

Assignment 3 README

Contents

[Contents](#)

[ivy](#) 

[How to run the code](#)

[Setting up the terminals.](#)

[How to reboot](#)

[What happens when you reboot?](#)

[Useful commands for the clients \(AKA 1,2,3,4...\)](#)

[Useful commands for the server \(AKA 1,2,...\)](#)

[Test results](#)

[Scenario 1: No failure](#)

[Scenario 2](#)

[Invariant 1: CM fails](#)

[Invariant 2: CM fails & reboots once](#)

[Scenario 3: Main CM fails and reboots multiple times](#)

[Scenario 4: Main CM and Backup fail multiple times](#)

[Inference](#)

[Is the fault tolerant version still sequentially consistent?](#)

[Some interesting design details](#)

[How do reads happen?](#)


[How do writes happen?](#)

ivy 

Implementation of the Integrated shared Virtual memory at Yale.

ivy command - github.com/fauzخان/ivy - Go Packages

This code is intended to handle the death of the central manager/ backup manager.

 <https://pkg.go.dev/github.com/fauzخان/ivy@v1.0.0#section-readme>

How to run the code

Setting up the terminals.

Each server, backup, and client will run on a separate terminal, or even separate systems on the same network. The reboots will work, provided the local DHCP doesn't allocate a new address for some arbitrary reason. This typically shouldn't happen if you didn't close your laptop between the crash and failure, as DHCPs only reassign IPs after a some preset timeout.

The first server you spin up will be the central manager.

```
go build && ./ivy -cm
```

The second server you spin up will be the backup, provided you pass in the IP of the first server to it. How do you find out the IP of the first server? Easy, just look at the logs on the first terminal:

```
There is no backup!
2023/12/05 21:02:34 Central manager!
My IP is:172.23.129.167:35677
Central manager is running at IP address:172.23.129.167:35677 <-- Look for this line, copy this onto your clipboard
```

Once you have copied the IP address of the central manager, you may now proceed to create:

1. Backup central manager using `go build && ./ivy -cm <IP ADDRESS THAT YOU JUST COPIED>`
2. As many clients as you want on separate terminals using `go build && ./ivy -u <IP ADDRESS THAT YOU JUST COPIED>`

Great! Once you have this setup, we can go over the commands to do the following:

1. Start the never ending cycle of randomly issuing read and write requests. You may stop this behaviour by adjusting the code to do it only once in `main.go`
2. Kill any server. Central or Backup. This can easily be done by pressing ctrl+C in the central manager/ backup manager's terminal.
3. View some important log information like Record cache at the server, or the Page Cache at the clients.

How to reboot

The code is originally designed to assign a random, unused port number for the first server. This is to save you from the agony of having to choose a port number 🤖🤖🤖

So in order to reboot at the same port number, simply issue the following command:

```
go build && ./ivy -r <IP ADDRESS OF THE SERVER THAT JUST WENT DOWN>
```

What happens when you reboot?

When the central manager (not backup) reboots, it will receive a PING message from the backup, and it will tell all the clients to contact it from here onwards.

When the backup manager reboots, it will simply wait for the central manager to die in order to do anything meaningful 😊 However, once the central manager dies (process exits), the backup will keep nagging (sending ping message) the central manager to see if it is back alive. Once it is back alive, the backup manager will send over all the details that it had updated when the central was dead, to the central manager.

Useful commands for the clients (AKA 1,2,3,4...)

"1" will print the clientlist

"2" will print the IP of the server you are currently contacting. This may be the central server, or the backup server.

"3" will start the never ending cycle of randomly issuing read or write requests. All read or write requests are made on randomized pages 1,2, or

3. You may increase the number of pages by changing the following line in `client.ReadRequest()` and `client.WriteRequest()`:

```
pageid := rand.Intn(3) // you may increase the number of pages
```

A read or a write request will be randomly issued once every 15 seconds. This is to prevent the logs from going 🤖🤖🤖

Only way you stop this never ending cycle is by killing the client.

Useful commands for the server (AKA 1,2,...)

"1" Will print the clientlist

"2" will print the records that you have in your possession. This is the list of pages, and their respective owners, and the list of IPs that have a copy of it.

Test results

For all the tests conducted, the following basis was maintained:

1. Two results were recorded -
 - a. the node that took the highest amount of time.
 - b. the node that took the lowest amount of time

For each scenario, 5 results were recorded for invariants 1a and 1b, and then averaged.

2. We don't consider cases where clients are able to retrieve pages from the cache - as this always yield lightning fast performance ~ 10ms.
3. Killing and rebooting of main central manager is to be done manually, by pressing ctrl+c, then passing the reboot command described in the previous section.

All results displayed are in ms. A sample of run with 10 clients, and two servers (one backup, one main) has been shown below:

Scenario 1: No failure

Easiest to test of them all. No need to kill anything, just spawn ten terminals, and one server and keep them running.

Highest time	14248	14245	14235	13498	14248
Lowest time	751	751	751	750	750

High time average: 14094.8

Low time average: 750.6

Average: 7422.7

Scenario 2

Spawn 2 servers- one central, and one backup, using the method described in the [Usage](#) section. Then press **ctrl+C** in the central managers terminal. Once you record the results, just reboot it using the command `go build && ./ivy -r <IP ADDRESS OF THE CENTRAL MANAGER>`

Invariant 1: CM fails

Highest time	14240	13461	14247	14240	14247
Lowest time	751	751	750	750	751

High time average: 14087

Low time average: 750.6

Average: 7418.8

Invariant 2: CM fails & reboots once

Highest time	14259	14248	13497	14239	14229
Lowest time	751	750	751	750	751

High time average: 14094.4

Low time average: 750.6

Average: 7422.5

Scenario 3: Main CM fails and reboots multiple times

You can run this scenario in the same way as the previous one, but trigger failure and reboot multiple times, manually.

Two servers, main fails, reboots multiple times	14247	14237	13496	13494	14250	14239	14238
	752	761	750	756	760	754	750

High time average: 14049.63

Low time average: 754.25

Average: 7401.938

Scenario 4: Main CM and Backup fail multiple times

For this scenario, you may sequentially perform the following:

- kill the main CM, then reboot main CM using main CM IP address
- kill backup CM, then reboot backup CM using backup CM IP address
- ... continue doing the above steps and record the observations

Two servers, main and backup fail, multiple times and reboot	13498	13497	14243	13500	13497	14247
	751	754	757	750	751	750

High time average: 13747

Low time average: 752.167

Average: 7249.583

Inference

The nature of this implementation is such that the presence or absence of the backup manager does not make a difference on the performance, as before either dies, they transfer their entire metadata to the other node. So it doesn't matter who crashes at which point in time, one is always able to pick up where the other left off, and continue the operation.

As a result, we see that the average timings remain consistent in almost any case.

Is the fault tolerant version still sequentially consistent?

When discussing sequential consistency, we need to take care of the following: *there needs to be some total ordering of reads and writes among all the clients.*

In case of reads, clients check their local cache. If the copy exists, it means no other node has written to it - we can be sure of this because, if some node had written to it, then the copy would not have existed in it's cache. Say some node n1 had requested to write to some page whose copy exists in another node n2; then one of two things might happen:

1. Invalidate message caused by n1's request may reach n2 after n2's local read request. In this case, n2 will read the last available copy that came before n1's write. Here sequential consistency is not broken, as the total ordering is still preserved.
2. Invalidate message caused by n1's request may reach n2 before n2's local read request. In this case, n2 will not have a copy of the page that it wants to read, therefore, it will forward the request to the central manager, who will then issue a `READ_FORWARD` message to the "owner". The owner of this page is guaranteed to have been changed, as n2's copy has already been invalidated. The implementation stipulates that the node with latest write request for that page is assigned to be the owner of that page. Refer to the code below, and the comments

```
// Write requests received by the client are handled by this function
func (central *Central) writeHandler(IP string, PageId int) bool {
    central.invalidateSender(PageId) // this blocks the central manager from sending WRITE_FORWARD
    for _, record := range central.Records {
        if record.PageId == PageId { // find the owner of that record, and send write forward
            go central.CallRPC(message.Message{Type: WRITE_FORWARD, PageId: PageId, From: IP}, record.OwnerIP)
            record.OwnerIP = IP // the owner id is updated as soon as this happens.
            return false
        }
    }
}
```

```

    return true
}

```

From the above snippet, we can deduce that if the copy I have is invalidated, then by the time my request reaches the central manager, it is guaranteed that the owner IP has been updated at the server. Therefore, the server will forward the read request to the owner, who has written something to it.

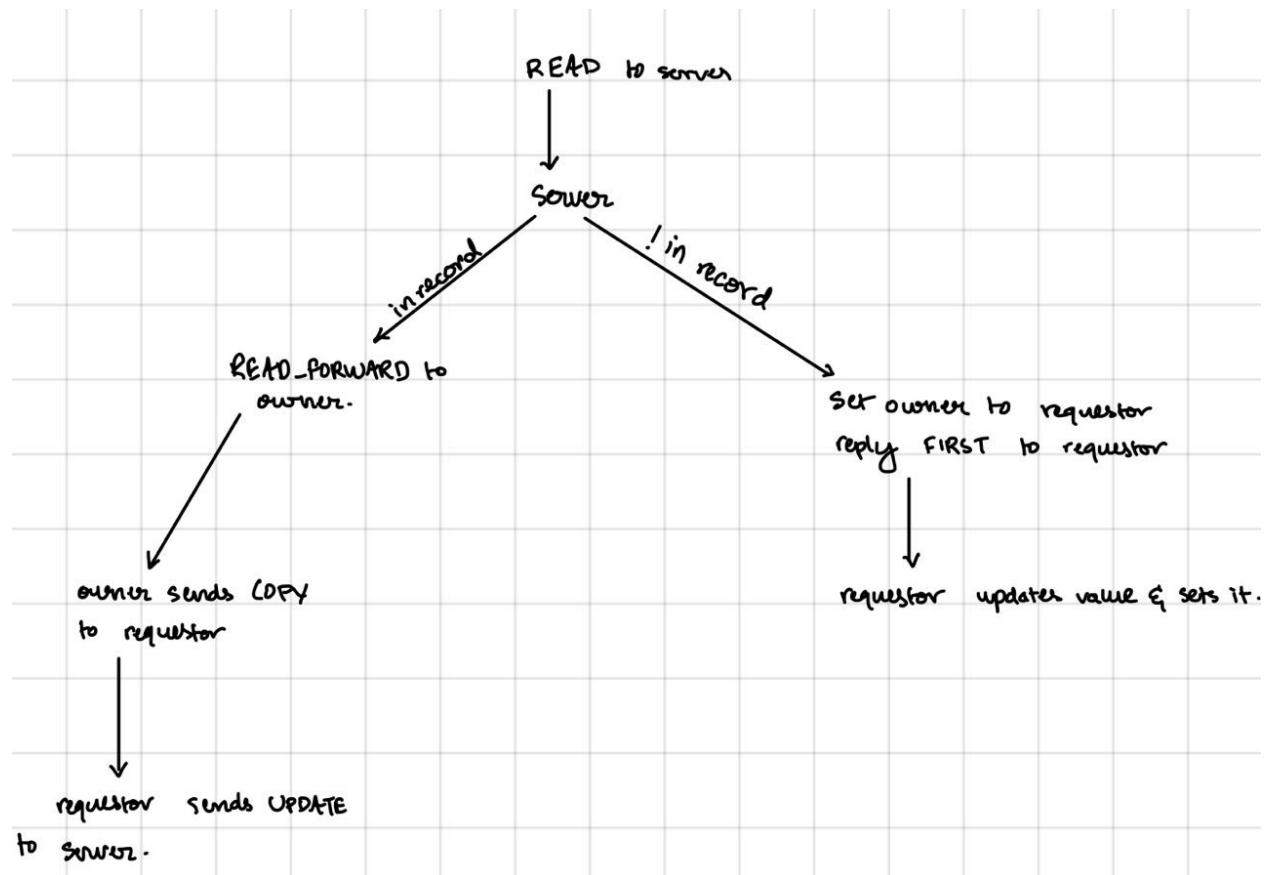
Therefore, it is guaranteed that the clients always read the latest copy, even if a write request has been raised elsewhere.

In case of writes, the sequential consistency is easy to deduce. This is because the central manager maintains a queue of write operations, and only carries out the operation for one write request at a time. Therefore, it is guaranteed that the total ordering of the writes is dictated by the ordering of writes received by the central manager. Any clients requesting to read in between the writes would have their requests forwarded to the client who is currently writing because they are set as the owner of the page currently, as shown in the code snippet above.

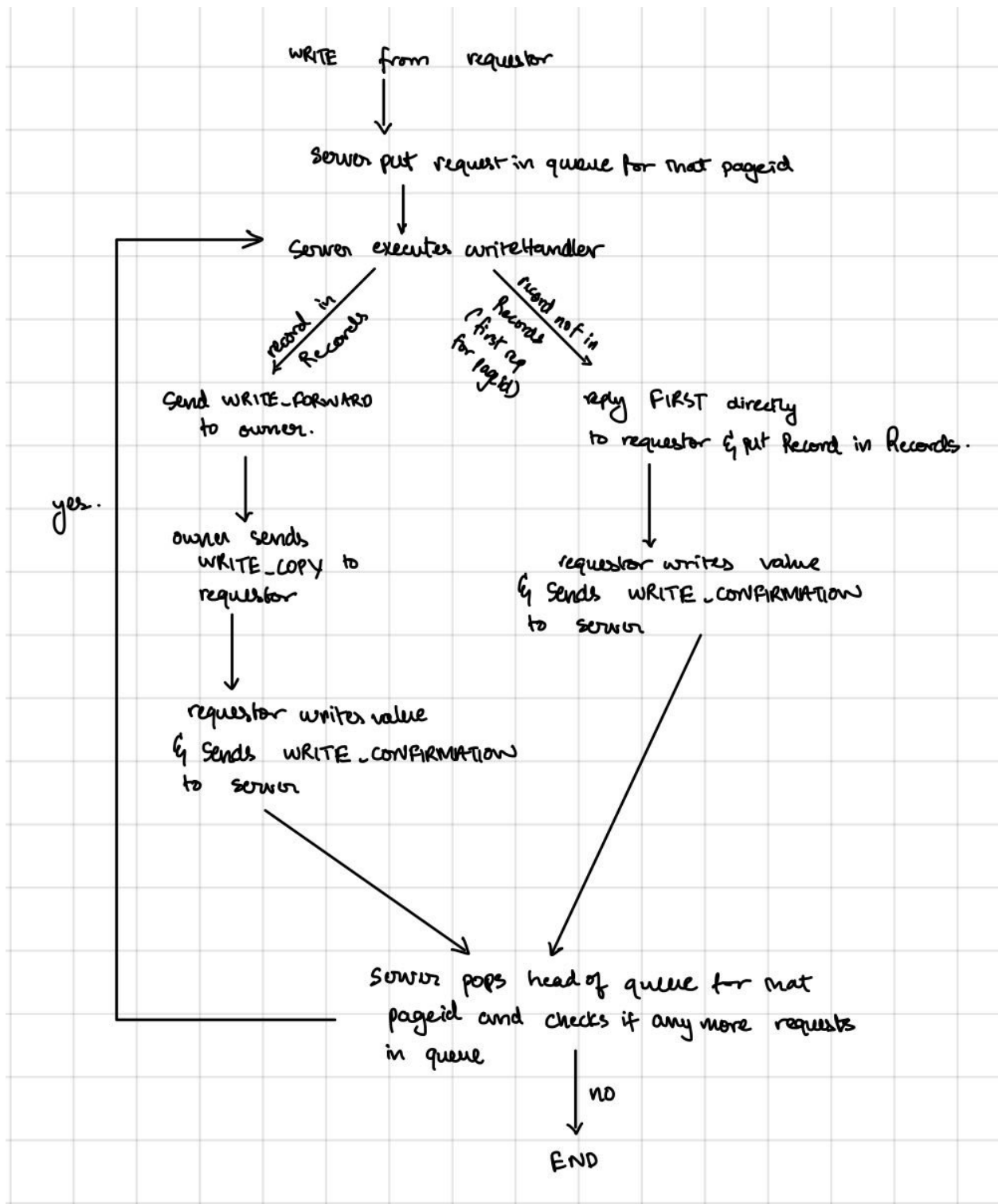
Some interesting design details

Details that needed documentation to capture the intricacy of the design.

How do reads happen?



How do writes happen?



Writes maintain a queue at the server. As of now, there is no inherent benefit to having a separate WRITE access mode at the client. This will be useful in scenarios where the client performs multiple computations on the page before falling back to write mode. However, the code only handles a single variable update for any page