

HADOOP DISTRIBUTED FILE SYSTEM

(WITH SMALL FILE PROBLEM SOLUTION)

BY

KARUNESH TRIPATHI

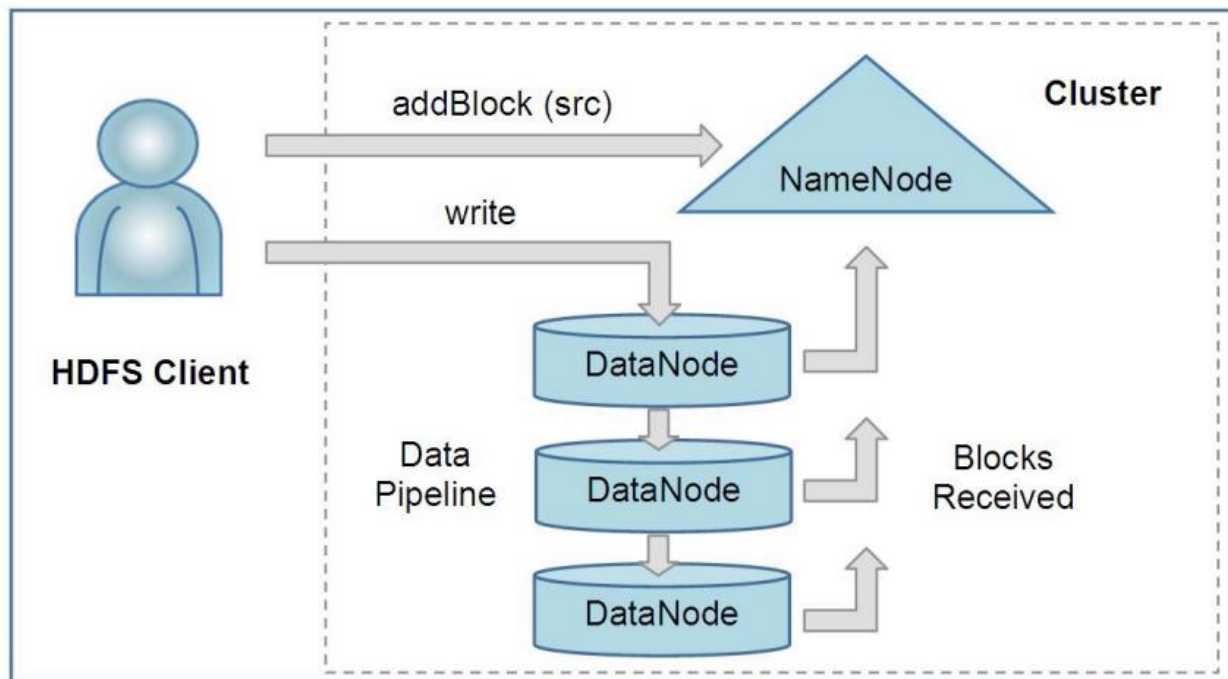
ABSTRACT

The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size.

INTRODUCTION

Hadoop provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce paradigm. An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and executing application computations in parallel close to their data. A Hadoop cluster scales computation capacity, storage capacity and IO bandwidth by simply adding commodity servers. Hadoop clusters at Yahoo! span 25 000 servers, and store 25 petabytes of application data, with the largest cluster being 3500 servers. One hundred other organizations worldwide report using Hadoop. HDFS is the file system component of Hadoop. While the interface to HDFS is patterned after the UNIX file system, faithfulness to standards was sacrificed in favor of improved performance for the applications at hand. HDFS stores file system metadata and application data separately. As in other distributed file systems, like PVFS , Lustre and GFS, HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols.

ARCHITECTURE



An HDFS client creates a new file by giving its path to the NameNode. For each block of the file, the NameNode returns a list of DataNodes to host its replicas. The client then pipelines data to the chosen DataNodes, which eventually confirm the creation of the block replicas to the NameNode.

Namenode

The namenode is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It is a software that can be run on commodity hardware. The system having the namenode acts as the master server and it does the following tasks –

- Manages the file system namespace.
- Regulates client's access to files.
- It also executes file system operations such as renaming, closing, and opening files and directories.

Datanode

The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. For every node (Commodity hardware/System) in a cluster, there will be a datanode. These nodes manage the data storage of their system.

- Datanodes perform read-write operations on the file systems, as per client request.
- They also perform operations such as block creation, deletion, and replication according to the instructions of the namenode.

Block

Generally the user data is stored in the files of HDFS. The file in a file system will be divided into one or more segments and/or stored in individual data nodes. These file segments are called as blocks. In other words, the minimum amount of data that HDFS can read or write is called a Block. The default block size is 64MB, but it can be increased as per the need to change in HDFS configuration.

SMALL FILE ISSUE

Due to hadoop is an open source , it was adopted by several companies under different names such as: HortonWorks distribution by yahoo, Cloudera from Facebook, Google, Oracle and Yahoo, Facebook corona by Facebook and some others . That's means hadoop is a novel successful tool to treats Big Data problem, thus, during these long years, there were a problem flout on the surface called "big data in small files".Namenode, is the master node of hadoop, this node stores all the meta data about datanodes, and all metadata about the data in the datanodes, on the other hand, datanodes are used to store the actual data, and due to Hadoop's architecture, every single file must be replicated into 3 copies in the cluster in order of availability purposes. In Datanode memory, every 64 MB of memory called data block, if the stored file is less than the data block, it considers as a small file, this data block can hold one file only whether the file is 1 MB or 64 MB, every data block is known to the Namenode by metadata, every metadata size is 120B, thus, if one Datanode has 1000000 data blocks, the total size of metadata that sent already to Namenode is $120 \times 1000000 = 120 \text{ MB/Datanode}$, if the total nodes in one hadoop cluster is 10000 nodes, this leads to $120 \text{ MB} \times 10000 = 1.2 \text{ TB}$ of metadata, this massive number of metadata cannot be handled by the Namenode which could leads to a dramatical shut down, and if Namenode goes down, the secondary Namenode will take the lead for a while then shutting down , and finally the wholly hadoop cluster will be use-less.

SOLUTIONS

Of course, many research papers proposed some exits for this problem, some of these researches are already in use while the rest still documented:

- HAR file: Hadoop Archive was the first novel exit for big data in small file issue, basically HAR file used to pack the small files into HDFS blocks that's can treats directly with Namenode. HAR file is created by running a hadoop archive command, means a Mapreduce job starts packing all the huge number of small files to be a small number of HAR files, thus treating with ten of HAR files is much more efficient and smoother rather than treating with thousands of small size files , and beside of this, HAR file doesn't alter the architecture of HDFS. HAR is an excellent exit to avoid Namenode metadata jam, HAR file gives an option to access the files directly inside it as well as the creation steps done in easy commands, but on the other side, as a shortage, HAR cannot be altered after being created, cannot add more files, or delete some unwanted files from it. The bumpiest shortage in HAR is every file inside it requires 2 index files (Master index, Index) to read , that's mean reading a file from HDFS it self is much easier than reading from HAR file. Another limitation of HAR is the memory; HAR files puts extra pressure on the file system due to generate a copy of the original files and takes space as much as they need.

- nHAR file: new Hadoop Archive is a correction of HAR file, its almost the same story but with some differences in architecture, first different is nHAR file needs only one index file to read, the second different is nHAR can be edit, you can add more files to the archive after create it. (Mir 2017). As well as HAR, nHAR is used to read files without altering HDFS architecture, but on the contrary of it, nHAR use only one index to access the files instead, which mean more reading performance and efficiency, however, the reading mechanism is done based on creating a hash table instead of master index approach. Figure 1 shown nHAR working mechanisim that's every small file 1...n, every file will be hashing and given an index number right before being saved inside a large file called "Part File". Hash table in nHAR is using instead of the index file that's adopted in HAR, to generate a hash code, the file name and the number of index file are used, then merging index files and moving to the part file, once part file being full, and new one will generate and continue filling. Similar to HAR, the actual file will be stored to the part file, so the same problem with HAR that's the actual file will add more data load to HDFS whole storage, even this, nHAR easier and simpler than HAR in adding a new file to an existing nHAR file. Both of HAR and nHAR techniques are preventing to avoid altering HDFS architecture, the rest of this paper is shown another small file reduction technique but with HDFS architecture upgrade.

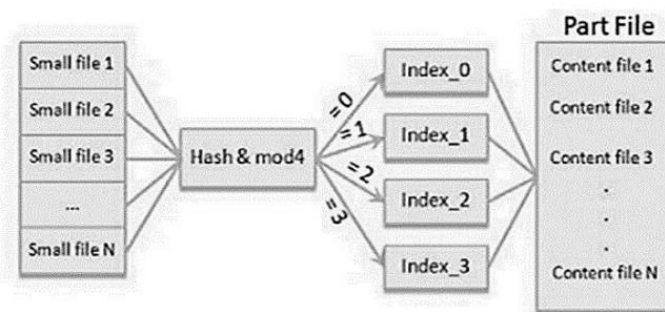


Figure 1. New archive technique [17, 24]

Improved HDFS

This technique stands of two parts, the first one is client components which packing the small files into one big file, the second one is data node component which treating the resource management. Improved HDFS is an index based that's used to collect all small files of the same HDFS directory in one file, on the other side, the cash manager will resident in the Datanode. The packing and storing method are running based on the small files offset and length, every small file will store in one big file one by one, then determine the total sum of the small files in that big file and finally comparing the result with the data block size is 64 MB, thus, if the large file total length goes greater than the block size, then multiple block will used to save the file separately . Figure 2 uncover the general file storing mechanism in HDFS that's only one file can be allocated in every data block, no matter if the file fits the block size or not, thus, every data block is connected to the Namenode via a metadata about it ID, Datanode location and some other info. As shown in Figure 2, once the file took place in the data block, the data block will be locked form inserting more files.

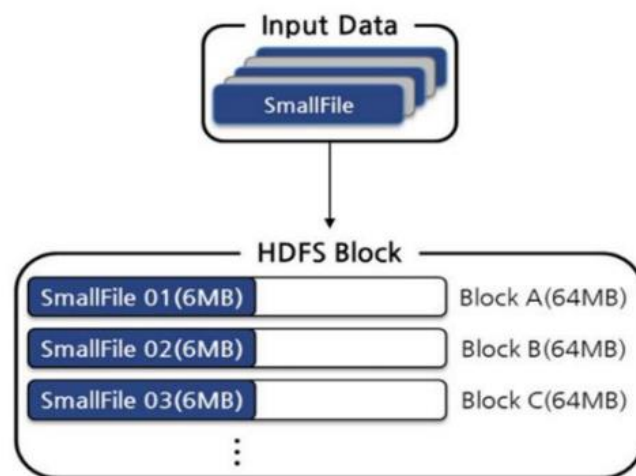


Figure 2. How the files being located in data blocks [26]

Hardware / Software Requirements

Minimum Hardware Requirements:

- 512 MB of system memory (RAM)
- 500 MHz x86 processor
- 5 GB of disk space
- Network Interface card

Maximum Software Requirements:

- Linux operating system capable or running GCC compiler

Block file size taken in program in 5kb

Features of HDFS

- It is suitable for the distributed storage and processing.
- Hadoop provides a command interface to interact with HDFS.
- The built-in servers of namenode and datanode help users to easily check the status of cluster.
- Streaming access to file system data.
- HDFS provides file permissions and authentication.

Assumptions and Goals

Hardware Failure

Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional. Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

Streaming Data Access

Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. POSIX imposes many hard requirements that are not needed for applications that are targeted for HDFS. POSIX semantics in a few key areas has been traded to increase data throughput rates.

Large Data Sets

Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.

Simple Coherency Model

HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. A MapReduce application or a web crawler application fits perfectly with this model. There is a plan to support appending-writes to files in the future.

“Moving Computation is Cheaper than Moving Data”

A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides interfaces for applications to move themselves closer to where the data is located.

Portability Across Heterogeneous Hardware and Software Platforms

HDFS has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a platform of choice for a large set of applications.

An example of HDFS

(FOR BETTER UNDERSTANDING)

Consider a file that includes the phone numbers for everyone in the United States; the numbers for people with a last name starting with A might be stored on server 1, B on server 2, and so on.

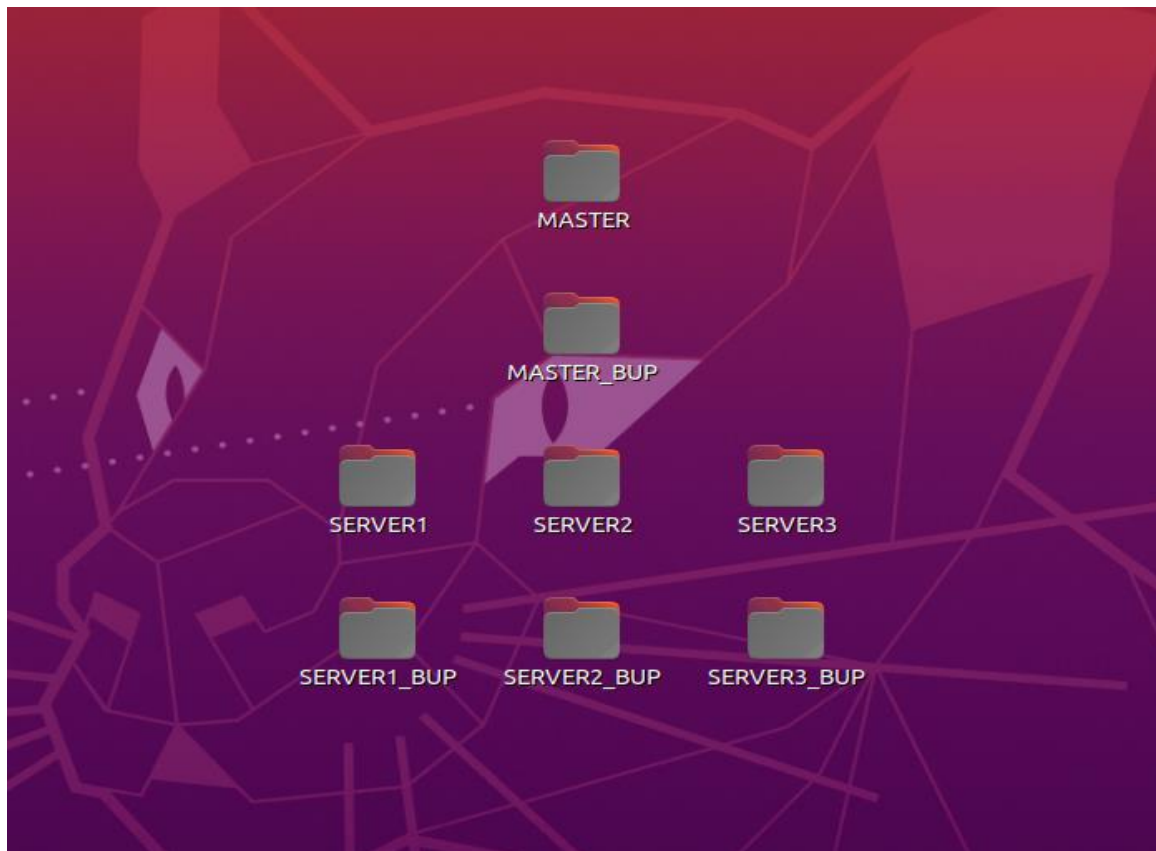
With Hadoop, pieces of this phonebook would be stored across the cluster, and to reconstruct the entire phonebook, your program would need the blocks from every server in the cluster.

To ensure availability if and when a server fails, HDFS replicates these smaller pieces onto two additional servers by default. (The redundancy can be increased or decreased on a per-file basis or for a whole environment; for example, a development Hadoop cluster typically doesn't need any data redundancy.) This redundancy offers multiple benefits, the most obvious being higher availability.

The redundancy also allows the Hadoop cluster to break up work into smaller chunks and run those jobs on all the servers in the cluster for better scalability. Finally, you gain the benefit of data locality, which is critical when working with large data sets.

Implementation

Storage Structure Used-



- We miniaturized the model of server into folder and file block size taken in 5kb.
- The master folder has all the information about the position of each file stored in our distributed file system.
- The uploaded file is distributed among SERVER1 , SERVER2 and SERVER3 and backup in corresponding _BUP folder. We have created only one backup per file for simplicity but there are at least 3 backups per file in real scenario.
- We have taken only 3 servers but in real world there can be many.
- **The small file problem** is solved by storing such file in rack named small files in SERVER1. The Rack is represented by folder named SMALLFILES inside SERVER1 as shown below.



and its back up in SERVER1_BUP in SMALLFILE_BUP. As shown



Client Code-

```
#include <arpa/inet.h>

#include <netinet/in.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <unistd.h>

#define IP_PROTOCOL 0

#define IP_ADDRESS "127.0.0.1"

#define PORT_NO 15050

#define Default_Size 1000

#define sendrecvflag 0

#define key 88

#define nofile "File Not Found!"

void clearBuf(char* b) {

    int i;

    for (i = 0; i < Default_Size; i++)

        b[i] = '\0';

}

char Cipher(char ch)

{

    return ch^key;

}


int SendFile(FILE* fp, char* buf, int s) {

    int i, len;

    if (fp == NULL) {
```

```

strcpy(buf, nofile);
len = strlen(nofile);
buf[len] = EOF;
for (i = 0; i <= len; i++)
buf[i] = Cipher(buf[i]);
return 1;
}

char ch, ch2;
for (i = 0; i < s; i++) {
ch = fgetc(fp);
ch2 = Cipher(ch);
buf[i] = ch2;
if (ch == EOF)
return 1;
}
return 0;
}

```

```

int RecieveFile(char* buf, int s)
{
int i,size;
char ch;
for (i = 0; i < s; i++) {
ch = buf[i];
size=s;
ch = Cipher(ch);
if (ch == EOF)
return 1;
}
}

```



```

else
printf("%c", ch);
}
return 0;
}

int main() {
int sockfddescriptor, nBytes,ch[1];
struct sockaddr_in addr_con;
int addrlen = sizeof(addr_con);
addr_con.sin_family = AF_INET;
addr_con.sin_port = htons(PORT_NO);
addr_con.sin_addr.s_addr = inet_addr(IP_ADDRESS);
char net_buf[Default_Size];
FILE* fp;
sockfddescriptor = socket(AF_INET, SOCK_DGRAM,IP_PROTOCOL);
if (sockfddescriptor < 0)
printf("\nfile descriptor not received!!\n");
else
printf("\nfile descriptor %d received\n", sockfddescriptor);
while (1) {
printf("\n Enter:\n 1. To Open File\n 2. To Upload new file\n 3. To Delete File\n 4. To Disconnect\n
Choice: ");
scanf("%d",&ch[0]);
sendto(sockfddescriptor, ch, sizeof(ch), 0, (struct sockaddr*)&addr_con, addrlen);

char path[3][200];
clearBuf(path[0]);
clearBuf(path[1]);
clearBuf(path[2]);
if(ch[0]==1)

```

```

{
printf("\nEnter name of the file to be opened: ");

scanf("%s", net_buf);

sendto(sockfddescriptor, net_buf, Default_Size,sendrecvflag, (struct sockaddr*)&addr_con,addrlen);

nBytes = recvfrom(sockfddescriptor, path, sizeof(path),sendrecvflag,(struct
sockaddr*)&addr_con,&addrlen);

printf("\n-----Receiving Data-----\n\n");

if(strlen(path[0])!=0)

{
while (1) {
clearBuf(net_buf);

nBytes = recvfrom(sockfddescriptor, net_buf, Default_Size,sendrecvflag,(struct
sockaddr*)&addr_con,&addrlen);

if (RecieveFile(net_buf, Default_Size)) {
break;
}
}
}

if(strlen(path[1])!=0)

{
while (1) {
clearBuf(net_buf);

nBytes = recvfrom(sockfddescriptor, net_buf, Default_Size,sendrecvflag,(struct
sockaddr*)&addr_con,&addrlen);

if (RecieveFile(net_buf, Default_Size)) {
break;
}
}
}

if(strlen(path[2])!=0)

{

```

```

while (1) {
clearBuf(net_buf);

nBytes = recvfrom(sockfddescriptor, net_buf, Default_Size,sendrecvflag,(struct
sockaddr*)&addr_con,&addrlen);

if (RecieveFile(net_buf, Default_Size)) {
break;
}
}

if((strlen(path[0])==0)&&(strlen(path[1])==0)&&(strlen(path[2])==0))
printf("File not found!\n");
}

else if(ch[0]==2)
{
int flag[1];
printf("\nEnter name of the file to be uploaded: ");
scanf("%s", net_buf);

sendto(sockfddescriptor, net_buf, Default_Size,sendrecvflag, (struct sockaddr*)&addr_con,addrlen);
fp = fopen(net_buf, "r");
printf("\nFile Name Received: %s\n", net_buf);
if (fp == NULL)
{
flag[0]=1;
printf("\n-----\n");
printf("\nFile not found!\n");
printf("\n-----\n");

sendto(sockfddescriptor, flag, sizeof(flag),sendrecvflag, (struct sockaddr*)&addr_con,addrlen);
continue;
}
}

```

```

}
else
{
flag[0]=0;
sendto(sockfddescriptor, flag, sizeof(flag),sendrecvflag, (struct sockaddr*)&addr_con,addrlen);
printf("\nFile Successfully opened!\n");
}
fseek(fp, 0L, SEEK_END);
long int fs[1];
fs[0]=ftell(fp);
fseek(fp, 0, SEEK_SET);
sendto(sockfddescriptor, fs, sizeof(fs), 0, (struct sockaddr*)&addr_con, addrlen);
printf("\n-----Data Sent-----\n\nUploading file....\n");
while (1) {
if (SendFile(fp, net_buf, Default_Size)) {
sendto(sockfddescriptor, net_buf, Default_Size,sendrecvflag,(struct sockaddr*)&addr_con, addrlen);
break;
}
sendto(sockfddescriptor, net_buf, Default_Size,sendrecvflag,(struct sockaddr*)&addr_con, addrlen);
clearBuf(net_buf);
}
if (fp != NULL)
fclose(fp);
printf("\nFile Successfully Uploaded.\n");
}

else if(ch[0]==3)
{
int flag[1];
printf("\nEnter name of the file to be Deleted: ");

```

```
scanf("%s", net_buf);

sendto(sockfddescriptor, net_buf, Default_Size,sendrecvflag, (struct sockaddr*)&addr_con,addrlen);

nBytes = recvfrom(sockfddescriptor, flag, sizeof(flag),sendrecvflag,(struct
sockaddr*)&addr_con,&addrlen);

if(flag[0]==1)
{
printf("\n-----\n");
printf("\nFile Successfully Deleted!\n");
}
if(flag[0]==0)
{
printf("\n-----\n");
printf("\nFile not found!\n");
}
}
else if(ch[0]==4)
exit(0);
else
printf("\nEnter correct choice.\n");
printf("\n-----\n");
}
return 0;
}
```

Server Code-

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#define IP_PROTOCOL 0
#define PORT_NO 15050
#define Default_size 1000
#define Key 88
#define sendrecvflag 0
#define nofile "File Not Found!"
#define BLOCK_SIZE 5000

void clearBuf(char* b) {
    int i;
    for (i = 0; i < sizeof(b); i++)
        b[i] = '\0';
}

char Cipher(char ch)
{
    return ch^Key;
}

int s1=0,s2=0,s3=0;
```

```

int RecieveFile1(char* buf, int s, FILE* smf, FILE* smfb)
{
    int i, size;
    char ch;
    for (i = 0; i < s; i++) {
        ch = buf[i];
        size = s;
        ch = Cipher(ch);
        if (ch == EOF)
            return 1;
        else
        {
            putc(ch, smf);
            putc(ch, smfb);
        }
    }
    return 0;
}

int RecieveFile(char* buf, int s, char *fl1, char* fl1b, char *fl2, char* fl2b, char *fl3, char* fl3b, char*
mas, char* masb)
{
    FILE* fp; FILE* fpb;
    FILE* ms; FILE* msb;
    int i, size;
    char ch;
    for (i = 0; i < s; i++) {
        ch = buf[i];
        size = s;
        ch = Cipher(ch);
        if (ch == EOF)

```

```
    return 1;
else
{
    ms=fopen(mas,"a");
    msb=fopen(masb,"a");
    if(s1<BLOCK_SIZE)
    {
        fp=fopen(fl1,"a");
        putc(ch,fp);
        fpb=fopen(fl1b,"a");
        putc(ch,fpb);
        if(s1==0)
        {
            fprintf(ms,"%s\n",fl1);
            fprintf(msb,"%s\n",fl1b);
        }
        s1++;
    }
    else if(s2<BLOCK_SIZE)
    {
        fp=fopen(fl2,"a");
        putc(ch,fp);
        fpb=fopen(fl2b,"a");
        putc(ch,fpb);
        if(s2==0)
        {
            fprintf(ms,"%s\n",fl2);
            fprintf(msb,"%s\n",fl2b);
        }
        s2++;
    }
}
```



```

    }
    else
    {
        fp=fopen(fl3,"a");
        putc(ch,fp);
        fpb=fopen(fl3b,"a");
        putc(ch,fpb);
        if(s3==0)
        {
            fprintf(ms,"%s\n",fl3);
            fprintf(msb,"%s\n",fl3b);
            s3++;
        }
    }
    fclose(fp);
    fclose(fpb);
    fclose(ms);
    fclose(msb);
}
}
return 0;
}

int SendFile(FILE* fp, char* buf, int s)
{
    int i, len;
    if (fp == NULL) {
        strcpy(buf, nofile);
        len = strlen(nofile);
        buf[len] = EOF;
        for (i = 0; i <= len; i++)

```

```

    buf[i] = Cipher(buf[i]);
    return 1;
}

char ch, ch2;
for (i = 0; i < s; i++) {
    ch = fgetc(fp);
    ch2 = Cipher(ch);
    buf[i] = ch2;
    if (ch == EOF)
        return 1;
}
return 0;
}

int main() {
    int sockfddescriptor, nBytes, ch[1];
    long int fs[1];
    struct sockaddr_in addr_con;
    int addrlen = sizeof(addr_con);
    addr_con.sin_family = AF_INET;
    addr_con.sin_port = htons(PORT_NO);
    addr_con.sin_addr.s_addr = INADDR_ANY;
    char net_buf[Default_size];
    FILE* fp; FILE* smf; FILE* smfb;
    sockfddescriptor = socket(AF_INET, SOCK_DGRAM, IP_PROTOCOL);
    if (sockfddescriptor < 0)
        printf("\nfile descriptor not received!!\n");
    else
        printf("\nfile descriptor %d received\n", sockfddescriptor);
    if (bind(sockfddescriptor, (struct sockaddr*)&addr_con, sizeof(addr_con)) == 0)
        printf("\nSuccessfully binded!\n");

```

```

else
printf("\nBinding Failed!\n");
while (1) {
printf("\n-----START-----\n");
printf("\nWaiting for choice...");
recvfrom(sockfddescriptor, ch, sizeof(ch), 0, (struct sockaddr*)&addr_con, &addrlen);
printf("\nChoice is: %d",ch[0]);
if(ch[0]==1)
{
printf("\nWaiting for file name...\n");
clearBuf(net_buf);

nBytes = recvfrom(sockfddescriptor, net_buf,Default_size, sendrecvflag,(struct sockaddr*)&addr_con,
&addrlen);

char masp[]="MASTER/";
int as=strlen(net_buf);
int bs=strlen(masp);
int s=as+bs;
char* mas=(char*)malloc(s*sizeof(char));
strcpy(mas,strcat(masp,net_buf));
fp = fopen(mas, "r");
printf("\nFile Name Received: %s\n", net_buf);
if (fp == NULL)
{
char path[3][200];
clearBuf(path[0]); clearBuf(path[1]); clearBuf(path[2]);
printf("\nFile open failed!\n");
sendto(sockfddescriptor, path, sizeof(path),sendrecvflag,(struct sockaddr*)&addr_con, addrlen);
}
else

```

```

{
    printf("\nFile Successfully opened!\n");
    char path[3][200];
    clearBuf(path[0]); clearBuf(path[1]); clearBuf(path[2]);
    fscanf(fp, "%s\n%s\n%s", path[0],path[1],path[2]);

    sendto(sockfddescriptor, path, sizeof(path),sendrecvflag,(struct sockaddr*)&addr_con, addrlen);

    if(strlen(path[0])!=0)
    {
        printf("%s",path[0]);
        FILE* fps;
        if (fp == NULL)
            printf("\nFile open failed!\n");
        else
        {
            printf("\nFile opened Successfully opened by Master!\n");
            fps = fopen(path[0], "r");
        }
        while (1) {
            if (SendFile(fps, net_buf, Default_size)) {
                sendto(sockfddescriptor, net_buf, Default_size,sendrecvflag,(struct sockaddr*)&addr_con, addrlen);
                break;
            }
            sendto(sockfddescriptor, net_buf, Default_size,sendrecvflag,(struct sockaddr*)&addr_con, addrlen);
            clearBuf(net_buf);
        }
        clearBuf(path[0]);
        if(fps!=NULL)
            fclose(fps);
    }
}

```

```
}  
if(strlen(path[1])!=0)  
{  
printf("%s",path[1]);  
FILE* fps;  
if (fp == NULL)  
printf("\nFile open failed!\n");  
else  
{  
printf("\nFile opened Successfully opened by Master!\n");  
fps = fopen(path[1], "r");  
}  
while (1) {  
if (SendFile(fps, net_buf, Default_size)) {  
sendto(sockfddescriptor, net_buf, Default_size,sendrecvflag,(struct sockaddr*)&addr_con, addrlen);  
break;  
}  
sendto(sockfddescriptor, net_buf, Default_size,sendrecvflag,(struct sockaddr*)&addr_con, addrlen);  
clearBuf(net_buf);  
}  
clearBuf(path[1]);  
if(fps!=NULL)  
fclose(fps);  
}  
if(strlen(path[2])!=0)  
{  
printf("%s",path[2]);  
FILE* fps;  
if (fp == NULL)  
printf("\nFile open failed!\n");
```

```

else
{
printf("\nFile opened Successfully opened by Master!\n");
fps = fopen(path[2], "r");
}
while (1) {
if (SendFile(fps, net_buf, Default_size)) {
sendto(sockfddescriptor, net_buf, Default_size, sendrecvflag, (struct sockaddr*)&addr_con, addrlen);
break;
}
sendto(sockfddescriptor, net_buf, Default_size, sendrecvflag, (struct sockaddr*)&addr_con, addrlen);
clearBuf(net_buf);
}
clearBuf(path[2]);
if(fps!=NULL)
fclose(fps);
}
if (fp != NULL)
fclose(fp);
}
}
if(ch[0]==2)
{
int flag[1];
printf("\nWaiting for file name...\n");
clearBuf(net_buf);

nBytes = recvfrom(sockfddescriptor, net_buf, Default_size, sendrecvflag, (struct sockaddr*)&addr_con,
&addrlen);

nBytes = recvfrom(sockfddescriptor, flag, sizeof(flag), sendrecvflag, (struct sockaddr*)&addr_con,
&addrlen);

if(flag[0]==1)

```

```

{
printf("\nFile not Found!\n");
continue;
}
else
printf("\nFile successfully Found!\n");
nBytes=recvfrom(sockfddescriptor, fs, sizeof(fs), 0, (struct sockaddr*)&addr_con, &addrlen);
printf("File size %ld bytes\n",fs[0]);
char fn[255];
char temp[Default_size];
strcpy(fn,net_buf);
printf("File name is: %s\n",fn);

char ch;char* t;
printf("\n-----Data Received-----\n\nUploading file....\n");
if(fs[0]>BLOCK_SIZE)
{
char fl1p[]="SERVER1/";
int as=strlen(fn);
int bs=strlen(fl1p);
int s=as+bs;
char* fl1=(char*)malloc(s*sizeof(char));
strcpy(fl1, strcat(fl1p,fn));

char fl1bp[]="SERVER1_BUP/";
bs=strlen(fl1bp);
s=as+bs;
char* fl1b=(char*)malloc(s*sizeof(char));
strcpy(fl1b, strcat(fl1bp,fn));

```

```
char fl2p[]="SERVER2/";  
bs=strlen(fl2p);  
s=as+bs;  
char* fl2=(char*)malloc(s*sizeof(char));  
strcpy(fl2, strcat(fl2p,fn));
```

```
char fl2bp[]="SERVER2_BUP/";  
bs=strlen(fl2bp);  
s=as+bs;  
char* fl2b=(char*)malloc(s*sizeof(char));  
strcpy(fl2b, strcat(fl2bp,fn));
```

```
char fl3p[]="SERVER3/";  
bs=strlen(fl3p);  
s=as+bs;  
char* fl3=(char*)malloc(s*sizeof(char));  
strcpy(fl3, strcat(fl3p,fn));
```

```
char fl3bp[]="SERVER3_BUP/";  
bs=strlen(fl3bp);  
s=as+bs;  
char* fl3b=(char*)malloc(s*sizeof(char));  
strcpy(fl3b, strcat(fl3bp,fn));
```

```
char masp[]="MASTER/";  
bs=strlen(masp);  
s=as+bs;  
char* mas=(char*)malloc(s*sizeof(char));  
strcpy(mas, strcat(masp,fn));
```



```

char masbp[]="MASTER_BUP/";
bs=strlen(masbp);
s=as+bs;
char* masb=(char*)malloc(s*sizeof(char));
strcpy(masb, strcat(masbp,fn));

printf("\nLarge File\n");
while (1) {
clearBuf(net_buf);

nBytes = recvfrom(sockfddescriptor, net_buf, Default_size,sendrecvflag,(struct
sockaddr*)&addr_con,&addrlen);

if (RecieveFile(net_buf, Default_size,fl1,fl1b,fl2,fl2b,fl3,fl3b,mass,massb)) {
break;
}
}
}
else
{
printf("\nSmall File\n");
char sf1[]="SERVER1/SMALLFILE/";
char* sf=(char*)malloc((strlen(sf1)+strlen(fn))*sizeof(char));
strcpy(sf, strcat(sf1,fn));

char sf1b[]="SERVER1_BUP/SMALLFILE_BUP/";
char* sfb=(char*)malloc((strlen(sf1b)+strlen(fn))*sizeof(char));
strcpy(sfb, strcat(sf1b,fn));

char p[]="MASTER/";
char* mas=(char*)malloc((strlen(p)+strlen(net_buf))*sizeof(char));
printf("%s",net_buf);

```

```

strcpy(mas, strcat(p, net_buf));
char pb[]="MASTER_BUP/";
char* masb=(char*)malloc((strlen(pb)+strlen(net_buf))*sizeof(char));
strcpy(masb, strcat(pb, net_buf));
FILE* ms;
FILE* msb;
ms=fopen(mas, "w");
fprintf(ms, "%s\n", sf);
msb=fopen(masb, "w");
fprintf(msb, "%s\n", sfb);
fclose(ms);
fclose(msb);
smf=fopen(sf, "w");
smfb=fopen(sfb, "w");
while (1) {
clearBuf(net_buf);

nBytes = recvfrom(sockfdescriptor, net_buf, Default_size, sendrecvflag, (struct
sockaddr*)&addr_con, &addrlen);

if (RecieveFile1(net_buf, Default_size, smf, smfb)) {
break;
}
}

if(smf!=NULL)
fclose(smf);
if(smfb!=NULL)
fclose(smfb);
}

printf("\nFile Successfully Uploaded.\n\n-----Data Received-----\n");
s1=0;
s2=0;

```

```

s3=0;
}
if(ch[0]==3)
{
int flag[1];
printf("\nWaiting for file name to delete...\n");
clearBuf(net_buf);

nBytes = recvfrom(sockfddescriptor, net_buf,Default_size, sendrecvflag,(struct sockaddr*)&addr_con,
&addrlen);

char p[]="MASTER/";
char* mas=(char*)malloc((strlen(p)+strlen(net_buf))*sizeof(char));
strcpy(mas, strcat(p, net_buf));

char pb[]="MASTER_BUP/";
char* masb=(char*)malloc((strlen(pb)+strlen(net_buf))*sizeof(char));
strcpy(masb, strcat(pb, net_buf));

fp = fopen(mas, "r");
if (fp == NULL)
{
printf("\nFile Not Found!\n");
flag[0]=0;
}
else
{
FILE* fpb;
fpb = fopen(masb, "r");
printf("\nMaster opened Successfully!\n");
flag[0]=1;

char path[3][200];
clearBuf(path[0]); clearBuf(path[1]); clearBuf(path[2]);
fscanf(fp, "%s\n%s\n%s", path[0],path[1],path[2]);

```

```
if (remove(path[0]) == 0)
    printf("\nDeleted successfully from %s",path[0]);
else
    printf("\nUnable to delete the file %s",path[0]);
if (remove(path[1]) == 0)
    printf("\nDeleted successfully from %s",path[1]);
else
    printf("\nUnable to delete the file %s",path[1]);
if (remove(path[2]) == 0)
    printf("\nDeleted successfully from %s",path[2]);
else
    printf("\nUnable to delete the file %s",path[2]);

clearBuf(path[0]); clearBuf(path[1]); clearBuf(path[2]);
fscanf(fpb, "%s\n%s\n%s", path[0],path[1],path[2]);
if (remove(path[0]) == 0)
    printf("\nDeleted successfully from %s",path[0]);
else
    printf("\nUnable to delete the file %s",path[0]);
if (remove(path[1]) == 0)
    printf("\nDeleted successfully from %s",path[1]);
else
    printf("\nUnable to delete the file %s",path[1]);
if (remove(path[2]) == 0)
    printf("\nDeleted successfully from %s",path[2]);
else
    printf("\nUnable to delete the file %s",path[2]);
fclose(fpb);
}
if(fp!=NULL)
```

```

    fclose(fp);
    if(flag[0]==1)
    {
        if (remove(mas) == 0)
        {
            printf("\nDeleted successfully from %s",mas);
            flag[0]=1;
        }
    }
    else
    {
        printf("\nUnable to delete the file %s",mas);
        flag[0]=0;
    }
    if (remove(masb) == 0)
    {
        printf("\nDeleted successfully from %s",masb);
        flag[0]=1;
    }
    else
    {
        printf("\nUnable to delete the file %s",masb);
        flag[0]=0;
    }
}

sendto(sockfddescriptor, flag, sizeof(flag),sendrecvflag,(struct sockaddr*)&addr_con, addrlen);
}

printf("\n-----END-----\n");
}

return 0;
}

```

WORKING

Starting Server-

```
kt@kt-VirtualBox: ~/Desktop
kt@kt-VirtualBox:~$ cd Desktop
kt@kt-VirtualBox:~/Desktop$ gcc Server.c
kt@kt-VirtualBox:~/Desktop$ ./a.out

file descriptor 3 received

Successfully binded!

-----START-----
█
```

Connecting Client to Server-

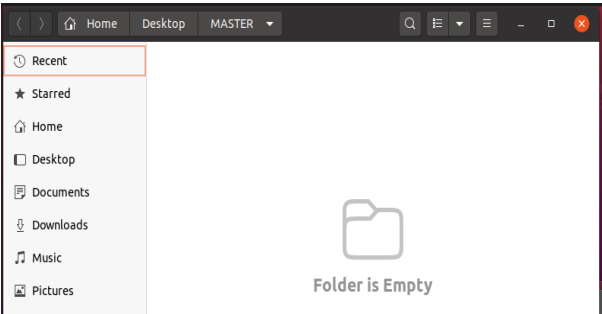
```
kt@kt-VirtualBox: ~/Desktop
kt@kt-VirtualBox:~$ cd Desktop
kt@kt-VirtualBox:~/Desktop$ gcc Client.c
kt@kt-VirtualBox:~/Desktop$ ./a.out

file descriptor 3 received

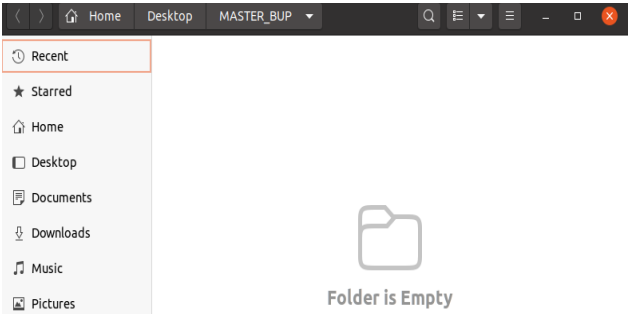
Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice:
```

Initial State of Master

MASTER

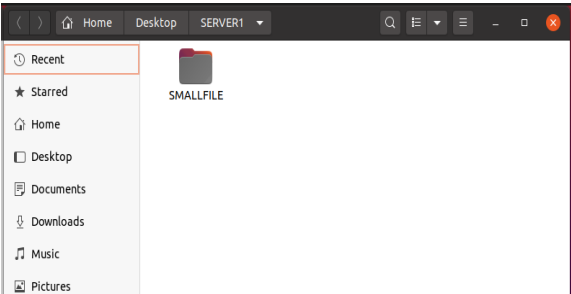


MASTER_BUP

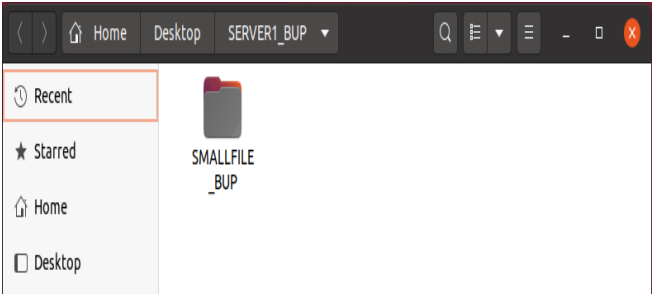


Initial State of Servers

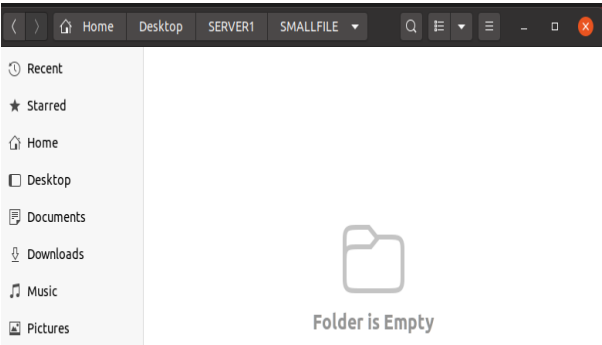
SERVER1



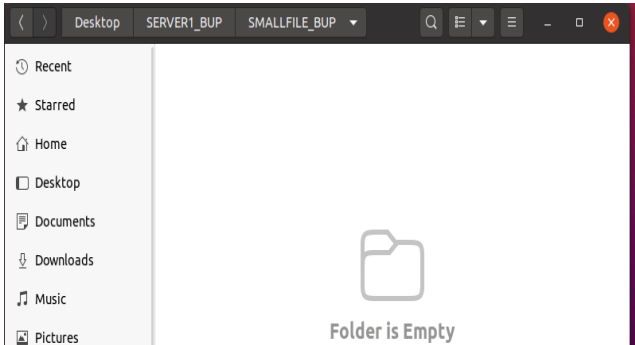
SERVER1_BUP-



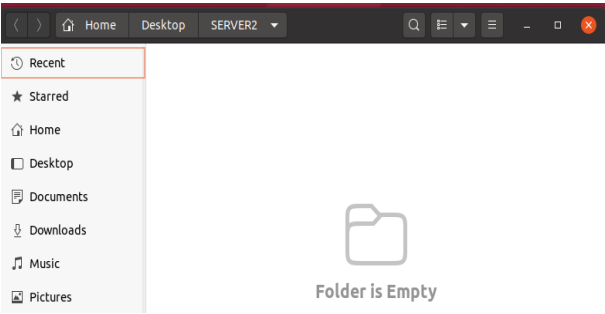
SMALLFILE Rack SERVER1



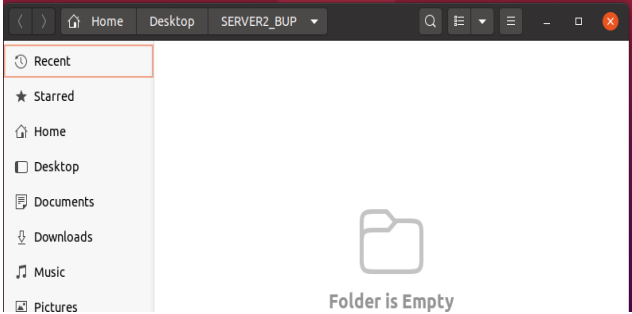
SMALLFILE_BUP Rack SERVER1_BUP



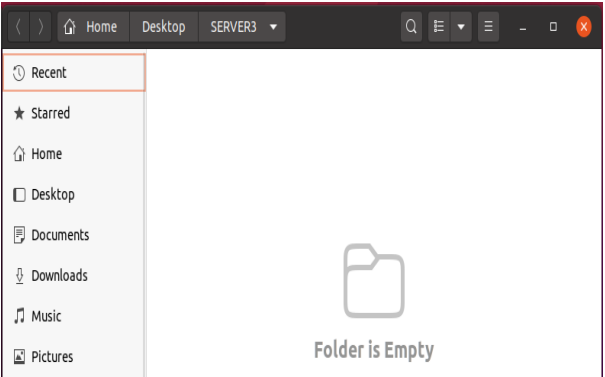
SERVER2



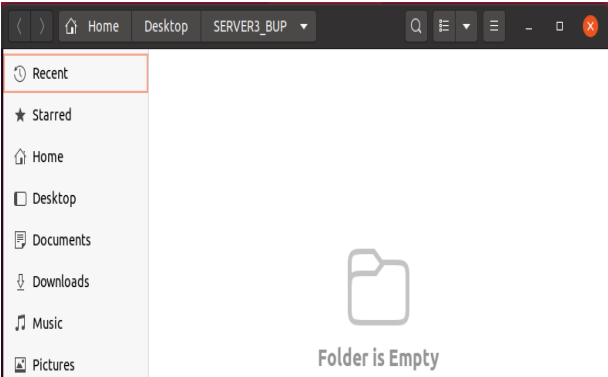
SERVER2_BUP



SERVER3

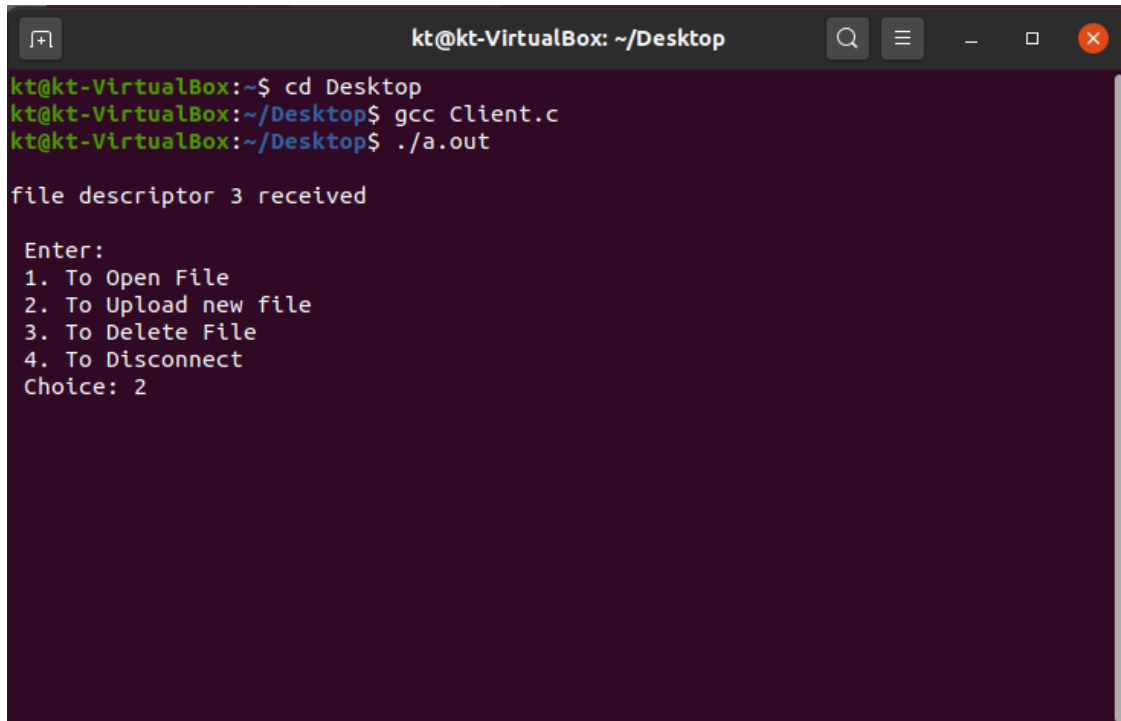


SERVER3_BUP



UPLOADING FILE (LARGE)-

- For uploading enter Choice 2

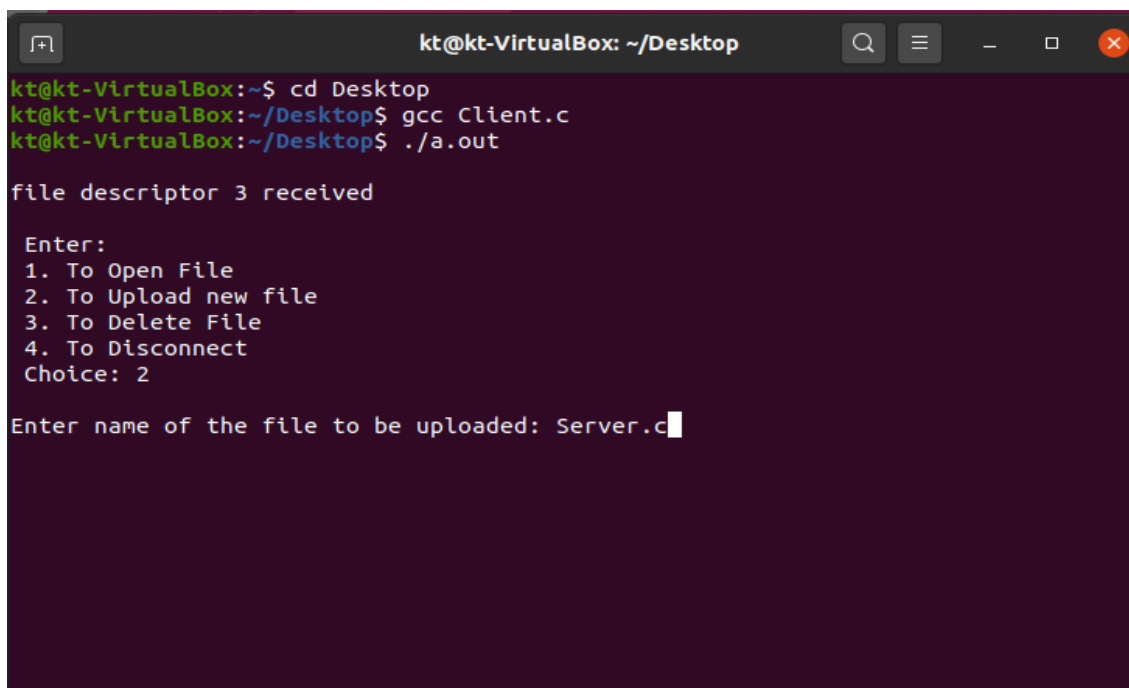


```
kt@kt-VirtualBox: ~/Desktop
kt@kt-VirtualBox:~$ cd Desktop
kt@kt-VirtualBox:~/Desktop$ gcc Client.c
kt@kt-VirtualBox:~/Desktop$ ./a.out

file descriptor 3 received

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 2
```

- Click on enter
- Now enter name of the text file to be uploaded



```
kt@kt-VirtualBox: ~/Desktop
kt@kt-VirtualBox:~$ cd Desktop
kt@kt-VirtualBox:~/Desktop$ gcc Client.c
kt@kt-VirtualBox:~/Desktop$ ./a.out

file descriptor 3 received

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 2

Enter name of the file to be uploaded: Server.c
```

- Click on enter
- The **client side** will show the message of successful uploading if file is present in the system.

```

kt@kt-VirtualBox: ~/Desktop
3. To Delete File
4. To Disconnect
Choice: 2

Enter name of the file to be uploaded: Server.c
File Name Received: Server.c
File Successfully opened!
-----Data Sent-----
Uploading file....
File Successfully Uploaded.
-----

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice:

```

- The **server side** will show the details of the uploaded file i.e. file size, file name, choice received, type of file i.e. Small file or large file.

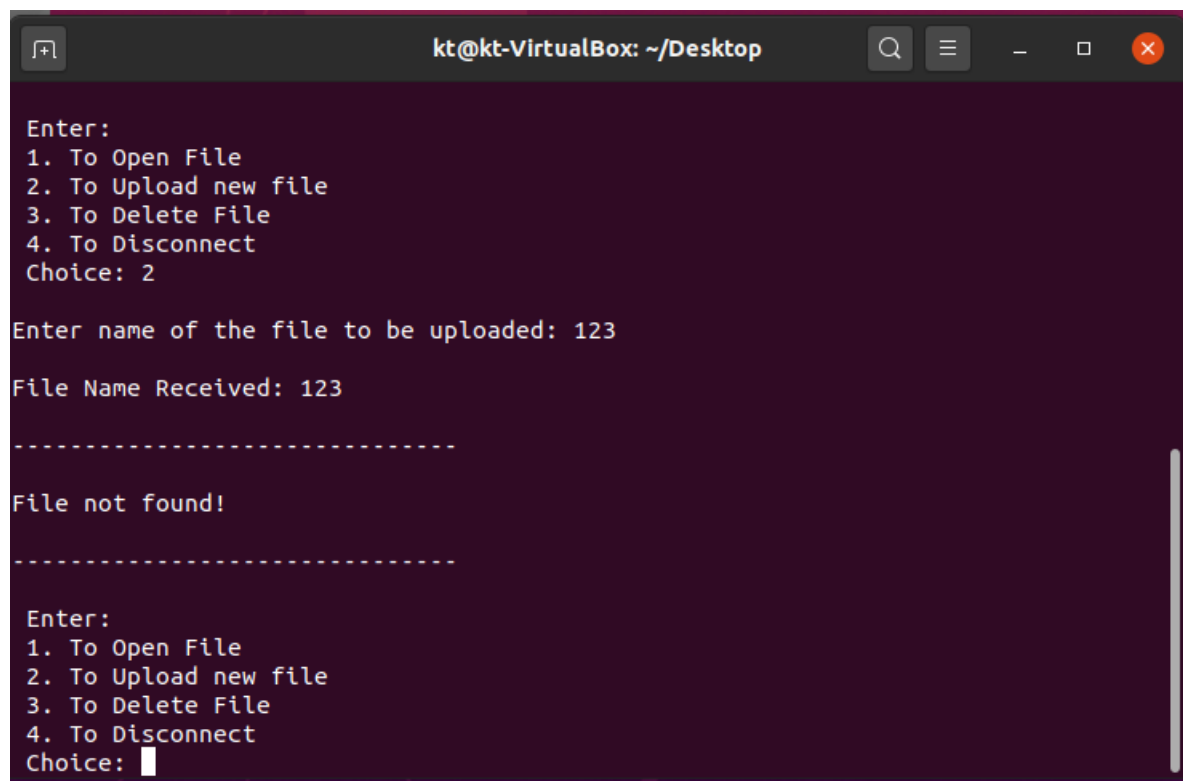
```

kt@kt-VirtualBox: ~/Desktop
Waiting for choice...
Choice is: 2
Waiting for file name...

File successfully Found!
File size 11169 bytes
File name is: Server.c
-----Data Received-----
Uploading file....
Large File
File Successfully Uploaded.
-----Data Received-----
-----END-----
-----START-----

```

- If we enter wrong file name the error message will be displayed in client side and server side as shown below



A terminal window titled 'kt@kt-VirtualBox: ~/Desktop' showing a menu-driven interface. The user selects option 2 to upload a file and enters '123' as the filename. The system responds with 'File not found!' and displays the menu again.

```
Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 2

Enter name of the file to be uploaded: 123

File Name Received: 123

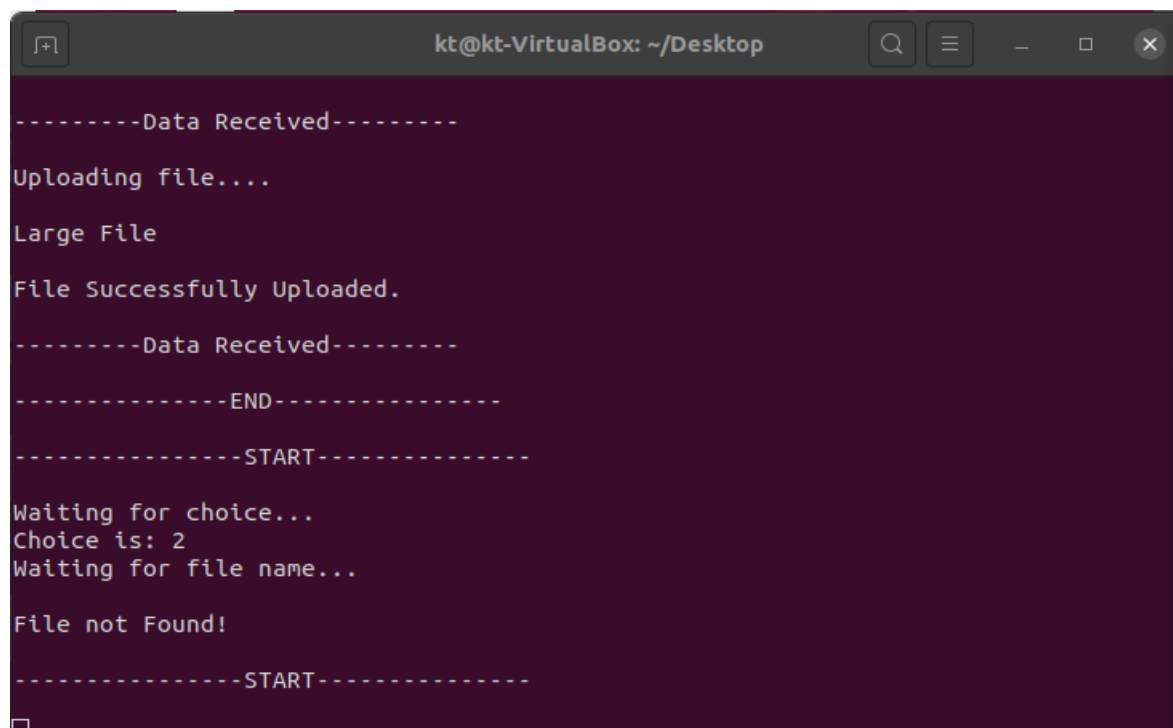
-----

File not found!

-----

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: █
```

Server Side-



A terminal window titled 'kt@kt-VirtualBox: ~/Desktop' showing server-side logs. It displays the receipt of data, an attempt to upload a 'Large File', and a 'File Successfully Uploaded.' message. After a series of status messages, it shows 'Waiting for choice...' and 'Choice is: 2', followed by 'Waiting for file name...' and 'File not Found!'.

```
-----Data Received-----
Uploading file....
Large File
File Successfully Uploaded.

-----Data Received-----
-----END-----
-----START-----

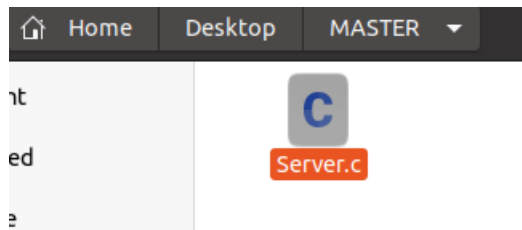
Waiting for choice...
Choice is: 2
Waiting for file name...

File not Found!

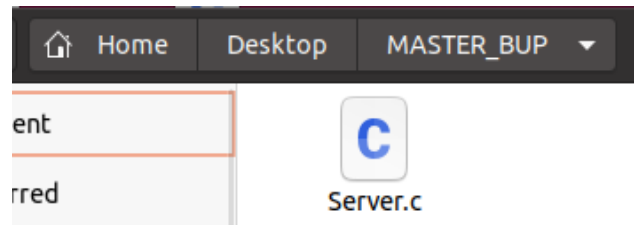
-----START-----
█
```

- File will be stored in 5kb blocks as shown below.

MASTER

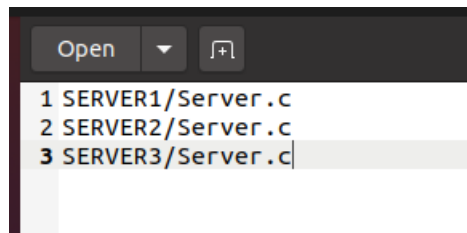


MASTER_BUP

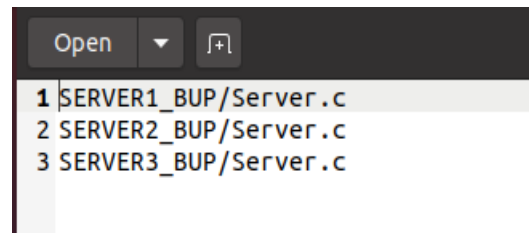


The master file will have following content

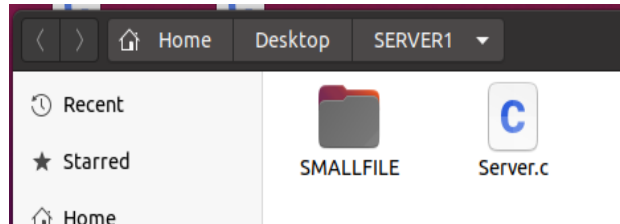
IN MASTER



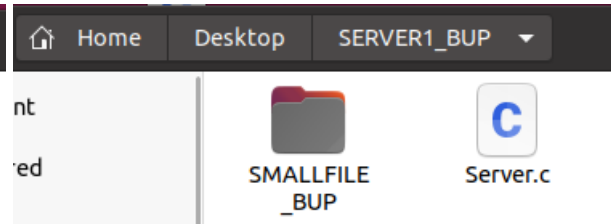
IN MASTER_BUP



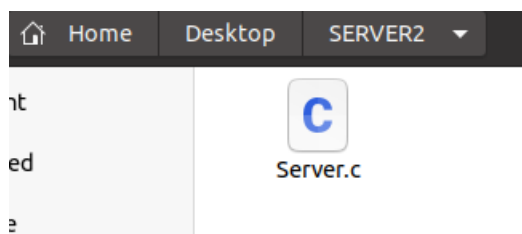
SERVER1 (5kb)



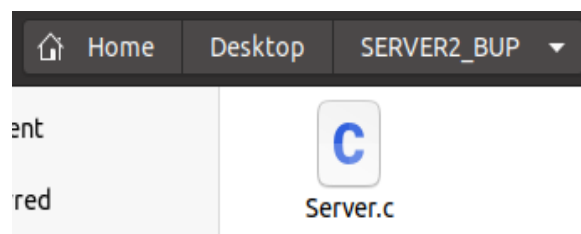
SERVER1_BUP (5kb)



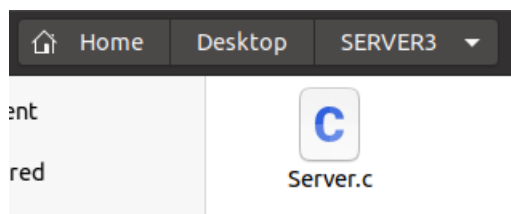
SERVER2 (5kb)



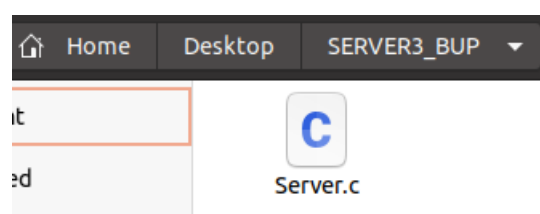
SERVER2_BUP (5kb)



SERVER3

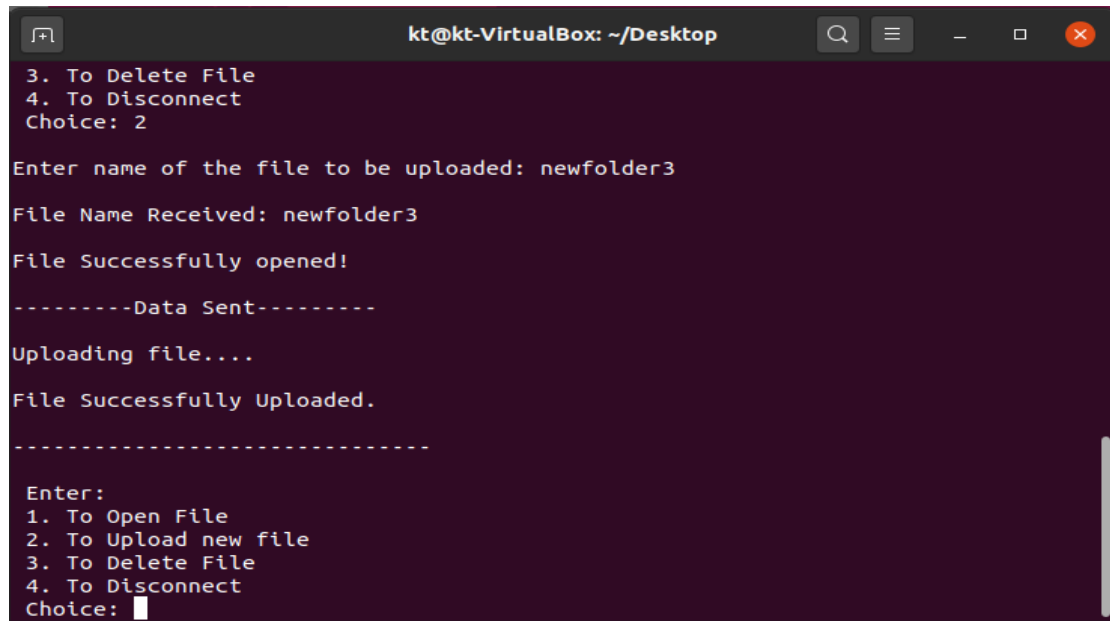


SERVER3_BUP



UPLOADING FILE (SMALL FILE)-

- For uploading enter Choice 2
- Click on enter
- Now enter name of the text file to be uploaded
- Click on enter
- The **client side** will show the message of successful uploading if file is present in the system.

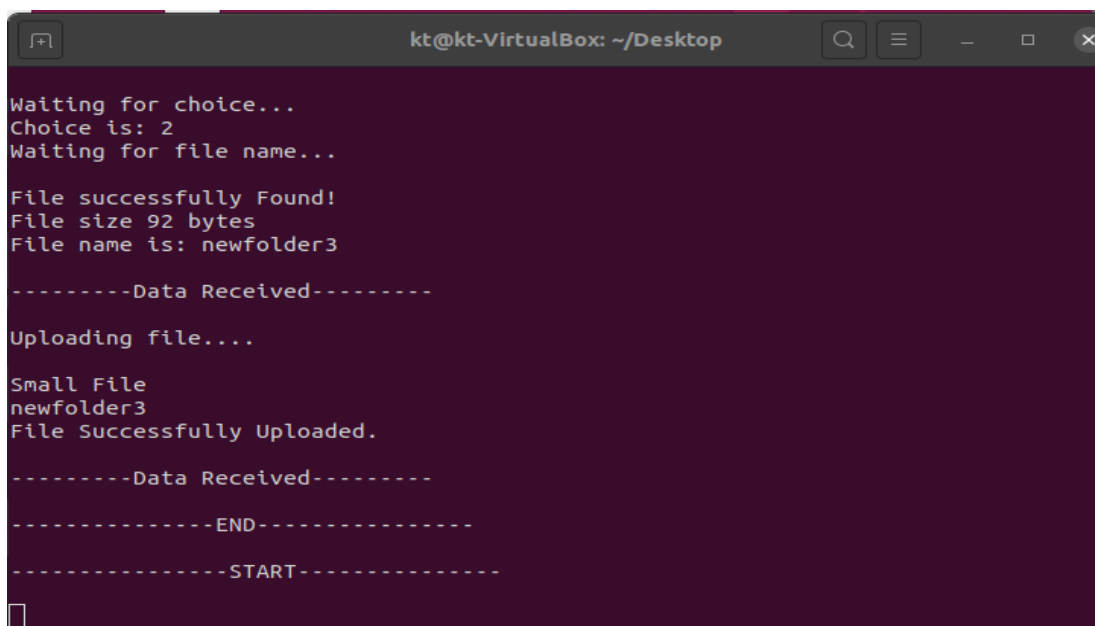


```
kt@kt-VirtualBox: ~/Desktop
3. To Delete File
4. To Disconnect
Choice: 2

Enter name of the file to be uploaded: newfolder3
File Name Received: newfolder3
File Successfully opened!
-----Data Sent-----
Uploading file....
File Successfully Uploaded.
-----

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 
```

- The **server side** will show the details of the uploaded file i.e. file size, file name, choice received, type of file i.e. Small file or large file.



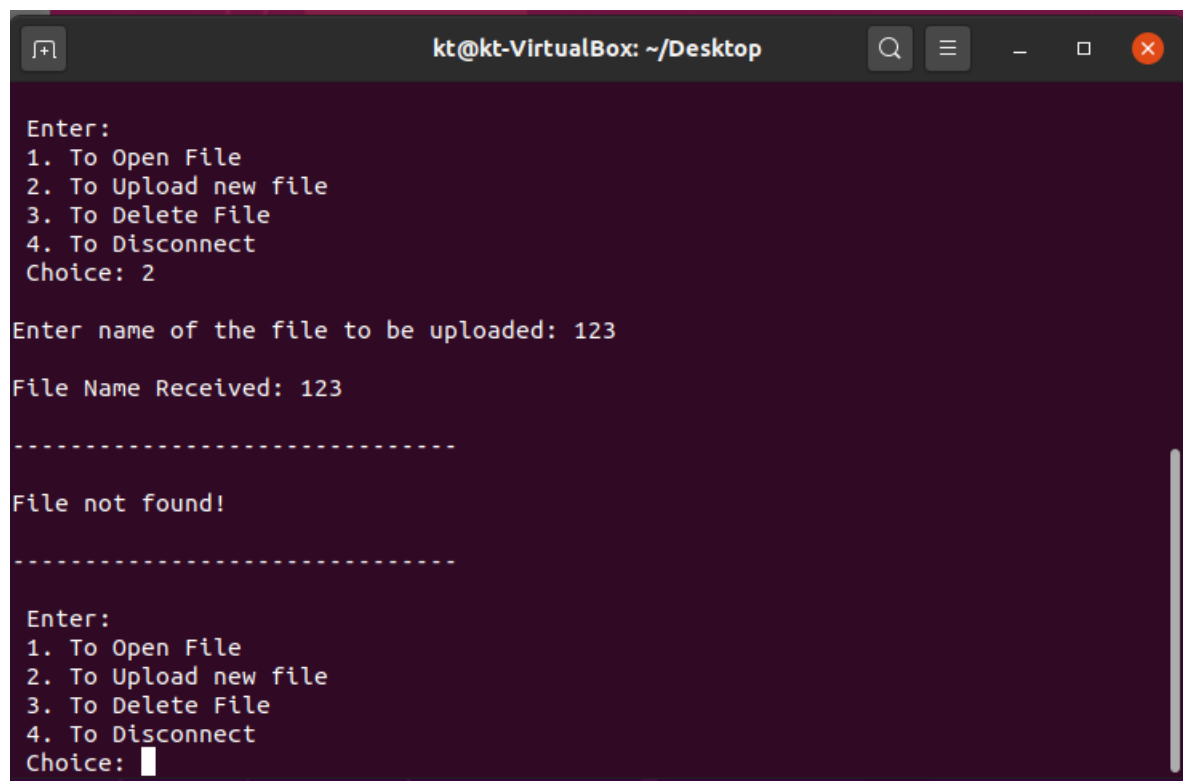
```
kt@kt-VirtualBox: ~/Desktop
Waiting for choice...
Choice is: 2
Waiting for file name...

File successfully Found!
File size 92 bytes
File name is: newfolder3
-----Data Received-----
Uploading file....

Small File
newfolder3
File Successfully Uploaded.
-----Data Received-----
-----END-----
-----START-----

```

- If we enter wrong file name the error message will be displayed in client side and server side as shown below



```
kt@kt-VirtualBox: ~/Desktop

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 2

Enter name of the file to be uploaded: 123

File Name Received: 123

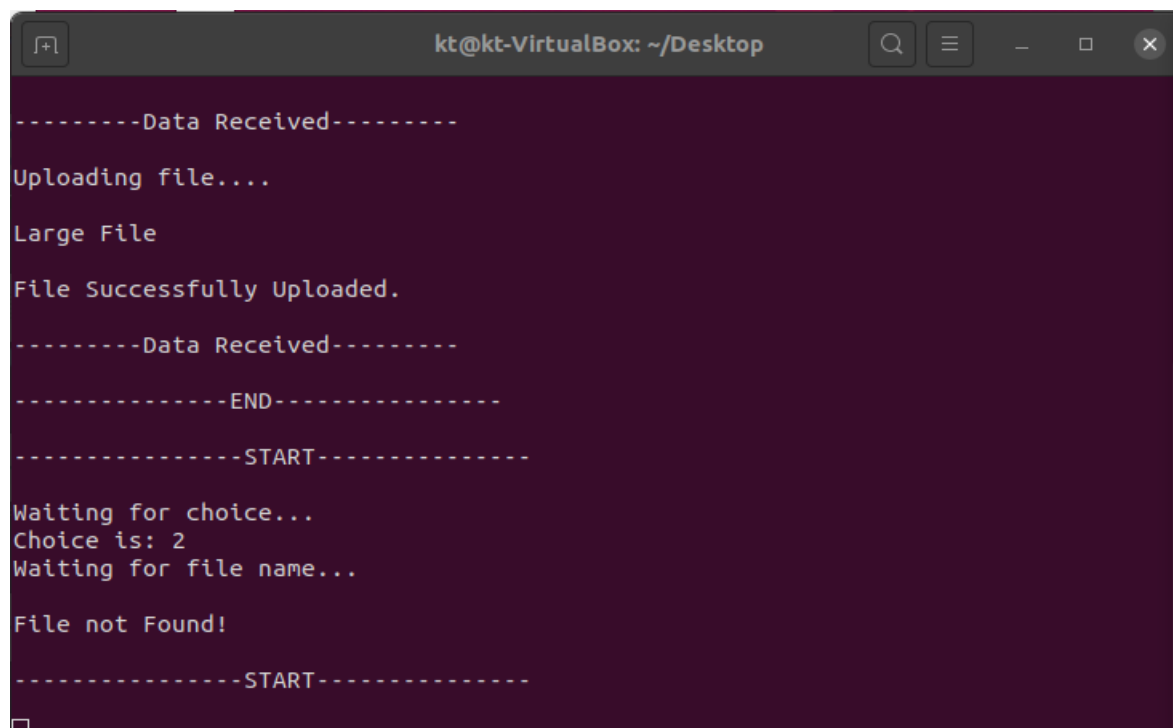
-----

File not found!

-----

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: █
```

Server Side-



```
kt@kt-VirtualBox: ~/Desktop

-----Data Received-----

Uploading file....

Large File

File Successfully Uploaded.

-----Data Received-----

-----END-----

-----START-----

Waiting for choice...
Choice is: 2
Waiting for file name...

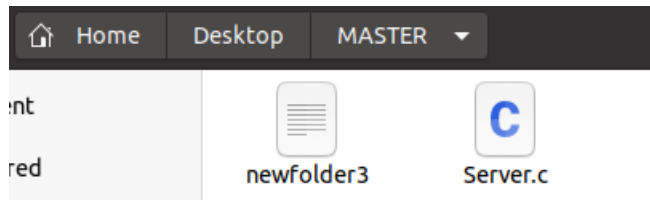
File not Found!

-----START-----

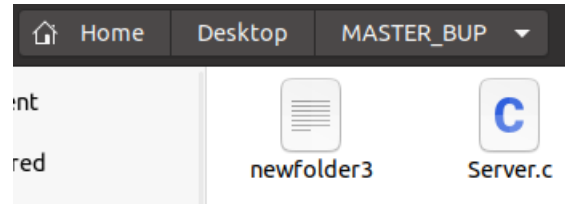
█
```

- File will be stored as shown below.

MASTER



MASTER_BUP

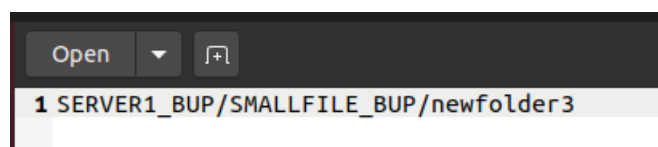


The master file will have following content

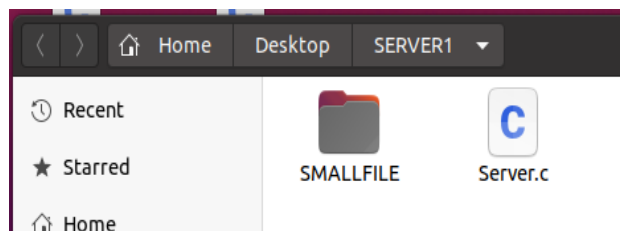
IN MASTER



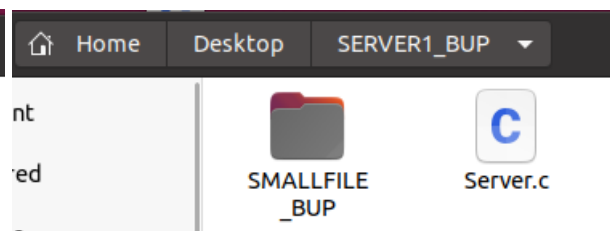
IN MASTER_BUP



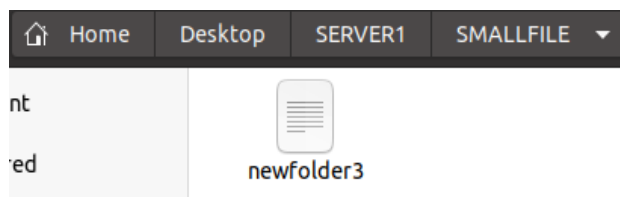
SERVER1



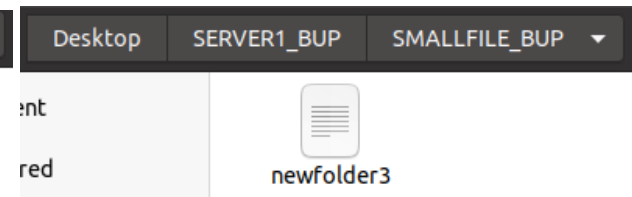
SERVER1_BUP



SMALLFILE

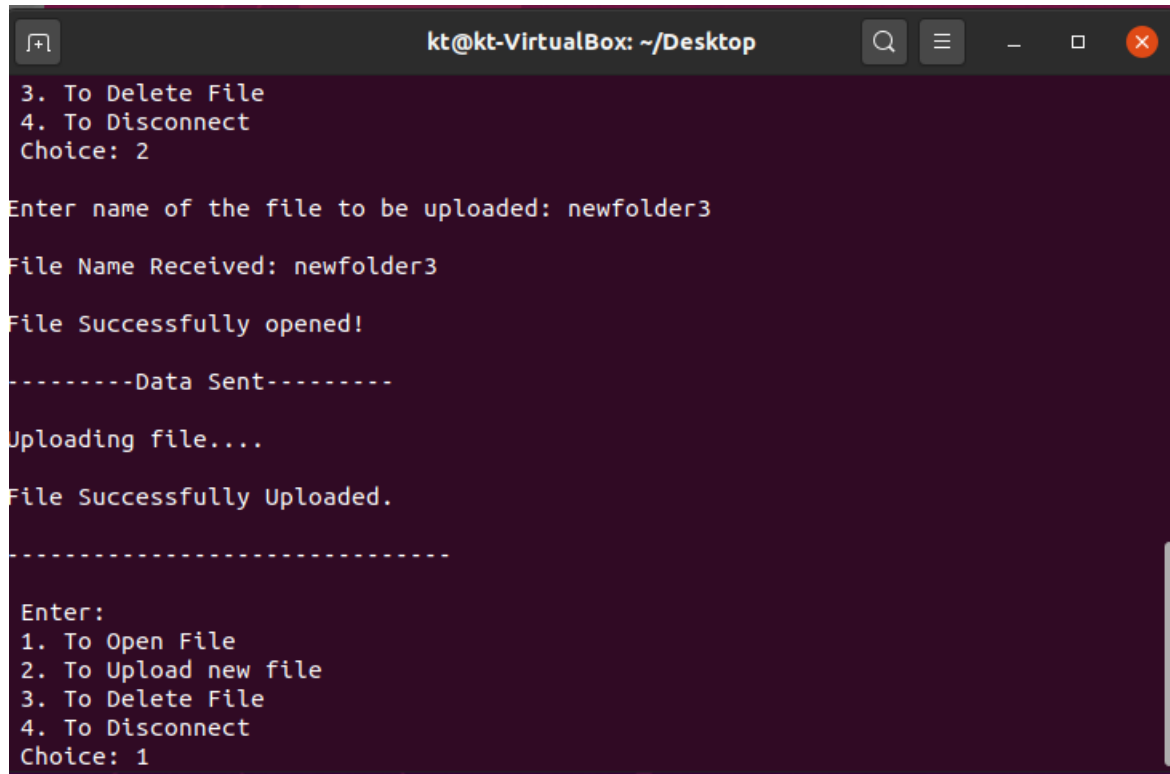


SMALLFILE_BUP



OPENING STORED FILE

- Enter 1 to open file
- Press enter

A terminal window titled 'kt@kt-VirtualBox: ~/Desktop' with standard window controls. The terminal shows a menu with options 3 (Delete File) and 4 (Disconnect). Choice 2 is entered. Then, the user is prompted for a file name to upload, and 'newfolder3' is entered. The terminal confirms the file name received and states the file was successfully opened. It then shows a progress bar for data sent, followed by 'Uploading file....' and 'File Successfully Uploaded.'. A separator line is shown. The menu is displayed again, and Choice 1 is entered.

```
3. To Delete File
4. To Disconnect
Choice: 2

Enter name of the file to be uploaded: newfolder3

File Name Received: newfolder3

File Successfully opened!

-----Data Sent-----

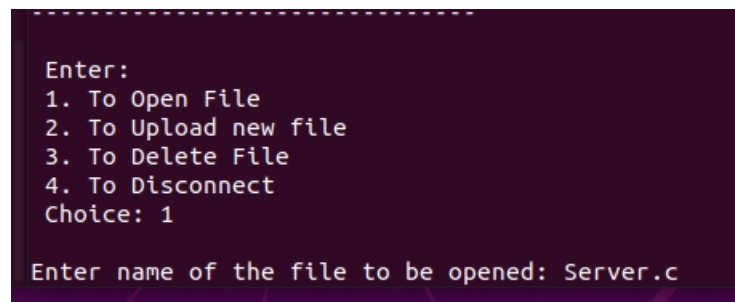
Uploading file....

File Successfully Uploaded.

-----

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 1
```

- Enter file name
- Press enter
- If entered file name is not present respective error message will be shown

A terminal window showing the menu from the previous screenshot. Choice 1 is entered, and the user is prompted for the file name to be opened. 'Server.c' is entered.

```
Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 1

Enter name of the file to be opened: Server.c
```


Complete file will be displayed on client side as shown below

```
kt@kt-VirtualBox: ~/Desktop
else
    printf("\nUnable to delete the file %s",path[1]);
if (remove(path[2]) == 0)
    printf("\nDeleted successfully from %s",path[2]);
else
    printf("\nUnable to delete the file %s",path[2]);
fclose(fpb);
}
if(fp!=NULL)
fclose(fp);
if(flag[0]==1)
{
if (remove(mas) == 0)
{
printf("\nDeleted successfully from %s",mas);
flag[0]=1;
}
else
{
printf("\nUnable to delete the file %s",mas);
flag[0]=0;
}
}
if (remove(masb) == 0)
{
printf("\nDeleted successfully from %s",masb);
flag[0]=1;
}
else
{
printf("\nUnable to delete the file %s",masb);
flag[0]=0;
}
}
sendto(sockfddescriptor, flag, sizeof(flag),sendrecvflag,(struct sockaddr*)&addr_con, addrlen);
}
printf("\n-----END-----\n");
}
return 0;
}

-----

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: █
```

Server Side will show message shown below

```
kt@kt-VirtualBox: ~/Desktop
-----END-----
-----START-----

Waiting for choice...
Choice is: 1
Waiting for file name...
File Name Received: Server.c

File Successfully opened!
SERVER1/Server.c
File opened Successfully opened by Master!
SERVER2/Server.c
File opened Successfully opened by Master!
SERVER3/Server.c
File opened Successfully opened by Master!

-----END-----
-----START-----
```

SIMILARLY SMALL FILE WILL BE OPENED AS SHOWN BELOW

CLIENT SIDE-

```
kt@kt-VirtualBox: ~/Desktop
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 1

Enter name of the file to be opened: newfolder3

-----Receiving Data-----

karunesh tripathi
19bce0880
vellore institute of technology
fall semester
operating system

-----

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: █
```

SERVER SIDE-

```
kt@kt-VirtualBox: ~/Desktop
SERVER2/Server.c
File opened Successfully opened by Master!
SERVER3/Server.c
File opened Successfully opened by Master!

-----END-----

-----START-----

Waiting for choice...
Choice is: 1
Waiting for file name...

File Name Received: newfolder3

File Successfully opened!
SERVER1/SMALLFILE/newfolder3
File opened Successfully opened by Master!

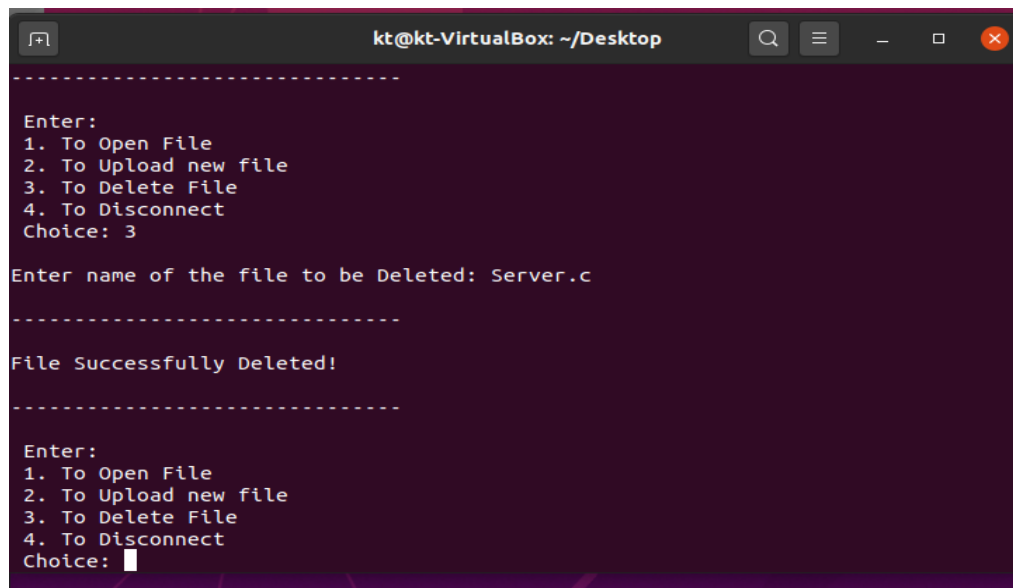
-----END-----

-----START-----
```

DELETE FILE

- For uploading enter Choice 3
- Click on enter
- Now enter name of the text file to be deleted
- Click on enter
- The **client side** will show the message of successful deletion if file is present in the system else respective error message will be shown

Client side-



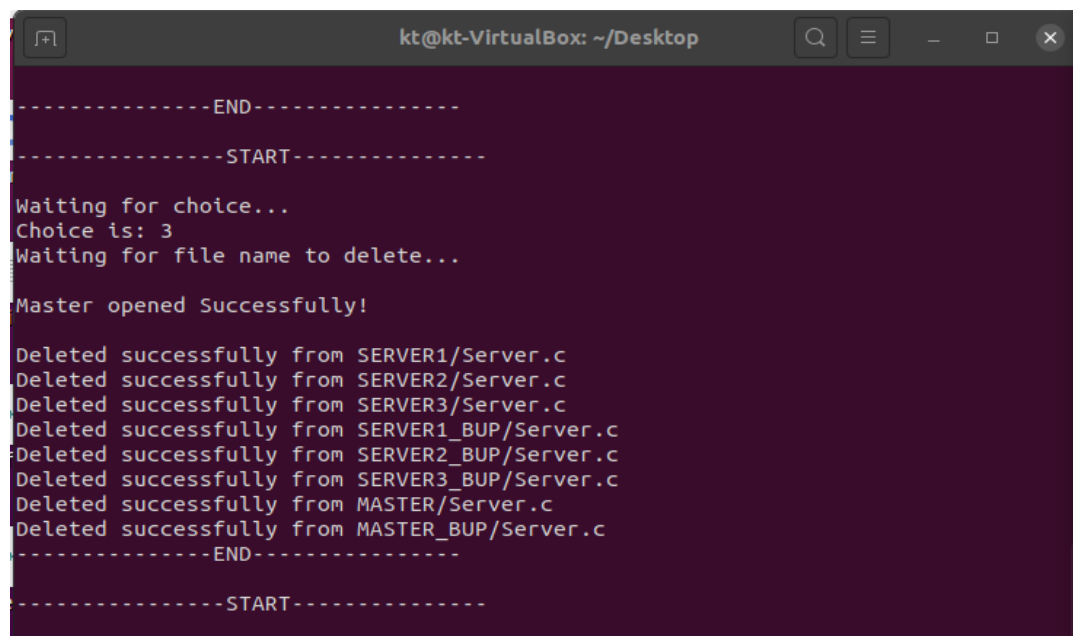
```
kt@kt-VirtualBox: ~/Desktop
-----
Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 3

Enter name of the file to be Deleted: Server.c

-----
File Successfully Deleted!

-----
Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: █
```

Server side-

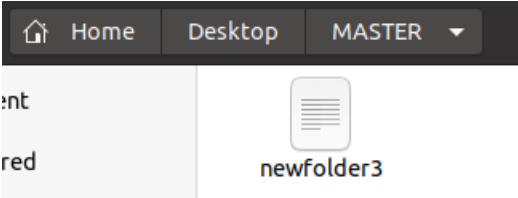


```
kt@kt-VirtualBox: ~/Desktop
-----END-----
-----START-----
Waiting for choice...
Choice is: 3
Waiting for file name to delete...
Master opened Successfully!

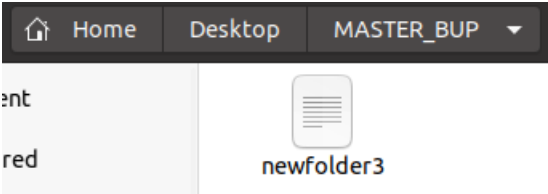
Deleted successfully from SERVER1/Server.c
Deleted successfully from SERVER2/Server.c
Deleted successfully from SERVER3/Server.c
Deleted successfully from SERVER1_BUP/Server.c
Deleted successfully from SERVER2_BUP/Server.c
Deleted successfully from SERVER3_BUP/Server.c
Deleted successfully from MASTER/Server.c
Deleted successfully from MASTER_BUP/Server.c
-----END-----
-----START-----
```

SERVER STATUS AFTER DELETION (LARGE FILE)

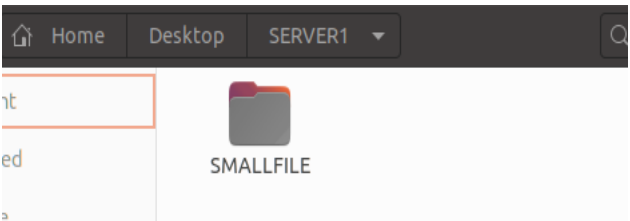
MASTER



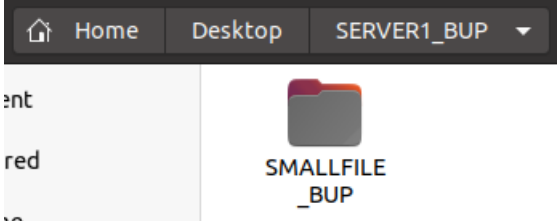
MASTER_BUP



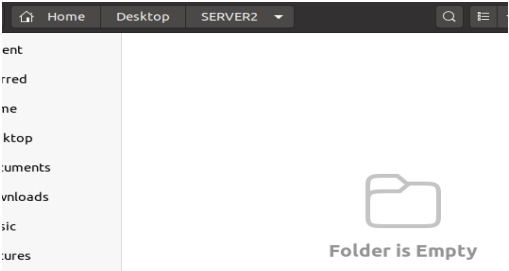
SERVER1



SERVER1_BUP



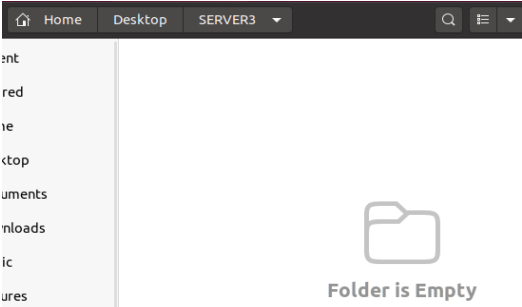
SERVER2



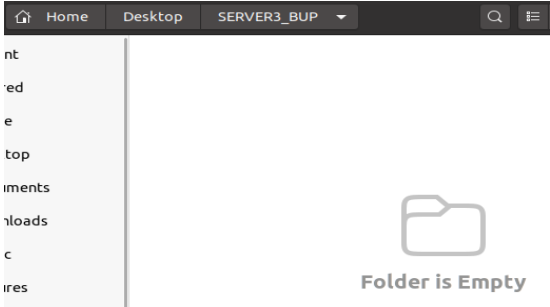
SERVER2_BUP



SERVER3



SERVER3_BUP



SIMILARLY DELETING SMALL FILE

CLIENT SIDE

```
kt@kt-VirtualBox: ~/Desktop
-----
Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 3

Enter name of the file to be Deleted: newfolder3

-----
File Successfully Deleted!

-----
Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: █
```

SERVER SIDE

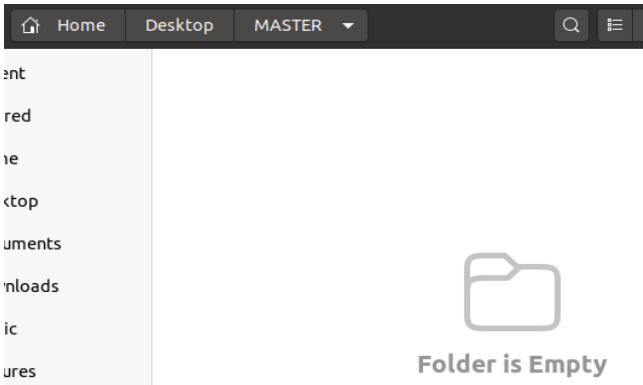
```
-----START-----
Waiting for choice...
Choice is: 3
Waiting for file name to delete...

Master opened Successfully!

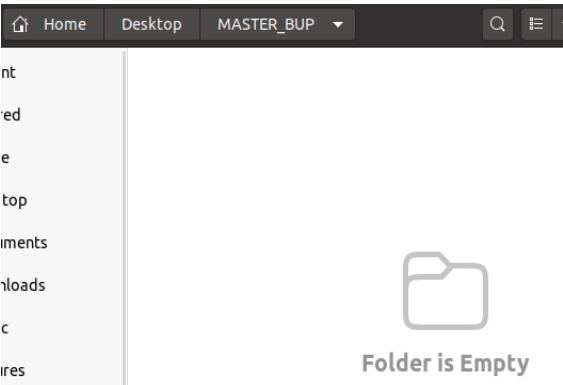
Deleted successfully from SERVER1/SMALLFILE/newfolder3
```

SERVER STATUS AFTER DELETION (SMALL FILE)

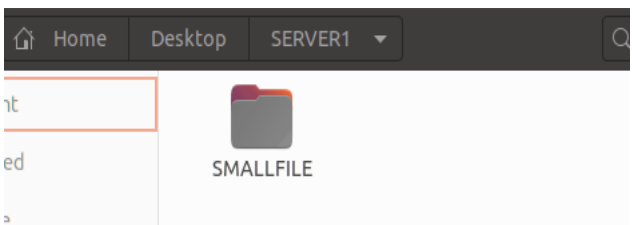
MASTER



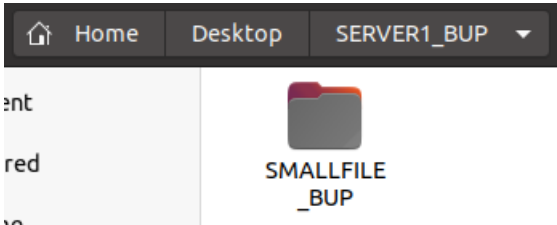
MASTER_BUP



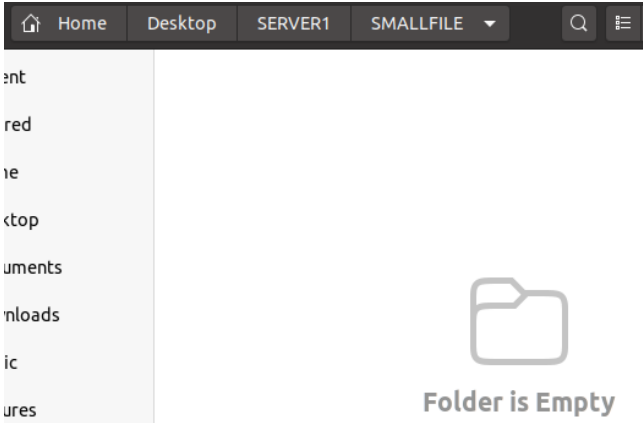
SERVER1



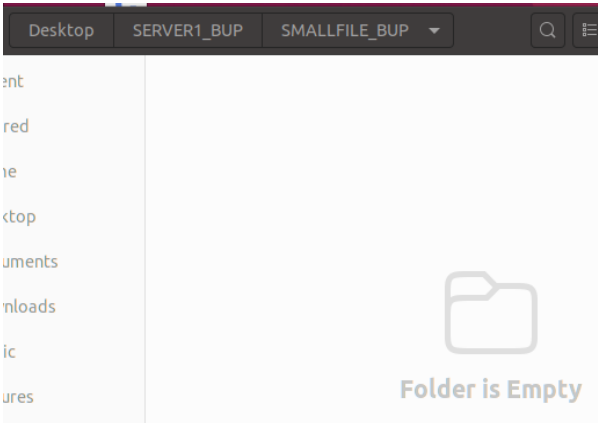
SERVER1_BUP



SMALLFILE

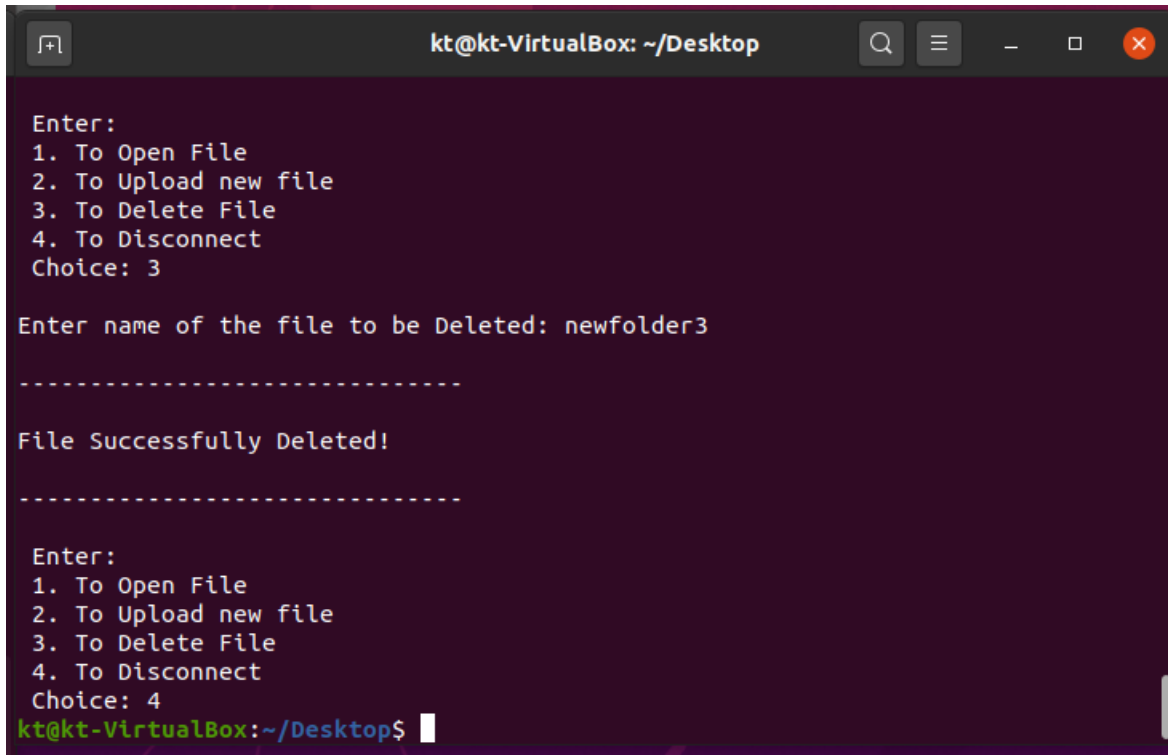


SMALLFILE_BUP



CLIENT EXITING

- Enter choice 4
- Press enter



A screenshot of a terminal window titled "kt@kt-VirtualBox: ~/Desktop". The terminal has a dark purple background and white text. It displays a menu with four options: "1. To Open File", "2. To Upload new file", "3. To Delete File", and "4. To Disconnect". The user has entered "Choice: 3", and the terminal prompts "Enter name of the file to be Deleted: newfolder3". After a separator line of dashes, it displays "File Successfully Deleted!". Another separator line follows. The menu is shown again, and the user enters "Choice: 4". The prompt "kt@kt-VirtualBox:~/Desktop\$" is visible at the bottom.

```
kt@kt-VirtualBox: ~/Desktop
Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 3

Enter name of the file to be Deleted: newfolder3

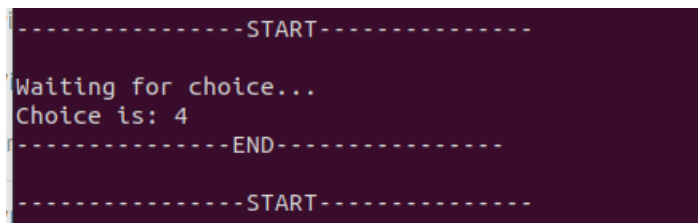
-----

File Successfully Deleted!

-----

Enter:
1. To Open File
2. To Upload new file
3. To Delete File
4. To Disconnect
Choice: 4
kt@kt-VirtualBox:~/Desktop$
```

SERVER SIDE-



A screenshot of a terminal window with a dark purple background and white text. It shows the server-side status with a separator line of dashes, the text "Waiting for choice...", the message "Choice is: 4", another separator line, and a final separator line.

```
-----START-----
Waiting for choice...
Choice is: 4
-----END-----
-----START-----
```

CONCLUSION

There are a lot of big data in small files solutions, few of them use HDFS as is while the rest of others force to alter HDFS attitude, however, using HDFS as is would be a recommended choice to treat a pure hadoop, due to adding more customization will clear the small file issue, but on the other hand could add more complexity on hadoop's work.

REFERENCES

<https://storageconference.us/2010/Papers/MSST/Shvachko.pdf> **(RESEARCH PAPER)**

https://www.researchgate.net/publication/340357673_Available_techniques_in_hadoop_small_file_issue **(RESEARCH PAPER)**

<https://www.youtube.com/> (FOR UNDERSTANDING CONCEPTS)

<https://www.geeksforgeeks.org/> (for syntax and errors understanding)

<https://stackoverflow.com/> (for syntax and errors understanding)