Code Notes:

Seafile Components: http://manual.seafile.com/develop/server-components.html
Seafile Data Model: http://manual.seafile.com/develop/data_model.html
Seafile Sync Algorithm: http://manual.seafile.com/develop/sync_algorithm.html

COMPONENTS OF SEAFILE:

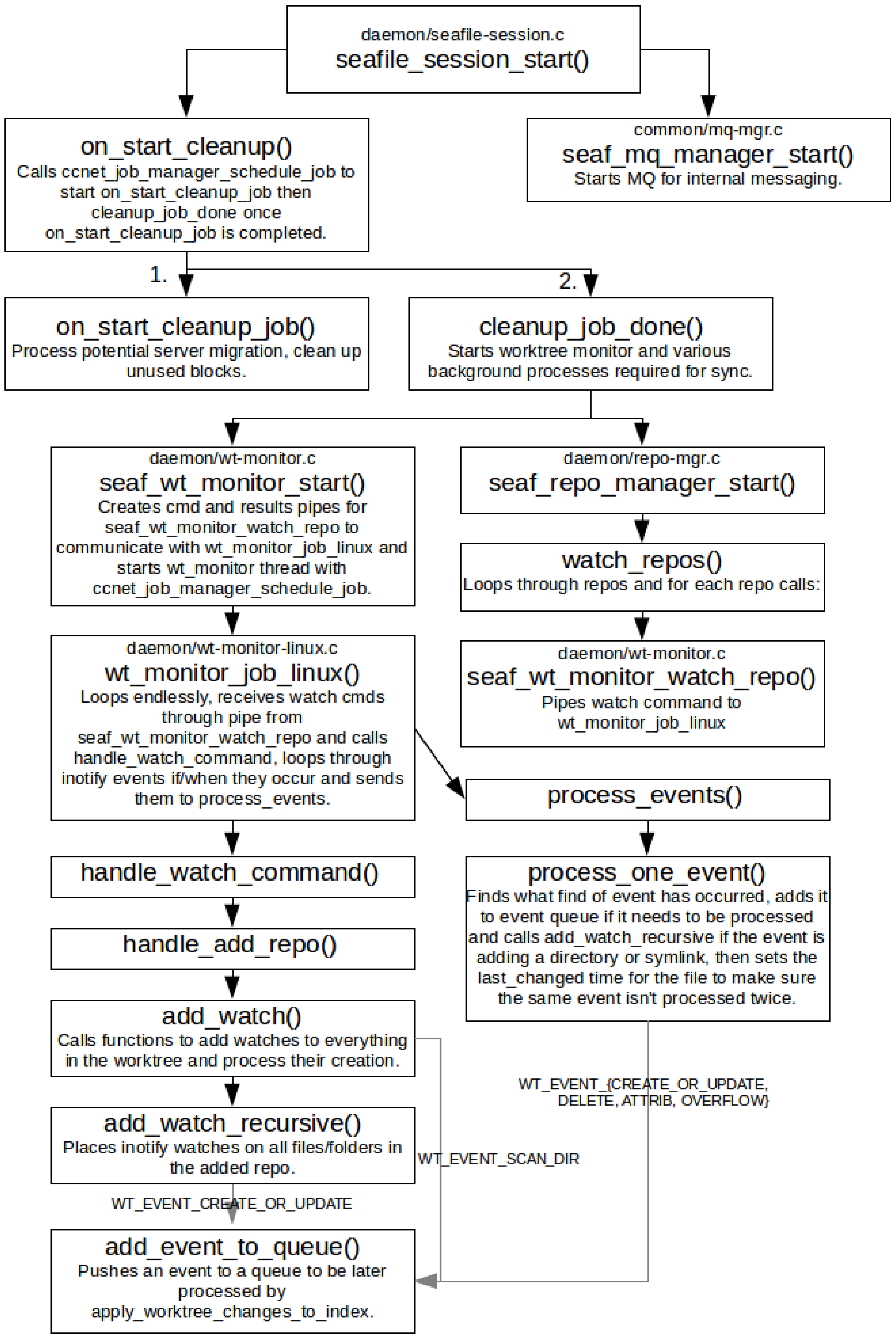
- Seafile Daemon is the daemon that runs on the client machine and processes added/removed/modified files and sends/receives the main file data (but not commit data and file metadata).
- Ccnet runs on both the client and server and handles internal messaging for the processes and threads, and transfers the commit data and file metadata from client to server and vice versa.
- Seaf-cli is the command line interface used on the client, it issues commands to Seafile Daemon and Ccnet.
- Seafile Server runs on the server, it is the application which does most of the tasks on the server.
- Seahub runs on the server and is the web frontend for seafile, it is needed to access it both from the website it generates and through the command line on the client.

We use Nginx and SQLite as a lightweight server and database respectively.

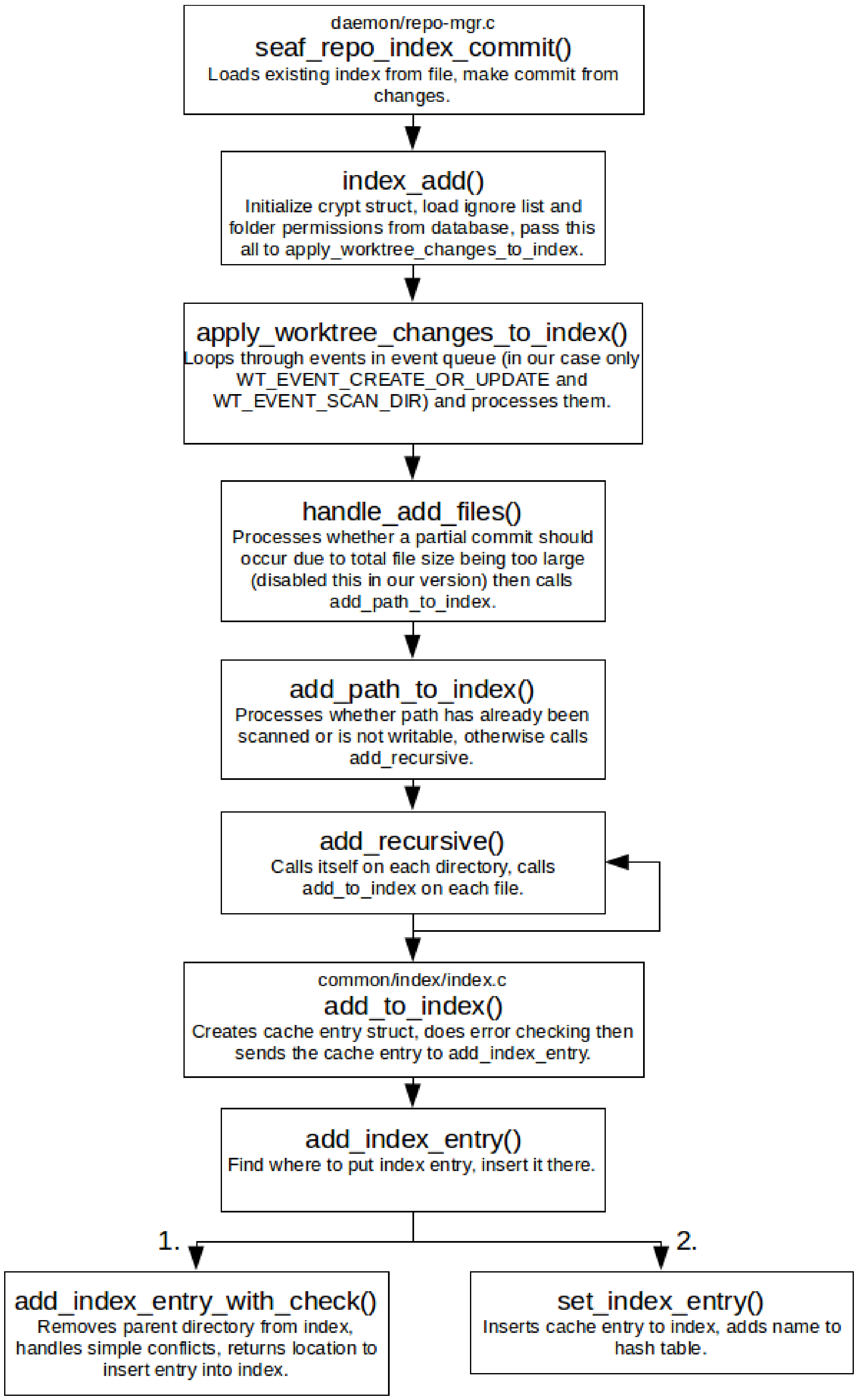
OVERVIEW OF PROGRAM FLOW FOR FILE SYNC (timed part of our sync tests):

1. Startup process & placing inotify watches

Section relevant to us:


daemon/seafile-session.c
seafile_session_start()
on_start_cleanup()
Calls ccnet_job_manager_schedule_job to start on_start_cleanup_job then cleanup_job_done once on_start_cleanup_job is completed.
common/mq-mgr.c
seaf_mq_manager_start()
Starts MQ for internal messaging.
1.
2.
on_start_cleanup_job()
Process potential server migration, clean up unused blocks.
cleanup_job_done()
Starts worktree monitor and various background processes required for sync.
daemon/wt-monitor.c
seaf_wt_monitor_start()
Creates cmd and results pipes for seaf_wt_monitor_watch_repo to communicate with wt_monitor_job_linux and starts wt_monitor thread with ccnet_job_manager_schedule_job.
daemon/repo-mgr.c
seaf_repo_manager_start()
watch_repos()
Loops through repos and for each repo calls:
daemon/wt-monitor-linux.c
wt_monitor_job_linux()
Loops endlessly, receives watch cmds through pipe from seaf_wt_monitor_watch_repo and calls handle_watch_command, loops through inotify events if/when they occur and sends them to process_events.
daemon/wt-monitor.c
seaf_wt_monitor_watch_repo()
Pipes watch command to wt_monitor_job_linux
process_events()
handle_watch_command()
process_one_event()
Finds what find of event has occurred, adds it to event queue if it needs to be processed and calls add_watch_recursive if the event is adding a directory or symlink, then sets the last_changed time for the file to make sure the same event isn't processed twice.
handle_add_repo()
add_watch()
Calls functions to add watches to everything in the worktree and process their creation.
WT_EVENT_{CREATE_OR_UPDATE, DELETE, ATTRIB, OVERFLOW}
add_watch_recursive()
Places inotify watches on all files/folders in the added repo.
WT_EVENT_SCAN_DIR
WT_EVENT_CREATE_OR_UPDATE
add_event_to_queue()
Pushes an event to a queue to be later processed by apply_worktree_changes_to_index.

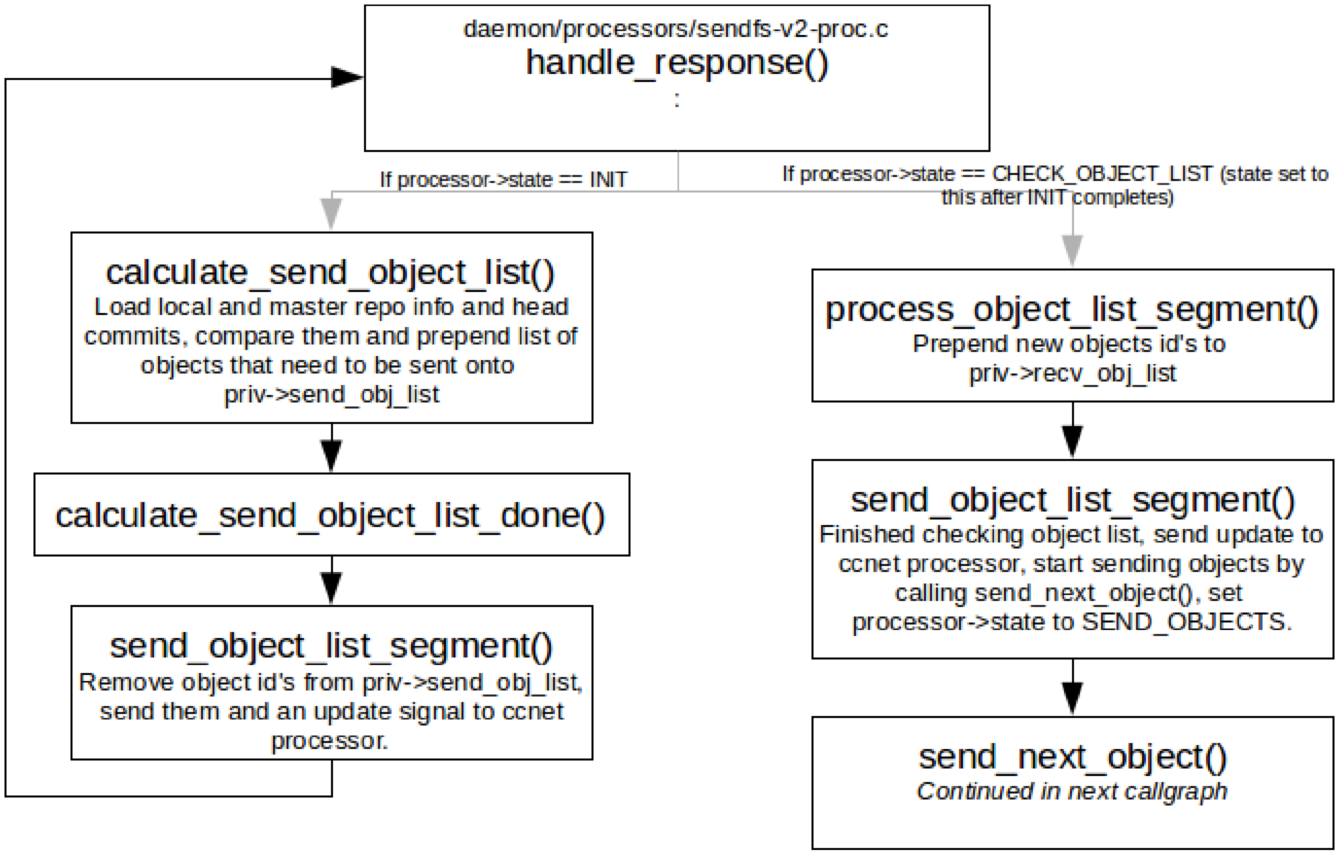
2. Update index with changes:


daemon/repo-mgr.c
seaf_repo_index_commit()
Loads existing index from file, make commit from changes.
index_add()
Initialize crypt struct, load ignore list and folder permissions from database, pass this all to apply_worktree_changes_to_index.
apply_worktree_changes_to_index()
Loops through events in event queue (in our case only WT_EVENT_CREATE_OR_UPDATE and WT_EVENT_SCAN_DIR) and processes them.
handle_add_files()
Processes whether a partial commit should occur due to total file size being too large (disabled this in our version) then calls add_path_to_index.
add_path_to_index()
Processes whether path has already been scanned or is not writable, otherwise calls add_recursive.
add_recursive()
Calls itself on each directory, calls add_to_index on each file.
common/index/index.c
add_to_index()
Creates cache entry struct, does error checking then sends the cache entry to add_index_entry.
add_index_entry()
Find where to put index entry, insert it there.
1.
2.
add_index_entry_with_check()
Removes parent directory from index, handles simple conflicts, returns location to insert entry into index.
set_index_entry()
Inserts cache entry to index, adds name to hash table.

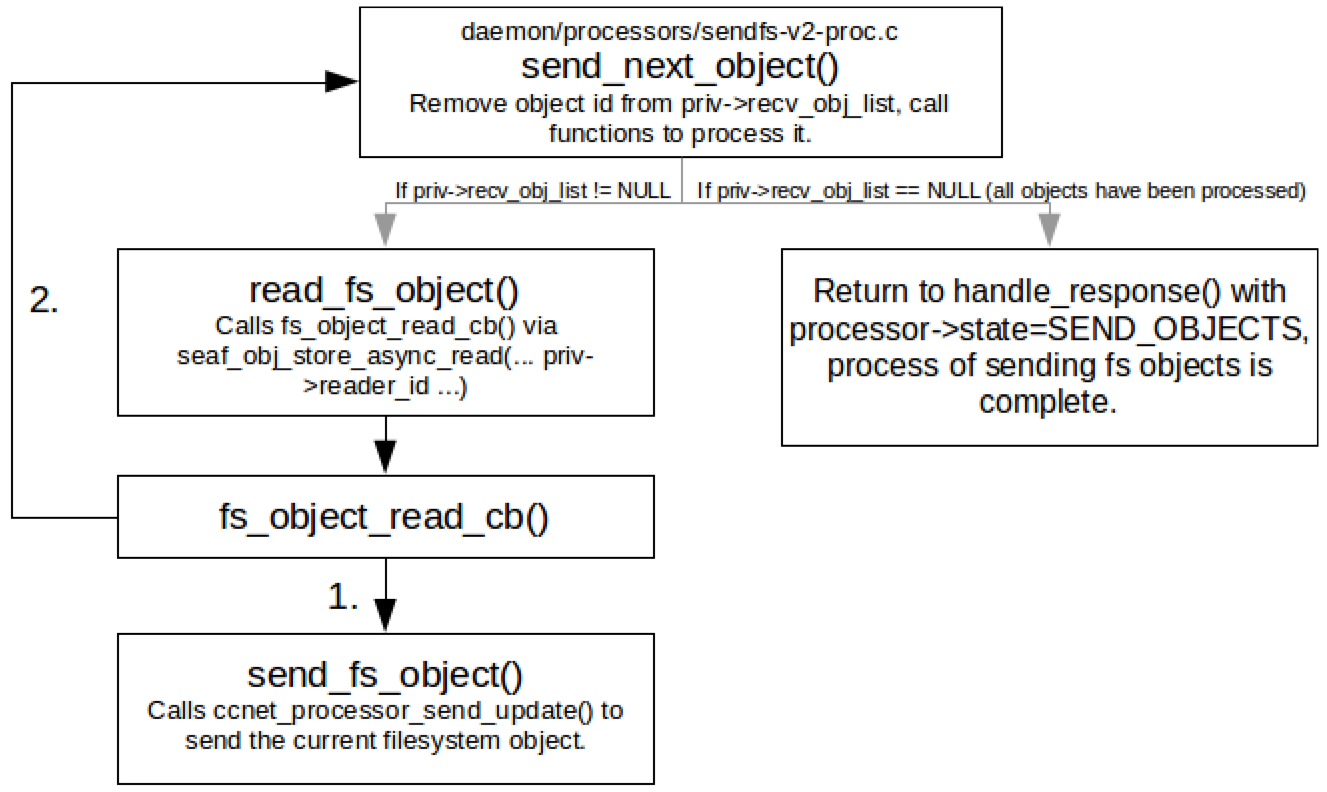
3. Send commit to server.

4. Calculate objects needed by client from server. (none in our use case as the repository on the server is empty)

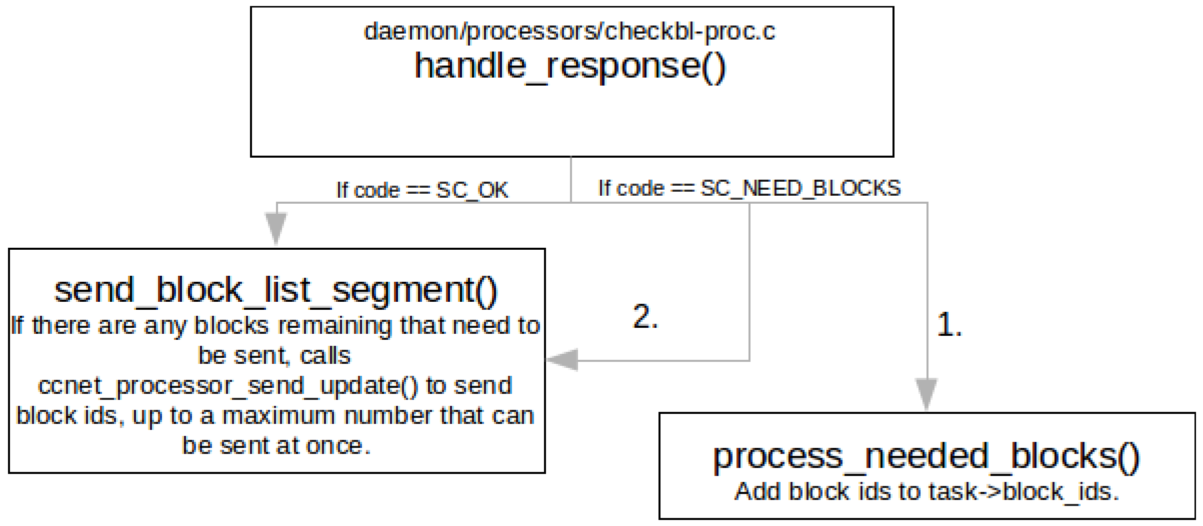
5. Calculate/send objects needed by server from client. (object id's of the new files we added).


daemon/processors/sendfs-v2-proc.c
handle_response()
:
If processor->state == INIT
If processor->state == CHECK_OBJECT_LIST (state set to this after INIT completes)
calculate_send_object_list()
Load local and master repo info and head commits, compare them and prepend list of objects that need to be sent onto priv->send_obj_list
calculate_send_object_list_done()
send_object_list_segment()
Remove object id's from priv->send_obj_list, send them and an update signal to ccnet processor.
process_object_list_segment()
Prepend new objects id's to priv->recv_obj_list
send_object_list_segment()
Finished checking object list, send update to ccnet processor, start sending objects by calling send_next_object(), set processor->state to SEND_OBJECTS.
send_next_object()
Continued in next callgraph

6. Calculate/send fs objects needed by server from client. (file system objects (metadata) of the new files we added).


daemon/processors/sendfs-v2-proc.c
send_next_object()
Remove object id from priv->recv_obj_list, call functions to process it.
If priv->recv_obj_list != NULL
If priv->recv_obj_list == NULL (all objects have been processed)
read_fs_object()
Calls fs_object_read_cb() via seaf_obj_store_async_read(... priv->reader_id ...)
Return to handle_response() with processor->state=SEND_OBJECTS, process of sending fs objects is complete.
2.
fs_object_read_cb()
1.
send_fs_object()
Calls ccnet_processor_send_update() to send the current filesystem object.

7. Calculate/send block ids needed by server from client. (ids of data blocks of the new files we added).


daemon/processors/checkbl-proc.c
handle_response()
If code == SC_OK
If code == SC_NEED_BLOCKS
send_block_list_segment()
If there are any blocks remaining that need to be sent, calls ccnet_processor_send_update() to send block ids, up to a maximum number that can be sent at once.
2.
1.
process_needed_blocks()
Add block ids to task->block_ids.

8. Calculate/send blocks needed by server from client. (data blocks of the new files we added).

The main loop for this is in client_thread_loop() in daemon/block-tx-client.c

9. Cleanup.

In order to get gprof output for the Seafile client, we have to compile both seafile and ccnet with the -pg flag. This is already done in our build scripts.

Seafile responds to the following inotify events:

- `IN_IGNORED`
- `IN_UNMOUNT`
- `IN_Q_OVERFLOW`
- `IN_MODIFY`
- `IN_CREATE`
- `IN_DELETE`
- `IN_MOVED_FROM`
- `IN_MOVED_TO`
- `IN_CLOSE_WRITE`
- `IN_ATTRIB`

SLEEPS:

The times where `g_usleep` occurred in the program have been replaced by `G_USLEEP`, which is defined as nothing (do {} while (0);). This is to prevent the sync from pausing when the upload/download limit has been passed, since with only 1 user in our tests we don't need it. `G_USLEEP` is defined in seafile/common/common.h, and `g_usleep` has been replaced by `G_USLEEP` in the following files:

- seafile/common/seaf-db.c
- seafile/common/processors/blocktx-common-impl-v2.h
- seafile/daemon/http-tx-mgr.c
- seafile/daemon/block-tx-client.c

The times where `usleep` occurred in the program have been replaced by `USLEEP`, which is defined as `usleep`, i.e. no change (legacy from replacing sleeps of all kinds). These are used to sleep before retrying if unable to access database, which is usually caused by multiple users trying to access it at once. usleep has been replaced with USLEEP in the following files:

- seafile/server/http-server.c
- seafile/server/repo-op.c
- seafile/server/processors/recvbranch-proc-v2.c