PNA Rust Project 3: Synchronous client-server networking

Task: Create a *single-threaded*, persistent key/value store *server and client with synchronous networking over a custom protocol*.

Goals:

- Create a client-server application
- Write a custom protocol with `std` networking APIs
- Introduce logging to the server
- Implement pluggable backends with traits
- Benchmark the hand-written backend against `sled`

Topics: `std::net`, logging, traits, benchmarking.

- Introduction
- Project spec
- Project setup
- Part 1: Command line parsing
- Part 2: Logging
- Part 3: Client-server networking setup
- Part 4: Implementing commands across the network
- Part 5: Pluggable storage engines
- Part 6: Benchmarking

Introduction

In this project you will create a simple key/value server and client. They will communicate with a custom networking protocol of your design. You will emit logs using standard logging crates, and handle errors correctly across the network boundary. Once you have a working client-server architecture, then you will abstract the storage engine behind traits, and compare the performance of yours to the [`sled`] engine.

Project spec

The cargo project, `kvs`, builds a command-line key-value store client called `kvs-client`, and a key-value store server called `kvs-server`, both of which in turn call into a library called `kvs`. The client speaks to the server over a custom protocol.

The `kvs-server` executable supports the following command line arguments:

- `kvs-server [--addr IP-PORT] [--engine ENGINE-NAME]`

  Start the server and begin listening for incoming connections. `--addr` accepts an IP address, either v4 or v6, and a port number, with the format `IP:PORT`. If `--addr` is not specified then listen on `127.0.0.1:4000`.

  If `--engine` is specified, then `ENGINE-NAME` must be either "kvs", in which case the built-in engine is used, or "sled", in which case sled is used. If this is the first run (there is no data previously persisted) then the default value is "kvs"; if there is previously persisted data then the default is the engine already in use. If data was previously persisted with a different engine than selected, print an error and exit with a non-zero exit code.

  Print an error and return a non-zero exit code on failure to bind a socket, if `ENGINE-NAME` is invalid, if `IP-PORT` does not parse as an address.

- `kvs-server -V`

  Print the version.

The `kvs-client` executable supports the following command line arguments:

- `kvs-client set <KEY> <VALUE> [--addr IP-PORT]`

  Set the value of a string key to a string.

  `--addr` accepts an IP address, either v4 or v6, and a port number, with the format `IP:PORT`. If `--addr` is not specified then connect on `127.0.0.1:4000`.

  Print an error and return a non-zero exit code on server error, or if `IP-PORT` does not parse as an address.

- `kvs-client get <KEY> [--addr IP-PORT]`

  Get the string value of a given string key.

  `--addr` accepts an IP address, either v4 or v6, and a port number, with the format `IP:PORT`. If `--addr` is not specified then connect on `127.0.0.1:4000`.

  Print an error and return a non-zero exit code on server error, or if `IP-PORT` does not parse as an address.

- `kvs-client rm <KEY> [--addr IP-PORT]`

  Remove a given string key.

  `--addr` accepts an IP address, either v4 or v6, and a port number, with the format `IP:PORT`. If `--addr` is not specified then connect on `127.0.0.1:4000`.

  Print an error and return a non-zero exit code on server error, or if `IP-PORT` does not parse as an address. A "key not found" is also treated as an error in the "rm" command.

- `kvs-client -V`

  Print the version.

All error messages should be printed to stderr.

The `kvs` library contains four types:

- `KvsClient` - implements the functionality required for `kvs-client` to speak to `kvs-server`
- `KvsServer` - implements the functionality to serve responses to `kvs-client` from `kvs-server`
- `KvsEngine` trait - defines the storage interface called by `KvsServer`
- `KvStore` - implements by hand the `KvsEngine` trait
- `SledKvsEngine` - implements `KvsEngine` for the [`sled`] storage engine.

The design of `KvsClient` and `KvsServer` are up to you, and will be informed by the design of your network protocol. The test suite does not directly use either type, but only exercises them via the CLI.

The `KvsEngine` trait supports the following methods:

- `KvsEngine::set(&mut self, key: String, value: String) -> Result<()>`

  Set the value of a string key to a string.

  Return an error if the value is not written successfully.

- `KvsEngine::get(&mut self, key: String) -> Result<Option<String>>`

  Get the string value of a string key. If the key does not exist, return `None`.

  Return an error if the value is not read successfully.

- `KvsEngine::remove(&mut self, key: String) -> Result<()>`

  Remove a given string key.

  Return an error if the key does not exit or value is not read successfully.

When setting a key to a value, `KvStore` writes the `set` command to disk in a sequential log. When removing a key, `KvStore` writes the `rm` command to the log. On startup, the commands in the log are re-evaluated and the log pointer (file offset) of the last command to set each key recorded in the in-memory index.

When retrieving a value for a key with the `get` command, it searches the index, and if found then loads from the log, and evaluates, the command at the corresponding log pointer.

When the size of the uncompacted log entries reach a given threshold, `KvStore` compacts it into a new log, removing redundant entries to reclaim disk space.

Project setup

Continuing from your previous project, delete your privous `tests` directory and copy this project's `tests` directory into its place. This project should contain a library named `kvs`, and two executables, `kvs-server` and `kvs-client`.

You need the following dev-dependencies in your `Cargo.toml`:

```toml
[dev-dependencies]
assert_cmd = "0.11"
criterion = "0.2.11"
predicates = "1.0.0"
rand = "0.6.5"
tempfile = "3.0.7"
walkdir = "2.2.7"
```

As with previous projects, add enough definitions that the test suite builds.

Part 1: Command line parsing

There's little new about the command line parsing in this project compared to previous projects. The `kvs-client` binary accepts the same command line arguments as in previous projects. And now `kvs-server` has its own set of command line arguments to handle, as described previously in the spec.

Stub out the `kvs-server` command line handling.

Part 2: Logging

Production server applications tend to have robust and configurable logging. So now we're going to add logging to `kvs-server`, and as we continue will look for useful information to log. During development it is common to use logging at the `debug!` and `trace!` levels for "println debugging".

There are two prominent logging systems in Rust: [`log`] and [`slog`]. Both export similar macros for logging at different levels, like `error!`, `info!` etc. Both are extensible, supporting different backends, for logging to the console, logging to file, logging to the system log, etc.

The major difference is that `log` is fairly simple, logging only formatted strings; `slog` is feature-rich, and supports "structured logging", where log entries are typed and serialized in easily-parsed formats.

`log` dates from the very earliest days of Rust, where it was part of the compiler, then part of the standard library, and finally released as its own crate. It is maintained by the Rust Project. `slog` is newer and maintained independently. Both are widely used.

For both systems, one needs to select a "sink" crate, one that the logger sends logs to for display or storage.

Read about both of them, choose the one that appeals to you, add them as dependencies, then modify `kvs-server` to initialize logging on startup, prior to command-line parsing. Set it up to output to stderr (sending the logs elsewhere additionally is fine, but they must go to stderr to pass the tests in this project).

On startup log the server's version number. Also log the configuration. For now that means the IP address and port, and the name of the storage engine.

Part 3: Client-server networking setup

Next we're going to set up the networking. For this project you are going to be using the basic TCP/IP networking APIs in `std::net`: [`TcpListener`] and [`TcpStream`].

For this project, the server is synchronous and single-threaded. That means that you will listen on a socket, then accept connections, and execute and respond to commands one at a time. In the future we will re-visit this decision multiple times on our journey toward an asynchronous, multi-threaded, and high-performance database.

Think about your manual testing workflow. Now that there are two executables to deal with, you'll need a way to run them both at the same time. If you are like many, you will use two terminals, running `cargo run --bin kvs-server` in one, where it runs until you press CTRL-D, and `cargo run --bin kvs-client` in the other.

This is a good opportunity to use the logging macros for debugging. Go ahead and log information about every accepted connection.

Before thinking about the protocol, modify `kvs-server` to listen for and accept connections, and `kvs-client` to initiate connections.

Part 4: Implementing commands across the network

In the last project you defined the commands your database accepts, and learned how to serialize and deserialize them to and from the log with `serde`.

Now it's time to implement the key/value store over the network, remotely executing commands that until now have been implemented within a single process. As with the file I/O you worked on in the last project to create the log, you will be serializing and streaming commands with the `Read` and `Write` traits.

You are going to design a network protocol. There are a number of ways to get data in and out of a TCP stream, and a number of decisions to make. Is it a text-based protocol, binary? How is the data translated from its format in memory to its format byte-stream format? Is there a single request per connection, or many?

Keep in mind that it must support successful results and errors, and there are two kinds of errors now: the ones generated by your storage engine, as well as network errors.

All the details of the protocol are up to you. The test suite does not care at all how the data gets from one end to the other, just that the results are correct.

Write your network protocol.

<!-- ## Part 5: More error handling

TODO write this section

- handle error responses by converting errors to a serializable format
- add context to errors

- replace `fn main() -> Result` with custom error reporting -->

Part 5: Pluggable storage engines

Your database has a storage engine, `KvStore`, implemented by you. Now you are going to add a second storage engine.

There are multiple reasons to do so:

- Different workloads require difference performance characteristics. Some storage engines may work better than other based on the workload.
- It creates a familiar framework for comparing different backends.
- It gives us an excuse to create and work with traits.
- It gives us an excuse to write some comparative benchmarks!

So you are going to *extract* a new trait, `KvsEngine`, from the `KvStore` interface. This is a classic *refactoring*, where existing code is transformed into a new form incrementally. When refactoring you will generally want to break the work up into the smallest changes that will continue to build and work.

Here is the API you need to end up with:

- `trait KvsEngine` has `get`, `set` and `remove` methods with the same signatures as `KvStore`.
- `KvStore` implements `KvsEngine`, and no longer has `get`, `set` and `remove` methods of its own.
- There is a new implementation of `KvsEngine`, `SledKvsEngine`. You need to fill its `get` and `set` methods using the `sled` library later.

It's likely that you have already stubbed out the definitions for these if your tests are building. *Now is the time to fill them in.* Break down your refactoring into an intentional sequence of changes, and make sure the project continues to build and pass previously-passing tests before continuing.

As one final step, you need to consider what happens when `kvs-server` is started with one engine, is killed, then restarted with a different engine. This case can only result in an error, and you need to figure out how to detect the case to report the error. The test `cli_wrong_engine` reflects this scenario.

Part 6: Benchmarking

As the course progresses we will increasingly concern ourselves with the performance of the database, exploring the impact of different architectures. You are encouraged to go beyond the model described herein and experiment with your own optimizations.

Performance work requires benchmarking, so now we're going to get started on that. There are many ways to benchmark databases, with standard test suites like [ycsb] and [sysbench]. In Rust benchmarking starts with the builtin tooling, so we will start there.

Cargo supports benchmarking with `cargo bench`. The benchmarks may either be written using Rust's built in benchmark harness, or an external one.

The built-in harness creates benchmarks from functions with the `#[bench]` attribute. It cannot be used on the Rust stable channel though, and is only documented briefly in the unstable book and the `test` crate docs. It is though widely used throughout the Rust ecosystem — crates that use it, even if they compile with stable releases, do benchmarking with nightly releases.

That system though is effectively deprecated — it is not being updated and will seemingly never be promoted to the stable release channel.

There are better benchmark harnesses for Rust anyway. The one you will use is [criterion]. And you will use it to satisfy your curiosity about the performance of your `kvs` engine compared to the `sled` engine.

These benchmarking tools work by defining a benchmarking function, and within that function iterating through a loop that performs the operation to be benchmarking. The benchmarking tool will iterate as many times as it needs to in order to know the duration of the operation with statistical significance.

See this basic example from the criterion guide:

```
fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("fib 20", |b| {
      b.iter(|| {
        fibonacci(20)
    });
  });
}
```

The call to `bench_function` defines the benchmark, and the call to `iter` defines the code that is run for the benchmark. Code before and after the call to `iter` is not timed.

Prepare for writing benchmarks by creating a file called `benches/benches.rs`. Like `tests/tests.rs`, cargo will automatically find this file and compile it as a benchmark.

Start by writing the following benchmarks:

- `kvs_write` - With the kvs engine, write 100 values with random keys of length 1-100000 bytes and random values of length 1-100000 bytes.
- `sled_write`- With the sled engine, write 100 values with random keys of length 1-100000 bytes and random values of length 1-100000 bytes.
- `kvs_read` - With the kvs engine, read 1000 values from previously written keys, with keys and values of random length.
- `sled_read` - With the sled engine, read 1000 values from previously written keys, with keys and values of random length.

(As an alternative to writing 4 benchmarks, you may also choose to write 2 benchmarks parameterized over the engine, as described in the criterion manual).

These are underspecified, and there's a fair bit of nuance to implementing them in a useful way. We need to consider at least three factors:

- What code should be timed (and be written inside the benchmark loop), and what code should not (and be written outside the benchmark loop)?

- How to make the loop run identically for each iteration, despite using "random" numbers.
- In the "read" benchmarks, how to read from the same set of "random" keys that were written previously.

These are all inter-related: some code needs to be carefully selected as un-timed setup code, and the seed values for random number generators need to be re-used appropriately.

In all cases, operations that may return errors should assert (with `assert!`) that they did not return an error; and in the read case, "get" operations should assert that the key was found.

Random numbers can be generated with the [`rand`] crate.

Once you have your benchmarks, run them with `cargo bench`.

Write the above benchmarks, and compare the results between `kvs` and `sled`.

Note: please run the benchmarks on an otherwise unloaded machine. Benchmark results are very sensitive to the environment they are run in, and while the criterion library does its best to compensate for "noise", benchmarks are best done on a clean machine without other active processes. If you have a spare machine just for development, use that. If not, an AWS or other cloud instance may produce more consistent results than your local desktop.

Nice coding, friend. Enjoy a nice break.

<!-- TODO

Extension 1: Signal handling

- Shutdown on KILL
- TODO need to figure out how to interrupt the tcp listener -->

<!--

Background reading ideas

- log docs
- slog docs
- TCP/IP basics
- refactoring overview
- traits and impl trait
- https://bheisler.github.io/post/benchmarking-with-criterion-rs/
- general overview of benchmarking
- conditional compilation of engines?

TODOs

- consider `Kvs_Engine_` trait vs `Kv_Store_` impl

-->