Program: 1 Write a program to print Hello World using implement runnable.

```java
package doslab;

public class DOSLAB implements Runnable {
    public void run() {
        System.out.println("Hello World");
    }

    public static void main(String[] args) {
        Thread t = new Thread(new DOSLAB()); t.start();
    }
}
```

**Output:**

![Output](image)
Program:-2 Write a program to print 1 to 10 using thread without synchronizable.

```java
package doslab;

public class DOSLAB implements Runnable {
    int i;
    public void run()
    {
        for(i=0;i<=10;i++)
        {
            System.out.print(Thread.currentThread());
            System.out.println(i);
        }
    }

    public static void main(String []args) throws InterruptedException
    {
        Thread t = new Thread(new DOSLAB()); t.setName("T1 :");
        t.sleep(2000);
        t.start();
        Thread t1 = new Thread(new DOSLAB()); t1.setName("T2 :");
        t1.start();
    }
}
```
Output:

```
run:
Thread[T2],{5,main};Thread[T1],{5,main};0
0
Thread[T1],{5,main};1
Thread[T2],{5,main};2
Thread[T2],{5,main};3
Thread[T1],{5,main};4
Thread[T1],{5,main};5
Thread[T1],{5,main};6
Thread[T1],{5,main};7
Thread[T1],{5,main};8
Thread[T1],{5,main};9
Thread[T2],{5,main};10
Thread[T2],{5,main};11
Thread[T2],{5,main};12
Thread[T2],{5,main};13
Thread[T2],{5,main};14
Thread[T2],{5,main};15
Thread[T2],{5,main};16
Thread[T2],{5,main};17
Thread[T2],{5,main};18
Thread[T2],{5,main};19
Thread[T2],{5,main};20
BUILD SUCCESSFUL (total time: 2 seconds)
```
Program: 3  Write a program to print 1 to 10 using thread using synchronizable.

```
package doslab;

public class DOSLAB implements Runnable {
    private int i;

    public synchronized void run() {
        for (i = 0; i <= 10; i++) {
            System.out.print(Thread.currentThread());
            System.out.println(i);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new DOSLAB());
        t.setName("T1 :");
        t.start();
        Thread t1 = new Thread(new DOSLAB());
        t1.setName("T2 :");
        t1.start();
    }
}
```
Output:

```
run:
Thread[T1] : 6, main
Thread[T2] : 6, main10

Thread[T2] : 6, main11
Thread[T1] : 6, main12
Thread[T2] : 6, main13
Thread[T1] : 6, main14
Thread[T2] : 6, main15
Thread[T1] : 6, main16
Thread[T2] : 6, main17
Thread[T1] : 6, main18
Thread[T2] : 6, main19
Thread[T1] : 6, main20
Thread[T2] : 6, main21

BUILD SUCCESSFUL (total time: 0 seconds)
```
Program: 4 Write a program to print string length using extended Thread class.

```java
import java.util.Scanner;
package dslab;

public class DSLAB extends Thread {
    public void run() {
        String s;
        Scanner s1 = new Scanner(System.in);
        s = s1.nextLine();
        System.out.println(s.length());
    }

    public static void main(String[] args) {
        Thread t = new Thread(new DSLAB());
        t.start();
    }
}
```

Output:

```
run:
SHYAM
BUILD SUCCESSFUL (total time: 3 seconds)
```
Program:-5 Write a Program to implement the echo server using socket.

P5_CLIENT: -

```java
import java.io.*;
import java.net.*;

public class P5_client
{
public static void main(String[] args) throws UnknownHostException, IOException
{
    Socket s = new Socket("localhost", 888);
    System.out.println("Connected to server.");
    DataOutputStream dos = new DataOutputStream(s.getOutputStream());
    BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputStream()));
    BufferedReader kb = new BufferedReader(new InputStreamReader(System.in));
    String str, str1;
    while (!(str = kb.readLine()).equals("exit"))
    {
        dos.writeBytes(str + 
        System.out.println("Sending : " + str);
        str1 = br.readLine();
        System.out.println("Receving : " + str1);
    }
    System.exit(0);
    dos.close();
    br.close();
    kb.close();
    s.close();
}
```
P5_server: -
import java.net.*;
import java.io.*;

public class P5_server
{
public static void main (String[] args) throws IOException {ServerSocket ss = new ServerSocket(888);

System.out.println("Exho server created at port no. 888");
Socket s = ss.accept();
PrintStream ps = new PrintStream(s.getOutputStream());
BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputStream()));

while (true)
{
    String str, str1;
    while ("((str = br.readLine()) != null)
    {
        System.out.println(str); ps.println(str);
    }

    ps.close();
    br.close();
    ss.close();
    s.close();
    System.exit(0);
}
}
Output:- P5_client

```
Connected to server.
Sending: hi
Receiving: hi
Sending: hello
Receiving: hello
```

Output:- P5_Server:-

```
Echo server created at port no. 888
hi
hello
```
Program:- 6 Write a Program to implement the time server using socket.

P6_client:-

```java
import java.net.*;
import java.io.*;

public class P6_client
{
  public static void main(String[] args) throws UnknownHostException, IOException {
    Socket s = new Socket("localhost", 888);
    System.out.println("Connected to server.");

    BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputStream()));
    String str, str1;
    System.out.println(" "+br.readLine());
    System.exit(0);
    br.close();
    s.close();
  }
}
```

P6_server:-

```java
import java.net.*;
import java.io.*;
import java.util.*;
import sun.util.BuddhistCalendar;

public class P6_server
{
  public static void main(String[] args) throws IOException {
    ServerSocket ss = new ServerSocket(888);
    System.out.println("Time server created at port no. 888");
    Socket s = ss.accept();

    PrintStream ps = new PrintStream(s.getOutputStream());
    Calendar calendar = new GregorianCalendar();
```
String am_pm;
int hour = calendar.get(Calendar.HOUR);
int minute = calendar.get(Calendar.MINUTE);
int second = calendar.get(Calendar.SECOND);

if (calendar.get(Calendar.AM_PM) == 0) {
    am_pm = "AM";
} else {
    am_pm = "PM";
}

ps.print(hour+" : ");
ps.flush();
ps.print(minute+" : ");
ps.flush();
ps.print(second+" : ");
ps.flush();
ps.print(" +am_pm");
ps.flush();
ps.close();
ss.close();
s.close();
System.exit(0);
}
Output:- P6_client:-

Output:

P6 Server:

Output:

Time server created at port no. 888
BUILD SUCCESSFUL (total time: 17 seconds)
Program: Write a program to implement RPC Calculator.

Calc.java

```java
import java.rmi.*;

public interface Calc extends Remote {
    public int add(int a, int b) throws RemoteException;
    public int sub(int a, int b) throws RemoteException;
    public double mul(int a, int b) throws RemoteException;
    public float div(int a, int b) throws RemoteException;
}
```

CalcImpl.java

```java
import java.rmi.*;
import java.rmi.server.*;

public class Calc_Impl extends UnicastRemoteObject implements Calc {
    public Calc_Impl() throws RemoteException {
        super();
    }

    @Override
    public int add(int a, int b) throws RemoteException { return a+b; }

    @Override
    public int sub(int a, int b) throws RemoteException { return a - b; }

    @Override
    public double mul(int a, int b) throws RemoteException { return a * b; }

    @Override
    public float div(int a, int b) throws RemoteException { if(b==0) { return a/b; } }
}
```
{
    return 0;
}
else
    return a /b;
}

Calc Client.java

import java.rmi.registry.*;
public class Calc_Client
{
    private static Registry registry;

    public static void main(String[] args) throws Exception {
        registry = LocateRegistry.getRegistry("localhost", 1099);
        Calc remoteApi = (Calc) registry.lookup(Calc.class.getSimpleName());

        System.out.println("Additiontion=" + remoteApi.add(4, 8));
        System.out.println("Substraction=" + remoteApi.sub(4, 8));
        System.out.println("Multiplication=" + remoteApi.mul(4, 8));
        System.out.println("Division=" + remoteApi.div(4, 8));
    }
}

Calc Server.java

import java.rmi.*;
import java.rmi.registry.*;

public class Calc_Server
{
    private static final int PORT = 1099;
    private static Registry registry;
    public static void startRegistry() throws RemoteException
    {
```
```java
registry = java.rmi.registry.LocateRegistry.createRegistry(PORT);
}
public static void registerObject(String name, Remote remoteObj) throws RemoteException, AlreadyBoundException
{
    registry.bind(name, remoteObj);
    System.out.println("Server started. \n Registry started.");
}
public static void main(String[] args) throws Exception
{
    startRegistry();
    registerObject(Calc.class.getSimpleName(), new Calc_Impl());
    Thread.sleep(5 * 60 * 1000);
}
}
Output Server:

Output Client:-
```

Output Server:
```java
run:
Server started.
Registry started.
```

Output Client:
```java
run:
Addition = 12
Subtraction = -4
Multiplication = 32.0
Division = 0.0
BUILD SUCCESSFUL (total time: 0 seconds)
```
Program: Write a program to implement TimeServer using RMI.

Time.java:

```java
package newpackage;
import java.rmi.*;

public interface Time extends Remote {
    public Data incrementCounter(Data value) throws RemoteException;
    public int hour() throws RemoteException;
    public int min() throws RemoteException;
    public int sec() throws RemoteException;
    public String am_pm() throws RemoteException;
}
```

ApicImple.java:

```java
package newpackage;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.util.*;

public class ApicImple extends UnicastRemoteObject implements Time {
    private static final long serialVersionUID = 1L;
    private int counter = 0;
    public ApicImple() throws RemoteException {
        super();
    }

    Calendar calendar = new GregorianCalendar();
    String am_pm;

    @Override
```
public int hour() throws RemoteException
{
    return calendar.get(Calendar.HOUR);
}

@Override
public int min() throws RemoteException
{
    return calendar.get(Calendar.MINUTE);
}

@Override
public int sec() throws RemoteException
{
    return calendar.get(Calendar.SECOND);
}

@Override
public String am_pm()
{
    if (calendar.get(Calendar.AM_PM) == 0)
    {
        am_pm = "AM";
    }
    else
    {
        am_pm = "PM";
    }
    return am_pm;
}

TimeServer.java :-

package newpackage;
import java.rmi.*;
import java.rmi.registry.*;
public class Server
{
    private static final int PORT = 1099;
    private static Registry registry;

    public static void startRegistry() throws RemoteException
    {
        registry = java.rmi.registry.LocateRegistry.createRegistry(PORT);
    }

    public static void registerObject(String name, Remote remoteObj) throws RemoteException, AlreadyBoundException
    {
        registry.bind(name, remoteObj);
        System.out.println(" Time Server started. \n Registry started. ");
        System.out.println("Registered: " + name + " -> " + remoteObj.getClass().getName() + "["+ remoteObj + "]");
    }

    public static void main(String[] args) throws Exception
    {
        startRegistry();
        registerObject(Time.class.getSimpleName(), new ApicImple());
        Thread.sleep(5 * 60 * 1000);
    }
}

TimeClient.java :-

package newpackage;
import java.rmi.registry.*;

public class Client
{
    private static Registry registry;
}
public static void main(String[] args) throws Exception {
    registry = LocateRegistry.getRegistry("localhost", 1099);

    Time remoteApi = (Time) registry.lookup(Time.class.getSimpleName());
    System.out.println(remoteApi.hour() + " : " + remoteApi.min() + " : " + remoteApi.sec() + " : " + remoteApi.am_pm());
}
}
Output Server:-

```
Time Server started.
Registry started.
```

Output client:-

```
BUILD SUCCESSFUL (total time: 0 seconds)
```
Program: -9 Write a program to implement Mutual Exclusion using Threads.

Main.java

package Program_10;

public class main
{
    public static void main(String []args) throws InterruptedException
    {
        Thread t = new Thread(new proces1()); t.setName("T");
        Thread t1 = new Thread(new process2()); t1.start();
        t.start();
    }
}

Critical.java

package Program_10;

public class critical
{
    boolean lock = false; static int a = 2;
    public int critical() { if (lock == false) {
        lock = true; a += 1;
        return a;
    }
    else
    {
        return 0;
    }
}

public void release_lock()
{
    lock = false;
}
Process1.java

package Program_10;

public class proces1 implements Runnable
{
critical cr = new critical(); int p1c = 0;

@Override
public void run()
{

}

while(p1c!=3)
{
int q = cr.critical(); if(q!=0)
{
    System.out.println("P1 is in critical section. " + q);
    p1c+=1;
    cr.release_lock();
}
else
{
    System.err.println("P1 ur section locked. please wait.");
}
}
}

Process2.java

package Program_10;
public class process2 implements Runnable
{
critical cr = new critical();
int p1c = 0;

@Override
public void run()
{
}
while (p1c != 3)
{
int q = cr.critical();

if (q != 0)
{
    System.out.println("P2 is in critical section. " + q);
p1c += 1;
cr.release_lock();
}
else
{
    System.err.println("P2 section locked. please wait.");
}
}
Output :-

:Output - DSLAB (run)

run:
P2 is in critical section. 3
P2 is in critical section. 5
P2 is in critical section. 6
P1 is in critical section. 4
P1 is in critical section. 7
P1 is in critical section. 8
BUILD SUCCESSFUL (total time: 0 seconds)
Program: 10 Write a program to implement Chat Application.
Server2.java

import java.io.*;
import java.net.*;
public class Server2 {
    public static void main(String arg[]) throws Exception {
        ServerSocket ss = new ServerSocket(888);
        Socket s = ss.accept();
        System.out.println("Connection Established..!!!");
        PrintStream ps = new PrintStream(s.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputStream()));
        BufferedReader kb = new BufferedReader(new InputStreamReader(System.in));
        while(true) {
            String s1, s2;
            while((s1 = br.readLine()) != null) {
                System.out.println(s1);
                s2 = kb.readLine();
                ps.println(s2);
            }
            ps.close();
            br.close();
            kb.close();
            ss.close();
            s.close();
            System.exit(0);
        }
    }
}
Client2.java

```java
import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.net.Socket;
public class Client2
{
    public static void main(String args[]) throws Exception
    {
        Socket s = new Socket("localhost",888);
        DataOutputStream dos = new DataOutputStream(s.getOutputStream());

        BufferedReader br = new BufferedReader(new InputStreamReader(s.getInputStream()));
        BufferedReader kb = new BufferedReader(new InputStreamReader(System.in));

        String s1,s2;
        while(!(s1 = kb.readLine()).equals("exit"))
        {
            dos.writeBytes(s1 + "\n");
            s2 = br.readLine();
            System.out.println(s2);
        }
        dos.close();
        br.close();
        kb.close();
        s.close();
    }
}
```

Output-client:-

```
Distributed_System (run-single) × Distributed_System (run-single) #2 ×
Enter YOUR ALL MESSAGE BELOW :

Enter exit for EXIT:

Ka Shyam,,,,

Client Say : Ka Shyam,,,,
Server Say : Are AA V AA V
What's Up ??
Client Say : What's Up ??
Server Say : FIne Fine...
```

```
Output-server:-

Enter YOUR ALL MESSAGE BELOW :

Enter exit for EXIT:

Are AAv AAv
Client Say : Ka Shyam....
Server Say : Are AAv AAv
Fine Fine...
Client Say : What's Up ??
Server Say : Fine Fine...
Program: 11 Write a Program to Increment a Counter in Shared Memory.

```java
import java.util.concurrent.Semaphore;
import java.util.logging.Level;
import java.util.logging.Logger;

public class MultithreadingExample {
    static int counter = 0;
    static Semaphore semaphore = new Semaphore(1);

    public static void incrementCounter() {
        try {
            semaphore.acquire();
            counter++;
            semaphore.release();
        } catch (InterruptedException ex) {
            Logger.getLogger(MultithreadingExample.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    public static void main(String[] args) {
        Thread thread1 = new Thread() {
            @Override
            public void run() {
                for (int i = 0; i < 5000; i++) {
                    incrementCounter();
                }
            }
        }
    }
}
```
Thread thread2 = new Thread()
{
    @Override
    public void run()
    {
        for (int i = 0; i < 5000; i++)
        {
            incrementCounter();
        }
    }
};
thread1.start();
thread2.start();
while (thread1.isAlive() || thread2.isAlive())
{
    System.out.println("Counter: "+ counter);
}

Output:
Run:
Counter: 10000
Build Successful (total time: 0 seconds)
Program:-12 Implement Network File System (NFS).

NFS allows a system to share directories and files with others over a network. By using NFS, users and programs can access files on remote systems almost as if they were local files.

Some of the most notable benefits that NFS can provide are:

1. Local workstations use less disk space because commonly used data can be stored on a single machine and still remain accessible to others over the network.
2. There is no need for users to have separate home directories on every network machine. Home directories could be set up on the NFS server and made available throughout the network.
3. Storage devices such as floppy disks, CDROM drives, and USB Thumb drives can be used by other machines on the network. This may reduce the number of removable media drives throughout the network.

Installation
At a terminal prompt enter the following command to install the NFS Server:

```
sudo apt install nfs-kernel-server
```

Configuration
You can configure the directories to be exported by adding them to the `/etc/exports` file. For example:

```
/ubuntu  *(ro,sync,no_root_squash)
/home    *(rw,sync,no_root_squash)
```

You can replace `*` with one of the hostname formats. Make the hostname declaration as specific as possible so unwanted systems cannot access the NFS mount.

To start the NFS server, you can run the following command at a terminal prompt:

```
sudo systemctl start nfs-kernel-server.service
```

NFS Client Configuration
Use the `mount` command to mount a shared NFS directory from another machine, by typing a command line similar to the following at a terminal prompt:

```
sudo mount example.hostname.com:/ubuntu /local/Ubuntu
```

The mount point directory `/local/ubuntu` must exist. There should be no files or subdirectories in the `/local/ubuntu` directory.
An alternate way to mount an NFS share from another machine is to add a line to the `/etc/fstab` file. The line must state the hostname of the NFS server, the directory on the server being exported, and the directory on the local machine where the NFS share is to be mounted.

The general syntax for the line in `/etc/fstab` file is as follows:

```
example.hostname.com:/ubuntu /local/ubuntu nfs
rsize=8192, wsize=8192, timeo=14, intr
```

If you have trouble mounting an NFS share, make sure the `nfs-common` package is installed on your client. To install `nfs-common` enter the following command at the terminal prompt:

```
sudo apt install nfs-common
```
Program:- 13 Study of CORBA File system.

What is CORBA?

The Common Object Request Broker Architecture (CORBA) is a standard developed by the Object Management Group (OMG) to provide interoperability among distributed objects. CORBA is the world's leading middleware solution enabling the exchange of information, independent of hardware platforms, programming languages, and operating systems. CORBA is essentially a design specification for an Object Request Broker (ORB), where an ORB provides the mechanism required for distributed objects to communicate with one another, whether locally or on remote devices, written in different languages, or at different locations on a network.

The CORBA Interface Definition Language, or IDL, allows the development of language and location-independent interfaces to distributed objects. Using CORBA, application components can communicate with one another no matter where they are located, or who has designed them. CORBA provides the location transparency to be able to execute these applications.

CORBA is often described as a "software bus" because it is a software-based communications interface through which objects are located and accessed. The illustration below identifies the primary components seen within a CORBA implementation.

Data communication from client to server is accomplished through a well-defined object-oriented interface. The Object Request Broker (ORB) determines the location of the target object, sends a request to that object, and returns any response back to the caller. Through this object-oriented technology, developers can take advantage of features such as inheritance, encapsulation, polymorphism, and runtime dynamic binding. These features allow applications to be changed, modified and re-used with minimal changes to the parent interface. The illustration below identifies how a client sends a request to a server through the ORB:
Interface Definition Language (IDL)

A cornerstone of the CORBA standards is the Interface Definition Language (IDL). IDL is the OMG standard for defining language-neutral APIs and provides the platform-independent delineation of the interfaces of distributed objects. The ability of the CORBA environments to provide consistency between clients and servers in heterogeneous environments begins with a standardized definition of the data and operations constituting the client/server interface. This standardization mechanism is the IDL, and is used by CORBA to describe the interfaces of objects.

IDL defines the modules, interfaces and operations for the applications and is not considered a programming language. The various programming languages, such as Ada, C++, or Java, supply the implementation of the interface via standardized IDL mappings.

Application Development Using ORBexpress

The basic steps for CORBA development can be seen in the illustration below. This illustration provides an overview of how the IDL is translated to the corresponding language (in this example, C++), mapped to the source code, compiled, and then linked with the ORB library, resulting in the client and server implementation.
The basic steps for CORBA development include:

1. **Create the IDL to Define the Application Interfaces**

   The IDL provides the operating system and programming language independent interfaces to all services and components that are linked to the ORB. The IDL specifies a description of any services a server component exposes to the client. The term "IDL Compiler" is often used, but the IDL is actually translated into a programming language.

2. **Translate the IDL**

   An IDL translator typically generates two cooperative parts for the client and server implementation, stub code and skeleton code. The stub code generated for the interface classes is associated with a client application and provides the user with a well-defined Application Programming Interface (API). In this example, the IDL is translated into C++.

3. **Compile the Interface Files**

   Once the IDL is translated into the appropriate language, C++ in this example, these interface files are compiled and prepared for the object implementation.

4. **Complete the Implementation**

   If the implementation classes are incomplete, the spec and header files and complete bodies and definitions need to be modified before passing through to be compiled. The output is a complete client/server implementation.

5. **Compile the Implementation**

   Once the implementation class is complete, the client interfaces are ready to be used in the client application and can be immediately incorporated into the client process. This client process is responsible
for obtaining an object reference to a specific object, allowing the client to make requests to that object in the form of a method call on its generated API.

6. Link the Application

Once all the object code from steps three and five have been compiled, the object implementation classes need to be linked to the C++ linker. Once linked to the ORB library, in this example, ORBexpress, two executable operations are created, one for the client and one for the server.

7. Run the Client and Server

The development process is now complete and the client will now communicate with the server. The server uses the object implementation classes allowing it to communicate with the objects created by the client requests.

In its simplest form, the server must perform the following:

- Create the required objects.
- Notify the CORBA environment that it is ready to receive client requests.
- Process client requests by dispatching the appropriate servant.
Program:-14 Study of Google File system.

Introduction

Google has designed and implemented a scalable distributed file system for their large distributed data intensive applications. They named it Google File System, GFS. Google File System is designed by Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung of Google in 2002-03. GFS provides fault tolerance, while running on inexpensive commodity hardware and also serving large number of clients with high aggregate performance. Even though the GFS shares many similar goals with previous distributed file systems, the design has been driven by Google's unique workload and environment. Google had to rethink the file system to serve their “very large scale” applications, using inexpensive commodity hardware.

Google give results faster and more accurate than other search engines. Definitely the accuracy is dependent on how the algorithm is designed. Their initial search technology is Page Rank Algorithm designed by Garry Brin and Larry Page in 1998. And currently they are merging the technology of using both software and hardware in smarter way. Now the field of Google is beyond the searching. It supports uploading video in their server, Google Video; it gives email account of few gigabytes to each user, Gmail; it has great map applications like Google Map and Google Earth; Google Product application, Google News application, and the count goes on. Like, search application, all these applications are heavily data intensive and Google provides the service very efficiently.

A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients. The basic analogy of GFS is master maintains the metadata; client contacts the master and retrieves the metadata about chunks that are stored in chunkservers; next time, client directly contacts the chunkservers. Figure 1 describes these steps more clearly.
Each of these is typically a commodity Linux machine running a user-level server process. Files are divided into fixed-size chunks. Each chunk is identified by an immutable and globally unique 64-bit chunk handle assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, three replicas are stored, though users can designate different replication levels for different regions of the file namespace. The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkservers in HeartBeat messages to give it instructions and collect its state. GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application.

Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. Neither the client nor the chunkservers caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

- **Chunk**
  Chunk in GFS is very important design decision. It is similar to the concept of block in file systems, but much larger than the typical block size. Compared to the few KBs of general block size of file systems, the size of chunk is 64 MB.

- **Metadata**
  The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the location of each chunk's replicas. Among these three, the first two types (namespaces and file-to-chunk mapping) are kept persistent by keeping the log of mutations to an operation log stored on the master's local disk.

- **Master**
  Master is a single process running on a separate machine that stores all metadata, e.g. file namespace, file to chunk mappings, chunk location information, access control information, chunk version numbers, etc. Clients contact master to get the metadata to contact the chunkservers.
System Interaction

Master and chunkservers communicate regularly to obtain the state, if the chunkserver is down, if there is any disk corruption, if any replicas got corrupted, which chunk replicas store chunkserver, etc. Master also sends instruction to the chunkservers for deleting existing chunks, creating new chunks.

• **Read Algorithm**
  Following is the algorithm for the Read operation.
  1. Application originates the read request
  2. GFS client translates the request form (filename, byte range) -> (filename, chunk index), and sends it to master
  3. Master responds with chunk handle and replica locations (i.e. chunkservers where the replicas are stored)

• **Write Algorithm**
  Following is the algorithm with related to Write operation.
  1. Application originates the request
  2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master
  3. Master responds with chunk handle and (primary + secondary) replica locations.

• **Record Append Algorithm**
  Algorithm for Record Append:
  1. Application originates record append request.
  2. GFS client translates requests and sends it to master.
  3. Master responds with chunk handle and (primary + secondary) replica locations.
  4. Client pushes write data to all replicas of the last chunk of the file.
  5. Primary checks if record fits in specified chunk
  6. If record doesn’t fit, then the primary:
     Pads the chunk
     Tell secondary to do the same and informs the client
     Client then retries the append with the next chunk
  7. If record fits, then the primary:
     Appends the record
     Tells secondary to write data at exact offset
     Receives responses from secondary
     And sends final response to the client
Name space management and locking

Multiple operations are to be active and use locks over regions of the namespace to ensure proper serialization. Unlike many traditional file systems, GFS does not have a per-directory data structure that lists all the files in that directory. Nor does it support aliases for the same file or directory (i.e, hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With prefix compression, this table can be efficiently represented in memory. Each node in the namespace tree (either an absolute file name or an absolute directory name) has an associated read-write lock.

Replica Placement

A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches. Additionally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack.

The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine’s network bandwidth. Chunk replicas are also spread across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline.