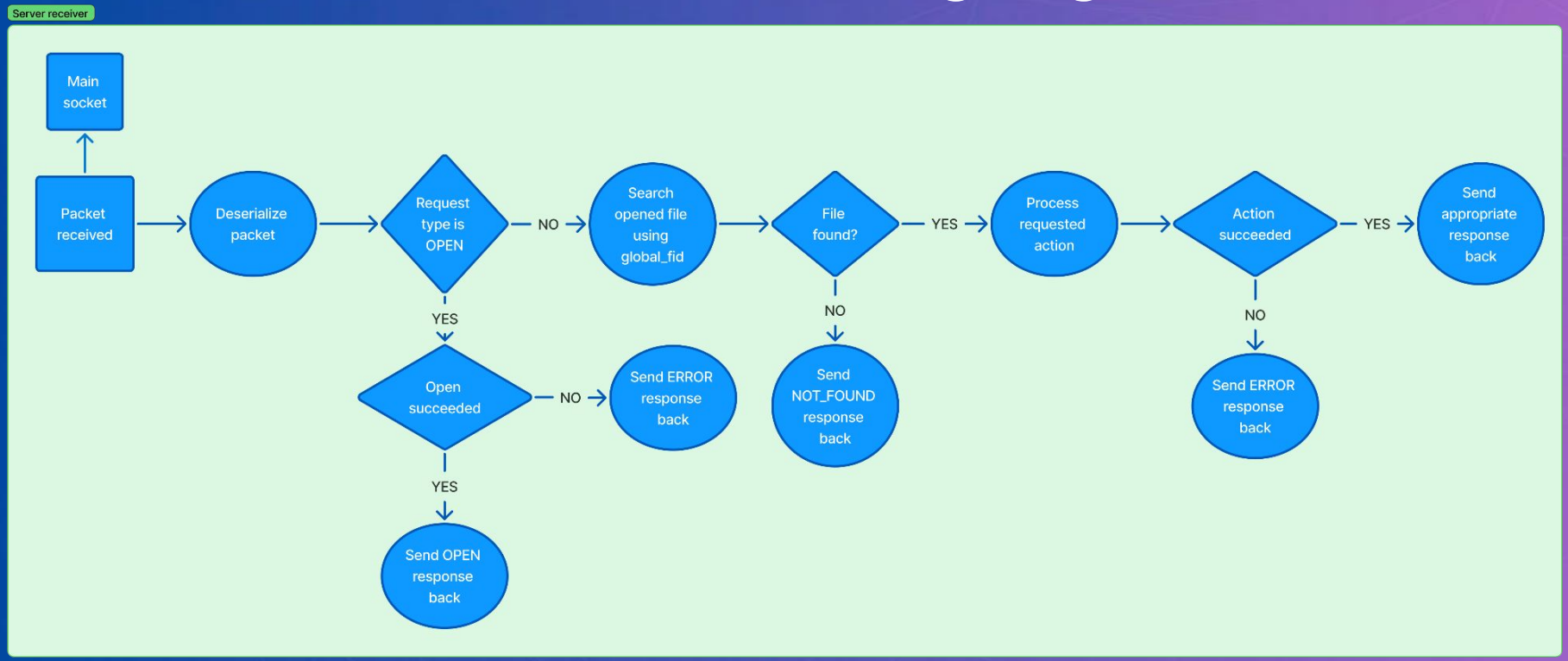
Value	Description
OPEN	Used by client to open a file on a server
READ	Used by client to read from a file on a server
WRITE	Used by client to write to a file on a server
TRUNCATE	Used by client to truncate a file on a server
NOT_FOUND	Used by server when a specific file (opened or not) was not found.
ERROR	Used by server to signify an error on an action

Server structure

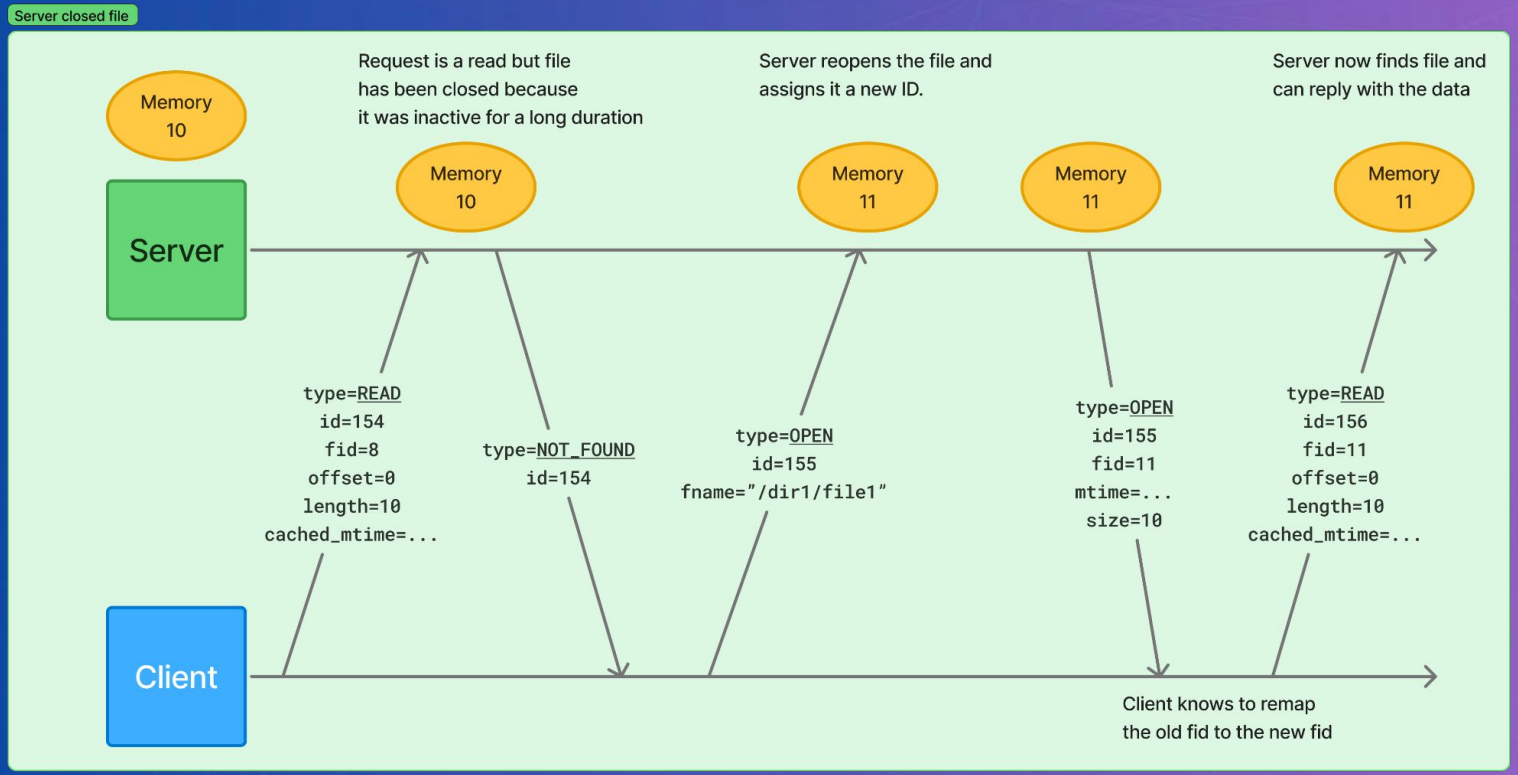
- Can perform the following actions on a file: Opening, reading, writing, truncating
- No state stored for clients
- Temporary state for opened files that stores a mapping from the file descriptor given to the clients and the file descriptor given to the server by the OS. Inactive files are removed from this map after some time
- Has a persistent memory file that stores the last file descriptor given to a file in case a restart occurs.



Server receiving logic



Server closed file/restarted

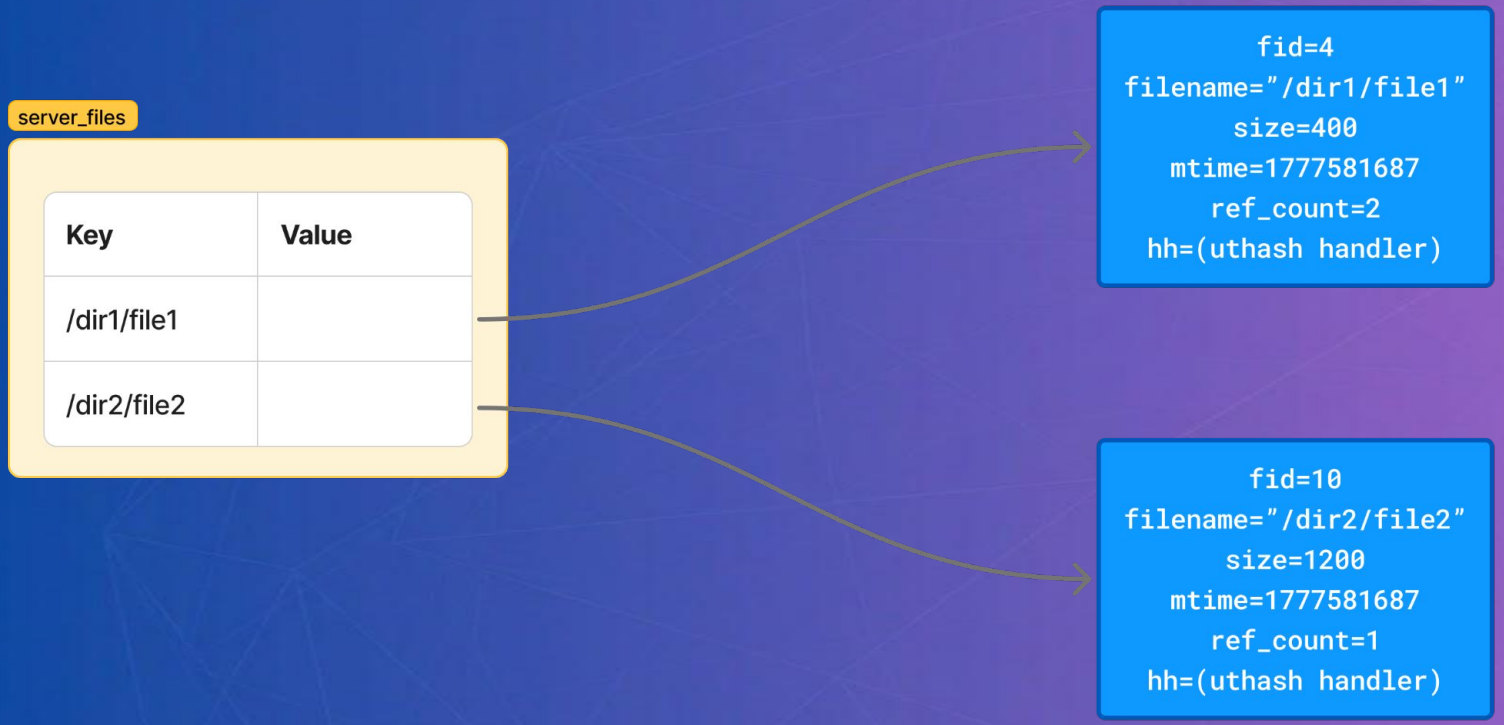


Client data structures

ServerFileT

Field	Type	Description
`fid`	`unsigned int`	File descriptor given by the server
`filename`	`char *`	The absolute path to the file
`size`	`uint64_t`	Size of the file
`mtime`	`uint64_t`	Modification timestamp of the file
`ref_count`	`unsigned int`	Counts how many time this file has been opened locally
`hh`	`UT_hash_handle`	Handler for uthash library

Client data structures

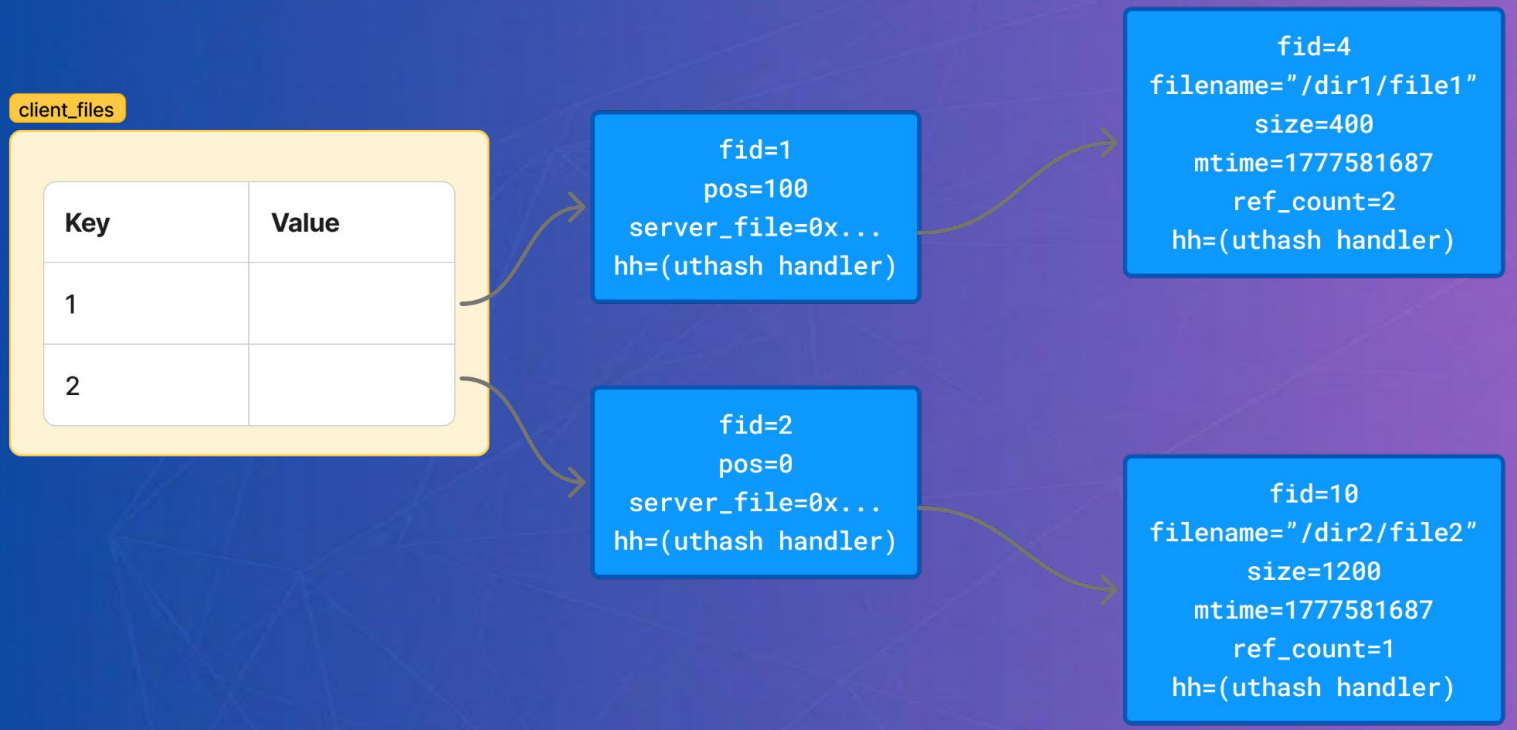


Client data structures

ClientFileT

Field	Type	Description
`fid`	`unsigned int`	File descriptor given to the user
`pos`	`long`	Position of the file offset
`server_file`	`ServerFile *`	Pointer to the server file this file is referring to
`hh`	`UT_hash_handle`	Handler for uthash library

Client data structures

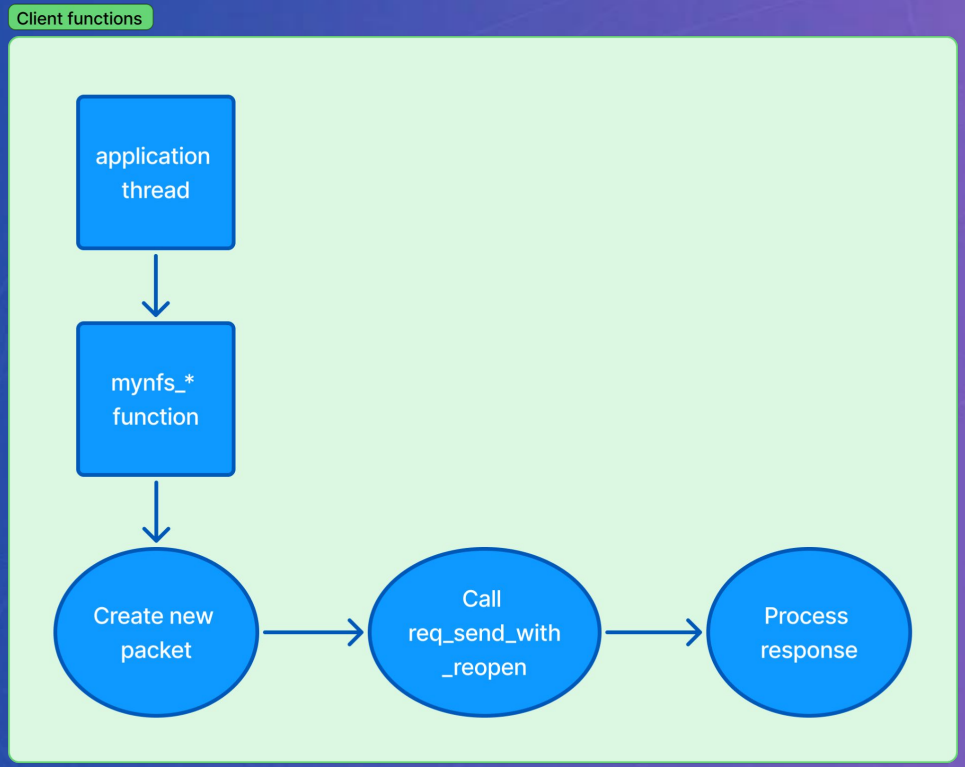


Client data structures

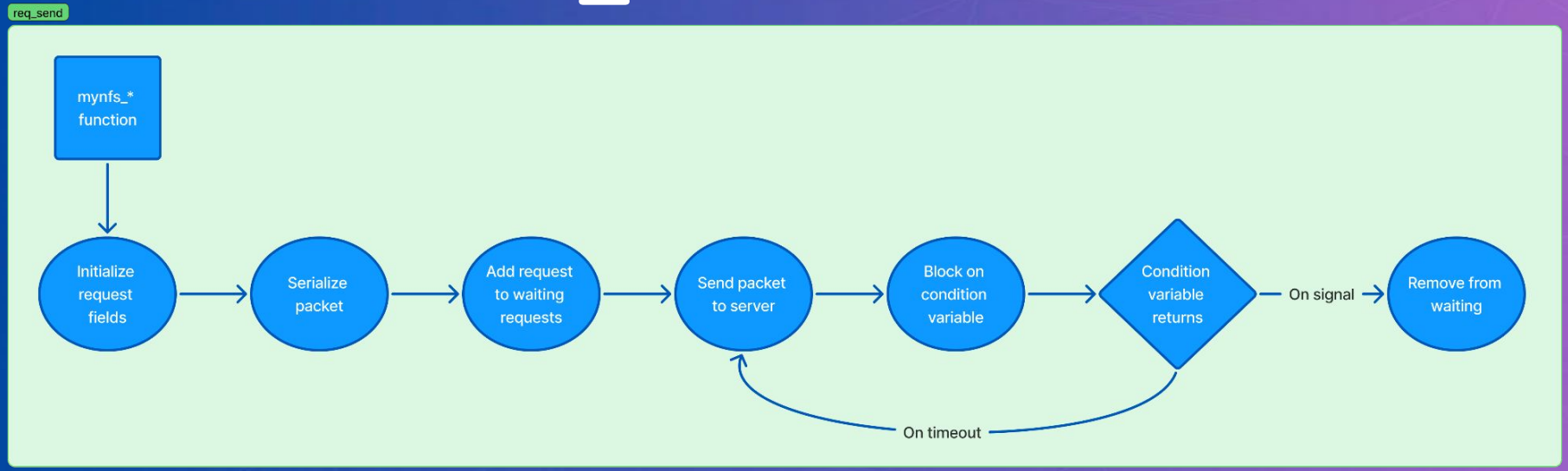
RequestT

Field	Type	Description
`id`	`uint32_t`	ID of the request
`packet`	`Packet *`	Request packet send to the server
`resp`	`Packet *`	Set by the receiver when response is received
`cond`	`pthread_cond_t`	Condition variable for sender to wait for the response
`hh`	`UT_hash_handle`	Handler for uthash library

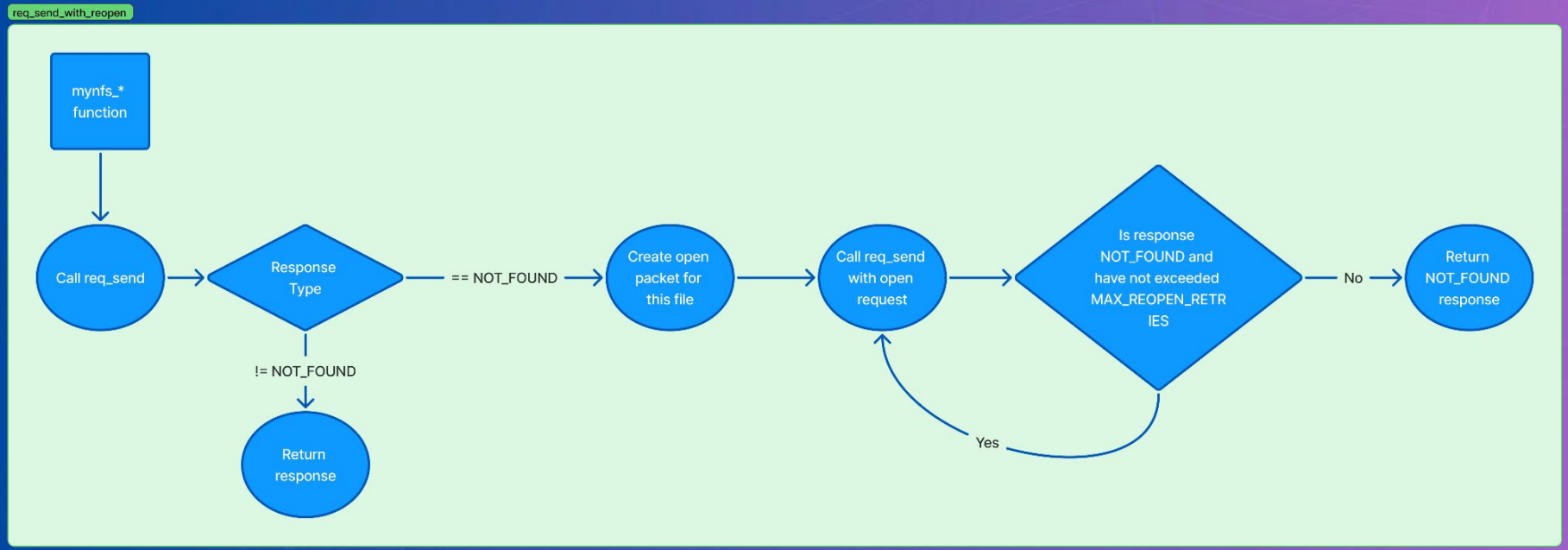
Client functions



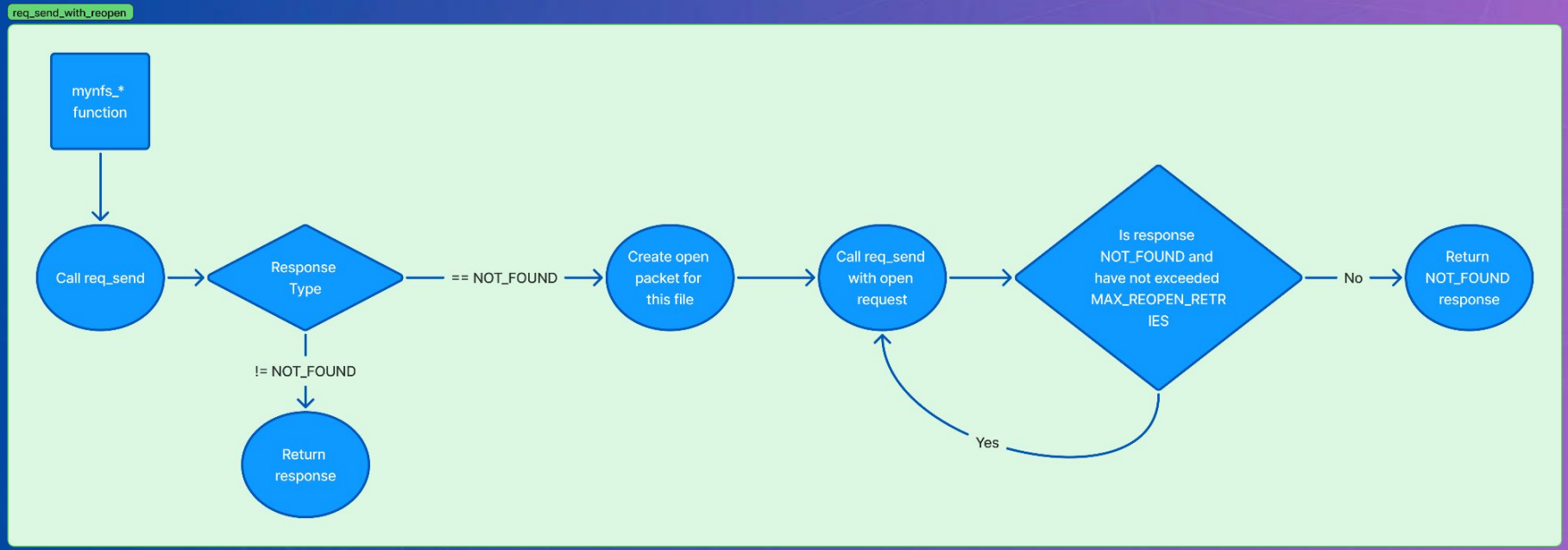
`req_send` function



`req_send_with_reopen` function



`req_send_with_reopen` function



Cache data structures

- Uses a hash map that maps the a struct key (file descriptor and block index) to a cache block struct that contains the data of the block and more.

CacheBlockT

Field	Type	Description
`key`	`CacheKey`	File descriptor + block index
`mtime`	`uint64_t`	Modification timestamp when this block was cached
`fetch_time`	`uint64_t`	When the block was fetched
`data_len`	`unsigned long`	The length of the data field
`data`	`void *`	The data of the block
`lru_prev`	`CacheBlockT *`	The previous block in the LRU list
`lru_next`	`CacheBlockT *`	The next in the LRU list
`hh`	`UT_hash_handle`	Handler for the uthash library

Cache API

Client side uses another API that implements the functions needed to use the cache. Each function automatically manages the eviction policy and keeps the LRU list up-to-date.

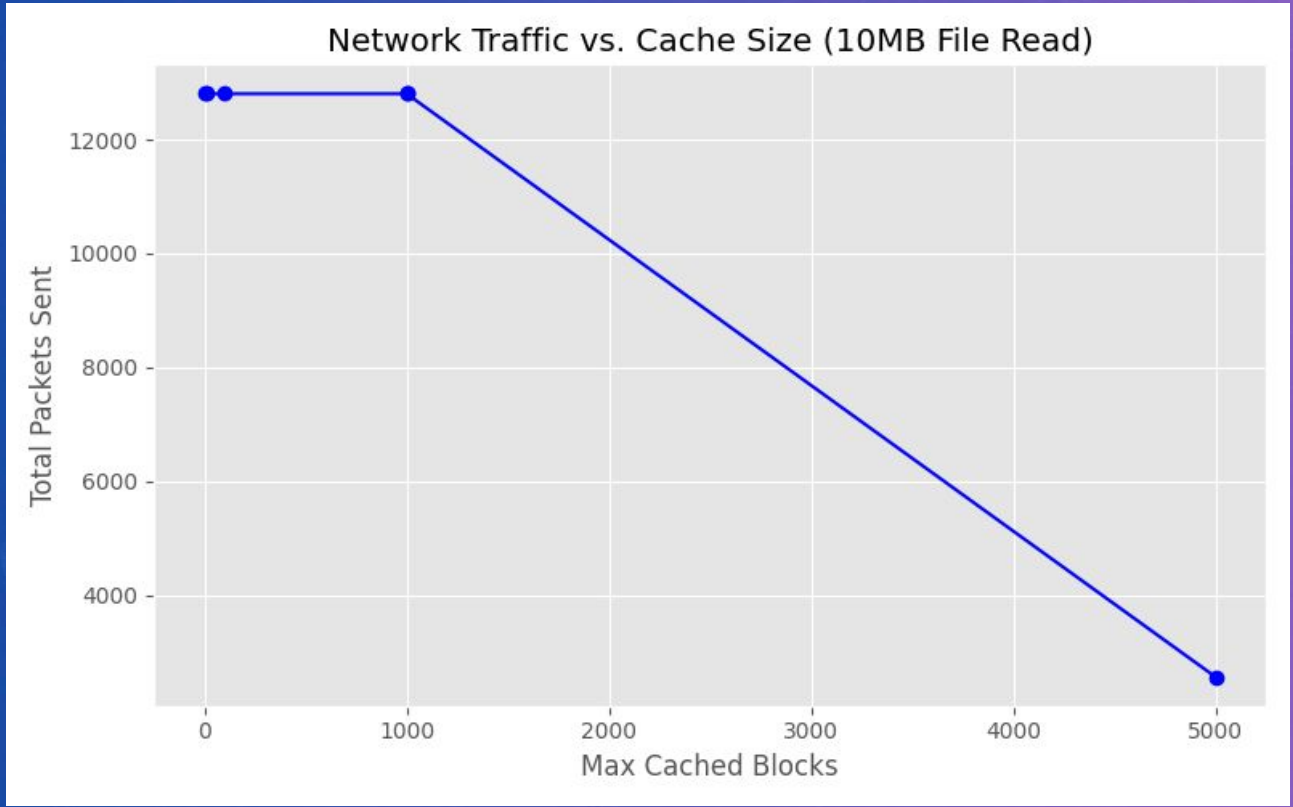
```
void cache_init(int max_blocks, int block_sz, int fresh_t);
CacheBlockT *cache_get(unsigned int server_fid, unsigned int block_index);
void cache_put(unsigned int server_fid, unsigned int block_index,
               unsigned char *data, unsigned int data_len, uint64_t mtime, uint64_t mtimensec);
int cache_is_fresh(CacheBlockT *block);

void cache_remap(unsigned int old_fid, unsigned int new_fid);
void cache_revalidate(CacheBlockT *block);
void cache_invalidate_past_offset(unsigned int server_fid, long new_length);
void cache_destroy();
```

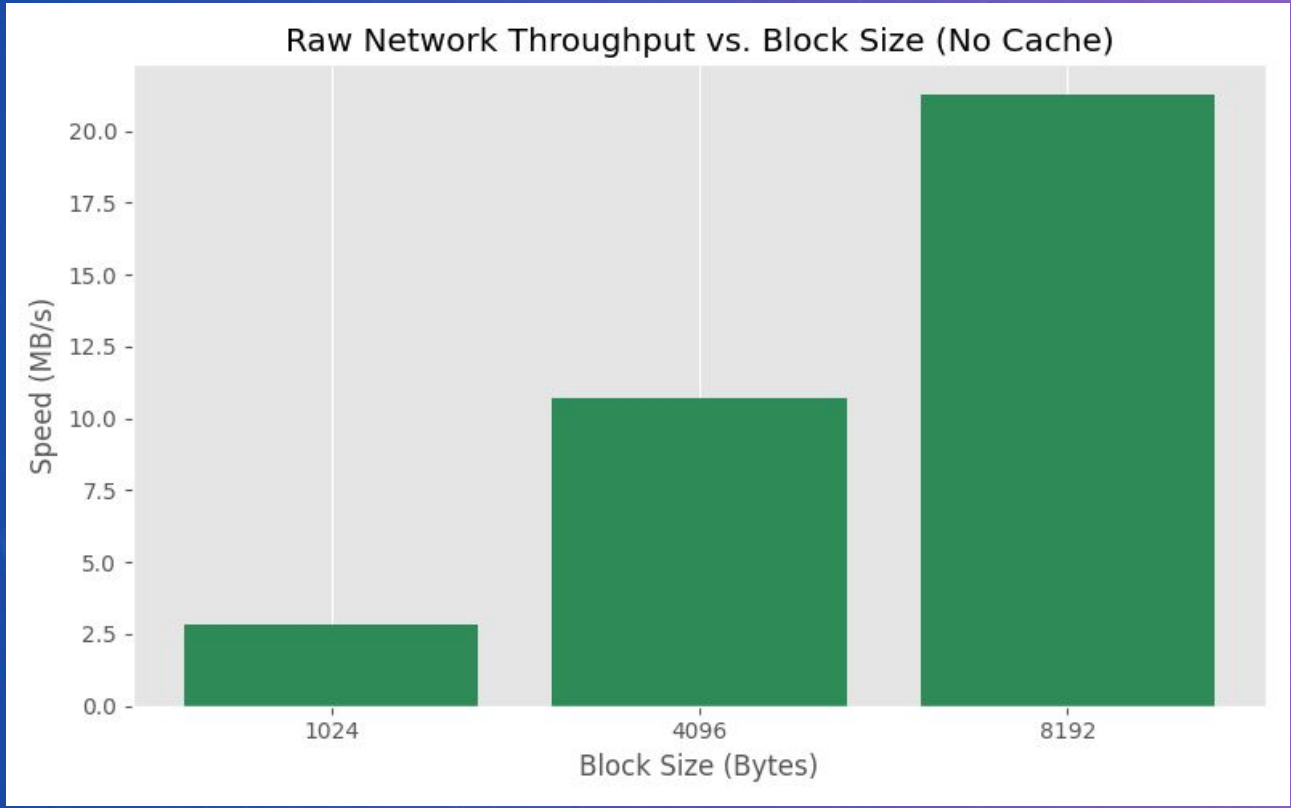
Client side

- The reading logic of the client needs to change when using a cache. We first want to check the data we need to read are in a cached block that's fresh. If not we need to communicate with the server.
- If the file hasn't been modified since the last update of the block the server responds with an empty read request with the same modified timestamp and the client refreshes the block
- On write, if we write a whole block the cache is updated locally on the client and a write request is sent to the server.

Charts: Traffic vs. Cache size



Charts: Network throughput vs. block size



Charts: Baseline vs. cache throughput

