Distributed Systems Large Assignment 2

Atomic Disk Drive

Your task is to implement a distributed block device that stores data in distributed registers. The solution shall take the form of a Rust library. A template, public tests, and additional files are provided in this package.

Although you will be using the module `System` from the first Large Assignment in some later assignments, we **do NOT** recommend using it for this assignment!

Background

On UNIX-like operating systems, a **block device** is, simplifying, a device that serves random-access reads and writes of data in portions called **blocks**. Arguably, the most common types of block devices are disk drives (e.g., HDD and SSD). However, the term is also applied to various abstractions that implement the interface of the block device (and thus, from a user's perspective, can be used like any other block device), but do not store data on physical devices. For instance, **Atomic Disc Drive**, which stores data in a **distributed register**.

A distributed register consists of multiple **processes** (entities of the distributed register, not operating system processes) running in user space, possibly on multiple physical (or virtual) machines. A Linux block device driver communicates with the processes using TCP. The processes themselves communicate using TCP too. The processes can crash and recover at any time. A number of the processes is fixed before the system is run, and every process is given its own directory where it can store its internal files.

The smallest physical unit inside a block device is called a **sector**. Its size is specific to each device. The size of a block is in turn always a multiple of the size of the sector. In the Atomic Disk Drive, every sector is a separate atomic value called **register** (and thus it is said that the system supports a set of atomic values/registers). The sector has a size of 4096 bytes.

As follows from the above description, a complete Atomic Disk Drive consists of two parts: a Linux block device driver, and a user-space library implementing the distributed register. The Linux block device driver is provided in the package (see instructions on how to use it), and you can use it to test your solution. Your task is to implement in Rust the user-space part as a distributed system.

Distributed register

Your implementation of the distributed register shall be based on an algorithm named **(N, N)-AtomicRegister**.

(N, N)-AtomicRegister

There is a fixed number of instances of the AtomicRegister module, `N`, and all instances know about each other. Crashes of individual instances can happen. Every instance can initiate both read and write operations (thus the `(N, N)` in the name of the algorithm). It is assumed that the system is able to progress on operations as long as at least a majority of the instances are working correctly.

The core algorithm, based on the lecture and modified to suit the crash-recovery model, is as follows (the definition of stubborn links remains the same as in the crash-stop model):

```
Implements:
    (N,N)-AtomicRegister instance nnar.

Uses:
    StubbornBestEffortBroadcast, instance sbeb;
    StubbornLinks, instance sl;

upon event < nnar, Init > do
    (ts, wr, val) := (0, 0, _);
    readlist := [ _ ] `of length` N;
    acklist := [ _ ] `of length` N;
    reading := FALSE;
    writing := FALSE;
    writeval := _;
    readval := _;
    write_phase := FALSE;
    store(wr, ts, val);

upon event < nnar, Recovery > do
    retrieve(wr, ts, val);
    readlist := [ _ ] `of length` N;
    acklist := [ _ ]  `of length` N;
    reading := FALSE;
    readval := _;
    write_phase := FALSE;
    writing := FALSE;
    writeval := _;

upon event < nnar, Read > do
    op_id := generate_unique_id();
    readlist := [ _ ] `of length` N;
    acklist := [ _ ] `of length` N;
    reading := TRUE;
    trigger < sbeb, Broadcast | [READ_PROC, op_id] >;

upon event < sbeb, Deliver | p [READ_PROC, id] > do
    trigger < sl, Send | p, [VALUE, id, ts, wr, val] >;

upon event <sl, Deliver | q, [VALUE, id, ts', wr', v'] > such that id == op_id and
 !write_phase do
    readlist[q] := (ts', wr', v');
    if #(readlist) > N / 2 and (reading or writing) then
        readlist[self] := (ts, wr, val);
        (maxts, rr, readval) := highest(readlist);
        readlist := [ _ ] `of length` N;
        acklist := [ _ ] `of length` N;
        write_phase := TRUE;
        if reading = TRUE then
            trigger < sbeb, Broadcast | [WRITE_PROC, op_id, maxts, rr, readval] >;
        else
            (ts, wr, val) := (maxts + 1, rank(self), writeval);
            store(ts, wr, val);
             trigger < sbeb, Broadcast | [WRITE_PROC, op_id, maxts + 1, rank(self), writeval] >;

upon event < nnar, Write | v > do
    op_id := generate_unique_id();
    writeval := v;
    acklist := [ _ ] `of length` N;
    readlist := [ _ ] `of length` N;
    writing := TRUE;
    trigger < sbeb, Broadcast | [READ_PROC, op_id] >;

upon event < sbeb, Deliver | p, [WRITE_PROC, id, ts', wr', v'] > do
    if (ts', wr') > (ts, wr) then
        (ts, wr, val) := (ts', wr', v');
        store(ts, wr, val);
    trigger < sl, Send | p, [ACK, id] >;

upon event < sl, Deliver | q, [ACK, id] > such that id == op_id and write_phase do
    acklist[q] := Ack;
    if #(acklist) > N / 2 and (reading or writing) then
        acklist := [ _ ] `of length` N;
        write_phase := FALSE;
        if reading = TRUE then
            reading := FALSE;
            trigger < nnar, ReadReturn | readval >;
        else
            writing := FALSE;
            trigger < nnar, WriteReturn >;
```

The `rank(*)` returns a rank of an instance, which is a static number assigned to an instance. The `highest(*)` returns the largest value ordered lexicographically by `(timestamp, rank)`.

Your solution will not be receiving special `Recovery` or `Init` events. Each time it starts, it shall try to recover from the persistent storage (during the initial run, the persistent storage will be empty). Crashes are expected to happen at any point, your solution shall work despite them. The algorithm presented above is only a pseudocode, so we suggest understanding ideas behind it.

Linearization

Usually, components of a distributed system do not share a common clock. Atomic Disk Device does not have one too, and thus the events can happen at different rates and in various orders in every process. However, the atomic register enforces constraints between events on processes, and thereby it makes it possible to put all read and write operations on a single timeline, and to mark the start and end of each operation. Every read returns the most recently written value. If an operation `o` happens before operation `o'` when the system is processing messages, then `o` must appear before `o'` on such a common timeline. This is called *linearization*.

To sum up, from a perspective of a single client (e.g., the Linux driver), there is a single sequence of read and write events. The client would not be able to distinguish between the distributed register and some single device if it was performing the operations instead.

Performance

The atomic register algorithm, as presented above, can only progress with one read or write operation at a time. However, this restriction applies only to a single value. Therefore, to improve the performance of Atomic Disk Device, one can run multiple instances of the atomic register logic, each progressing on a different sector. Your solutions are expected to provide this kind of concurrency and be able to process many sectors at once.

Solution specification

Your solution shall take the form of a cargo library crate. Its main function is `run_register_process()` from the `solution/src/lib.rs` file, which shall run a new process of the distributed register. This function will be used by a simple wrapper—program to run your solution. The process will be passed all necessary information (e.g., addresses for TCP communication, directory for persistent storage, HMAC keys, etc.) via the `Configuration` struct (`solution/src/domain.rs`) provided as an argument to the function.

The solution shall be asynchronous. It will be run using Tokio.

TCP communication

Components of Atomic Disk Device shall communicate using TCP. The communication shall work as follows:

Client-to-process communication

In the following description, we name a *client* any application communicating with the distributed register using TCP. The target client is the Linux device driver, which is a part of Atomic Disk Drive. However, other clients will be also used when evaluating your solution (for instance, when unit testing).

Every process of the system can be contacted by a client. Clients connect using TCP, and send `READ` and `WRITE` commands. Your process must issue replies after a particular command is safely completed by the distributed register. Semantics of commands are mostly self-explanatory, so only their format is presented below. All numbers in the messages are always in the network byte order.

`READ` and `WRITE` commands have the following format:

```
0       7 8     15 16    23 24    31 32    39 40    47 48    55 56    64
+--------+--------+--------+--------+--------+--------+--------+--------+
|             Magic number          |       Padding            |  Msg  |
| 0x61     0x74      0x64     0x64  |                          |  Type |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Request number                              |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Sector index                                |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Command content ...
|
+--------+--------+--------...
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
```

`READ` operation type is `0x01`, `WRITE` operation type is `0x02`. Client can use the request number for internal identification of messages.

`WRITE` has content of 4096 bytes to be written. `READ` has no content. The HMAC tag is a `hmac(sh256)` tag of the entire message (from the magic number to the end of the content).

After the system completes any of these two operations, it shall reply with a response of the following format:

```
0       7 8     15 16    23 24    31 32    39 40    47 48    55 56    64
+--------+--------+--------+--------+--------+--------+--------+--------+
|             Magic number          |     Padding     | Status |  Msg   |
| 0x61     0x74      0x64     0x64  |                 |  Code  |  Type  |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Request number                              |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Response content ...
|
+--------+--------+--------+...
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
```

Again, the HMAC tag is `hmac(sha256)` of the entire response.

Possible status codes are listed in `StatusCode` (`solution/src/domain.rs`), and it is documented there when each code is expected. Status codes shall be consecutive numbers (starting with `0x00` as `Ok`), and be encoded as a single byte.

The response content for a successful `WRITE` is empty, for a successful `READ` it is 4096 bytes read from the system. If a command fails for any reason, the corresponding response shall have an empty content. The response message type is always `0x40` added to original message type: response type for `READ` is `0x41`, response type for `WRITE` is `0x42`.

Requests with an invalid HMAC tag shall be discarded with an appropriate status code returned. A HMAC key for client commands and responses is provided in `Configuration.hmac_client_key` (`solution/src/domain.rs`). When evaluating your solution, the key will be provided to the clients by a testing framework.

You can assume that a client will not send multiple commands with the same sector index at the same time. In other words, if a client sends a command regarding some sector, it will wait for the response before sending another command regarding this sector. However, there can be multiple clients, and different clients can each send a command with the same sector index at the same time.

Operations submitted by clients can be executed in an arbitrary order. The system shall start sending a response when a command is completed by the register.

Since one atomic register can execute only one operation at a time (for a given sector), the operations shall be queued. We suggest using a TCP buffer itself as the queue.

Process-to-process communication

All internal messages (i.e., messages sent between processes) shall have a common header as follows:

```
0       7 8     15 16    23 24    31 32    39 40    47 48    55 56    64
+--------+--------+--------+--------+--------+--------+--------+--------+
|             Magic number          |       Padding   | Process|  Msg   |
| 0x61     0x74      0x64     0x64  |                 |  Rank  |  Type  |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           UUID                                        |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           UUID                                        |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Sector                                      |
|                           index                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Message content ...
|
+--------+--------+--------...
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
```

These internal messages shall be signed with a different HMAC key than messages to/from clients. The key is provided in `Configuration.hmac_system_key` (`solution/src/domain.rs`). The process rank is the rank of a process that sends the command. The message types and their content are as follows:

1. `READ_PROC`

   Type `0x03`, no content.

2. `VALUE`

Type `0x04`, content:

```
0       7 8     15 16    23 24    31 32    39 40    47 48    55 56    64
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Timestamp                                   |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                  Padding                                     | Value  |
|                                                              | wr     |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Sector data ...
|
+--------+--------+--------+...
```

The `value wr` is a write rank of a process, which delivered the last write. The sector data is 4096 bytes long.

3. `WRITE_PROC`

   Type `0x05`, content:

```
0       7 8     15 16    23 24    31 32    39 40    47 48    55 56    64
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Timestamp                                   |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                  Padding                                     | Value  |
|                                                              | wr     |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           Sector data ...
|
+--------+--------+--------+...
```

   The sector data contains 4096 bytes of data to be written. The `value wr` is a write rank of a process, which delivered the last write.

4. `ACK`

   Type `0x06`, no content.

Moreover, you are allowed to use acknowledgment responses to the internal messages. They shall contain the UUID of a message that is being acknowledged, and they shall be signed with the `Configuration.hmac_system_key` HMAC key too. The message format is:

```
0       7 8     15 16    23 24    31 32    39 40    47 48    55 56    64
+--------+--------+--------+--------+--------+--------+--------+--------+
|               Magic number         |     Padding     |Process |  Msg   |
| 0x61     0x74      0x64     0x64   |                 |  Rank  |  Type  |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           UUID                                        |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           UUID                                        |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                           HMAC tag                                    |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                          HMAC tag                                     |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                          HMAC tag                                     |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
|                          HMAC tag                                     |
|                                                                       |
+--------+--------+--------+--------+--------+--------+--------+--------+
```

The process rank is the rank of a process, which sends the response (i.e., acknowledges the message). The message type is always a `0x40` added to the type of the original message. For example, it is `0x46` for `ACK` (as `0x06 + 0x40 = 0x46`).

Handling incorrect messages

When a message, which does not comply with the presented above formats, is received, it shall be handled as follows:

- The solution shall slide over bytes in the stream until it detects a valid magic number. This marks the beginning of a message.
- If a message type is invalid, the solution shall discard the magic number and the following 4 bytes (8 bytes in total).
- In case of every other error, the solution shall consume the same number of bytes as if a message of this type was processed successfully.

Other requirements

We expect that within 300 milliseconds after calling `run_register_process()` a TCP socket will be bound. We suggest the function binds to the appropriate socket before executing other actions.

Your internal TCP client is not allowed to lose any messages, even when a target process crashes and the TCP connection gets broken. Remember that the (N,N)-AtomicRegister algorithm relies on stubborn links, which send messages forever (unless the sender crashes). This behavior models retransmissions. In an actual implementation, messages should be retransmitted (with some delay interval) until the sender learns that further retransmissions are guaranteed to have no influence on the system’s progress.

You are allowed to use custom messages, for instance, for debugging purposes. Their message types shall be numbers equal to or greater than `0x80`.

Messages sent by a process to itself should skip TCP, serialization, deserialization, HMAC preparation and validation phases to improve the performance.

Internal interfaces

Aside from integration tests, which evaluate your solution as a whole, your solution will also be evaluated with unit tests. To this end, the template splits the implementation of the distributed register into multiple parts and defines their interfaces. Most of the interfaces are asynchronous, since running the register will result in multiple IO tasks, and cooperative multitasking seems to be notably profitable in such application.

In the package, we provide a diagram (`atdd.svg`) presenting how Atomic Disk Device might be implemented. Every process of the distributed register is wrapped in a Linux process. Tokio is used as the executor system. The Linux block device driver sends commands to processes over TCP. Processes communicate with one another using internal messages, and then complete the commands and return responses over TCP back to the driver. Every process of the distributed register has multiple copies of the atomic register code, to support concurrent writes/reads on distinct sectors. The copies share a component, which handles communication (`RegisterClient`), and a component, which handles storing sectors data.

Your solution shall implement the following interfaces (they are defined in `solution/src/lib.rs`):

AtomicRegister

`AtomicRegister` provides functionality required of an instance of the atomic register algorithm. It is created by calling `build_atomic_register()` (`solution/src/lib.rs`). All its methods shall follow the atomic register algorithm presented above. When implementing `AtomicRegister`, you can assume that `RegisterClient` passed to the function implements `StubbornLink` required by the algorithm.

Every sector is logically a separate atomic register. However, you should not keep `Configuration.public.n_sectors` `AtomicRegister` objects in memory; instead, you should dynamically create and delete them to limit the memory usage (see also the Technical Requirements section).

SectorsManager

`SectorsManager` facilitates storing sectors data in the filesystem directory. Sector data shall be stored together with necessary basic information, such as the logical timestamp and the write rank (see the pseudocode of the atomic register algorithm).

Sectors are numbered from `0` inclusive to `Configuration.public.n_sectors` (`solution/src/domain.rs`) exclusive. You can assume that `Configuration.public.n_sectors` will not exceed `2^21`.

If a sector was never written, we assume that both the logical timestamp and the write rank are `0`, and that it contains 4096 zero bytes.

No particular storage scheme is required, it must just provide atomic operations. No caching is necessary.

The `build_sectors_manager()` function (`solution/src/lib.rs`) shall create an instance of `SectorManager` for, among others, unit testing. You can assume that the unit tests will not perform concurrent operations on the same sector, even though the trait is marked as *Sync*.

`SectorsManager` is a specialized persistent storage for sector data and metadata. Such specialization allows for optimizing disk usage (see also the Technical Requirements section), and possibly performance.

A directory for `SectorsManager` is provided in `Configuration.public.storage_dir` (`solution/src/domain.rs`).

RegisterClient

`RegisterClient` manages TCP communication between processes of the distributed register. An instance is passed to instances of `AtomicRegister` to allow them communicating with each other.

This trait is introduced mainly for the purpose of unit testing.

When a process sends a message to itself, it is suggested to transfer the message in some manner more directly than through TCP to increase the performance.

Serialization and deserialization

Concerning serialization, your solution shall provide two methods: `deserialize_register_command()` and `serialize_register_command()`. They convert bytes to a `RegisterCommand` object and in the other direction, respectively. They shall implement the message formats as described above (see the description of TCP communication).

Serialization shall complete successfully when there are no errors when writing to the provided reference that implements `AsyncWrite`. If errors occur, the serializing function shall return them.

Deserialization shall return a pair `(message, hmac_valid)` when a valid magic number and a valid message type is encountered. Incorrect messages shall be handled as described above (see the description of TCP communication). An error shall be returned if an error occurs when reading from the provided `AsyncRead` reference.

Technical requirements

Your solution will be tested with the latest stable Rust version.

Interface

You must not modify the public interface of the library. You can, however, implement more derivable traits for public types via the `derive` attribute.

Dependencies

You can use only crates listed in `solution/Cargo.toml` for main dependencies, and anything you want for `[dev-dependencies]` section, as this section will be ignored when evaluating your solution. You are not allowed to use `[build-dependencies]`, but you can specify any number of binaries in your `Cargo.toml` (these will also be ignored during evaluation). If you need any other crate, ask on Moodle for permission. If a permit is granted, every student is allowed to use it.

Using asynchronous libraries makes it easy to scale your solution to the number of available cores, and wait for completions of hundreds of concurrent IO tasks. This is necessary to reach an acceptable performance.

Storage

Because crashes are to be expected, the `Configuration` struct specifies a directory for an exclusive use by a process. You are allowed to create subdirectories within the directory. You are not allowed to touch any other directory.

There is a limit on the number of open file descriptors: 1024. We suggest utilizing it for maximum concurrency. You can assume that there will not be more than 16 client connections.

You can assume that the local filesystem stores data in blocks of 4096 bytes, the same size as the sectors.

Your solution is allowed to use at most 10% more filesystem space than the size of sectors, which have been written to. That is, if there were writes to `n` distinct sectors, it is expected that the total directory size does not exceed `1.1 * n * 4096 + constant` bytes (with the constant being reasonable). Temporary files used for ensuring atomicity do not count towards this limit. However, they must be removed when they are no longer necessary. In particular, this means when the system is not handling any messages, the filesystem usage should be below the limit.

Hint: to fulfill the above requirement, you can try storing sector metadata in filenames. The recovery of `SectorsManager` can have O(n) time complexity.

Memory

Your solution can use memory linear in the number of sectors, which have been written to. It should not use memory linear in the total number of sectors.

Logging

You are allowed to use logging as you wish, as long as your solution does not produce a huge volume of messages at levels ≥ `INFO` when the system is operating correctly. All logging must be done via the `log` crate.

Performance

To receive the maximum number of points for performance, your solution must be able to process at least 50 sectors per second when run on the students machine with 4 system processes and 3 threads (used by tokio runtime) per process. Remember that your solution will probably run slower on students than on a lab computer!

Assignment specification

You are given a subset of official tests (see `public-tests/` in the package). Their intention is to make sure that the public interface of your solution is correct, and to evaluate basic functionality.

Submitting a solution

Your solution must be submitted as a single `.zip` file with its name being your login at students (e.g., `ab123456.zip`). After unpacking the archive, a directory path named `ab123456/solution/` must be created. In the `solution` subdirectory there must be a Rust library crate that implements the assignment. Project `public-tests` must be able to be built and tested cleanly when placed next to the `solution` directory.

Grading

Your solution will be graded based on the results of automatic tests and code inspection. The number of available and required points is specified in the Passing Rules described at the main website of the course. If your solution passes the public tests, you will receive at least the required number of points. Solutions, which will not actually implement a distributed system (e.g., they will keep all data in RAM only, execute commands on a single node, etc.) will be rejected.

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