პროექტი კომპილირდება make-ით. პროგრამები ეშვება როგორც პირობაში მითითებული ინსტრუქციით.
იმისთვის რომ კოდი დაკომპილირდეს საჭიროა დაყენებული იყოს შემდეგი package-ები:

$ sudo apt install pkg-config libfuse-dev
$ sudo apt-get install libpcap-dev libssl-dev

სულ მაქვს 3 ფაილი: net_raid_client.c, net_raid_server.c და shared.h
მთელი ლოგიკა სრულდება net_raid_client.c-ში, სერვერი უბრალოდ კლიენტის გამოგზავნილ ბრძანებებს ასრულებს. shared.h-ში არის ისეთი include-ები, define-ები, სტრუქტურები და ა.შ., რაც ორივეს სჭირდება.

კონფიგურაციის ფაილი
კონფიგურაციის ფაილს ვხსნი სახელის მიხედვით და ხაზ-ხაზად ვკითხულობ, თითოეულ ხაზს ვყოფ strtok-ით ":"-ის მიხედვით. თუ ცარიელი ხაზი შემხვდა ანუ ახალ სეგმენტზე გადავედით. პირველი სეგმენტი არის ფაილური სისტემის ზოგადი მახასიათებლები და მის შესანახად მაქვს სტრუქტურა client:

```
struct client {
   char* errorlog;  - ლოგ ფაილის მისამართი
   uint64_t cache_size; - ქეშის ზომა (რიცხვი და უნდა მოყვებოდეს K, M ან G, კილობაიტის, მეგაბაიტის ან გიგაბაიტის აღსანიშნად, შესაბამისად)
   char* cache_replacement; - გამოძევების ალგორითმი (თუმცა ეს სულ lru არის)
   size_t timeout; - სერვერის თაიმაუთი
};
```

შემდეგი სეგმენტები არის თითოეული storage disk-ის მახასიათებლები და ინახება disk სტრუქტურაში (სხვა საჭირო ინფორმაციასთან ერთად):

```
struct disk {
   char* diskname; - დისკის სახელი
   char* mountpoint; - კლიენტის მაუნთფოინთის მისამართი
   size_t raid; - რეიდი (1 ან 5)
   char** servers; - სერვერების მასივი (მხოლოდ ip-ები)
   int* ports; - პორტების მასივი
   int* sfds; - სერვერებთან დამაკავშირებელი სოკეტების file descriptor-ები
   char* hotswap; - ჰოტსვაპის სერვერის ip
   int hotswap_port; - ჰოტსვაპის პორტი
   size_t num_servers; - სერვერების რაოდენობა
   size_t servers_size; - სერვერების მასივის ზომა
```

char** fdpaths; - raid1-ის შემთხვევაში მეორე სერვერზე გახსნილი ფაილების file descriptor-ების მასივი
size_t fdpaths_size; - fdpaths მასივის ზომა
pthread_mutex_t* mutexes; - სერვერებთან ურთიერთობის ატომურობისთვის მუტექსების მასივი
struct cache* cache; - ქეში
};

ამ disk სტრუქტურის მასივი მაქვს, რადგან შეიძლება ბევრი დისკი იყოს.

ხაზებს მხოლოდ თითო space-ს ვაცლი აქეთ-იქიდან(თუ აქვს საერთოდ space-ები), რადგან პირობაში ეგეთი კონფიგურაციის ფაილის მაგალითია, ბევრი space-ები რომ ჰქონდეს მაინც თითოს მოვაცლი.

კონფიგურაციის ფაილის დაპარსვის შემდეგ გადავუყვები ყველა დისკს და თითოეულისთვის ვუშვებ ახალ პროცესს, რადგან თითოსთვის თავისი fuse უნდა გაეშვას. შემდეგ თითოეულ პროცესში გადავუყვები შესაბამისი დისკის სერვერებს და პორტებს და ვუკავშირდები და file descriptor-ს ვინახავ disk სტრუქტურის sfds მასივში. თითოეული პროცესი exit-დება შესაბამისი fuse_main-ის დაბრუნებული სტატუსით. ვინახავ თითოეული ამ შვილობილი პროცესის id-ებს და main სრედი ელოდება ყველა შვილობილს, შემდეგ კი ასუფთავებს malloc-ებით გამოყოფილ მეხსიერებებს.

RAID 1 (mirroring)
გადავტვირთე fuse-ის შემდეგი syscall-ები:

```
struct fuse_operations nr_operations = {
    .getattr = nr_getattr,
    .mknod = nr_mknod,
    .open = nr_open,
    .release = nr_release,
    .unlink = nr_unlink,
    .mkdir = nr_mkdir,
    .opendir = nr_opendir,
    .readdir = nr_readdir,
    .rmdir = nr_rmdir,
    .releasedir = nr_releasedir,
    .rename = nr_rename,
    .read = nr_read,
    .write = nr_write,
    .truncate = nr_truncate,
    .utime = nr_utime,
    .access = nr_access,
};
```

ფიუზს private_data-დ ვატან შესაბამის დისკ სტრუქტურას, იქიდან საჭირო ინფორმაციის ამოსაღებად, მაგალითად სოკეტების fd-ების.

თითოეულ გადატვირთულ syscall-ში კლიენტი უგზავნის სერვერებს შესაბამისი მოქმედებისთვის საჭირო ინფორმაციას, სერვერი ასრულებს syscall-ს და კლიენტს უბრუნებს სტატუსს და სხვა დატასაც, თუ საჭიროა. რადგან სერვერმა არ იცის რომელი syscall გამოუგზავნის დატას, კლიენტიდან სერვერისთვის გასაგზავნად მაქვს საერთო info სტრუქტურა:

```
struct info {
   enum operation_id id; - რომელი syscall-ია გამოძახებული
   char path[PATH_MAX]; - ფაილის მისამართი mountpoint-ისადმი relative (სერვერი ამ
მისამართს თავისი storage-ის მისამართზე მიადგამს და მიიღებს მთლიან მისამართს)
   mode_t mode; - რა ნებართვებით იხსნება/იქმნება ფაილი
   dev_t dev; - mknod-ისთვის საჭირო ატრიბუტები
   int flags; - ფაილის გახსნის ფლაგები
   char newpath[PATH_MAX]; - ფაილის ახალი მისამართი rename-ისთვის (ესეც
mountpoint-ისადმი relative)
   int fd; - read/write/release-ში საჭირო file descriptor
   DIR* dir; - readdir/releasedir-ში საჭირო დირექტორიაზე მიმთითებელი
   size_t size; - read/write-ში წასაკითხი/ჩასაწერი ბაიტების რაოდენობა
   off_t offset; - read/write-ში ჩაწერა/წაკითხვა საიდანაც უნდა დაიწყოს (ფაილის დასაწყისიდან
ათვლილი)
   off_t newsize; - truncate-ში ფაილის ახალი ზომა (რაც უნდა მივიღოთ)
   int mask; - access-მა რა permission-ებიც უნდა შეამოწმოს
   int mode_change; - სერვერის მხარეს read/write-ის დროს მინდა თუ არა, რომ სერვერმა
გადაცემული fd-ს გამოყენების მაგივრად თავიდან გახსნას ეს ფაილი საჭირო permission-ებით
};
```

სერვერი მუდმივად ელოდება კლიენტისგან info სტრუქტურის მიღებას, როდესაც მიიღებს, ამოწმებს id-ს და იძახებს შესაბამის მეთოდს.

```
enum operation_id {
   zero,
   _getattr,
   _mknod,
   _open,
   _release,
   _unlink,
```

```
    _mkdir,
    _opendir,
    _readdir,
    _rmdir,
    _releasedir,
    _rename,
    _read,
    _write,
    _truncate,
    _utime,
    _access,
    _check
};
```

0 ნომრის მქონეს არ ვიყენებ საერთოდ, რადგან როცა შეცდომა ხდება მაშინაც 0-ს უგზავნის. სერვერი კლიენტს ჯერ უგზავნის რამე დამატებითი დატა თუ აქვს გასაგზავნი, ხოლო ბოლოს სტატუსს. opendir და getattr შემდეგი სტრუქტურების საშუალებით უგზავნიან თავიანთ დატას:

```
struct getattr_t {
    struct stat statbuf;
    int status;
};

struct opendir_t{
    DIR* dir;
    int status;
};
```

თუმცა ასე ერთი სტრუქტურით გაგზავნის საჭიროება აღარ იყო და სხვა syscall-ებზე აღარ გავაკეთე ასე და ცალ-ცალკე ვგზავნი.

getattr, opendir, readdir, releasedir, read და access syscall-ები მეორე სერვერზე მხოლოდ მაშინ სრულდება პირველი სერვერი თუ გათიშული იყო (თაიმაუთი რომც არ იყოს ჯერ გასული მაინც) ან ცუდი პასუხი დააბრუნა პირველმა სერვერმა.

open, release, unlink, mkdir, rmdir, rename და utime ორივე სერვერზე სრულდება (ცხადია, გათიშული თუა რომელიმე, იმაზე ვერ შესრულდება).

mknod, write და truncate მეორე სერვერზე მხოლოდ მაშინ სრულდება, თუ პირველ სერვერზე წარმატებით შესრულდა ან პირველი სერვერი გათიშული იყო იმ მომენტში.

ესენი იმას უზრუნველყოფს რომ მარტო ერთი სერვერიც რომ დამრჩეს, ან ცოტა ხნით გაითიშოს მეორე, დარჩენილ სერვერზე სწორად ხდებოდეს ყველაფერი.

Stable storage-ის უზრუნველსაყოფად, თითოეულ ფაილს extended attribute-ად ვუსეტავ content-ის md5 ჰეშს, რომელსაც openssl-ის საშუალებით ვაგენერირებ. Mknod-ისა და ყოველი write-ის დროს თავიდან ვითვლი ფაილის ჰეშს და ხელახლა ვუსეტავ ატრიბუტად. ყოველი open-ის დროს სერვერი ითვლის გახსნილი ფაილის ჰეშს და ატრიბუტად შენახულ ჰეშთან ერთად უგზავნის კლიენტს. თუ პირველი სერვერის ახალი დათვლილი ჰეში არ დაემთხვა ატრიბუტად შენახულს, პირველ სერვერზე გაფუჭებულია ფაილი და მეორედან გადმოვწერთ, თუ მეორეს ახალი ჰეში არ დაემთხვა ატრიბუტს, მაშინ პირიქით, პირველიდან გადმოვწერთ მეორეზე, ხოლო თუ არცერთ სერვერზე არ დაემთხვა ახალი ჰეშები შენახულებს, მაშინ გაფუჭებულია ორივეგან და ორივე სერვერზე წავშლით ამ ფაილს; ასევე პირველიდანგადმოვწერთ მეორეზე, თუ თითოეული სერვერის დაბრუნებული ძველი და ახალი ჰეშები ემთხვევა ერთმანეთს, თუმცა პირველ სერვერზე ჰეში არ ემთხვევა მეორე სერვერზე ჰეშს. ფაილის გადაწერისთვის კლიენტის მხრიდან ვიძახებ სერვერების read/write-ებს. მაგალითად, თუ პირველი სერვერიდან ვწერ მეორე სერვერზე, ვიძახებ პირველ სერვერზე 4096-იანი chunk-ის read-ს, რაც დამიბრუნდება იმის მეორე სერვერზე ჩასაწერად ვიძახებ მის write-ს შესაბამის ზომაზე. რადგან ჩაწერები 4096-იანი chunk-ებით ხდება fuse-ში, ამ შემთხვევაშიც ასე დავწერე. ფაილების ერთი სერვერიდან მეორეზე გადასაწერად მჭირდება struct info-ში int mode_change, რადგან ეს ფაილი კი გახსნილი იყო აქამდე და მაქვს file descriptor, მაგრამ შეიძლება ისეა გახსნილი, რომ მხოლოდ წაკითხვის ნებართვა აქვს და მაშინ ჩაწერა რომ მომინდება ამ ფაილში (მეორე სერვერიდან ფაილის გადმოსაწერად) შეცდომა მოხდება. თუ mode_change 1-ია write ხსნის “wb” mode-ით, ეს ნიშნავს, რომ რაც მანამდე ეწერა წაშალოს, თუ 2-ია, “ab” ფლაგით ხსნის, რადგან პირველი ჩაწერის შემდეგ უბრალოდ უნდა მიადგას დატა წინა ჩაწერილებს, ხოლო თუ 0-ია არ ხსნის თავიდან და გადმოცემულ fd-ს იყენებს. read-ის შემთხვევაში 0 თუა არ ხსნის თავიდან, ხოლო 0 თუ არაა, ხსნის “rb” mode-ით. თუ თავიდან გავახსნევინე ფაილი, syscall-ის ბოლოს იხურება. თუ open-ისას ერთ-ერთ სერვერზე საერთოდ აღარ არსებობს რაღაც ფაილი და მეორეზე არსებობს, ანუ ერთმა ვერ გახსნა საერთოდ ამ მიზეზით და მეორემ გახსნა, მაშინ ჯერ mknod-ს ვიძახებ იმისთვის, რომელზეც წაშლილია, შემდეგ ვწერ ფაილის content-ს და ბოლოს ვხსნი ამ ახალ შექმნილ ფაილს და fd-ს ვუბრუნებ open-ს, რომ შეინახოს შემდგომი გამოყენებისთვის.

მაღალმდგრადობა

fuse_init-ში ვუშვებ 2-2 სრედს (რამდენი სერვერიცაა იმდენს ვუშვებ, მაგრამ raid5 არ მიწერია მაინც, ამიტომ 2 გამოდის). fuse_init-ში იმიტომ, რომ ისე თიშავს fuse_main მისი პროცესის სრედებს.

თითოეულს სრედს დატად გადაეცემა server_checker_data სტრუქტურა:

```
struct server_checker_data {
   size_t ind; - რომელი სერვერია (პორტიც და sfd-ც შესაბამის ინდექსებზე წერია)
   time_t** last_server_checks; - მასივი სადაც ინახება თითოეულმა სერვერმა ბოლოს როდის
დააბრუნა check-ზე პასუხი
   pthread_mutex_t* mutex; - შესაბამისი სოკეტის მუტექსი
   struct disk* d; - შესაბამისი დისკი
```

};

თითო სრედი აკონტროლებს თითო სერვერს, ანუ უგზავნის check-ებს, სპეციალური id არის ამისთვის და სერვერს რომ ამ id-ის მქონე info მიუვა, უბრალოდ 0-ს აბრუნებს. თუ ამ სრედმა წარმატებით მიიღო ეს 0-იანი, ჩაწერს ახლანდელ დროს დროების მასივში, რომელშიც ინახება თითოეული სერვერისთვის ბოლო წარმატებული კავშირი სერვერთან სრედის მიერ. თუ ვერ მიიღო სერვერისგან პასუხი, ანუ გაითიშა სერვერი და ვცდილობთ თავიდან connect-ს, თუ ვერ დავუკავშირდით ვამოწმებ თაიმაუთის დრო ჰო არ გასულა ბოლო წარმატებული კავშირიდან, თუ არ გასულა, ვაგრძელებ ისევ გათიშული თუა დაკავშირების მცდელობას, თუ არაა check-ის გაგზავნას; თუ გავიდა, ვნახულობ ჰოტსვაპი უკვე გამოყენებული მაქვს თუ არა, თუ არ მაქვს, ვანაცვლებ ამ სერვერს ჰოტსვაპით და მთლიანი ფაილური სისტემა გადმომაქვს ცოცხალი სერვერიდან ჰოტსვაპზე რეკურსიულად (path-ს გადავცემ ფუნქციას და თუ ფაილია პირდაპირ მთლიანი ფაილი გადმომაქვს მეორე სერვერიდან, როგორც stable storage-ის დროს, ხოლო თუ დირექტორიაა, ვქმნი ამ დირექტორიას და რეკურსიულად ვიძახებ ამ ფუნქციას ყველა მისი შიდა ფაილისა თუ დირექტორიისთვის). თუ უკვე გამოყენებულია ჰოტსვაპი ანუ მოკვდა ეს სერვერი საბოლოოდ, ვინახავ მასივში, რომ მოკვდა და სრედს ვთიშავ. თუ კავშირი მანამდე აღდგა სანამ თაიმაუთი გავიდოდა, უბრალოდ ძველ sfd-ს ჩავანაცვლებ ახლით sfds მასივში. ამ დროს ფაილში ცვლილებები თუ მოხდა ცოცხალ სერვერზე, open-ის დროს აღდგება გაცოცხლებულზეც. რადგან ეს სრედები fuse-ის ფუნქციები პარალელურადაა გაშვებული, ერთმანეთს რომ ხელი არ შეუშალონ და ერთმანეთის დატა არ იკითხონ სოკეტებიდან, თითოეული სოკეტისთვის მაქვს მუტექსი, რომელიც ილოქება ამ სოკეტზე პირველი send-ის წინ და ლოქი ეხსნება ბოლო recv-ის შემდეგ თითოეულ fuse-ის გადატვირთულ syscall-ში და ასევე ამ სრედების ჩეკების გაგზავნებში. ვედმეტ ნაწილებს ტყუილად არ ვლოქავ. ჰოტსვაპზე გადაწერა მთლიანად იმ სერვერის ლოქშია რომელსაც ვანაცვლებთ, რადგან ამ დროს ჰოტსვაპზე წაკითხვა ან რაიმე სხვა ოპერაცია არ მოხდეს, სანამ მთლიანი დატა არ ჩაიწერება. ჩანაცვლებისას ჰოტსვაპის sfd ჩაიწერება მის მიერ ჩანაცვლებული სერვერის sfd-ს ადგილას მასივში, ისევე როგორც სერვერი და პორტი.

ლოგირება

ვაკეთებ ყველა მნიშვნელოვანი მოვლენის ლოგირებას, რაც სისტემის წარმადობაზე მოქმედებს. თითოეული syscall-ის გამოძახებას არ ვლოგავ, რადგან ძალიან ბევრი იქნება და მაგათში ისინიც აღარ გამოჩნდება, რაც მართლა წარმადობისთვის მნიშვნელოვანია. ამიტომ ვლოგავ თვითონ სერვერებს რაც ეხება ისეთ მოვლენებს: სერვერებთან დაკავშირებას, სერვერს თუ ვეღარ ვუკავშირდები/არ მპასუხობს, სერვერთან კავშირის აღდგენას, სერვერის დაკარგვას, ჰოტსვაპთან დაკავშირებას და ჰოტსვაპზე ყველაფრის გადაწერას. ჰოტსვაპით ჩანაცვლების შემდეგ მასზეც ყველაფერი ჩვეულებრივ სერვერივით ილოგება.

ქეშირება

ქეშის თითოეული entry წარმოდგენილია cache_entry სტრუქტურის სახით:

```
struct cache_entry {
    char* path; - ფაილის მისამართი
    off_t offset; - ფაილის დასაწყისიდან დაშორება
    size_t size; - ბაიტების რაოდენობა
    char* data; - ფაილის content: offset დაშორებიდან დაწყებული size ცალი ბაიტი
    time_t last_use; - ქეშიდან ამ entry-ს წაკითხვის ბოლო დრო
};
```

თავად ქეში წარმოდგენილია cache სტრუქტურის სახით:

```
struct cache {
    struct cache_entry* entries; - entry-ების მასივი
    size_t log_len; - მასივში რამდენი entry-ა
    size_t alloc_len; - რამდენი entry-ს ადგილია გამოყოფილი მასივში
    size_t cache_size; - რამდენ ბაიტს იკავებს ქეშში შენახული ფაილების ნაწილები ჯამში
};
```

cache სტრუქტურა თითოეულ დისკს თავისი აქვს.
read-ის დროს გადავუყვები ქეშს და ვნახულობ ამ ფაილის ამ offset-ზე დაწყებული დატა თუ მაქვს, რომლის ზომაც <= ია read-სთვის გადაცემულ ზომაზე (რადგან ყოველთვის არ არის იმდენი ბაიტი წასაკითხი რამდენიც read-ს გადაეცემა), თუ მაქვს ეს chunk ქეშში, მაშინ პირდაპირ ქეშიდან მომაქვს და სერვერს აღარ ვუკავშირდები (თან ვსეტავ, რომ ეს entry ამ დროს მოვითხოვე ბოლოს), თუ არ მაქვს, სერვერიდან ვკითხულობ და რასაც წავიკითხავ იმას ჩავწერ ქეშში. თუ ამ chunk-ის დამატებით client სტრუქტურაში შენახულ ქეშის ზომას გადავცდები, მაშინ უნდა გავაძევო ჯერ სხვა chunk. გადავუყვები ყველა entry-ს და ვნახულობ ყველაზე ადრე რომელი იყო ბოლოს გამოყენებული და თუ ყოფნის მისი გაძევება, ვაძევებ და ახალს ვწერ მის ადგილას, თუ არადა საერთოდ არ ვწერ ახალს, რადგან შეიძლება ძალიან დიდი chunk იყოს და ბევრი სხვა entry-ს გაძევება მომიწიოს, რომლებსაც ხშირად ვიყენებ, ამიტომ სადაა რომელი უფრო ოპტიმალურია, ანუ ღირს თუ არა იმდენის გამოძევება რამდენიც საჭიროა.

write-ის დროს სერვერებზე ვწერ პირდაპირ, ხოლო ქეშიდან ვაძევებ ამ ფაილის ყველა chunk-ს, რადგან ფაილებში აუცილებლად ჩასაწერია, კლიენტის გათიშვის შემთხვევაში დატა რომ არ წაიშალოს და ქეშშიც იგივენაირად ჩაწერა არაოპტიმალურია, რადგან read-ები შეიძლება სხვანაირი offset-ებით და size-ებით მოხდეს ამ write-ის შემდეგ და ნაწილ-ნაწილ იქნება entry-ებში ძებნა საჭირო.

unlink-ის დროსაც ვშლი ამ ფაილის ყველა entry-ს ქეშიდან. ცარიელ ადგილებს არ ვტოვებ entry-ების მასივში, ბოლოდან მოვყვები და თითოეულ წაშლაზე memcpy-თი ვაჩოჩებ მის მარჯვნივ მყოფებს.

rename-ის დროს გადავყვები ქეშს და სადაც path-ის მქონე entry-ს ვნახავ newpath-ით ვანაცვლებ მისამართს.

Epoll API

სერვერის მხარეს დამატებული მაქვს epoll, მაგრამ რადგან სულ ერთი კლიენტი ყავს თითო სერვერს, ოპტიმიზაციის მხრივ არაფერს ცვლის, უბრალოდ წაკითხვის წინ epoll აძლევს socket file descriptor-ს და იქიდან კითხულობს სერვერი.