Lab guidance

Hardness of assignments

Each task indicates how difficult it is:

- Easy: A few hours.
- Moderate: ~ 6 hours (per week).
- Hard: More than 6 hours (per week). If you start late, your solution is unlikely to pass all tests.

These times are rough estimates of our expectations.

The tasks in general require not many lines of code (a few hundred lines), but the code is conceptually complicated and often details matter a lot. Some of the tests are difficult to pass.

> **Important:**
>
> Don't start a lab the night before it is due; it's much more time efficient to do the labs in several sessions spread over multiple days. Tracking down bugs in distributed system code is difficult, because of concurrency, crashes, and an unreliable network. Problems often require thought and careful debugging to understand and fix.

Tips

- Do the Online Go tutorial and consult How to write Go code. See Editors to set up your editor for Go.
- Use Go's race detector, with `go build -race` and `go run -race`. Fix any races it reports.
- Read this guide for Raft specific advice.
- Advice on locking in labs.
- Advice on structuring your Raft lab.
- This Diagram of Raft interactions may help you understand code flow between different parts of the system.
- It may be helpful when debugging to insert print statements when a peer sends or receives a message, and collect the output in a file with `go test -race > out`. Then, by studying the trace of messages in the `out` file, you can identify where your implementation deviates from the desired behavior.
- Structure your debug messages in a consistent format so that you can use `grep` to search for specific lines in `out`.
- You might find `DPrintf` useful instead of calling `log.Printf` directly to turn printing on and off as you debug different problems.
- You can use formatting options like colors or columns to make understanding the log output easier. This post explains how to build a tool that labels logs with topics and then pretty prints the logs using colorized terminal output with separate columns for each server.
- To learn more about git, look at the Pro Git book or the git user's manual.

Debugging as a formal process

Debugging complex concurrent systems is not magic. Debugging is a science and a skill, and it can be learned and practiced. When done well, debugging is a process that will lead you inerringly from an initial observation of an error back to the exact fault that caused that error. In general, most bugs in complex systems cannot be solved using ad-hoc or "guess-and-check" approaches; you are likely to have the most success by being as methodical in your approach as possible.

Helpful definitions:

- A **fault** is an underlying cause of a bug. This is the mistake in a piece of code that allows for something to go wrong. They can be anything from a typo to a fundamental misunderstanding of a protocol that is enshrined in your code.
- An **error** is a deviation at a particular point in time between the theoretical correct state of a program and the actual state of the program at that time. An error is either a latent error, an observable error, or a masked error.
    - A **latent error** is not visible to you; it is silently propagating within your code. It will eventually become an observable error or a masked error.
    - An **observable error** is visible; it has been **surfaced** in the observable output of the program, either as an error message, an unexpected output, or the lack of an expected output.
    - A **masked error** is where wherein some property of the implementation or design allows a previous latent error to be ignored. For example, if a log entry is incorrectly added in Raft (a latent error), but then overwritten later by a correct entry before it is reported anywhere, the original error has been masked and can never become observable.
- **Instrumentation** refers to the parts of your code that report on the current state of your program. This includes both code to explicitly detect particular possible errors, and code to print out state so that it can later be manually examined to see whether or not it indicates the presence of any errors.

You can debug forwards (from fault to error), or backwards (from error to fault). The forward approach tends to devolve into a "guess-and-check" approach, and frequently does not allow for consistent or repeatable progress. By contrast, the backwards approach is always applicable to the 6.824 labs, and allows for making consistent progress, so we will focus on it here.

The backwards approach is always applicable for our labs because you always have an observable error to start with: the test cases themselves function as a form of instrumentation, which will detect certain kinds of errors and surface them to you.

Working backwards to find an error

The execution of a program that you are debugging can generally be to have three phases:

1. The phase in which the execution of the program is correct and error-free. A fault may exist in the program, but it has not yet been manifested as an actual error.
2. The phase in which a fault has manifested, and has produced one or more latent errors in the program state. These are not yet visible, and may silently propagate within the program state.
3. The phase in which the latent errors have surfaced, and become observable errors. At this point, you have an indication that there's something wrong with the running program.

The goal of debugging is to narrow in on the exact location of the fault within your implementation. This means you need to identify the point in time when the fault is manifested, and the program moves from phase 1 to phase 2. It's hard to find the exact extent of phase 1, because it is generally

prohibitively difficult to analyze the complete state of a program in sufficient detail to be certain that a fault has not yet manifested itself at any point in time.

Instead, the best approach is usually to work backwards and narrow down the size of phase 2 until it is as small as possible, so that the location of the fault is readily apparent. This is done by expanding the instrumentation of your code to surface errors sooner, and thereby spend less time in phase 2. This generally involves adding additional debugging statements and/or assertions to your code.

Making progress

When adding instrumentation, you want to focus on making clear and deliberate progress at narrowing down the cause of the fault. You should identify the first observable error you can in your debugging output, and then attempt to narrow down the most proximate cause of that error. This is done by forming a hypothesis about what the most proximate cause of the error could be, and then adding instrumentation to test that hypothesis. If your hypothesis is true, then you have a new "first observable error" and can repeat the process. If false, then you must come up with a new hypothesis about the proximate cause and test that instead.

It's important to maintain a sense of exactly what your current first observable error is, as you move backwards, so that you don't get lost among other errors that may be observable in your output.

At the beginning, you don't know anything about where the fault could be; it could be anywhere in your entire program. But as you advance, the possible interval in which the fault could be will narrow, until you reach a single line of code that must contain the fault. When you're down to the execution of a single function or a single block of code, it can be helpful to use a "bisection" approach, where you repeatedly add instrumentation halfway through the interval, and narrow down to one half or the other. (This is like binary search, but for finding bugs.)

Adding instrumentation

As a debugger, your main challenge is to pick the best locations for your instrumentation to best narrow down the location of your fault, along with deciding the most useful pieces of information to report in that piece of instrumentation. You can and should use your knowledge of your implementation to speed this up, but you must always check your assumptions by actually running your code and observing the results of your instrumentation.

Because instrumentation is so essential to your debugging process, you should take the time to design and implement it carefully. Consider some of the following questions when designing your instrumentation:

- How much detail do you need from your instrumentation? Either in general, or just for the current step in your debugging process? How can you make it easier to adjust the level of detail and the main focus of your instrumentation? Can you turn on or off different pieces of debugging without deleting them from your code?

  (In particular, consider using an approach like the provided DPrintf function does, and defining one or more constant boolean flags to turn on or off different aspects of your instrumentation.)

- How can you optimize your own ability to quickly read and understand what your instrumentation is indicating? Can you use text colors, columns, consistent formats, codewords, or symbols to make it easier to read?

  The best approach will be personalized to the particular way that YOU best percieve information, so you should experiment to find out what works well for you.

- How can you enhance your own ability to add instrumentation? Can you use existing tools (like the go "log" package) to help? (I recommend turning on the Lmicroseconds flag, if you do.)

  Can you build your own helper functions, so that a common set of data (current server, term, and role, perhaps?) will always be displayed?

One specific note: make sure to learn about format strings if you aren't already familiar with them. You can refer to the Wikipedia article on Printf format strings, or to the Go-specific documentation. You will likely be much happier using functions like 'log.Printf' or 'fmt.Printf' than trying to achieve the same effects with multiple print functions.

You might also consider trying to condense each individual event you report into a single line to facilitate your ability to scan output quickly.

Tips on asking for help

As mentioned above, debugging is a complex and iterative process. It can be hard for TAs to help you debug your code over Piazza; we strongly encourage you to bring your difficult debugging challenges to Office Hours instead! We will be happy to help walk you through the debugging approaches that are best applicable to your situation. Of course, you are likely to get the most mileage out of your time in Office Hours if you make a significant attempt at debugging the problem yourself in as methodical a manner as possible.

Tips on timeouts

It should be noted that tweaking timeouts rarely fixes bugs, and that doing so should be a last resort. We frequently see students willing to keep making arbitrary tweaks to their code (especially timeouts) rather than following a careful debugging process. Doing this is a great way to obscure underlying bugs by masking them instead of fixing them; they will often still show up in rare cases, even if they appear fixed in the common case.

In particular, in Raft, there are wide ranges of timeouts that will let your code work. While you CAN pick bad timeout values, it won't take much time to find timeouts that are functional.

Additional debugging tips

- It's worth noting that there may be multiple faults in your code! It is often easiest to narrow down a bug by focusing on the first fault, rather than the last, because then there will be fewer errors in the program state to consider.

- Try to avoid ruling out a bug in one piece of code simply because you believe that it's correct. If you're dealing with a bug, that usually implies that there's something wrong with your mental model of your code -- either how your implementation actually works, or else your understanding of how it's supposed to work. As such, you can't rely too much on your mental model; always verify your assumptions.

- Debugging is a complex and multifaceted skill, and doing it well takes discipline and clear consideration. Try to think carefully about why your debugging approaches are succeeding or failing (or even just being convenient or painful), and take the time to find and explore new approaches that may help you. Time invested now in improving your debugging knowledge will pay off well in later labs.

- Making premature fixes are often dangerous when debugging. They may or may not solve the issue you're locking at, and are just as likely to shift the exact location and presentation of the

fault you're trying to track down. Even worse, they may simply mask the real fault, rather than solve it. Wait until you're confident about the exact fault causing your observable error before you attempt a fix.

- Sometimes, making a fix doesn't immediately solve all of your problems. Sometimes, the same test case still fails, because the old fault was corrected, and a new fault that was previously hidden is now visible.

- Don't neglect the possibility of bugs in your glue code. Elements like main loops, locks, and channel I/O may seem simple and unlikely to be wrong, but they can sometimes hide extremely challenging bugs.

- When possible, consider writing your code to "fail loudly". Instead of trying to tolerate unexpected states, try to explicitly detect states that should never be allowed to happen, and immediately report these errors. Consider even immediately calling the Go 'panic' function in these cases to fail especially loudly. See also the Wikipedia page on Offensive programming techniques. Remember that the longer you allow errors to remain latent, the longer it will take to narrow down the true underlying fault.

- When dealing with a bug that occurs sporadically, the best approach is usually to log aggressively and dump the output to a file. It's easier to filter out irrelevant parts of a verbose log (such as with a separate script) than it is to wait for the error to reappear after N runs.

- When you're failing a test, and it's not obvious why, it's usually worth taking the time to understand what the test is actually doing, and which part of the test is observing the problem. It can be helpful to add print statements to the test code so that you know when events are happening.

- And one last note: locking strategy matters. A lack of races reported by the race detector does not indicate that your locking strategy is correct. In particular, fine-grained locks can be dangerous, because they can introduce more changes for interleaving.

Happy debugging!