CS380P: Parallel Systems

Lab: Two-Phase-Commit

The functional goal of this assignment is to use Rust to implement a simple 2PC (two-phase commit) protocol.

The pedagogical goals of this assignment are two-fold:

- To gain exposure to using programming tools that leverage language-level type systems to manage concurrency and provide safety guarantees. Indeed, **the primary goal of this assignment is to increase your facility with programming in this kind of idiom**, as we feel this is going to be increasingly important in future systems and languages. If you've had no previous exposure to Rust, you will likely find it's type system and approach to concurrency quite restrictive, and learning it will take some time.
- To gain some exposure to implementing distributed protocols like 2PC. However, since the primary goal exposure to Rust's concurrency paradigm, the project **does not require** an implementation of 2PC that handles difficult cases like recovery (although you can handle them for extra credit, see below). The project requires you to implement 2PC across multiple processes communicating with each other using the IPC offered by ipc-channel and the correctness properties we will evaluate are restricted to checking for agreement between the coordinator and participants that all transactions are either committed or aborted.

In this assignment, you will write a two phase commit protocol that works with a single coordinator an arbitrary number of concurrent clients (issuing requests to the coordinator) and an arbitrary number of 2PC participants. To restrict the scope of the assignment, your solution may make the following assumptions:

- The coordinator does not fail. Recall from lecture that handling this requires additional mechanism such as 3-phase commit.
- Participants may fail either by explicitly voting abort in phase one, or by not responding to requests. You can handle failed participants by allowing them to rejoin silently. You do not need to implement a recovery protocol (although you can for extra credit).
- Requests can be represented by a single operation ID. This simplification comes at no loss of generality for the 2PC protocol itself.

Inputs/Outputs

Your program should accept the command line parameters below. If you start with the skeleton code provided, the command line parameter passing and initialization of a number of important bits of infrastructure should be already taken care of for you.

```
./target/debug/two_phase_commit --help

USAGE:
    two_phase_commit [OPTIONS]

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -l                              Specifies path to directory where logs are stored
    -m                                  Mode: "run" starts 2PC, "client" starts a client process, "particip
                                        starts a participant process, "check" checks logs produced by
    -c                           Number of clients making requests
    -p                       Number of participants in protocol
    -r                           Number of requests made per client
    -s         Probability participants successfully execute requests
    -S              Probability participants successfully send messages
    -v                              Output verbosity: 0->No Output, 5->Output Everything
        --ipc_path                   Path for IPC socket for communication
        --num                             Participant / Client number for naming the log files. Ranges from
                                               num_clients - 1 or num_participants - 1
```

When you run the project in "run" mode, it should start the coordinator and launch the client and participant processes and store the commit logs in ./logs. Clients and participants are launched by spawning new children processes with the same arguments except in the "client" and "participant" mode respectively. In addition, the parent (coordinator) process will need to utilize the --ipc_path to set up IPC communication and the --num parameter to properly name the participant log files. When you run the project in "check" mode, it will analyze the commit logs produced by the last run to determine whether your 2PC protocol handled the specified number of client requests correctly.

Using the Skeleton Code

We provide a skeleton implementation that should help get started and insulate you from some of the details of managing the rust build system. The skeleton code provided includes a specification for the cargo build tool (see below) and the simple module decomposition overviewed in the table below. The "Edits Needed?" Column indicates whether we expect you to need to write code in that file or not. The description column summarizes the module functionality and (if appropriate) the changes we expect you to make in that module.

File	Edits Needed?	Description
Cargo.toml	no	Build tool input, specifies dependences
src/main.rs	yes	Starts application, required changes to launch clients, participants, coordinator.

src/client.rs	yes	Implements client-side protocol and state tracking
src/participant.rs	yes	Implements participant-side protocol and state tracking
src/coordinator.rs	yes	Implements coordinator-side protocol and state tracking
src/message.rs	maybe	Provides message format and related enums.
src/tpcoptions.rs	maybe	Implements command line options and tracing initialization
src/oplog.rs	no	Implements a commit log API to use in coordinator and participant
src/checker.rs	no	Implements the correctness check for "check" mode.

In general, the code in the skeleton is documented to be explicit about what functionality must be implemented and where. For example, the following code fragment from `src/main.rs` documents what the `run_participant()` function should do, provides specifications for each parameter, a handful of HINTS to keep in mind and an explicit block comment where we expect you to need to implement something. While in most cases you have considerably more design freedom than is illustrated by this example, you should find that the skeleton provides reasonably complete documentation of what is expected in each module.

```
///
/// pub fn run_participant(opts: &tpcoptions:TPCOptions, running: Arc<AtomicBool>)
///     opts: An options structure containing the CLI arguments
///     running: An atomically reference counted (ARC) AtomicBool(ean) that is
///         set to be false whenever Ctrl+C is pressed
///
/// 1. Connects to the coordinator to get tx/rx
/// 2. Constructs a new participant
/// 3. Starts the participant protocol
///
fn run_participant(opts: & tpcoptions::TPCOptions, running: Arc<AtomicBool>) {
    let participant_id_str = format!("participant_{}", opts.num);
    let participant_log_path = format!("{}//{}.log", opts.log_path, participant_id_str);

    // TODO
}
```

Using cargo build

The skeleton code provided includes a specification for the cargo build tool (see below for more about building) to simplify getting up and running. We expect that with a functional rust installation (if it is not part of your default environment, you can find simple instructions here), building the project should be a matter of typing:

```
cargo build
```

When you build the skeleton code, you should get a functional executable right away. You will see a large number of warnings in the skeleton code, reflecting the fact that that the rust compiler does not like the unreferenced variables and modules left by removing logic from our solution to create the skeleton.

Step 1: 90/100 points

Implement the 2PC protocol, handling arbitrary success/failure of operations at participants.

What we will check: We will test your code with various numbers of concurrent clients, participants, and numbers of requests per client, by first running the project in "run" mode, and next running with the same set of parameters in "check" mode. A scenario is illustrated below where we first run with 10 participants, 4 clients issuing 10 requests each, with a 95% probability that each participants successfully executes each request (and thus votes commit in phase 1). Next, we run with the same parameters in "check" mode, causing the program to load the commit logs produced by the first run, and analyze them for correctness. Note that the checker logic is implemented in `src/checker.rs` and is provided for you. If you use the message codes and message types specified in `src/message.rs`, the checker should "just work" for you.

```
rossbach@moonstone> ./target/debug/two_phase_commit -s .95 -c 4 -p 10 -r 10 -m run
CTRL-C!
coordinator:    C:26    A:14    U:0
participant_2:  C:26    A:14    U:0
participant_1:  C:26    A:14    U:0
participant_8:  C:26    A:14    U:0
participant_3:  C:26    A:14    U:0
participant_6:  C:26    A:14    U:0
participant_0:  C:26    A:14    U:0
participant_5:  C:26    A:14    U:0
participant_9:  C:26    A:14    U:0
participant_4:  C:26    A:14    U:0
participant_7:  C:26    A:14    U:0
rossbach@moonstone> ./target/debug/two_phase_commit -s .95 -c 4 -p 10 -r 10 -m check -v 0
participant_9 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
participant_4 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
participant_3 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
participant_1 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
participant_5 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
participant_6 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
```

```
participant_8 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
participant_0 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
participant_2 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
participant_7 OK: C:26 == 26(C-global), A:14 <= 14(A-global)
```

Step 2: 10/100 points

Extend your solution to support message send failure and operation failure. This means handling runs that specify "-S" as well as "-s". The former manages send success probability, while the latter manages execution success rate of requests per participant.

Step 3: Extra Credit Options: 10 points each

- Extend your solution to use a fault tolerant log such as commitlog.
- Implement a recovery protocol. Allow a failed participant to rejoin.
- Support coordinator failure.

Hints

- Use the trace/info/debug macros. We have initialized `stderrlog` for you, so you should have a rich set of tools for instrumenting your application to simplify debugging. We have left many fragments of code that use `trace!()` and `info!()` to illustrate how to use them. You just need to vary the verbosity argument to set the logging level.
- Rust documentation is very good. We found The Rust Programming Language and Rust by Example to be invaluable during the development of this programming project.
- The documentation for IpcOneShotServer should help with setting up ipc-channel.

Deliverables

When you submit your solution, you should include the following:

- A written report of the assignment in either plain text or PDF format. Please explain your approach, present performance results, discuss any insights gained, etc.
- Your source code, instructions to build it if "cargo build" does not suffice.

Please report how much time you spent on the lab.