პროექტის დოკუმენტაცია

1. კონფიგურაციის ფაილი
2. RAID 1, იერარქიული დირექტორიის სტრუქტურა
3. მაღალმდგრადობა
4. ქეშირება
5. Epoll API
6. ლოგირება
7. ხარვეზები
8. გაშვება

პროექტი მოიცავს შემდეგ ფაილებს: parser.h და parser.c (კონფიგურაცისს ფაილის დაპარსვა), net\_raid\_client.c (კლიენტი), net\_raid\_server.c (სერვერი), net\_raid\_protocol.h (მოიცავს იმ სტრუქტურებსა და კონსტანტებს, რომლებსაც ვიყენებ როგორც კლიენტში, ისე სერვერში), cache\_object.h (მოიცავს სტრუქტურების, რომლებიც გამოიყენება ქეშირებისთვის), makefile.

**კონფიგურაციის ფაილი**

კონფიგურაციის ფაილის ინფორმაციას ვპარსავ და ვინახავ ჩემ მიერ შექმნილ სტრუქტურებში. ინფორმაციის შესანახად შევქმენი შემდეგი სტრუქტურები:

struct cfg {

char\* errorlog;

char\* cacheSize;

char\* cacheReplacment;

int timeout;

struct storage\* storages;

int numStorages;

};

struct cfg-ში ვინახავ ზოგად ინფორმაციას:

* char\* errorlog: იმ ფაილის path სადაც ხდება ლოგირება.
* char\* cacheSize: ქეშის ზომა. ჩათვლილი მაქვს, რომ კონფიგურაციის ფაილში ქეშის ზომა ჩაწერილია შემდეგ ფორმატთაგან ერთ-ერთით (n დადებითი მთელი რიცხვვია):

cache\_size = n // n ბაიტი

cache\_size = nK // n კილობაიტი

cache\_size = nM // n მეგაბაიტი

cache\_size = nG // n გიგაბაიტი

* char\* cacheReplacment: ქეშიდან გამოძევების ალგორითმი. ამ ეტაპზე მოცემული ცვლადის მნიშვნელობას ტექნიკურად აზრი არ აქვს, რადგან ქეშიდან გამოძევების მხოლოდ ერთ ალგორითმს (LRU) ვიყენებ ისე, რომ ამ ცვლადის მნიშვნელობას არანირ ყურადღებას არ ვაქცევ.
* int timeout: თუ რომელიმე სერვერი გაითიშება, timeout დროის განმავლობაში არ უნდა გამოცხადდეს დაკარგულად და უნდა ვცდილობდე მასთან დაკავშირებას. ჩათვლილი მაქვს, რომ კონფიგურაციის ფაილში შესაბამისი მნიშვნელობა იქნება წამების რაოდენობა.
* struct storage\* storages: struct storage-ს მასივი. *struct storage-ს აღწერა ქვემოთაა.*
* int numStorages: storage-ების რაოდენობა.

struct storage {

char\* diskName;

char\* mountpoint;

char\* raid;

char\*\* servers;

char\* hotswap;

int numServers;

};

struct storage-ში ვინახავ ინფორმაციას თითოეული სთორიჯისთვის:

* char\* diskName: დისკის/სთორიჯის სახელი.
* char\* mountpoint: path to mountpoint.
* char\* raid: 1 (RAID 1) or 5 (RAID 5)
* char\*\* servers: სტრინგების მასივი, თითოეული ელემენტი წარმოადგენს აიპისა და პორტის ერთობლიობას, მაგ: 127.0.0.1:10001
* char\* hotswap: ჰოთსვაპ სერვერის აიპისა და პორტის ერთობლიობა, მაგ: 127.0.0.1:10002
* int numServers: სერვერების რაოდენობა

**RAID 1, იერარქიული დირექტორიის სტრუქტურა**

პროგრამა თითოეული სთორიჯისთვის 2 სერვერზე ინფორმაციას ინახავს RAID 1 პრინციპით. მომხმარებელს დამაუნთებულ დირექტორიაში შეუძლია იერარქიული დირქტორიების სტრუქტურის შექმნა. გადავტვირთე FUSE-ს შემდეგი syscall-ები: getattr, opendir, readdir, mknod, mkdir, rename, rmdir, unlink, open, read, releasedir, release, truncate და write.

client-ის გაშვებისას, ხდება მონაცემთა დაპარსვა ზემოთ აღწერილ სტრუქტურებში, სთორიჯების რაოდენობის შესაბამისად ხდება პროცესის fork. შემდეგ მონაცემები უფრო მოქნილ სტრუქტურაში გადამაქვს, რათა დავუსეტო FUSE-ს private data-დ.

typedef char fixed\_size\_string[256]; // ვიყენებ სერვერის სახელის შესანახად.

struct data {

char disk\_name[256];

int sfd[3];

int sfd\_status[3];

fixed\_size\_string server\_names[3];

FILE\* errorlog\_fptr;

char hotswap[256];

int num\_of\_healthy\_servers;

int num\_of\_working\_servers;

int hotswap\_status;

int timeout;

};

* char disk\_name[256]: სთორიჯის/დისკის სახელი
* file descriptor-ების მასივი, სადაც ხდება socket() ფუნქციის მიერ დაბრუნებული მნიშვნელობის დასეტვა. ვინაიდან პროექტი დავწერე მხოლოდ RAID 1-სთვის, int fd-ების რაოდენობა 3ია: პირველი ორი ძირითადი სერვერებისთვის და მესამე სარეზერვო სერვერისთვის.
* int sfd\_status[3]: მოცემულ სამ ელემენტიან მასივში ვიმახსოვრებ int sfd[3]-ში არსებული sfd-ების სტატუსებს. -1 ნიშნავს შესაბამის სერვერთან კავშირის არ ქონას, ხოლო 1 - კავშირს. თავიდან 3-ვე სტატუსი -1ია, როგორც კი ხდება connection, სტატუს ვცვლი 1-ით.
* fixed\_size\_string server\_names[3]: ამ მასივში ვწერ თითოეული სერვერის სახელს, ვიყენებ ლოგირებისთვის.
* FILE\* errorlog\_fptr: იმ ფაილის მიმთითებელი, სადაც ხდება ლოგირება.
* char hotswap[256]: სარეზერვო სერვერზე ინფორმაცია (ip, port).
* int num\_of\_healthy\_servers: სერვერების რაოდენობა, რომლებთანაც არის კავშირი ან პოტენციურად შესაძლებელია დაქონექთება. თავიდან მნიშვნელობა არის 3.
* int num\_of\_working\_servers: სერვერების რაოდენობა, რომლებთანაც ამჟამად არის დამყარებული კავშირი.
* int hotswap\_status: გამოყენებულია თუ არა სარეზერვო სერვერ (-1/1)
* int timeout: ნებისმიერი სერვერის გათიშვის შემთხვევაში, timeout წამის განმავლობაში მიმდინარეობს კავშირის აღდგენის მცდელობა.

მას შემდეგ, რაც დაამყარებს კლიენტი კავშირს სერვერებთან და გამოიძახებს fuse\_main-ს, მომხმარებელს შეეძლება RAID 1-ით იერარქიული დირექტორიების სტრუქტურის შექმნა, ფაილების შექმნ, გახსნა, ჩაწერა, კოპირება (ყველა ის ფუნქცია, რაც შესაძლებელია ზემოთ აღწერილი სისტემქოლებით).

კლიენტი და სერვერი კომუნიკაციას net\_raid\_protocol.h-ში აღწერილი სტრუქტურების და და კონსტანტების საშუალებით ამყარებენ:

struct protocol {

int function\_number;

char path[PATH\_MAX];

char newpath[PATH\_MAX];

mode\_t mode;

off\_t newsize;

size\_t size;

off\_t offset;

int flags;

int recover;

int append;

};

თუ კლიენტს ესაჭიროება ინფორმაციის მიმოცვლა სერვერთან, ნებისმიერი ფუნქციიდან სერვერს უნდა გაუგზავნოს struct protocol შესაბამისი მნიშვნელობებით:

* int function\_number: ინახავს ინ ფუნქციის ნომერს, რომლიდანაც მოხდა სერვერთან დაკავშირება. შესალო მნიშვნელობაბი (ასევე net\_raid\_protocol.h-ში აღწერილი):

static const int GETATTR = 1;

static const int OPENDIR = 2;

static const int READDIR = 3;

static const int MKNOD = 4;

static const int MKDIR = 5;

static const int RENAME = 6;

static const int RMDIR = 7;

static const int UNLINK = 8;

static const int OPEN = 9;

static const int READ = 10;

static const int RELEASEDIR = 11;

static const int RELEASE = 12;

static const int TRUNCATE = 13;

static const int WRITE = 14;

static const int INFINITE = 15;

static const int FILE\_PATH = 16;

ამათგან პირველი 14 FUSE-ს სიტემქოლს შეესაბამება, ხოლო ბოლო 2 პროგრამის მაღალმდგრადობასთანაა დაკავშირებული და მათზე ქვემოთ ვისაუბრებ.

* char path[PATH\_MAX]: ფაილის path. გამოიყემებს FUSE-ს ყველა ფუნქცია.
* char newpath[PATH\_MAX]: როდესაც ხდება ფაილისი rename, მოცემულ ცვლადში ვიმახსოვრებ ახალ path-s.
* mode\_t mode: mode, რომლითაც ხდება ფაილის/დირექტორიის შექმნა.
* off\_t newsize: გამოიყენება truncate-ში.
* size\_t size: ფაილში ჩაწერის/ფაილიდან კითხვისთვის გამოიყენება.
* int flags: ფაილის გახსნისთვის ვიყენებ (open).
* int recover: ვანიჭებ მხოლოდ ორ მნიშვნელობას: 1 ნიშნავს იმას, რომ კონკრეტულ ფაილში ჩაწრა ემსახურება მასში ინფორმაციის აღდგენას მეორე სერვერიდან; -1 უბრალოდ ჩაწერა მომხმარებლისგან.
* int append: გამოიყენება write-ში, აქვს 2 მნიშვნელობა: 1: ჩაწერისას ფაილში მოხდეს append; -1: დასაწყისიდან მოხდეს ჩაწერა.

სერვერი მიღებული struct protocol-ის ინფორმაციის დამუშავებისას ახდენს გადამისამართებას შესაბამის ფუნქციებში. უკან პასუხი შეიძლება დააბრუნოს როგორც int, წინასწარ შეთანხმებული ზომის char-ების მასივი ან მოცემულ სტრუქტურათაგან ერთ-ერთი:

* ამ სტრუქტურის სახით უბრუნდება getattr-ს result:

struct getattr\_protocol {

int status;

struct stat sb;

};

* დირქტორიის გახსნისას მიღებული შედეგი და DIR\*.

struct opendir\_protocol {

int status;

DIR\* dp;

};

* ფაილის გასნისას მიღებული შედეგი და file descriptor.

struct open\_protocol {

int status;

int fd;

};

ჩაწერა ხდება stable storage მექანიზმით. ფაილში ნებისმიერი ცვლილების შეტანისას განახლებულ ინფორმაციას ვჰეშავ და მიღებულ ჰეშს ვუსეტავ შესაბამის ფაილს. ფაილის გახსნისას კიდევ ერთხელ ვჰეშავ და ახალ ჰეშჰს ვადარებ ფაილის ატრიბუტად შენახულ ჰეშს. ვითვალისწინებ შემდეგ 4 ვარიანტს:

1. ორივე სერვერის დონეზე ძველი და ახალი ჰეშები ემთხვეა ერთმანეთს: ვთვლი, რომ ინფორმაცია არ დაზიანებულა.
2. პირველ სერვერზე ძველი და ახალი ჰეში დაემთხვა ერთმანეთს, ხოლო მეორეზე არა: პირველი სერვერიდან შესაბაისი ფაილიდან ინფორმაციას გადავწერ მეორე სერვერზე შესაბამის ფაილში.
3. მესამე შემთხვევა 2-ეს სიმეტრიულია.
4. არცერთი სერვერის დონეზე ძველი და ახალი ჰეშები არ ემთხვევა ერთმანეთს: ვთვლი, რომ ინფორმაცია დაზიანებულია ორივე სერვერზე და, შესაბაისად, მოცემულ ფაილს ორივე სერვერიდან ვშლი.

ჰეშირებას ვაკეთებ MD5-ით. **შესაბამისი კოდი აღებული მაქვს** [**www.stackoverflow.com**](http://www.stackoverflow.com) **დან**. net\_raid\_server.c-ში შესაბამის ადგილას კომენტარად მითითებული მაქვს სტეკოვერფლოუს კოდი გამოყენება

ფაილის დაზიანებასთან დაკავშირებით ხდება ლოგირება. მესიჯების აღწერა ქვემოთაა.

**მაღალმდგრადობა**

მაღალმდგრადობისთვის კლიენტს ნაკადებში ვუშვებ. თითოეული სერვერისთვის კლიენტი 2 ნაკადს უშვებს. შესაბამისად, ჩვეულებრი ნორმალურ მდგომარეობაში არის 4 ნაკადი, მათგან ერთი პირველი სევერთან კავშირის არსებობას ამოწმებს, მეორე - მეორე სერვერთან კავშირის არსებობას ამოწმებს, ხოლო დანარჩენი 2 ემსახურება FUSE-ს ფუნქციების გამოძახებას (სისტემქოლების გადატვირთვას)

ნაკადი, რომელსაც ევალება სერვერთან კავშირის შემოწმება, დორის რაღაც ინტერვალში ერთხელ (1000 მიკროწამი) სერვერს უგზავნის struct protocol-ს, რომელშიც ფუნქციის ნომრად მითითებულია INFINITE. ამის საპასუხოდ სერვერს მხოლოდ და მხოლოდ ის ეველება, რომ კლიენტის ამ ნაკადს დაუდასტუროს კავშირის არსებობა და უგზავნის int-ს OK\_STATUS. თუ კლიენტი წაიკითხავს OK\_STATUS-ს, დროის ინტერვალის გასვლის შემდეგ ისევ იმავეს გააკეთებს.

იმ შემთხვევაში, თუ კლიენტი ვერ მიიღებს სერვერიდან შესაბამის პასუხს, მოცემული სერვერის შესაბამის filedescriptors სტატუსად დაუსეტავს მნიშვნელობას -1, შესაბამისად FUSE-ს სიტემქოლების გამოძახებისას, ამ სერვერს ბრძანებეი არ ეგზავნება. timeout წამის განმავლობაში ეს ნაკადი ცდილობს სერვერთან რექონექთს. თუ რექონექთი ხდება, შესაბაიმის filedescriptors სტატუსად კვლავ 1 დაესეტება და პროგრამა მუშაობას ისევ 2 სერვერით გაააგრძელებს. მანამ სანამ timeout წამი გავა, პროგრამა ერთი სერვერით მუშაობს. კლიენტის მხარე ისეა დაწერილი, რომ ამ დროს მომხარებელს **შეუძლია** დირექტორიებში შესვლა/ გამოსვლა, ფაილების გახსნა/წაკითხვა, თუმცა **არ შეუძლია** ჩაწერა/შექმნა/წაშლა.

თუ timeout წამი გავა ისე, რომ გათიშულ სერვერთან რექონექთი არ მოხდება, ამ სერვერს სარეზერვო სერვერით ვანაცვლებ. ხდება სარეზერვო სერვერთან კავშირის დამყარება დარჩენილი სერვერიდან მონაცემთა სრული დუბლირება სარეზერვო სერვერზე (მონაცემთა დუბლირებისთვის დამჭირდა ფაილურ სიტემაზე რეკურსიული გადავლა და ყველა დირექტორიის/ფაილის path-ის გაგება. **stackoverflowze ვნაზე პატარა მეთოდი და იმაზე დაყრდნობით დავწერე Traversing a Filesystem with fts(3)**). პროგრამა ისევ 2 სერვერთან იმუშავებს. თუ კიდევ ერთი სერვერი გაითიშება, მოხმარებელს ვაძლევ უფლებას ერთ სერვერთან იმუშაოს ყველანაირი access-ის (წერა/კითხვა/შექმნა/წაშლა).

იმისათვის რომ ერთ server file descriptor-ში ჩაწერა რამდენიმე ნაკამდმა არ სცადოს ვიყენებ ლოქებს. კოდის ყველა ის ფრაგმენტი, სადაც ხდება სერვერთან ინფორმაციის გაგზავნა/სერვერიდან ინფორმაციის მიღება დალოქილია.

მაღალმდგრადობასთან დაკავშირებული მოვლენების ლოგირება ხდება ლოგ ფაილიში. ქვემოთ ჩამოწერილი მაქვს ლოგ მესიჯები და მათი განმარტებები.

**ქეშირება**

კონფიგურაციის ფაილში მითითებული ლიმიტის ფარგლებში კლიენტის მხარეს გამოვყოფ შესაბამის მეხსიერებას და მასში ვინახაავ იმ ბლოკებს, რომლებზეც მოხდა read-ის გამოძახება. თითოეულ ბლოკს ვინახავ შემდეგი სტრუქტურის სახით:

struct cache\_object {

char name[PATH\_MAX];

char\* content;

off\_t offset;

size\_t size;

};

* char name[PATH\_MAX]; ფაილის path.
* char\* content: ბლოკის შესაბამისი content.
* off\_t offset: ფაილის დონის offset, საიდანაც იწყება მოცემული ბლოკი.
* size\_t size: მოცემული ბლოკის ზომა.

ქეშზე ზოგადი ინფორმაციის შესანახად კლიენტში ვქმნი მხოლოდ ერთ cache\_info სტრუქტურას:

struct cache\_info {

int num\_elems;

int capacity;

size\_t cur\_size;

size\_t max\_size;

};

* int num\_elems: ქეშში არსებული ბლოკების რაოდენობა.
* int capacity: ბლოკებისთვის გამოყოფილი რაოდენობა, (დინამიური).
* size\_t cur\_size: ქეშის მიმდინარე ზომა ბაიტებში.
* size\_t max\_size: ქეშის მაქსიმალური ზომა ბაიტებში.

ქეში წარმოადგენს cache\_object-ების მასივს. როდესაც მომხმარებელი იძახებს reads, ქეშში ვეძებ ისეთ ბლოკს, რომლიდანაც შევძლებ წავაკითხო შესაბამისი ფაილის ოფსეტიდან შესაბამისი რაოდენობის ბაიტი. თუ ასეთი ბლოკი არ მაქვს, ინფორმაციას სერვერიდან ვკითხულობ და მასივის ბოლოში ვამატებ ახლად შექმნილ cache\_object-ს. თუ ასეთი ბლოკი მაქვს მომხარებელს ქეშიდან ვაძლევ შესაბამის ინფორმაციას (სერვრზე მიუსვლელად) და მოცემული cache\_object გადამაქვს მასივის ბოლოში.

როდესაც ხდება ქეშის გადავსება, ვაძევებ მასივის პირველ ელემენტს, რადგან ის ყველაზე ადრე ჩამატებული ელემენტია (LRU). გამოძევებამდე ვამოწმებ ახალი ბლოკის ზომა ქეშის ზომაზე დიდი ხომ არაა, რადგან არ მოხდეს ტყუილად გამოძევეაბ. თუ ბლოკის ზომა ქეშის ზომაზე ნაკლებია, ვაძევებ იმდენ cache\_object-ს, რამდენიც საჭიროა ახალი ბლოკის ჩასამატებლად.

თუ ხდება ფაილის rename ან unlink, ქეშში შესაბამის ობიექტებს შესაბამის ცვლილებებს ვუკეთებ.

თუ ხდება ისეთ ფაილში ჩაწრა, რომელიც მაქვს ქეშში, ქეშიდან ამ ფაილის შესაბამის ბლოკებს ვაძევებ.

**Epoll API**

სერვერის მხარეს ვიყენებ Epoll-ს. გამომდინარე ჩვენი პროექტის სპეციფიკიდან, სერვერი მოხლოდ 1 კლიენტს ემსახურება, და ამიტომაც პრაქტიკულად Epoll-ს ამჟამად დანიშნულება არ აქვს. თუმცა პროექტის გადაკეთების შემთხვევაში მოხდება რესურსების დაზოგვა.

**ლოგირება**

2018-08-12 01:19:03: STORAGE1: disconnected from server: 127.0.0.1:10001

როდესაც ხდება ნებისმიერ სერვერთან კავშირის გაწყვეტა.

2018-08-12 01:19:04: STORAGE1: Reconnected to 127.0.0.1:10001 after 1.000000 seconds. connection is renewd.

როდესაც ხდება ნებისმიერ სერვერთან კავშირის აღდგენა.

2018-08-12 01:19:04: STORAGE1: Status is OK, server name: 127.0.0.1:10001

სერვერთან კავშირის აღდგენის შემთხვევაში სერვერის მიერ დაბრუნებული სტატუსი დამაკმაყოფილებელია.

2018-08-12 01:19:16: STORAGE1: 3.000000 seconds waited, can not reconnect to 127.0.0.1:10002. Need to connect to a new server

ტაიმაუტ წამის განმავლობაში ვერ აღდგა სერვერთან კავშირი.

2018-08-12 01:19:16: STORAGE1: closing server file descriptor, server name: 127.0.0.1:10002

სერვერი გამოცხადდა დაკარგულად.

2018-08-12 01:19:16: STORAGE1: Connecting to hotswap server...

საჭიროა სარეზერვო სერვერის გამოყენება. იწყება კავშირის დამყარება.

2018-08-12 01:19:16: STORAGE1: Connected to hotswap server

სარეზერვო სერვერთან კავშირი დამყარებულია.

2018-08-12 01:19:16: STORAGE1: number of healthy servers: 2

სარეზერვო სერვერთან კავშირის დამყარების შემდეგ იწერება აქტიური და პოტენციურად აქტიური სერვერების ჯამური რაოდენობა.

2018-08-12 01:19:16: STORAGE1: number of working servers: 2

სარეზერვო სერვერთან კავშირის დამყარების შემდეგ იწერება აქტიური სერვერების რაოდენობა.

2018-08-12 01:19:16: STORAGE1: All data is copied to hotswap server

მესიჯი, რომელიც იწერება სარეზერვო სერვერზე ჯანსაღი სერვერიდან მონაცემთა სრული დუბლირების შემდეგ.

2018-08-12 01:25:20: STORAGE1: Can not conect to hotswap server.

თუ საჭიროა სარეზერვო სერვერის ჩართვა, მაგრამ მასთან კავშირი ვერ დამყარდა.

2018-08-12 01:19:29: STORAGE1: There is only one healthy server

როდესაც დაკარგულად გამოცხადებულია ორი სერვერი და აქტიურია მხოლოდ 1.

2018-08-12 01:19:34: STORAGE1: There are no healthy servers. shutting down the program...

როდესაც დაკარგულად გამცხადებულია ყველა სერვერი.

recovere file from 127.0.0.1:10002 to 127.0.0.1:10001. file path: /a

დაზიანებული ფაილის აღდგენა.

File is damaged on both servers: /c

ფაილი დაზიანებული იყო ორივე სერვერზე და წაიშალა.

**ხარვეზები**

პროგრამა ვაგრანტზე გავტესტე ძალიან ბევრი სცენარის მიხედვით, გამართულად მუშაობს ზემოთ აღწერილი ყველა ფუნქცია. ვაგრანტზე გატესტვისას ხანდახან (მაგრამ არა ყოველთვის) პროგრამას ექმნება პრობლემები, როდესაც დიდ ფაილებს (1 მეგაბაიტზე დიდი) ვწერ/ვკითხულობ. მეგაბაიტზე ნაკლები ფაილებისთვის პრობლემა არასდროს არ შემქნია.

**გაშვება**

გასაშვებად საკმარისია ტერმინალით დადგეთ იმ დირექტორიაში, სადც მოათავსებთ პროგრამის ფაილებს, და გამოიძახოთ **make.**

make-ს გამოძახების შემდეგ სერვერს გაუშვებთ შემდეგნაირად:

**./net\_raid\_server [ip] [port] [path/to/server/directory]**

make-ს გამოძახების შემდეგ კლიენტს გაუშვებთ შემდეგნაირად:

**./net\_raid\_client –f [path/to/configfile]**