## CpuOperations.java

```
class FloatOps implements Runnable {

	int noOfThreads;

	FloatOps(int noOfThreads)

	{
		this.noOfThreads = noOfThreads;

	}

	public void run() {

		float a = 1.9f, b = 3.7f, c = 4.4f, d = 1.2f, e = 2.1f, f =
2.1f, g = 1.2f, h = 2.9f;
		float j = 1.9f, k = 3.7f, l = 2.1f, m = 1.2f, n = 2.9f, o =
33.f, p = 1.2f, q = 2.2f, r = 3.1f;
		float s = 1.2f, t = 1.2f, u = 1.6f, v = 1.3f, w = 1.4f;
		long i;

		// The loop will be executed for (1 Billion times/ NoofThreads)
by each Thread
		for (i = 0; i < 1000000000 / noOfThreads; i++) {

			a = w + b;
			c = k + b;
			b = m + j;
			d = k - o;
			e = c - a;
			f = p + b;
			g = l + m;
			h = c + m;
			j = m + k;
			l = a + k;
			m = n - u;
			n = a + b;
			o = c - v;
			p = l + c;
			q = m + t;
			r = j + s;
			s = p + a;
			t = k + g;
			u = h + g;
			v = g + e;
			w = a + b;
			q = f + w;
		}
	}
}

class IntOps implements Runnable {

	int noOfThreads;

	IntOps(int noOfThreads)

	{
		this.noOfThreads = noOfThreads;
```

```
	}

	public void run() {

		int a = 1, b = 4, c = 4, d = 8, e = 2, f = 2, g = 5, h = 5;
		int j = 3, k = 3, l = 2, m = 4, n = 2, o = 33, p = 1, q = 2, r
= 3;
		int s = 5, t = 7, u = 4, v = 3, w = 7;
		long i;

		// The loop will be executed for ( 1Billion times/ NoofThreads)
by each Thread

		for (i = 0; i < 1000000000 / noOfThreads; i++) {

			a = w + b;
			c = k + b;
			b = m + j;
			d = k - o;
			e = c - a;
			f = p + b;
			g = l + m;
			h = c + m;
			j = m + k;
			l = a + k;
			m = n - u;
			n = a + b;
			o = c - v;
			p = l + c;
			q = m + t;
			r = j + s;
			s = p + a;
			t = k + g;
			u = h + g;
			v = g + e;
			w = a + b;
			q = f + w;
		}
	}
}

public class CpuOperations {

	public static void main(String args[])

	{

		CpuOperations fops = new CpuOperations();
		int[] noOfThreads = { 1, 2, 4 };

		for (int j : noOfThreads) {

			fops.computeFops(j);

		}

		for (int j : noOfThreads) {

			fops.computeIops(j);

		}
```

```
	}

	public void computeFops(int noOfThreads) {
		// Function for calculating Giga Flops
		try {
			long startTime = System.currentTimeMillis();
			FloatOps tf = new FloatOps(noOfThreads);
			Thread[] threads = new Thread[noOfThreads];
			for (int i = 0; i < threads.length; i++) {
				threads[i] = new Thread(tf);
				threads[i].start();

			}

			for (Thread thread : threads) {
				thread.join();
			}
			long endTime = 0;
			endTime = System.currentTimeMillis();
			long timeneeded = endTime - startTime;

			float timetaken = (float) (timeneeded / 1000.0);
			System.out.println("Time Taken for running with " +
noOfThreads + " Thread is " + timetaken + " Seconds");

			double flops = (1000000000.0 / timetaken);

			// Divide the flops by 1 billion to get Giga Flops
			// Multiply FLOPS with no of operations
			double Gigaflops = 44 * flops / 1000000000;
			System.out.println("Total No Of Giga Flops is " +
Gigaflops + "\n");

		}

		catch (Exception e) {

		}
	}

	public void computeIops(int noOfThreads) {

		try {
			// Function for calculating Giga Iops
			long startTime = System.currentTimeMillis();
			IntOps ti = new IntOps(noOfThreads);
			Thread[] threads = new Thread[noOfThreads];
			for (int i = 0; i < threads.length; i++) {
				threads[i] = new Thread(ti);
				threads[i].start();

			}

			for (Thread thread : threads) {
				thread.join();
			}
			long endTime = 0;
			endTime = System.currentTimeMillis();
			long timeneeded = endTime - startTime;
```

```
                    float timetaken = (float) (timeneeded / 1000.0);
                    System.out.println("Time Taken for running with " +
noOfThreads + " Thread is " + timetaken + " Seconds");

                    double Iops = (1000000000.0 / timetaken);

                    // Divide the Iops by 1 billion to get Giga Ilops
                    // Multiply IOPS with no of operations

                    double GigaIops = 44 * Iops / 1000000000;
                    System.out.println("Total No Of Giga Iops is " + GigaIops
+ "\n");

                }

                catch (Exception e) {

                }
        }

}
```

## CpuOperationSamples.java

```
  class FloatOpsSamples implements Runnable {

        int noOfThreads;


        FloatOpsSamples (int noOfThreads)

        {
              this.noOfThreads=noOfThreads;

        }

      public  void run()
                  {

                        float
a=1.9f,b=3.7f,c=4.4f,d=1.2f,e=2.1f,f=2.1f,g=1.2f,h=2.9f;
                        float
j=1.9f,k=3.7f,l=2.1f,m=1.2f,n=2.9f,o=33.f,p=1.2f,q=2.2f,r=3.1f;
                        float s=1.2f,t=1.2f,u=1.6f,v=1.3f,w=1.4f;
                        long i;
                        // The loop will be executed for (1 Billion times/
NoofThreads) by each Thread
                        for (i=0;i<1000000000/noOfThreads;i++)
                        {

                              a=w+b;
                              c=k+b;
                              b=m+j;
                              d=k-o;
                              e=c-a;
                              f=p+b;
                              g=l+m;
                              h=c+m;
                              j=m+k;
                              l=a+k;
                              m=n-u;
                              n=a+b;
                              o=c-v;
                              p=l+c;
                              q=m+t;
                              r=j+s;
                              s=p+a;
                              t=k+g;
                              u=h+g;
                              v=g+e;
                              w=a+b;
                              q=f+w;
                        }
                  }
        }


  class IntOpsSamples implements Runnable {

        int noOfThreads;
```

```
        IntOpsSamples (int noOfThreads)

        {
              this.noOfThreads=noOfThreads;

        }

      public  void run()
                  {

            int a=1,b=4,c=4,d=8,e=2,f=2,g=5,h=5;
            int j=3,k=3,l=2,m=4,n=2,o=33,p=1,q=2,r=3;
            int s=5,t=7,u=4,v=3,w=7;
                        long i;
                        // The loop will be executed for (1 Billion times/
NoofThreads) by each Thread
                        for (i=0;i<1000000000/noOfThreads;i++)
                        {
                              a=w+b;
                              c=k+b;
                              b=m+j;
                              d=k-o;
                              e=c-a;
                              f=p+b;
                              g=l+m;
                              h=c+m;
                              j=m+k;
                              l=a+k;
                              m=n-u;
                              n=a+b;
                              o=c-v;
                              p=l+c;
                              q=m+t;
                              r=j+s;
                              s=p+a;
                              t=k+g;
                              u=h+g;
                              v=g+e;
                              w=a+b;
                              q=f+w;
                        }
                  }
        }


  public class CpuOperationSamples {

       public static void main(String args[])

       {

             CpuOperationSamples fops = new CpuOperationSamples();

            System.out.println("Following are 600 Samples for Floating
Point Operations per Second With 4 Threads");
             //This loop will provide 600 Samples for Giga FLOPS with 4
threads
            for(int i=0;i<600;i++)
            {
             fops.computeFops(4);
```

```
            }
            System.out.println("Following are 600 Samples for Integer
Operations Per Seconds With 4 Threads");

            //This loop will provide 600 Samples for Giga IOPS with 4
threads
            for(int i=0;i<600;i++)
                  {
                   fops.computeIops(4);

                   }




      }

      public void computeFops(int noOfThreads)
            {
            // Function for calculating Giga Flops
                  try
                  {
                   long startTime = System.currentTimeMillis();
                   FloatOpsSamples tf = new FloatOpsSamples(noOfThreads);
                  Thread[] threads = new Thread[noOfThreads];
                   for (int i = 0; i < threads.length; i++)
                   {
                         threads[i] = new Thread (tf);
                         threads[i].start();

                   }

                   for (Thread thread : threads) {
                               thread.join();
                     }
                        long endTime = 0;
                        endTime = System.currentTimeMillis();
                        long timeneeded = endTime - startTime;

                        float timetaken = (float) (timeneeded/1000.0);

                        double flops =    (1000000000.0/timetaken);

                        // Divide the flops by 1 billion to get Giga Flops
                        // Multiply FLOPS with no of operations
                        double Gigaflops =44*flops/1000000000;
                        System.out.println(Gigaflops);

            }

                  catch(Exception e)
                  {

                  }
            }

      public void computeIops(int noOfThreads)
            {
            // Function for calculating Giga Iops
                  try
```

```
                {
                 long startTime = System.currentTimeMillis();
                 IntOpsSamples ti = new IntOpsSamples(noOfThreads);
                Thread[] threads = new Thread[noOfThreads];
                 for (int i = 0; i < threads.length; i++)
                 {
                       threads[i] = new Thread (ti);
                       threads[i].start();

                 }

                 for (Thread thread : threads) {
                             thread.join();
                       }
                      long endTime = 0;
                      endTime = System.currentTimeMillis();
                      long timeneeded = endTime - startTime;

                      float timetaken = (float) (timeneeded/1000.0);

                      // Divide the Iops by 1 billion to get Giga Ilops
                      // Multiply IOPS with no of operations
                      double Iops =   (1000000000.0/timetaken);
                      double GigaIops = 44*Iops/1000000000;
                      System.out.println(GigaIops);

          }

                catch(Exception e)
                {

                }
          }



}
```

## DiskRandRead.java

```
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.util.Scanner;

class DiskReadR implements Runnable {

	int bufferSize;
	String path;
	int noOfThreads;

	DiskReadR(int noOfThreads, int bufferSize, String path)

	{
		this.bufferSize = bufferSize;
		this.path = path;
		this.noOfThreads = noOfThreads;
	}

	public void run() {
		try {
			System.gc();
			RandomAccessFile File = new RandomAccessFile(path, "rw");
			FileChannel inChannel = File.getChannel();
			long size = inChannel.size();
			long noOfLoops = (inChannel.size() / bufferSize);
			// Creating varying size Buffer Blocks
			ByteBuffer buf = ByteBuffer.allocate(bufferSize);
			int bytesRead = inChannel.read(buf);
```

```
            long value = size - 1048576;

            for (long i = 0; i <= noOfLoops; i++) {

                  // Creating a Random Function to obtain random inChannel
                  // position
                  long random = (long) ((Math.random() * (value)) + 1);
                  // Obtaining Random inChannel position
                  inChannel.position(random);
                  // Flips buffer to make it ready for read
                  buf.flip();

                  while (buf.hasRemaining()) {
                        // Reads the byte at buffers current position
                        buf.get();
                  }
                  // Clear buffer to make it ready for writing
                  buf.clear();
                  // Read bytes from this channel to given buffer
                  bytesRead = inChannel.read(buf);

            }
            File.close();
            System.gc();

}

catch (Exception E)

{
```

```
                }

        }
}

public class DiskRandRead {

        public static void main(String args[])

        {
                String s = null;
                ;

                DiskRandRead CalRandRead = new DiskRandRead();
                int[] noOfThreads = { 1, 2 };
                int[] buffSize = { 1, 1024, 1024 * 1024 };

                for (int i : noOfThreads) {

                        for (int j : buffSize) {

                                if (j == 1)

                                {
                                        s = "myshortfile.txt";
                                }

                                else {
                                        s = "myfile.txt";
                                }
                                CalRandRead.DiskReadRandCompute(i, j, s);
```

```
                }

        }

}

public void DiskReadRandCompute(int noOfThreads, int buffersize, String path)

{
        // Function to perform Disk Random Read Operations with varying block
        // sizes and varying concurrent Threads
        try {
                RandomAccessFile readFile = new RandomAccessFile(path, "rw");
                DiskReadR readRand = new DiskReadR(noOfThreads, buffersize, path);
                long startTime = System.currentTimeMillis();
                Thread[] threads = new Thread[noOfThreads];
                for (int i = 0; i < threads.length; i++) {

                        threads[i] = new Thread(readRand);
                        threads[i].start();

                }
                for (Thread thread : threads) {
                        thread.join();
                }

                long endTime = System.currentTimeMillis();
                long totalTime = (endTime - startTime);
                float timetaken = (float) (totalTime / 1000.0);
                float data = (float) (readFile.length() / 1048576.0);
```

```
                        float throughput = noOfThreads * data / timetaken;
                        float Latency = (float) ((buffersize * timetaken) / (noOfThreads *
readFile.length()));
                        System.out.println("Time " + timetaken);
                        System.out.println("No of Threads " + noOfThreads);
                        System.out.println("Buffer Size " + buffersize);
                        System.out.println("Total Data Read is " + data + " MB");
                        System.out.println("Throughput = " + throughput + " MB/S");
                        System.out.println("Latency = " + Latency + " Seconds\n");

                } catch (Exception e)

                {

                }

        }

}
```

## DiskRandWrite.java

```
import java.io.File;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

class DiskRandR implements Runnable {

	int bufferSize;

	int noOfThreads;

	DiskRandR() {
	}

	DiskRandR(int noOfThreads, int bufferSize)

	{
		this.bufferSize = bufferSize;

		this.noOfThreads = noOfThreads;
	}

	public void run() {

		try {
			long noOfLoops = 1000;

			byte[] bytes = new byte[bufferSize];
			// Generates random bytes and place them in bytes array
			ThreadLocalRandom.current().nextBytes(bytes);
			// Creating varying size byte blocks
			ByteBuffer buffer = ByteBuffer.wrap(bytes);
			RandomAccessFile readFile = new
RandomAccessFile("DiskWriteRand.txt", "rw");
			FileChannel inChannel = readFile.getChannel();

			long range = bufferSize * noOfLoops;
			Random r = new Random();

			for (long i = 1; i <= noOfLoops / noOfThreads; i++) {
				long number = (long) (r.nextDouble() * range);
				// Writes a sequence of bytes to this channel from
given buffer
				inChannel.write(buffer);
				// Sets the position of buffer back to zero so we
can re-read
				// the data
				buffer.rewind();
				// Sets the inChannel to Random positions
				inChannel.position(number);
			}
			inChannel.close();
			readFile.close();

		}

		catch (Exception E) {
```

```
			}

		}

		public void DiskRandCompute(int i, int j) {
			// TODO Auto-generated method stub

		}
}

public class DiskRandWrite {

		public static void main(String args[])

		{

			DiskRandWrite CalRandWrite = new DiskRandWrite();
			int[] noOfThreads = { 1, 2 };
			int[] buffSize = { 1, 1024, 1024 * 1024 };

			for (int i : noOfThreads) {

				for (int j : buffSize) {
					CalRandWrite.DiskRandCompute(i, j);

				}

			}
		}

		public void DiskRandCompute(int noOfThreads, int buffersize) {
			// Function to perform Disk Random Write Operations with
varying block
			// sizes and varying concurrent Threads

			try {

				DiskRandR writeRand = new DiskRandR(noOfThreads,
buffersize);
				long startTime = System.currentTimeMillis();
				Thread[] threads = new Thread[noOfThreads];
				for (int i = 0; i < threads.length; i++) {

					threads[i] = new Thread(writeRand);
					threads[i].start();

				}
				for (Thread thread : threads) {
					thread.join();
				}

				long latency = 0;
				long endTime = System.currentTimeMillis();
				long totalTime = (endTime - startTime);
				float timetaken = (float) (totalTime / 1000.0);
				float throughput = 1000 * buffersize / (timetaken *
1048576);
				float totaldata = (float) (1000 * buffersize /
1048576.0);
				float datainbytes = (float) (1000 * buffersize);
```

```
                    float Latency = (float) ((buffersize * timetaken) /
(datainbytes));
                    System.out.println("Time " + timetaken + "Seconds");
                    System.out.println("No of Threads " + noOfThreads);
                    System.out.println("Buffer Size " + buffersize);
                    System.out.println("Total Data Write is " + totaldata + "
MB");
                    System.out.println("Throughput = " + throughput + "
MB/S");
                    System.out.println("Latency = " + Latency + "
seconds\n");
            } catch (Exception e)

            {

            }

      }
}
```

## DiskSeqRead.java

```
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.util.Scanner;

class DiskReadS implements Runnable {

	int bufferSize;
	String path;
	int noOfThreads;

	DiskReadS() {

	}

	DiskReadS(int noOfThreads, int bufferSize, String path)

	{
		this.bufferSize = bufferSize;
		this.path = path;
		this.noOfThreads = noOfThreads;
	}

	public void run() {

		try {
			System.gc();
			RandomAccessFile readFile = new RandomAccessFile(path, "rw");
			FileChannel inChannel = readFile.getChannel();
```

```
                    // Creating varying size Buffer Blocks
                    ByteBuffer buffer = ByteBuffer.allocate(bufferSize);
                    // Read bytes from this channel to given buffer
                    int bytesRead = inChannel.read(buffer);

                    while (bytesRead != -1) {
                        // Flips buffer to make it ready for read
                        buffer.flip();

                        while (buffer.hasRemaining()) {
                            // Reads the byte at buffers current position
                            buffer.get();
                        }
                        // Clear buffer to make it ready for writing
                        buffer.clear();
                        // Read bytes from this channel to given buffer
                        bytesRead = inChannel.read(buffer);

                    }
                    readFile.close();
                    System.gc();

                }

                catch (Exception E) {
                }

        }

}
```

```
public class DiskSeqRead {

	public static void main(String args[])

	{

		String s;

		s = "myfile.txt";

		DiskSeqRead CalSeqRead = new DiskSeqRead();
		int[] noOfThreads = { 1, 2 };
		int[] buffSize = { 1, 1024, 1024 * 1024 };

		for (int i : noOfThreads) {

			for (int j : buffSize) {
				CalSeqRead.DiskReadSeqCompute(i, j, s);

			}

		}
	}

	public void DiskReadSeqCompute(int noOfThreads, int buffersize, String path)

	{
		// Function to perform Disk Sequential Read Operations with varying
		// block sizes and varying concurrent Threads
		try {
```

```
                RandomAccessFile readFile = new RandomAccessFile(path, "rw");
                DiskReadS readSeq = new DiskReadS(noOfThreads, buffersize, path);

                long startTime = System.currentTimeMillis();
                Thread[] threads = new Thread[noOfThreads];
                for (int i = 0; i < threads.length; i++) {

                        threads[i] = new Thread(readSeq);
                        threads[i].start();

                }
                for (Thread thread : threads) {
                        thread.join();
                }

                long endTime = System.currentTimeMillis();
                long totalTime = (endTime - startTime);
                float timetaken = (float) (totalTime / 1000.0);
                float data = (float) (noOfThreads * readFile.length() / 1048576.0);
                float throughput = data / timetaken;
                float Latency = (float) ((buffersize * timetaken) / (noOfThreads *
readFile.length()));
                System.out.println("Time " + timetaken);
                System.out.println("No of Threads " + noOfThreads);
                System.out.println("Buffer Size " + buffersize);
                System.out.println("Total Data Read is " + data + " MB");
                System.out.println("Throughput = " + throughput + " MB/S");
                System.out.println("Latency = " + Latency + " seconds\n");
        }

        catch (Exception e) {
```

```
                }
            }


}
```

## DiskSeqWrite.java

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.util.concurrent.ThreadLocalRandom;

class DiskWriteS implements Runnable {

        DiskWriteS()
        {}

        int bufferSize;
        String path;
        int noOfThreads;

        DiskWriteS (int noOfThreads,int bufferSize)

         {
                this.bufferSize=bufferSize;

                this.noOfThreads=noOfThreads;
         }

                public  void run()
                {
                        try{

                                byte[] bytes = new byte[bufferSize];
```

```
                //Generates random bytes and place them in bytes array
                 ThreadLocalRandom.current().nextBytes(bytes);
                // Creating varying size byte blocks
ByteBuffer buffer = ByteBuffer.wrap(bytes);
File file = new File("DiskWriteSeq.txt");
boolean append = false;
FileChannel SeqChannel = new FileOutputStream(file, append).getChannel();
long noOfLoops=0;
if(bufferSize==1)

{
    noOfLoops= 52428800;
}
else if (bufferSize==1024)

{
    noOfLoops= 5242880;
}
else
{

    noOfLoops= 5120;
}

for(long i=1;i<=noOfLoops/noOfThreads;i++)
{
   //Writes a sequence of bytes to this channel from given buffer
    SeqChannel.write(buffer);
    //Sets the inChannel to Random positions
  buffer.rewind();
}
```

```
                    SeqChannel.close();

                        System.gc();



                        }

                        catch (Exception E)

                        {



                        }


                }
        }

public class DiskSeqWrite {

        public static void main(String args[])

        {


                DiskSeqWrite CalSeqWrite = new DiskSeqWrite();
                 int[] noOfThreads = {1,2};
                 int[] buffSize = {1,1024,1024*1024};

                 for (int i : noOfThreads) {
```

```
                    for (int j : buffSize)
                    {
                            CalSeqWrite.DiskWriteSeqCompute(i,j);

                    }

            }

    }


     public void DiskWriteSeqCompute (int noOfThreads,int buffersize)
     {

            try{
                    // Function to perform Disk Sequential Write Operations with varying block sizes and varying concurrent Threads

                     DiskWriteS writeSeq = new DiskWriteS(noOfThreads,buffersize);
                     long startTime = System.currentTimeMillis();
                     Thread[] threads = new Thread[noOfThreads];
                     for (int i = 0; i < threads.length; i++)
                     {

                            threads[i] = new Thread (writeSeq);
                            threads[i].start();

                     }
                     for (Thread thread : threads) {
                                    thread.join();
                             }
```

```
                                        float Latency=0;
                                        long endTime = System.currentTimeMillis();
                                        long totalTime = (endTime-startTime);
                                        float timetaken = (float) (totalTime/1000.0 );
                                        float data=0;
                                        if(buffersize==1)
                                        {

                                                data = (float) (50);
                                                 Latency = ((buffersize*timetaken)/(50));
                                        }
                                        else
                                        {
                                                data = (float) (5120);
                                                Latency = ((buffersize*timetaken)/(5120));
                                        }

                                        float throughput = data/timetaken;
                                        System.out.println("Time " + timetaken +" Seconds" );
                                        System.out.println("No of Threads " + noOfThreads );
                                        System.out.println("Buffer Size " + buffersize);
                                        System.out.println("Total Data Write is " + data + " MB");
                                        System.out.println("Throughput = " + throughput + " MB/S"
);
                                        System.out.println("Latency = " + Latency/1048576 +"
Seconds\n");

                        }
                        catch(Exception e)

                        {
```

```
                    }


            }

    }
```

## Network_Tcp_Client.java

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.nio.ByteBuffer;
import java.util.Random;
import java.util.Scanner;
import java.util.concurrent.ThreadLocalRandom;

class Tcp_Client_N implements Runnable{

	String localhost = null;
	int buff = 0;
	int port=0;

	public Tcp_Client_N(){
	}


	public Tcp_Client_N(int buff, int port,String host) {
		this.buff = buff;
		this.port=port;
		this.localhost=host;
	}


	public void run() {
```

```
DataOutputStream Dataoutput ;
DataInputStream Datainput ;
Socket socket ;
OutputStream output ;
InputStream input ;



try {
        socket = new Socket(localhost, port);
        //Returns Output Stream for this socket
        output = socket.getOutputStream();

        byte[] sendPackets = new byte[buff];
        //Generates random bytes and place them in sendPackets array
         ThreadLocalRandom.current().nextBytes(sendPackets);

        Dataoutput = new DataOutputStream(output);
        Dataoutput.writeInt(sendPackets.length);
        //Writes Data into Output Stream provided
        Dataoutput.write(sendPackets);
        //Returns Input Stream for this socket
        input = socket.getInputStream();
        Datainput = new DataInputStream(input);
        //Creating byte array for receiving packets back from server
        byte[] RecievePackets = new byte[buff];
        //Reads the byte from contained input stream
        Datainput.readFully(RecievePackets);
        System.out.println("Communicating With Thread :
"+Thread.currentThread().getName());
        System.out.println("Recieved Packet Size: "+ RecievePackets.length);
        socket.close();
```

```
                output.close();
            } catch (Exception e) {


                System.out.println("Please Enter appropriate Server Address");
                System.out.println("Error Message "+ e.getMessage());
                System.exit(0);

            }

        }


}

public class Network_Tcp_Client{

 public static void main(String args[])

        {
        String s=null;
        String nthread=null;
        int threadcount=0;

    Scanner in = new Scanner(System.in);
    System.out.println("Please Enter IP address of Server you want to Communicate with");
    s = in.nextLine();


    Scanner in1 = new Scanner(System.in);
    System.out.println("Please Enter no of Threads 1 or 2");
```

```
    nthread = in1.nextLine();

    threadcount=Integer.parseInt(nthread);

            Network_Tcp_Client tcp= new Network_Tcp_Client();
            int[] noOfThreads = {threadcount};
            int[] buffSize = {1,1024,64*1024};


            for (int i : noOfThreads) {

                    for (int j : buffSize)
                    {
                            tcp.TcpNetworkCalculation(i,j,s);

                    }

            }
}

public void TcpNetworkCalculation(int noOfThreads,int buffersize,String host)

{

           // Function to perform Network Operations using TCP Protocol

            long startTime = System.currentTimeMillis();


            try{
```

```
Thread[] threads = new Thread[noOfThreads];
int[] port = {11379,11279};
for (int i = 0; i < threads.length; i++)
{
        Tcp_Client_N th = new Tcp_Client_N(buffersize,port[i],host);
        threads[i] = new Thread (th);
        threads[i].start();

}
for (Thread thread : threads) {
                thread.join();
         }

}
catch (Exception e)

{

}
        long endTime = System.currentTimeMillis();
        long totalTime = (endTime-startTime);
        float timetaken = (float) (totalTime/1000.0 );
        float throughput = noOfThreads*buffersize*2*8/(timetaken*1000000);
        System.out.println("Time " + timetaken );
        System.out.println("No of Threads " + noOfThreads );
        System.out.println("Buffer Size " + buffersize);
        System.out.println("Throughput = " + throughput + " Mb/S" );

}
```

}

## Network Tcp Server.java

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class Network_Tcp_Server extends Thread {

	protected Socket sendersocket;

	public static void main(String[] args) throws IOException {

			Scanner in = new Scanner(System.in);
	    System.out.println("Please Number of threads 1 or 2");
	    String s = in.nextLine();

		int a = Integer.parseInt(s);

		ServerSocket Socket1 = null;
		ServerSocket Socket2 = null;

		try {
			//Creates server Socket bound to specified port provided
			Socket1 = new ServerSocket(11379);

			Socket2 = new ServerSocket(11279);
```

```
                System.out.println("Socket has been Created");
                try {

                        System.out.println("Waiting for Connection");
                        while (true) {
                //Waits for the connection to get made with this socket and accepts it
                                new Network_Tcp_Server(Socket1.accept());
                        if(a==2)
                        {
                                new Network_Tcp_Server(Socket2.accept());
                        }
                        }
                } catch (Exception e) {
                        System.out.println("Connection failure.");
                        System.exit(1);
                }
        } catch (Exception e) {

                System.exit(1);
        } finally {
                try {
                        // Socket is closed
                        Socket1.close();
                        Socket2.close();
                } catch (Exception e) {

                        System.exit(1);
                }
        }
}
```

```
	private Network_Tcp_Server(Socket clientSoc) {
		sendersocket = clientSoc;
		start();
	}

	public void run() {

		try {

			InputStream input = sendersocket.getInputStream();
			OutputStream output = sendersocket.getOutputStream();
			DataInputStream Datain = new DataInputStream(input);
			int length = Datain.readInt();
			byte[] RecievedPacket = new byte[length];
			Datain.readFully(RecievedPacket);

			System.out.println(Thread.currentThread().getName() + " aknowledgement
recieved from client side");

			DataOutputStream Dataout = new DataOutputStream(output);
			Dataout.write(RecievedPacket);

		} catch (Exception e) {
			System.exit(1);
		}
	}
}
```

## Network Udp Client.java

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.Scanner;
import java.util.concurrent.ThreadLocalRandom;

class UDP_Client_N implements Runnable{


        int buff = 0;
        int port=0;
        String localhost=null;

        public UDP_Client_N(int buff, int port,String localhost) {
                this.buff = buff;
                this.port=port;
                this.localhost=localhost;
        }


        public void run() {
           DatagramSocket socket = null;

    try  {
                //Determines the IP address of Server to which connection need to be created
         InetAddress host = InetAddress.getByName(localhost);
         socket = new DatagramSocket();

         byte[] SendPackets = new byte[buff];
        //Generates random bytes and place them in sendPackets array
```

```
            ThreadLocalRandom.current().nextBytes(SendPackets);
            //Create DatagramPacket to send to server with specified IP address and port number
            DatagramPacket Sndpackets = new DatagramPacket(SendPackets, SendPackets.length,
host, port);
            //Sends the pack from specified socket
            socket.send(Sndpackets);
           //Creating byte array for receiving packets back from server
            byte[] RecievedPackets = new byte[buff];
            // constructing DatagramPacket for receiving packets
            DatagramPacket Recvpackets = new DatagramPacket(RecievedPackets,
RecievedPackets.length);
            //Receives  data gram socket from this specified socket
            socket.receive(Recvpackets);
                        System.out.println("Recieved Packet Size: "+ RecievedPackets.length);

      }catch(Exception e)
      {
          System.out.println("Please Enter appropriate Server Address");
                        System.out.println("Error Message "+ e.getMessage());
                        System.exit(0);

      }
          }


}

public class Network_Udp_Client {

          public static void main (String args[])

          {
```

```
int buff=0;
int nthread=0;
String host=null;

System.out.println("Please Enter IP address of Server you want to Communicate
with ");
Scanner in1 = new Scanner(System.in);
host=in1.nextLine();

System.out.println("Please Enter no of Threads 1 or 2");
Scanner in2 = new Scanner(System.in);
nthread=Integer.parseInt(in2.nextLine());

System.out.println("Please Enter the size of Buffer Packets in Bytes");
Scanner in3 = new Scanner(System.in);
buff=Integer.parseInt(in3.nextLine());


Network_Udp_Client udp= new Network_Udp_Client();
 int[] noOfThreads = {nthread};


 for (int i : noOfThreads) {


                udp.UDPNetworkCalculation(i,buff,host);


 }
```

```
}

public void UDPNetworkCalculation(int noOfThreads,int buffersize,String path)

{

        // Function to perform Network Operations using UDP Protocol

         long startTime = System.currentTimeMillis();



         try{

         Thread[] threads = new Thread[noOfThreads];
         int[] port = {10129,12979};
         for (int i = 0; i < threads.length; i++)
         {
                  UDP_Client_N th = new UDP_Client_N(buffersize,port[i],path);
                  threads[i] = new Thread (th);
                  threads[i].start();

         }
         for (Thread thread : threads) {
                           thread.join();
                   }

         }
         catch (Exception e)

         {
```

```
                }
                        long endTime = System.currentTimeMillis();
                        long totalTime = (endTime-startTime);
                        System.out.println("Time " + totalTime + " ms" );
                        long timetaken = (long) (totalTime );
                        float throughput = (float)(noOfThreads*buffersize*2*8/(timetaken));
                        System.out.println("No of Threads " + noOfThreads );
                        System.out.println("Buffer Size " + buffersize);
                        System.out.println("Throughput = " + throughput/1000 + " Mb/S\n" );


        }


}
```

## Network Udp Server.java

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;



class Network_Udp_Ser implements Runnable {

int portNum;

Network_Udp_Ser (int portNum){
      this.portNum = portNum;
}

public void run() {

      try{

            //Create byte array of maximum size to receive data from client
                  // Packet size for UDP cannot exceed 64 KB
            byte receivingdata[]=new byte[1024*62];
            //Create Server Socket
            DatagramSocket socket=new DatagramSocket(portNum);

            System.out.println("Waiting for Data Packets");
            //Create DatagramPacket to receive packet from client
            DatagramPacket recievepacket=new DatagramPacket(receivingdata,receivingdata.length);
             //Receives  data gram socket from this specified socket
            socket.receive(recievepacket);

          //Creating byte array for sending packets back to client
```

```
            byte[] sendData = recievepacket.getData();
            //Create DatagramPacket to send to client with specified IP address and port number
            DatagramPacket sendpacket = new
DatagramPacket(sendData,sendData.length,recievepacket.getAddress(),recievepacket.getPort());
           //Sends the packet from this specified socket
            socket.send(sendpacket);

            System.out.println("Data Received and replied back to client
"+recievepacket.getAddress().getHostAddress() );
            System.out.println(Thread.currentThread().getName() + " is completed");

      }catch (Exception e) {
            e.printStackTrace();
      }

}

}

public class Network_Udp_Server {

public static void main(String args[]) throws IOException {

           Thread[] threads = new Thread[2];
            int[] port = {10129,12979};

            for (int i = 0; i < threads.length; i++)
            {
                  Network_Udp_Ser  udp= new Network_Udp_Ser(port[i]);
```

```
                threads[i] = new Thread (udp);
                threads[i].start();

        }


}

}
```