პროექტის არქიტექტურა:

ორი მთავარი ფაილია net_raid_client.c და nete_raid_server.c რომლებიც სტარტავენ სერვერს და კლიენტს როგორც კონფიგურაციის ფაილშია მოთხოვნილი. დეფოლტზე foreground-ში ეშვება fuse-ის პროცესები და სერვერებიც, რადგაც ზოგი ისეთი პრინტია, რაც ლოგებში არ უნდა იყოს, მაგრამ კონსოლში პრინტვისას კარგად ასახავს თუ რას აკეთებენ კონკრეტულ მომენტში სერვერები ან fuse-ის პროცესები.

სერვერის გაშვებისას პარსავთს თავის პარამეტრებს, იწყებს კლიენტის ლოდინს, და შემდეგ კი იღებს მისგან ტასკებს(struct task_R1-ებს, რის შესახებაც მერე ვილაპარაკებ). თითო ტასკს თავისი ჰენდლერი აქვს რომლებიც task_handling_R1.c-შია აღწერილი.

კლიენტის გაშვებისას კონფიგურაციის ფაილიდან თითო დისკისთვის საჭირო ინფორმაცია ამოიკითხება და გაეშვება ცალკე პროცესი, რომელიც აგვარებს ერთი დისკის საქმეს. საქმის მოგვარებაში იგულისხმება ის, რომ თითო სისტემ ქოლისას იგზავნება საჭირო ტასკი სერვერებზე, სერვერები კი აბრუნებენ რისპონსებს(struct server_response_R1, რომლის შესახებაც მერე ვილაპარაკებ), სადაც წერია საჭირო ინფორმაცია იმისთვის, რომ კლიენტი მიხვდეს თუ როგორ შესრულდა გაგზავნილი ტასკი.

ლოგირების მესიჯები:

ყველა ლოგს წინ ეწერება თავისი დრო. ილოგება ნებისმიერი გამოძახებული სისტემ ქოლი ნებისმიერ სერვერზე(ip:port syscall ფორმატით) და ასევე სერვერთან წვდომასთან დაკავშირებული შემდეგი ინფორმაციები: სერვერთან დაკავშირება, კავშირის დაკარგვა, hotswap-ზე გადასვლა იმის გამო, რომ ერთ-ერთ სერვერთან კავშირის აღდგენა დიდი ხნის განმავლობაში ვერ მოხერხდა.

გამოყენებული მონაცემთა სტრუტურები:

კონსტანტები და სტრუქტურები ინახება shared_types.h და shared_constants.h ფაილებში.

```
struct initial_task {
  int task_type;
};
```

ეგზავნება სერვერს როგორც პირველი ინფორმაცია. task_type განსაზღვრავს თუ რისთვის ვიყენებთ ამ სერვერს(raid1-ს ერთ-ერთი სერვერი, raid5-ის ერთ-ერთი სერვერი თუ hotswap სერვერი)

```
struct server_and_port {
  char server[32];
  int port;
};
```

ამ სტრუქტურით ვინახავ თითო სერვერის ip-ს და პორტს.

```
struct storage_info {
  char diskname[32];
  char mountpoint[256];
  int raid;
  int server_count;
  struct server_and_port servers[32];
  struct server_and_port hotswap;
};
```

ამ სტრუქტურაში ინახება თითო სთორიჯის შესახებ საჭირო ინფორმაცია: სახელი, დასამაუნტებელი ფოლდერის მისამართი, რომელი რეიდის ალგორითმი უნდა გამოიყენოს, გამოყოფილი სერვერების რაოდენობა(hotswap-ის გარეშე), სერვერები.

```
struct task_R1 {
  char comment[64];
  int task_type;
  char path[MAX_PATH];
  char buf[MAX_BUF];
  int size;
  int offset;
  int mask;
  mode_t mode;
  dev_t rdev;
  char from[MAX_PATH];
  char to[MAX_PATH];
  int flags;
};
```

ეგზავნება სერვერს როგორც ტასკი კლიენტისგან, თან მიყვება პარამეტრები, რომლების გატანაც შეიძლება დასჭირდეს fuse-ის სისტემ ქოლს. Comment გამოიყენებოდა მხოლოდ დებაგირებისთვის, დანარჩენი ცვლადები ფიგურირებს fuse-ის გადატვირთულ სისტემ ქოლებში.

```
struct server_response_R1 {
  char comment[32];
  int ret_val;
  struct stat stbuf;
  char buf[MAX_BUF];
  int files_in_dir;
  char file_names[MAX_FILES][MAX_PATH];
  struct stat stats[MAX_FILES];
  unsigned char old_hash[16];
  unsigned char cur_hash[16];
  int hashes_match;
  int success;
  int is_dir;
  int files_ended;
  int size;
};
```

გამოიყენება სერვერისგან პასუხის მისაღებად თუ როგორ შეასრულა ტასკი. Ret_val-ში იწერება საბოლოო შედეგი, დანარჩენ პარამეტრებში კი ის ინფორმაცია, რომელიც საჭიროა კლიენტის მხარეს გამოძახებული სისტემ ქოლის წარმატებით(თუ სერვერის მხარესაც წარმატებით შესრულდა) დასამთავრებლად.

ამ ცვლადების შესახებ უფრო დეტალურად ალგორითმების მხარეს ვილაპარაკებ.

ალგორითმები:

სისტემ ქოლების ჰენდლინგი:

getattr:

ნებისმიერ ცოცხალ სერვერზე იგზავნება ტასკი, რომლის task_type-ითაც სევრეი ხვდება რომელი სისტემ ქოლი უნდა გაუშვას თავის მხარეს(ამ სისტემ ქოლისთვის საჭირო ცვლადები ტასკ-ს მოყვება). გაშვების შემდეგ იქმნება server_response_R1 ტიპის ცვლადი, რომლის ret_val-შიც იწერება სერვერზე გამოძახებული სისტემ ქოლის დაბრუნებული შედეგი, და წარმატების შემთხვევაში ასევე ივსება კლიენტის მხარეს სისტემ ქოლის დასასრულებლად საჭირო ცვლადები(მაგალითად getattr-ს სჭირდება stbuf, readdir-ის შემთხვევაში ფაილების სახელები და ა.შ.)

Write:

მუშაობს ჩვეულებრივი write-სავით, მაგრამ ასევე ააფდეითებს ქეშსაც(ქეშის იმპემენტაციის შესახებ ქვემოთ ვილაპარაკებ). Write-ი ჯერ ხდება პირველ სერვერზე, და მიღებული პასუხის შემდეგ ხდება მეორე სერვერზე(თუ რომელიმე არ ფუნქციონირებს ამ მომენტისთვის, მაშინ მხოლოდ ერთზე). შესაძლოა, რომ ჩაწერა(ან ჰეშის დააფდეითება(ჰეშების შესახებ open-ში ვილაპარაკებ)) შუა პროცესში შეწყდეს. ეს შეიძლება მოხდეს პირველ სერვერზე წერისას ან მეორე სევერზე წერისას. პრობლემას open სისტემ ქოლი აგვარებს.

Read:

მუშაობს write-ის მსგავსად, ჯერ იხედება ქეშში და ნახულობს ხომ არ არის ქეშსი ის, რისი წაკითხვაც გვინდა, თუ არის პირდაპირ კითხულობს ქეშიდან და აღარ გზავნის სევერზე read-ის ტასკს, თუარადა ჩვეულებრივად გზავნის სერვერზე read-ის ტასკს, რის შემდეგაც სერვერი აბრუნებს საჭირო ინფორმაციას ამ სისტემ ქოლის დასამთავრებლად(წაკითხული ბაიტების რაოდენობა, შემდეგ კი თვითონ ბაიტები).

Open:

Open-ი გარდა თავისი საქმისა, აგვარებს write-ის ან ფაილის შექმნის დროს მომხდარ ხარვეზებს. ფაილის შექმანსთან დაკავშირებით ხავეზი შეიძლება იყოს შემდეგნაირი, ფაილის შექმნისას, მოესწრო მხოლოდ ერთ სერვერზე მისი შექმნა, და მეორეზე არა(რაღაცის გამო გაითიშა კლიენტი). ამ ფაილზე open-ის დროს ჯერ ორივე სერვერიდან პასუხი ბრუნდება(თუ ერთი სერვერია მხოლოდ ცოცხალი, მაშინ ამ სერვერიდან დაბრუნებული პასუხი საბოლოოა), თუ აღმოჩნდა რომ ერთ-ერთ სერვერზე ასეთი ფაილი არის, და მეორეზე არაა, მაშინ ხდება ამ ფაილის კოპირება, და მხოლოდ ამის შემდეგ სრულდება სისტემ ქოლი.
თუ ხარვეზი ჰქონდა write-ს, მაშინ რამდენიმე ვარიანტია შესაძლებელი. Write-ი ჯერ კონტენტს წერს, შემდეგ ააფდეითებს ფაილის ჰეშს(extended attribute-ში ინახება). თუ არ მოესწრო ფაილის მთლიანად ჩაწერა პირველ სერვერზე, მაშინ ჰეშიც არ დააფდეითდება, შესაბამისად მეორე სერვერზე ჯერ კიდევ სწორი ჰეშია, ამიტომ open-ის დროს, მეორე სერვერის ინფორმაცია გადმოეწერება პრიველს. თუ ჰეშის დააფდეითება მოესწრო პირველ სერვერზე, მაშინ ხარვეზი მეორე სერვერზე რომ იყოს(მთლიანად ვერ ჩაიწერა, ან ჰეში ვერ დააფდეითდა), მაშინ პირველი სერვერიდან გადაეწერება ინფორმაცია მეორე სერვერზე(რადგან პირველზე ჰეში სწორი იქნება, და რადგან ჯერ პირველზე ვწერთ ესეიგი იქ უფრო 'ახალი' ვერსიაა ამ ფაილის).

დანარჩენ სისტემ ქოლებს რაც შეეხება ყველა მოქმედებს ერთნაირად, გადმოცემული პარამეტრებით ამზადებენ task_R1 ტიპოს ცვლადს, რომელსაც უგზავნიან სერვერს, სერვერზე გამოიძახება ნამდვილი სისტემ ქოლი, რომლის დაბრუნებული შედეგი და შევსებული ცვლადები იწერება server_response_R1 ტიპის ცვლადში, და ეს ცვლადი ეგზავნება უკან კლიენტს, რომელიც ამ ვლადიდან საჭირო პარამეტრებით ამთავრებს თავის სისტემ ქოლს.

Cache:

ქეშის იმპლემენტაციისთვის ავარჩიე ლინკლისტის ალგორითმი, რადგან გვევალებოდა LRU cache replacement.

შევქმენი შემდეგი სტრუქტურა:

```
struct Cnode {
        int size;
        int pseudo_size;
        int offset;
        char* path;
        char* buf;
        Cnode* prev;
        Cnode* next;
};
```

რომელიც საშუალებას მაზლევს ქეშის ენთრიები ერთმანეთს გადავაბა და ასევე ვინახო ინფორმაცია მათ შესახებ. ეს Cnode-ები იქმნება read-ის დროს. Offset ის პარამეტრია, რომელიც სისტემ ქოლს გადმოეცა, size არის თუ რამდენი ბაიტის წაკითხვა მოხერხდა სისტემ ქოლის დამთავრებისას, ხოლო pseudo_size-ში ინახება თუ რამდენი ბაიტის წაკითხვა მოითხოვეს სისტემ ქოლით. Pseudo_size მაძლევს იმის საშუალებას, რომ შემდგომში write-ის გამოძახებისას მივხვდე, რომ ეს ენთრი დასააფდეითებელია იმ შემთხვევაშიც, როდესაც write-ის ინტერვალი pseudo_size-ის რაღაც ნაწილზე წერს(რასაც ვერ ვიზამმდი მხოლოდ size-ის შენახვით, რადგან ვერ მივხვდებოდი წესით უნდა ეკუთვნოდეს თუ არა ამ ქეშ ენთრის ის დააფდეითებული ბაიტები). რადგან მთელი ქეში მაქვს აგებული ლინკდ ლისტზე, შესაბამისად LRU cache replacement-ის დაცვაც მარტივი ხდება, ყოველ ჯერზე როდესაც რაღაც ენთრი გამოიყენება read-ისთვის ან write-ისთვის, ის ლისტის მარჯვნივ გადავარდება, ხოლო როდესაც მჭირდება ქეშში ახალი ენტრის დამატება და ადგილი არ მყოფნის, ქეშის მარცხნიდან ვიწყებ ენთრიების ამოყრას რადგან ისინი იქნებიან ყველაზე ნაკლებად გამოყენებადები.

Hotswap:

ჰოტსვაპის ჰენდლინგისთვის მაქვს გაშვებული ცალკე სრედი, რომელიც ამოწმებს სერვერების მდგომარეობას(პერიოდულად გზავნის ინფორმაციას, რომელიც მარტო იმაში გამოიყენება, რომ შემოწმდეს ცოცხალია თუ არა სერვერი). ამ გაგზავნებს სრედი აკეტებს უსასრულო ციკლში. როდესაც რომელიმე სერვერი ვერ იღებს ინფორმაციას, იმახსოვრებს გათიშვის დროს, მონიშნავს სერვერს როგორც არაფუნქციონირებად სერვერად და შემდგომ timeout წამის განმავლობაში ამ სერვერთან ინფორმაციის გაგზავნის მაგივრად უკვე ცდილობს კავშირის აღდგენას(ამ პერიოდში ყველა სისტემ ქოლი გამოიძახება მხოლოდ ერთ, მუშა სერვერზე). თუ timeout პერიოდში მოხერხდა კავშირის აღდგენა, მაშინ მეორე სერვერიდან ყველაფერი გადაიწერება ამ სერვერზე, თუარადა ეს სერვერი ჩანაცვლდება hotswap-ით, რომელზეც გადაიწერება მთლიანი მეორე სერვერის ინფორმაცია, და შემდგომში გაგრძელდება მუსაობა ჩვეულიბრივ ორ სერვერთან.

გამოყენებული დამატებითი ტასკები:
TASK_CPYFL - უგზავნის სერვერს ფაილის მისამართს, რომლის წამოღებაც უნდა კლიენტს(კოპირება), სერვერი ჯერ აბრუნებს ფაილის ზოომას, შემდეგ კი მაგ ზომის კონტენტს.

TASK_RCVFL - ეგზავნება სერვერს ინფორმაცია, რომ უნდა მიიღოს გარკვეული ზომის ფაილი(პირველ გაგზავნაზე მხოლოს მისამართი და ზომა ეგზანება, შემდეგ კი მაგ ზომის კონტენტი)

TASK_HEALTHCHECK - ამ ტასკს გზავნის კლიენტის ის სრედი, რომელიც სერვერების ფუნქციონირებას ამოწმებს.

TASK_SMWYG - კლიენტი ამ ტასკს უგზავნის სერვერს, რის შემდეგაც სერვერი თავის ყველა ფაილს(ფოლდერებიანად) უგზავნის უკან კლიენტს, ისეტი თანმიმდევრობით, რომ შესაძლებელი იყოს მათი შექმნა(ანუ ჯერ ფაილი, და მერე ფოლდერი, რომელშიც ეს ფაილია, ასეთი თანმიმდევრობით არ მოვა). ეს ტასკი TASK_RCVFL-თან კომბინაციით გამოიყენება ჰოტსვაპზე მუშა სერვერიდან ინფორმაციის გადასატანად.

TASK_HARAKIRI - ამ ტასკის გაგზავნისას სერვერი შლის ყველა თავის ფაილს სტორიჯში.

სანამ მიმდინარეობს სერვერების შემოწმება ან ინფორმაციის ჰოტსვაპზე გადატანა, სისტემ ქოლები გაჩერებულია, რაც უბრალოდ ერთი სემაფორითაა მოგვარებული.