# Distributed Systems Lab 06

## Stable storage

During the previous lab, we discussed secure network communication between processes of a distributed system and presented how to implement it in Rust. Another crucial abstraction, especially in distributed algorithms for the crash-recovery failure model, is *stable storage*. This lab thus presents stable storage and discusses how to implement it on top a filesystem. To this end, we will use files contained in this package, which you should download and extract locally.

# Learning section

## Stable storage

In the **crash-recovery failure model**, when a process crashes, the content of its transient memory is lost. However, usually a recovering process should resume its operation rather than restart it from the beginning. To this end, it needs to be able to recover its state (e.g., a current epoch number or a set of pending messages) from before the crash. It is possible if the state is stored in **stable storage**, sometimes called also **persistent storage**.

During this course, we will use stable storage that implements a **key-value** interface. Such storage provides two basic **atomic operations**: `store` and `retrieve`. A value can be stored (written) under some key using `store` and then retrieved (read) via the same key using `retrieve`. Once the `store` operation completes successfully (which in practice means that a call to the corresponding method returns), the written value is reliably saved in the storage and can be read (using `retrieve`) even after process crashes. A subsequent `store` with the same key reliably overwrites the saved value.

The stable storage you will be implementing in today's Small Assignment additionally includes a `remove` operation, which gives guarantees analogous to the ones of `store`. Note that while in practice a `remove` operation might be convenient and used to reduce the size of stored data, it is not necessary for theoretical algorithms, as it can be emulated by overwriting a key's data with a special "nothing" value.

### Stable storage on top a filesystem

An implementation of stable storage can be realized on top of a filesystem provided by an operating system. Strictly speaking, such stable storage requires an assumption that the implementation of the filesystem fully adheres to system call specification, is bug-free and that the underlying drive will not fail. However, the required level of fault tolerance can be, in practice, achieved relatively easy by using some mature filesystem and by increasing the robustness of hardware using information redundancy (e.g., RAID 1). As always, we cannot be *absolutely certain* a single machine would recover, but it should be *virtually impossible* to observe the storage in an inconsistent state.

To implement stable storage on top of a filesystem, an ability to reliably and atomically write data is required. On a POSIX system, we can rely on the `fsync and fsyncdata` system calls as primitive building blocks. These calls block until all modified data of a specified file descriptor is flushed onto the disk. The minor difference between those calls is that `fsync` always flushes metadata associated with the descriptor, while `fsyncdata` flushes only these strictly necessary. Neither of them updates the list of files in the containing directory, and any failure should be considered fatal for the system.

With such an assumption, we can use the following scheme to store data the data in `dstdir/dstfile`:

1. Write the data with a checksum (e.g., CRC32) to a temporary file `dstdir/tmpfile`.
2. Call the POSIX `fsyncdata` function on `dstdir/tmpfile` to ensure the data is actually transferred to a disk device (in Rust, one can use the `tokio::fs::File::sync_data()` method).
3. Call `fsyncdata` on `dstdir` to transfer the data of the modified directory to the disk device. (Again, in Rust, one can use the `tokio::fs::File::sync_data()` method. Even though the struct is called `File`, here it can be used for directories as well, for example: `tokio::fs::File::open("dir").await.unwrap().sync_data().await.unwrap()`).
4. Write the data (without the checksum) to `dstdir/dstfile`.
5. Call `fsyncdata` on `dstdir/dstfile`.
6. Call `fsyncdata` on `dstdir` (only necessary if `dstfile` did not exist before the previous step).
7. Remove `dstdir/tmpfile`.
8. Call `fsyncdata` on `dstdir`.

On recovery, one needs to check the `tmpfile`. If it does not exist, this means that `dstfile` contains valid data (the most recent write was not interrupted). On the other hand, if it exists: - If the checksum is incorrect, this means that a crash during a write caused a corrupted `tmpfile`. This crash must have happened before `dstfile` was modified, so `tmpfile` can be simply removed, thus cancelling the write. - If the checksum is correct, `tmpfile` was fully written and a crash happened later. In this case the write can be resumed using the data from `tmpfile` (regardless of whether `dstfile` is corrupted).

Note that this scheme is atomic with respect to crashes, but it is not atomic with respect to concurrent accesses. Conversely, the `rename` function guarantees atomicity with respect to concurrent accesses, but not with respect to crashes. Furthermore, notice we cover data integrity only across crashes, and rely on filesystem integrity checks otherwise.

# Small Assignment

Your task is to write a Rust implementation of stable storage on top of a filesystem directory. The stable storage shall implement the `StableStorage` trait, which defines key-value storage having the following properties:

- The `put()` method implements the `store` operation described above.
- The `get()` methods implements the `retrieve` operation described above.
- The `remove()` method implements the `remove` operation described above.
- The storage supports keys that are at most 255 bytes long, and values that are at most 65535 bytes long.
- An attempt to store an invalid key or an invalid value returns `Err` with an error message of your choosing. The attempt does not result in a malfunction of the storage.
- Likewise, removing a nonexistent key does not corrupt the storage and just returns `false`.
- If a key has been successfully inserted, updated, or removed before a crash (i.e., a `put()` ended with `Ok` or a `remove()` ended with any result), after the restart the storage returns the most recently stored value for that key (or `None` if the most recent operation for that key was `remove`). This includes crashes of the operating system too.
- Values are stored atomically: either a whole value is stored under a key or removed, or the value is not inserted/updated/removed at all.
- The `get()` method returns `None` for a key that has been removed or has never been inserted.

- The `remove()` method returns `true` if the call actually removed a key and its data; otherwise, the method returns `false`.

Your implementation shall also provide the `build_stable_storage()` function, which returns a trait object implementing the stable storage.

The execution time and number of accessed files for any stable storage operation (including recovery) should be independent of the number of keys kept in the storage (assuming the time it takes to read, write, or remove a file in a directory is independent of the number of files in that directory).

If a filesystem operation fails (e.g., due to insufficient disk space), your solution shall panic. It must not make the stable storage enter an invalid state.

You can assume that the directory provided to the `build_stable_storage()` function already exists and that it will be used exclusively by your stable storage instance.

You can store all keys in memory, but not values.

You are allowed to create subdirectories within the provided directory. You are not allowed to touch any other directories.

You can use the `sha2` and `base64` crates, as imported in `Cargo.toml`. If you use `sha2`, you can assume that there will be no SHA-256 collisions.

The second Large Assignment may require implementing such stable storage, and you can then reuse your solution for this assignment.

## Additional Homework

Observe that databases are an excellent example of systems requiring stable storage: after a transaction is committed, the modified data are expected to be stored reliably despite any software and hardware failures. One of the techniques used by databases to provide stable storage for complex data is a **Write Ahead Log (WAL)**. You can read more about it in a blog post by Daniel Chia. The concept of using a log to guarantee reliable data operations is used also in **journaling filesystems**. You can read about them in this Wikipedia article. In general, append-only data structures are easier to implement safely. However, in order to prevent the data storage from growing uncontrollably, an atomic compaction operation is required.

If you wonder how many bugs can be present in a popular filesystem implementation, we recommend reading the A Study of Linux File System Evolution paper by L. Lu et al., in which the authors examine over 5000 patches from six major filesystems available in Linux.

In the lab, we assumed correctness of the system primitives. However, the practical world is full of corner cases and underspecified behavior. For instance, a hard drive is some configurations may lie about storing the transfer in non-volatile memory, while keeping them in an internal battery-powered buffer. On the other hand, most of the popular filesystems trust the disk for verifying data integrity against corruption, but these checksums are relatively weak. You may find these links useful to deepen your knowledge:

- PostgreSQL fsyncgate 2018: can an application not notice a write failure?
- Can Applications Recover from fsync Failures?: a research paper analyzing various filesystems and applications.
- Linux Kernel documentation on block device data integrity written in 2007.
- Files are fraught with peril: a transcript of a talk covering overall guaranteed persistence.

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski, M. Matraszek.