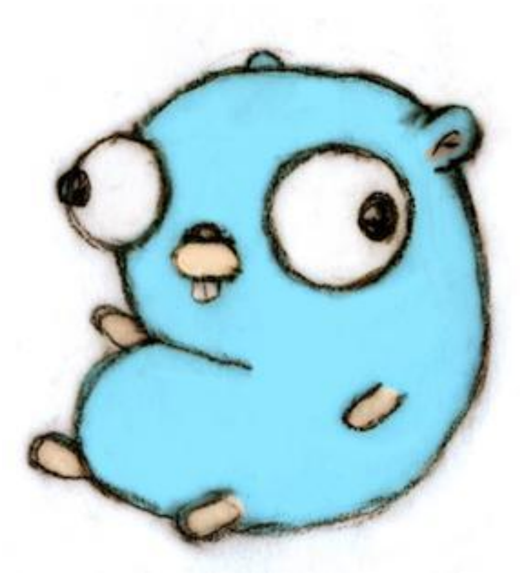
P1 : Distributed Bitcoin Miner



15-440/15-640

Fall 2021

Overview

- OH/Piazza Policy Reminder
- P1 Part B
- Q&A
- Appendix: P1 Post-checkpoint Unit Tests

OH/Piazza Policy Reminder

- You do have a partner to work together! Make sure to discuss with your partner when you are blocked.
- Review the slides from debugging recitation and make sure you follow the everything listed there + check out FAQ and other posts on piazza.
- Provide context when asking questions on Piazza or during OH.
- If you do not make enough effort before asking for help, the TAs can refuse to help you.
- Please do not make private post about debugging questions.

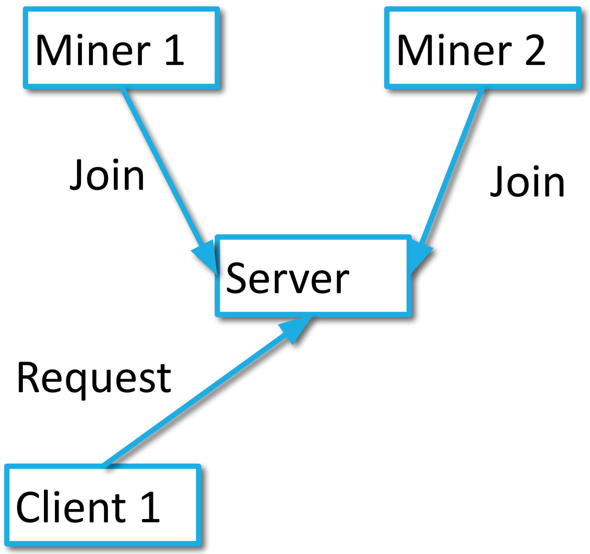
P1 Part B

- Implement a distributed mining infrastructure on top of the LSP you develop for part A

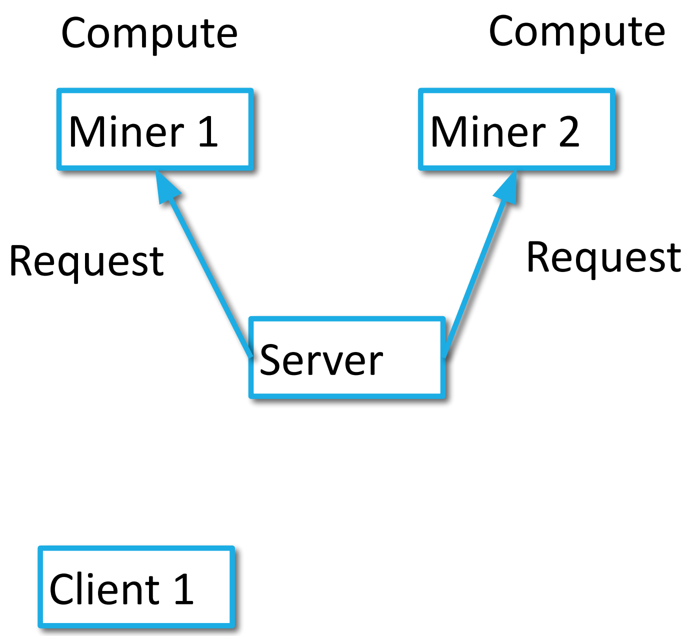
Mining

- Given
  - Message M
  - Unsigned integer N
- Find n such that
  - $0 \leq n \leq N$
  - $\mathrm{h}ash(M, n) \leq \mathrm{h}ash(M, n') \; \forall \; 0 \leq n' \leq N$
- Run brute-force search to enumerate all possible scenarios across multiple machines

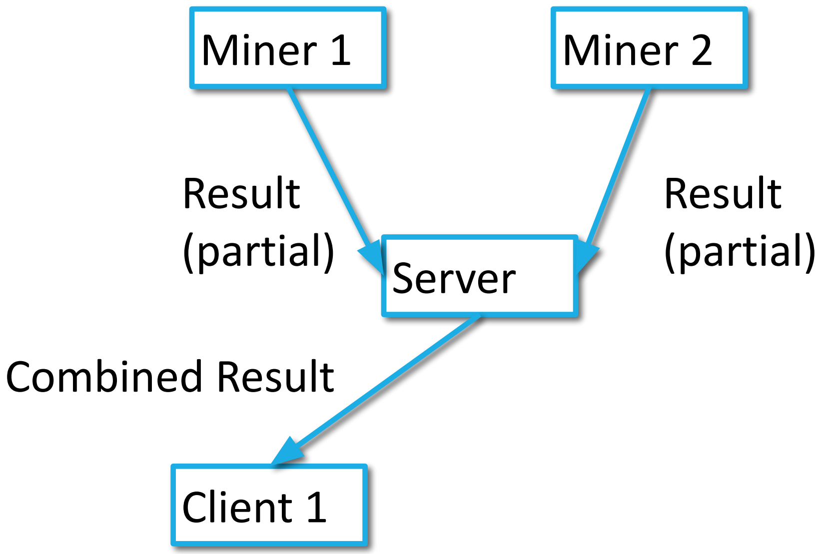
Architecture



Architecture



Architecture



Handling Failures

- When a miner loses contact with the server it should shut itself down.
- When a request client loses contact with the server, it should **print Disconnected** to standard output and exit.

Handling Failures

- When the server loses contact with a miner, **it should reassign any job that the worker was handling to a different worker**. If there are no available miners left, the server should wait for a new miner to join before reassigning the old miner's job.
- When the server loses contact with a request client, **it should cease working on any requests being done on behalf of the client** (you need not forcibly terminate a job on a miner—just wait for it to complete and ignore its results).

Scheduler

- You should design the server so that the processing time per request is proportionate to the request size.
- This means that the server should respond quickly to small requests and can respond slowly to larger ones.
  - If the server gets a very large request, and a small one right afterward, your design should ensure the small request completes fast and does not wait for the larger one to finish first!

Scheduler (continue)

- On the other hand requests should be given some priority based on when they are received.
- We have purposefully not given you the design of the scheduler in the handout. You should brainstorm to find scheduling techniques that satisfy this requirement.
- **Your code should clearly document how you implemented scheduling in your server. -> the documentation will be considered in the style grading.**

Questions I know you will ask

- Are there hidden tests in part B?
  - Yes, stests are not provided to you, we also have hidden mtest and ctest.
- Does passing the public test mean we can pass the hidden tests?
  - No, not even close
- If we fail the hidden tests on Gradescope, can we get useful hints on what is wrong?
  - Barely. Don’t waste 15 submission attempts on debugging. So write good tests!!
  - Also make use of the miner, client, and server binary
- How can we split the work?
  - It depends.
  - Total LOC of bitcoin implementation in our reference solution: ~360
  - Total LOC in hidden tests: >700
  - **Write good tests from day 1!!! or at least try to test your implementation.**

APPENDIX: P1
Post-checkpoint Unit Tests

- TestSendReceive*
- TestRobust*
- TestWindow*
- TestExpBackOff*
- TestMaxUnackedMessages*
- TestServerSlowStart*
- TestServerClose*
- TestServerCloseConns*
- TestClientClose*
- TestServerFastClose*
- TestServerToClient*
- TestClientToServer*
- TestRoundTrip*
- TestVariableLengthMsgServer
- TestVariableLengthMsgClient
- TestCorruptedMsgServer
- TestCorruptedMsgClient
- TestCAck*. TODO(natre)

TestSendReceive*

- TestSendReceive* test that all messages sent from one side are received by the other (without relying on epochs to resend any messages)
- Window size = 1
- Otherwise similar to TestBasic*
  (Do not need window)

TestRobust*

- TestRobust* test robustness by inserting random delays in between client/server reads or writes, and by increasing the packet loss to up to 20%
- Window size up to 10
- Client count up to 5
- (Need epoch, does not intentionally test on window implementation)

TestWindow*

- TestWindow1~3 test the case that …
  - The sliding window has reached its maximum capacity.
- TestWindow4~6 test the case that …
  - Messages are returned by Read in the order they were sent (i.e. in order of their sequence numbers).
  - If messages 1-5 are dropped and messages 6-10 are received, then the latter 5 should not be returned by Read until the first 5 are received.
  - **Need Epochs implemented to work!**

TestMaxUnackedMessages*

- TestMU1~3 test the case that …
  - The maxUnacked has reached its maximum capacity.
- TestMU4~6 test the case that …
  - Messages are returned by Read in the order they were sent (i.e. in order of their sequence numbers).
  - M =10, W = 20
    - If messages 1-5 are unacked, then 6-10 are acked, then 10-15 should be sent, 15-20 should not.
  - **Need Epochs implemented to work!**

TestExpBackOff*

- TestExpBackOff* test that the number of messages sent due to exponential back-off falls within a reasonable range
- We sniff messages sent through lspnet
- Up to 10 clients
- Up to 15 messages
- *The test is not comprehensive (we set up a range of acceptable answers, which is not "precise"), so design the epoch part carefully.*

TestServerSlowStart*

- TestServerSlowStart* test that a client is able to connect to a slow-starting server
  - if the server starts a few epochs later than a client, the presence of epoch events should ensure that the connection is eventually established
- Up to 3 clients
- Timeout after 5 epochs

TestServerClose* TestServerCloseConns* TestClientClose*

- Check that the client/server Close methods work correctly
- Pending messages should be returned by Read and pending messages should be written and acknowledged by the other side before Close returns
- CloseConn should return immediately without blocking
- Check that no extra messages are received on the client/server
- After close is called, Read() and Write() should return an error or block indefinitely.

TestServerFastClose*

- Streaming messages in large batches and the network is toggled on/off (i.e. drop percent is set to either 0% or 100%) throughout.

TestServerFastClose* (Cont.)

- Test procedure at high level (the test case log also records this procedure)

1. Wait for all servers and clients to be ready
2. Shut down network
3. Client application starts writing…
4. Turn on network and delay (server-client communication resumed)
5. Shut down network
6. Server application starts reading…
7. Server application starts writing…
8. Start closing server (pending messages need to be ready for send)
9. Turn on network and delay (server-client communication resumed)
10. Shut down network
11. Client application starts reading…
12. Start closing client

TestServerToClient*
TestClientToServer*
TestRoundTrip*

- Variants of

  TestServerFastClose*

- For more details, read

  lsp4_test.master()

TestVariableLengthMsgServer TestVariableLengthMsgClient

- Check that server/client…
  - Can read normal length message
  - Truncates long messages
  - Doesn't read short messages

TestCorruptedMsgServer TestCorruptedMsgClient

- Check that server/client…
  - Drop **Data** messages whose calculated and recorded **checksums** don't match