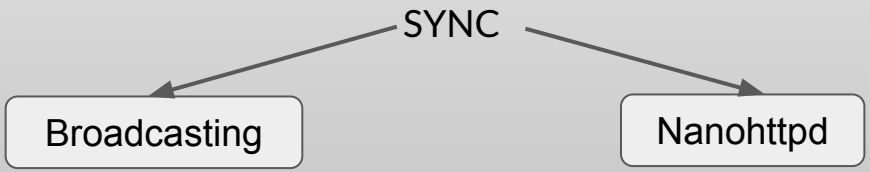
Sync Protocol design using nanohttpd

MCNRG Research Group
Department of Computer Science & Engineering

Sync Protocol

Our main motive is to a develop a multi server and multi client system, such that after implementation the files in each node in an common network are balanced.

1. Identifies the nodes in a network.
2. The device shouts its IP address and listens
3. Server is started using Nanohttpd.
4. Prioritize the files in the system.
5. Summary vector exchange.
6. As soon as it receives an IP address, it starts a thread for downloading the file.
7. Downloading of files using TCP Client.



1. Identifies the nodes in a network.
2. The device shouts its IP address and listens for the other IP addresses.
3. As soon as it receives an IP address, it starts a thread for downloading the file.

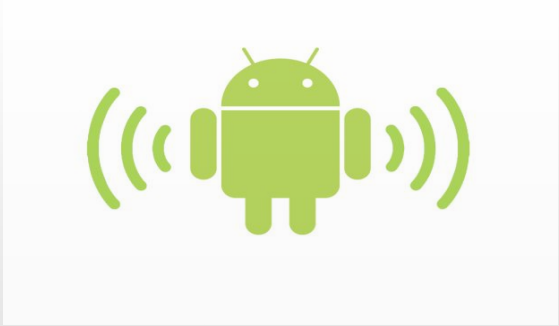
1. Server is started using Nanohttpd.
2. Prioritize the files in the system.
3. Summary vector exchange.
4. Downloading of files using TCP Client.

Broadcasting

The problem statement of the project: syncing data among devices connected in a network.

Usage	Reasons
Identifies all nodes in connected in a network.	Because our system was single server and multiple clients. Multiple clients would only sync data with single known server as ip of server was hard coded in client's code
The device broadcast its ip and listen other node's ip address in a network	This is done because we need system where every node should behave as server and client simultaneously.
As soon as it receives ip address of other nodes , it starts a thread for downloading the data for each devices.	As it listens other nodes ip it makes new thread and transfer data it . By doing this each node in the network behaves as both server and client simultaneously.
Nodes broadcast its ip (Shouts ip).	As in our previous system clients already knew the ip of the server but using broadcasting ip feature each nodes can identify all the other nodes (server).

Why broadcasting ?

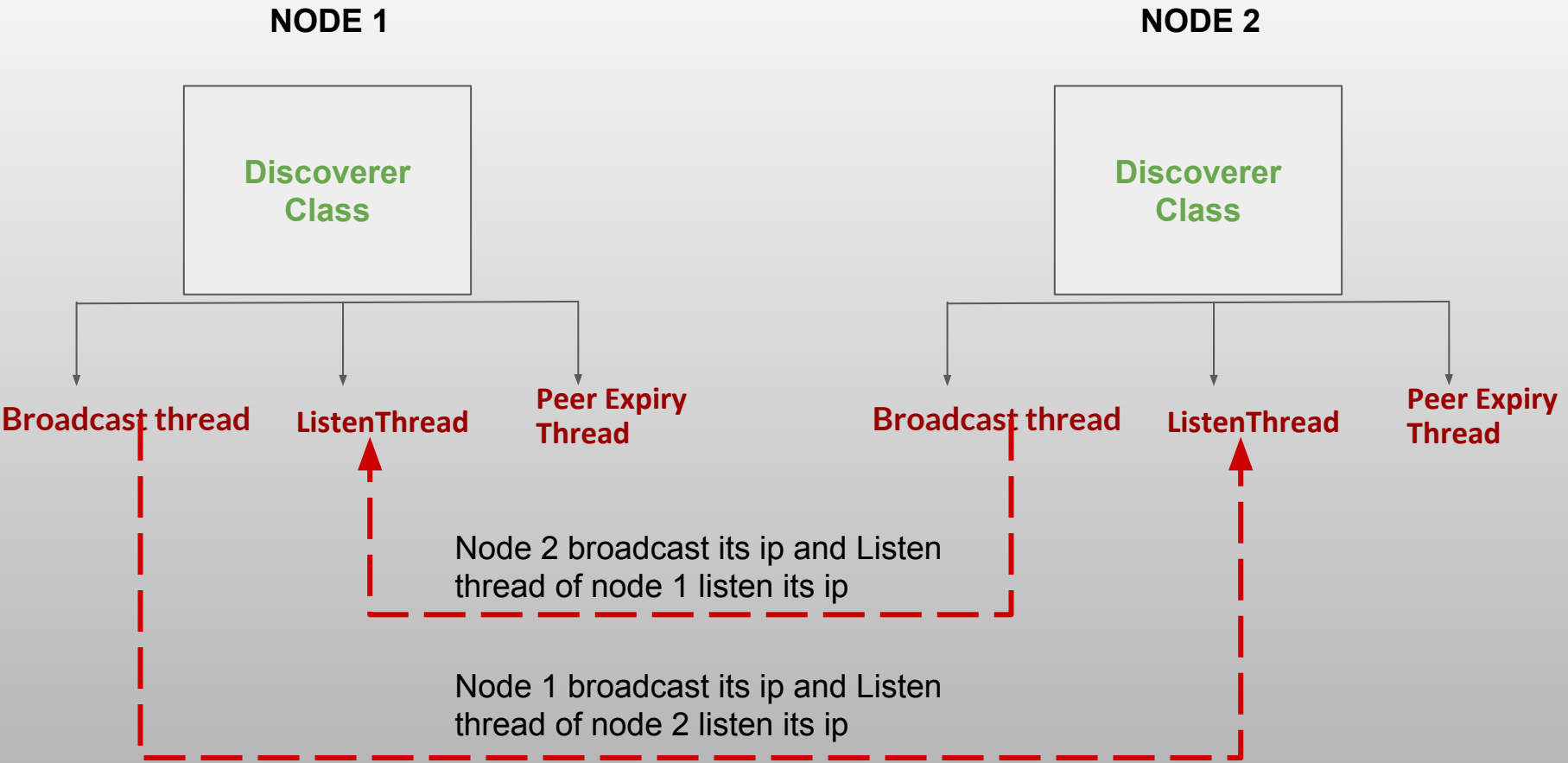


Initially, we had developed the system in which a single server and multiple clients connection were only possible i.e, we could not use each device as both server and client running simultaneously. **The ip address of the server was hard coded in client's code so it could only identify one server. This was major drawback.**

Using broadcasting we could make each device shout their respect internet protocol address and simultaneously receive the internet protocol address of other devices within the range. once a device started receiving the internet address of other devices it starts a new thread for each device for transferring data.

Using broadcasting we made each devices work as both client and server simultaneously .

Broadcasting step by step



//Three main threads (Declaration) :-

```
final Thread[] thread = new Thread[3];              // 3 threads
final BroadcastThread broadcastThread;              // First thread for broadcasting node's ip
final ListenThread listenThread;                    // second thread for listening other node's ip in a network
final PeerExpiryThread peerExpiryThread;            // third one is for peer expiry
```

// Initializing threads

```
broadcastThread = new BroadcastThread(BROADCAST_IP, PORT);
listenThread = new ListenThread();
peerExpiryThread = new PeerExpiryThread();
thread[0] = new Thread(broadcastThread);
thread[1] = new Thread(listenThread);
thread[2] = new Thread(peerExpiryThread);
```

→ **3 Separate threads for broadcasting ip , listening ip and peer expiry**

NOTE :-

1. **final-When a variable is declared with final keyword, its value can't be modified, essentially, a constant.**

Broadcast Thread

1. Our main aim is to determine the IP addresses of all devices within a common network.
2. When a device joins the network, then a new instance of discovery object with specified peer ip (recogniser) is created.
3. startDiscoverer() is initiated.
4. Node starts broadcasting Datagram packet in the network.
5. Broadcasting occurs within specific interval.

Thread to broadcast Datagram packets

```
public class BroadcastThread implements Runnable {
        String BROADCAST_IP;
        int PORT;                                   //port to which socket is to be bound
        DatagramSocket datagramSocket;   //a datagramSocket object is created to carry the packet to the destination and to receive it whenever
                                                        the server sends any data.
        byte buffer[ ] = null;                    //the packet data.
        DatagramPacket datagramPacket; // containing packet data, packet length, internet addresses and port.

       public BroadcastThread(String BROADCAST_IP, int PORT) {
        this.BROADCAST_IP = BROADCAST_IP;
        this.PORT = PORT;
        }

        @Override
        public void run() {
        try {
        datagramSocket = new DatagramSocket();
        datagramSocket.setBroadcast(true);    // Broadcast is on
        buffer = PEER_ID.getBytes("UTF-8"); // UTF-8  -> For converting string to Byte
        this.isRunning = true;
        while(!this.exit) {

                datagramPacket = new DatagramPacket(buffer, buffer.length, InetAddress.getByName(BROADCAST_IP), PORT);
                                                                                                        // buffer.length - > the packet data length.
                try {
                datagramSocket.send(datagramPacket);
                logger.d("DEBUG", "Broadcast Packet Sent");
                }
                catch (Exception e){
                e.printStackTrace();
                logger.d("DEBUG", "Broadcast Packet Sending Failed");
}
```

ListenThread

1.) Listener thread starts listening to the broadcasting from other devices and recognises them.

Thread to listen for broadcasts

```
public class ListenThread implements Runnable    {
     DatagramPacket datagramPacket; // // containing packet data, packet length, internet addresses and port.

      byte buffer[ ];
      DatagramSocket datagramSocket;
  @Override
   public void run() {
            try{
            datagramSocket = new DatagramSocket(PORT, InetAddress.getByName("0.0.0.0"));// get ip address of the host
            datagramSocket.setBroadcast(true);
            datagramSocket.setSoTimeout(200);
            this.isRunning = true;
            while(!this.exit) {
            buffer = new byte[15000];
            datagramPacket = new DatagramPacket(buffer, buffer.length);
            try {
            datagramSocket.receive(datagramPacket); // receiving packet containing information
            System.out.println(datagramPacket.getAddress().toString()); // byte to string
            new Thread(new client(datagramPacket.getAddress())).start();  //New thread    }
```

Peer Expiry Thread

This thread detects if a device exists in the network or not?

This thread runs in specific time-interval and updates the value of Timestamp by 1.

When Timestamp value reaches a specific limitation (user defined) ,then we accept that the device is no longer in the network.

Our implementation for discovery

- We have implemented the discovery class which can detect various peers and show the ip and id of the peer.
- If any peer stops, others can detect the change in the network.
- We have implemented it in android which works properly as it is mentioned.

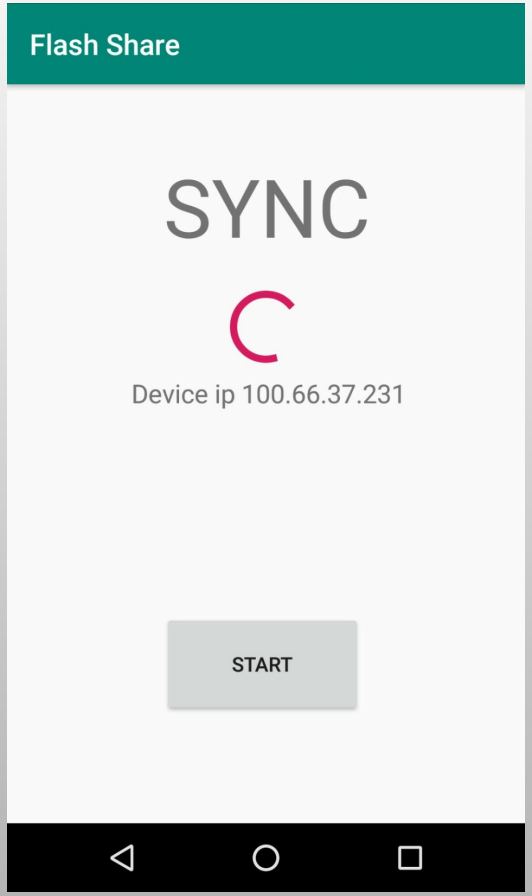
System's Design

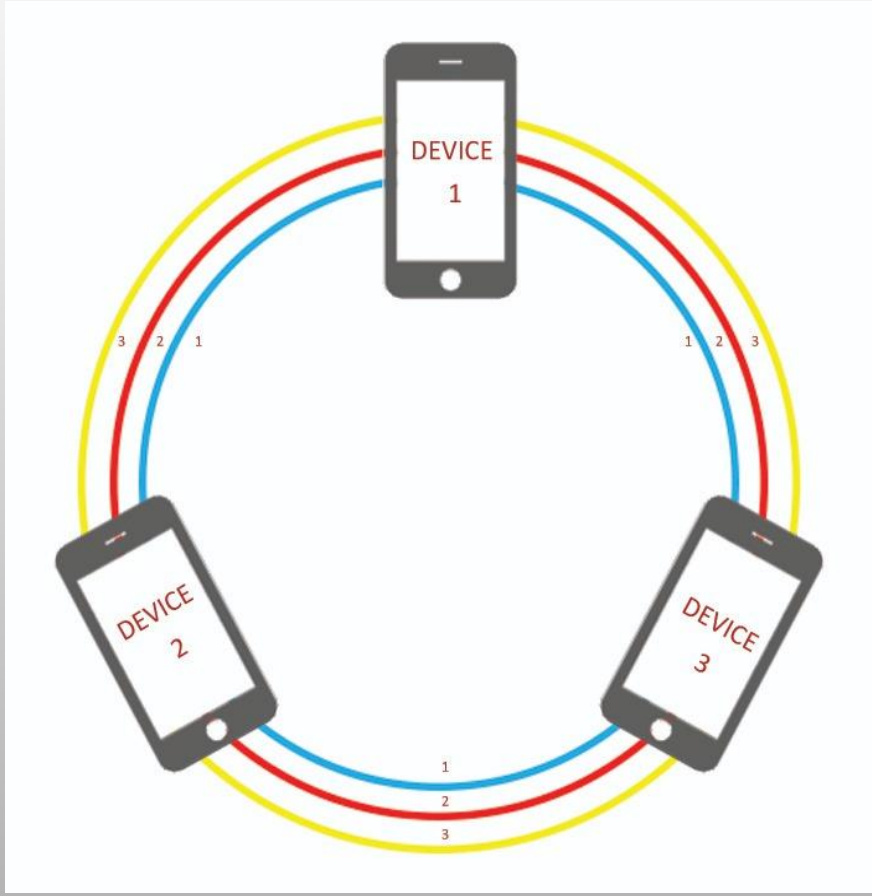
Android application name : Flash Share

1 --> Broadcast own ip

2 -->listen others ip

3 --> Make thread and transfer





let us consider, we have 3 devices connected to a common network say "NIT-1". As soon as we open the application the web server is started.

Clicking on the start button broadcasts the IP of the device i.e, device shouts its IP and receives IP of other devices and makes the new thread for each device for the data transfer.

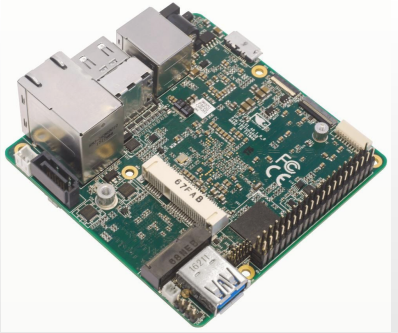
Let each device contains the test folder in the internal storage and lets us assume that "Device-1" contains files {1,2,3,9,8} , "Device-2" contains files {12,13,7,6,3,9} and "Device-3" contains file {11,12,13,15,14,10}.

After broadcasting (clicking the start button), each device shouts its IP and receive other devices IP and connect to each other.
Now, all the three devices should have a list of missing files according to their opposite Node's file list.Then missing files are transferred from one Node to another Node.

So, after syncing, each device should have files {1,2,3,6,7,8,9,10,11,12,13,14,15} in their respective test folder.

A device should not download a file which is already in the file list of the specified node and this should be taken into care by pSync.
This type of synchronization can be observed in heterogeneous devices like – mobile, laptop, raspberry pi etc.

NANOHTTPD



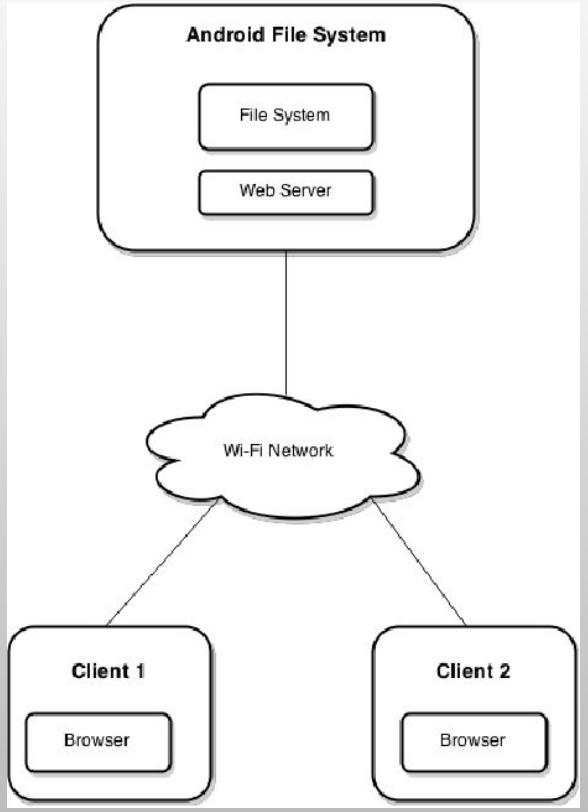
NanoHttpd is an open-source, web server that is suitable for embedding in **embedded device like raspberry pi , up-squared etc. ,smartphones etc** written in the Java programming language. The source code consists of a single *.java* file. It can be used as a library component in developing other software for serving files.

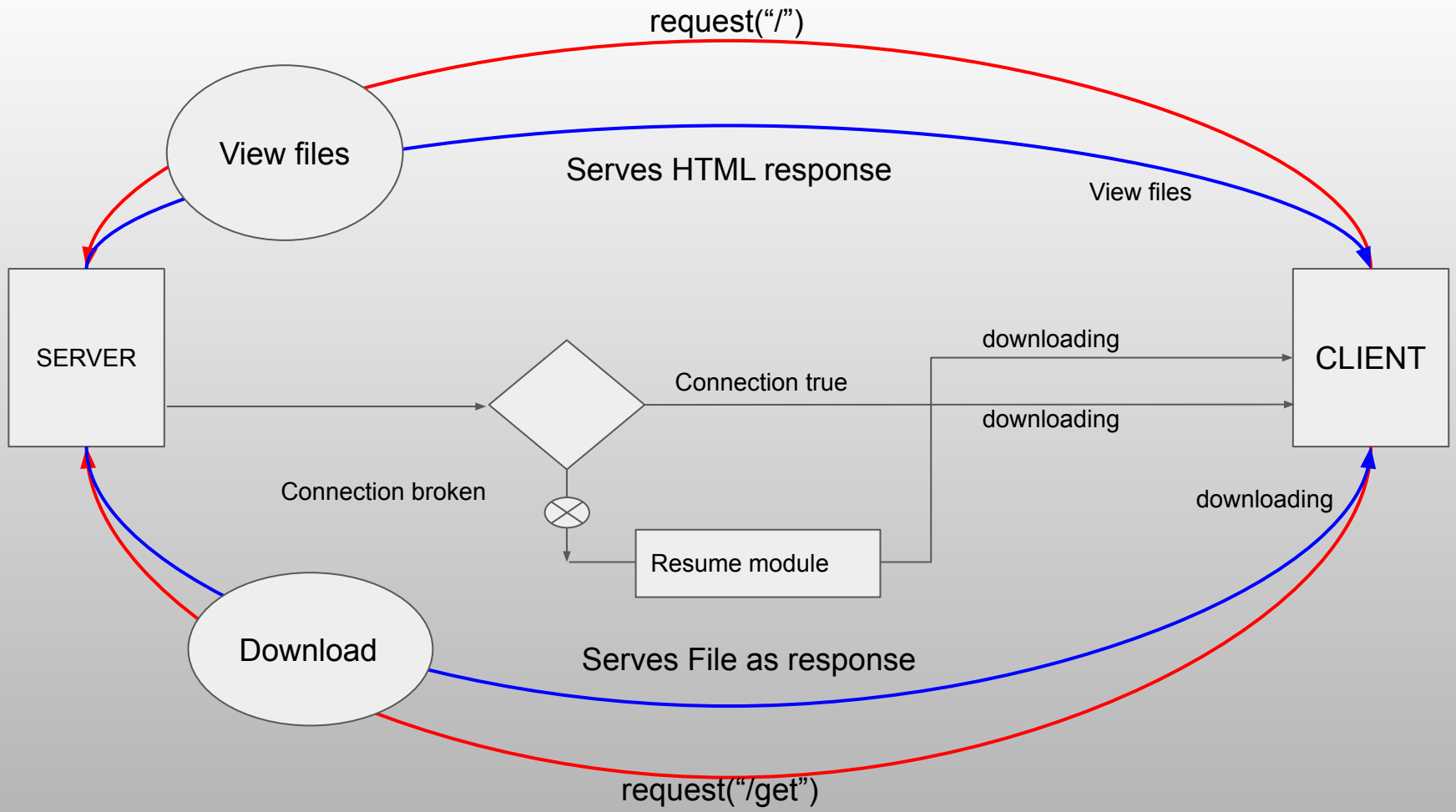
Why NANOHTTPD ?

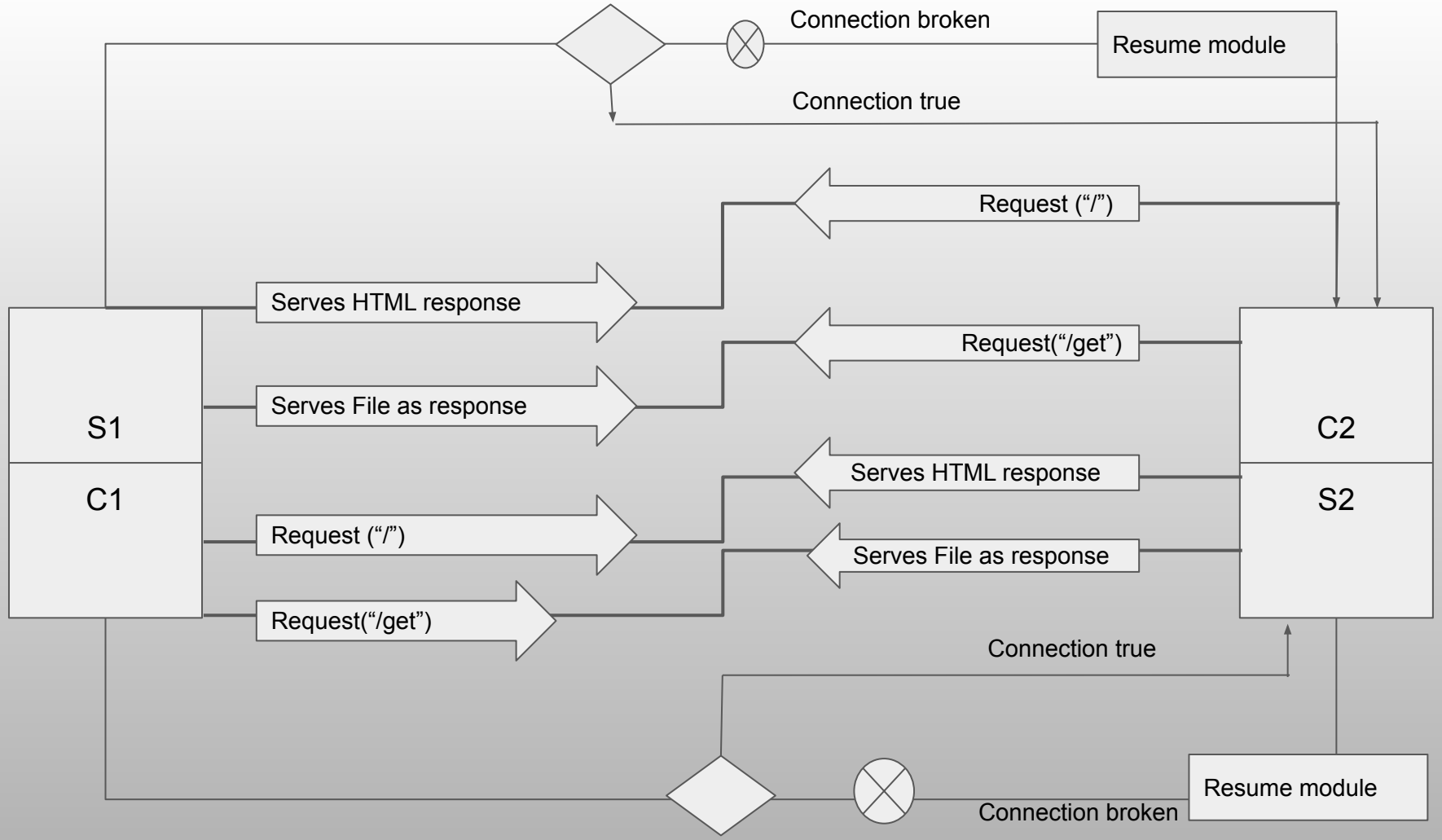
- Due to better functionality of HTTP Protocols at application layer in Network Hierarchy, we have chosen NanoHTTPD server.
- This serves the files to the client peers .
- We have implemented a NanoHTTPD Server where a file can be downloaded when the file name is given in the url.

System architecture

- As shown in figure, our system consists of an android device with http server running on it and client device/s.
- The client need to use the IP address port number displayed on the android device to connect with the server. After connection is established, the server serves the files of that android device to the client browser.
- Then the client can browse through the files displayed.




request(“/”)
View files
Serves HTML response
View files
SERVER
CLIENT
downloading
Connection true
downloading
Connection broken
downloading
Resume module
Download
Serves File as response
request(“/get”)


Connection broken
Resume module
Connection true
Request (“/”)
Serves HTML response
Request(“/get”)
Serves File as response
S1
C1
C2
S2
Serves HTML response
Request (“/”)
Serves File as response
Request(“/get”)
Connection true
Resume module
Connection broken

Basic principle of NanoHttpd

A NanoHttpd Web server uses the client/server model and the World Wide Web Hypertext Transfer Protocol to serve web pages to the client. **The HTTP protocol is a request/response protocol.** A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a successor, followed by a containing server information, entity meta information, and possible entity-body content.

- **Uniform Resource Identifier (URI):** URI is basically a combination of Uniform Resource Locator (URL) and
  Uniform Resource Name (URN). URIs in HTTP is simply used to identify a resource.

- **HTTP Request:** A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

- **HTTP Response:** After receiving and interpreting a request message, a server responds with an HTTP response message. The first line of a Response message is the Status-Line, consisting of the protocol version followed by numeric status code and its associated textual phrase.

HTTP Methods:

- The HTTP **GET** request method is designed to retrieve information from the server. Requests using GET should only retrieve data and should have no other effect on data.
- The **POST** request method is designed to request that a web server accepts the data enclosed in the request message body for storage.
- The **HEAD** method is used to retrieve information same as GET, but transfers the header section only.
- The **PUT** method is used to request the server to store the included entity -body at a location specified by the given URL.

Code Snippet(JAVA)

```
//TO CREATE THE WEBPAGE

if (uri.equals("/"))  {     //if the client requests to show the  files in the server.

 String st = "";

 String x = "";

  for (Map.Entry<Integer, List<String>> en : prio.entrySet()) {  //fetching the names of files from "prio" list

     for (String obj : en.getValue()) {

        x = obj;

       st = st + "<a href=\"/get?name=" + x + "\">" + x + "</a>";

        st = st + "<br>";            }         }    //creates the HTML webpage consisting the name of files.

      return new Response(Response.Status.OK, MIME_HTML, st);   //server returns as reponse a webpage consisting of
                                                                                                  the name  of the files

}
```

//TO DOWNLOAD

```
else if (uri.equals("/get"))   {                //if client requests for downloading of a particular file

        FileInputStream fis = null;

        File f = null;

        try {

          f = new File(Environment.getExternalStorageDirectory().getAbsolutePath()+"/test/"+parameters.get("name"));
          //path exists and its correct

          fis = new FileInputStream(f);

         }  catch (FileNotFoundException e) {

                    e.printStackTrace();          }

          MimetypesFileTypeMap mimeTypesMap = new MimetypesFileTypeMap();

          String mimeType = mimeTypesMap.getContentType(filename);              //getting the MIME type of the file

        return new NanoHTTPD.Response(Response.Status.OK, mimeType, fis);    //server returns as response a fileinputstream
                                                                                    along with the MIME type of the file


        }
```

Resuming download

If a particular file is in downloading stage, some error occured and its not completely downloaded. After reconnecting, the file must not be downloaded from starting (it's not efficient), it must have to continue from the point where the breakdown occurred.

The way we implemented this is as follows :-

```
URL url = new URL(link);

HttpURLConnection http = (HttpURLConnection)url.openConnection();

BufferedInputStream in = new BufferedInputStream(http.getInputStream());

FileOutputStream fos;

if(out.exists())

fos = new FileOutputStream(out,true);      //makes the stream append if the file exists

else {

fos = new FileOutputStream(out);           //creates a new file.                    }
```

File priority

We have included the File priority so that the transfer of the files between server and client can take place according to the priority as follows :-

1. .txt file
2. .pdf file
3. .jpeg/.jpg/.png
4. .mp3
5. .mp4

In our code we have used Map <integer,List<Strings>>. There are basically 5 lists of string. Each of which stores the name of the files in the order given above. Example-

Priority(Int)	**Name of files**
1	A.txt, abc.txt, exm.txt
2	Info.pdf, xyz.pdf
3	Img1.jpg, img2.jpeg, z.jpg
4	list.mp3
5	Go.mp4, rescue.mp4, aid.mp4

```
File folder = new File(Environment.getExternalStorageDirectory().getAbsolutePath() + "/test/");

        Map<Integer, List<String>> prio = new HashMap<>();
        List<String> list1 = new ArrayList<>();
        List<String> list2 = new ArrayList<>();
        List<String> list3 = new ArrayList<>();
        List<String> list4 = new ArrayList<>();
        List<String> list5 = new ArrayList<>();
        for (File file : folder.listFiles()) {
                if (GetFileExtension.getFileExtension(file).equals("txt")) {   // if extension of the file is pdf , priority is 1st
                list1.add(file.getName());
                prio.put(new Integer(1), list1);            }

        if (GetFileExtension.getFileExtension(file).equals("pdf")) {   // if extension of the file is pdf , priority is 2nd
                list2.add(file.getName());
                prio.put(new Integer(2), list2);            }

        if (GetFileExtension.getFileExtension(file).equals("jpg")|| GetFileExtension.getFileExtension(file).equals("jpeg"))           {
                list3.add(file.getName());
                prio.put(new Integer(3), list3);            }                  // if extension of the file is jpg || jpeg ||   , priority is 3rd

        if (GetFileExtension.getFileExtension(file).equals("mp3")) {
                list4.add(file.getName());
                prio.put(new Integer(4), list4);            }                // if extension of the file is mp3 , priority is 4th

        if (GetFileExtension.getFileExtension(file).equals("mp4")) {
                list5.add(file.getName());
                prio.put(new Integer(5), list5);}           }              // if extension of the file is mp4 , priority is 5th
```

Summary vector

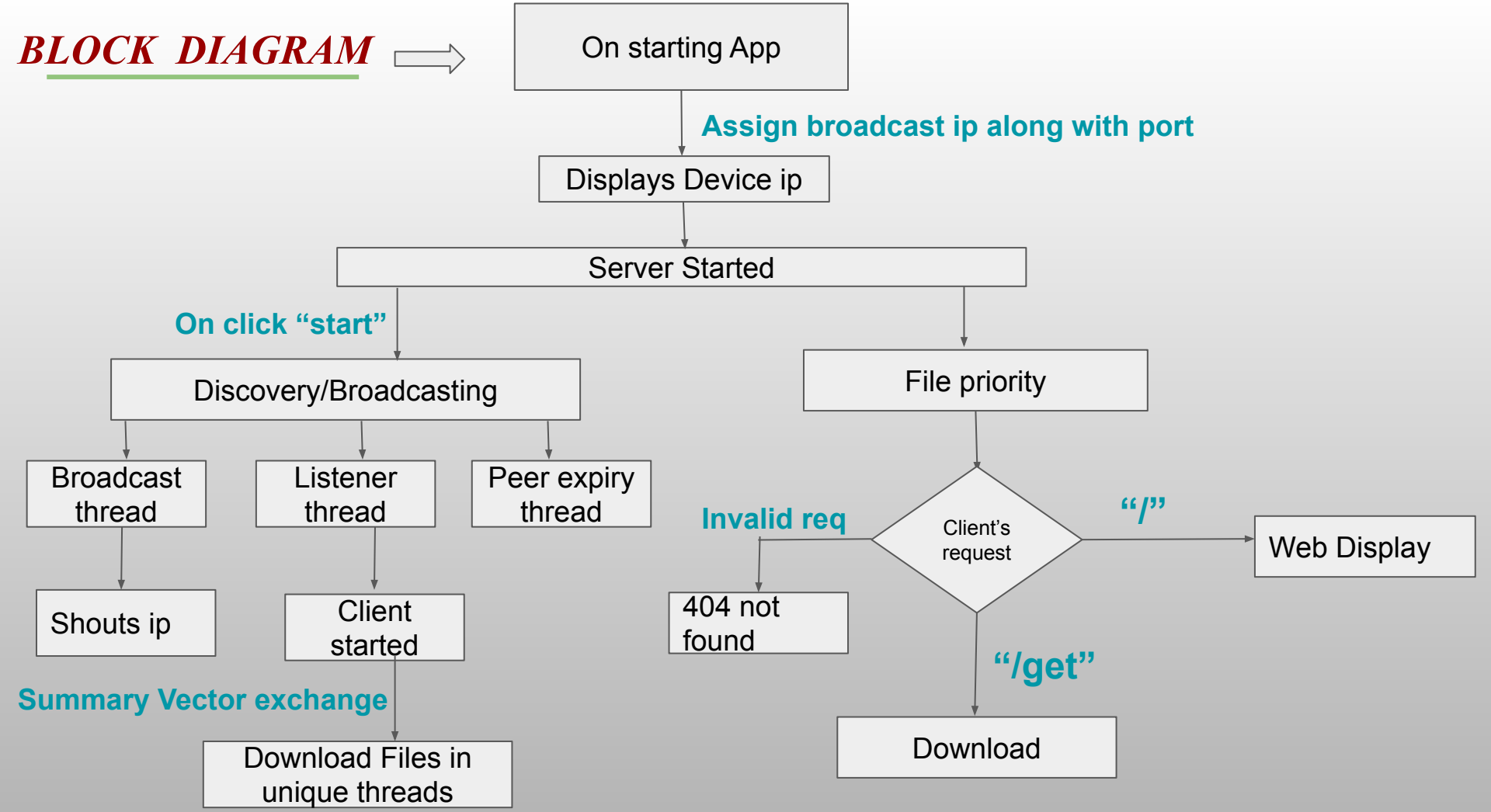
We use summary vector to avoid transfer of common files between nodes. We implemented summary vector as follows : -

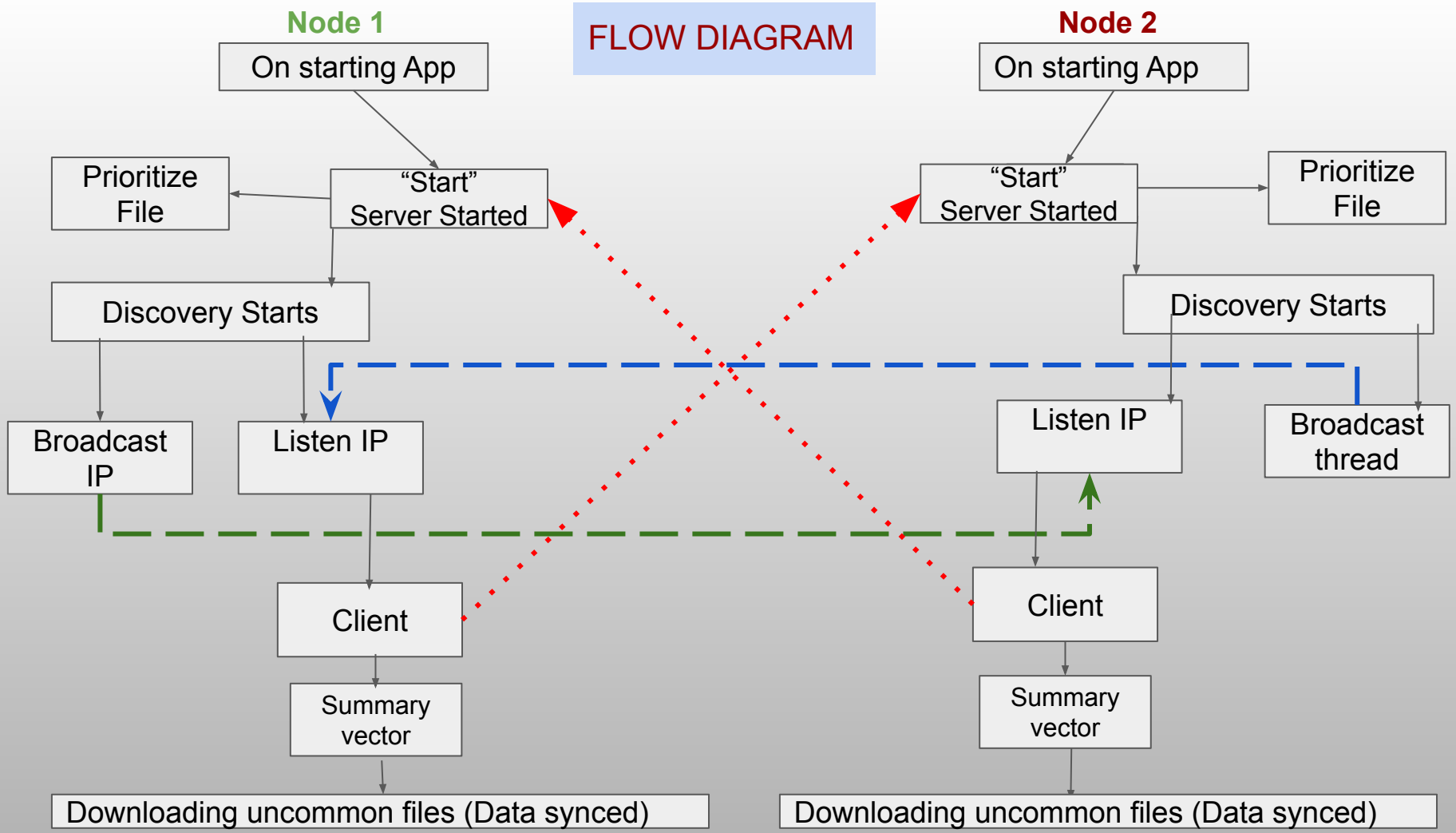
1.Server : Serves its file as web page according to priority.
2.Client: Reads the file from the page stores it in a list, Calculates the difference between the files in its own system and the server, and downloads the uncommon files.

In this way we have implemented summary vector exchange.

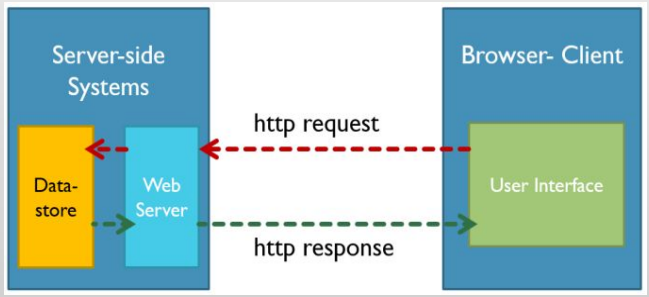
```
Collection A = new ArrayList();                                  //list of files in the Server.
Collection B = new ArrayList();                                  //list of files in the Client.
Collection diff = new ArrayList(A);
Diff.remove All(B);                                              //Calculating the uncommon files
System.out.println("File To be Taken from Server\n"+diff);
Iterator it = diff.iterator();
while (it.hasNext()) {
String x = (String) it.next();
fileURL = "http:/" + IP + ":8080/get?name=" + x;
saveDir = Environment.getExternalStorageDirectory().getAbsolutePath()+"/test/"+x;
File out = new File(saveDir);
new Thread(new download(fileURL, out)).start();
}
```

BLOCK DIAGRAM

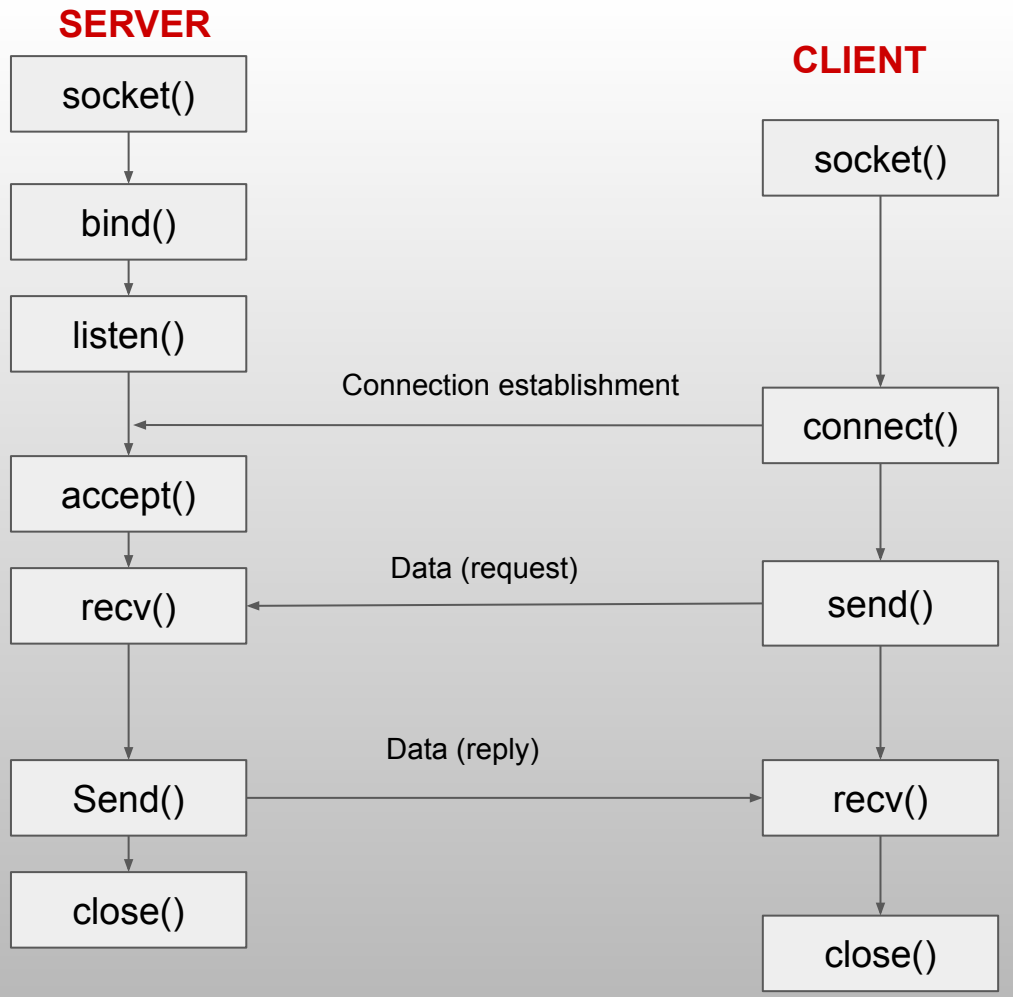



Node 1
FLOW DIAGRAM
Node 2
On starting App
On starting App
Prioritize File
"Start" Server Started
"Start" Server Started
Prioritize File
Discovery Starts
Discovery Starts
Broadcast IP
Listen IP
Listen IP
Broadcast thread
Client
Client
Summary vector
Summary vector
Downloading uncommon files (Data synced)
Downloading uncommon files (Data synced)

FLOW DIAGRAM



Client-Server model



Let's consider the flow of communication between client and server of previous slide

Server:

1. socket() to get socket identifier.
2. bind() bind server to a specific IP and port number.
3. listen() make server listen on port for incoming requests.
4. accept() whenever incoming request is received.
5. Exchange data using send() and recv() functions.
6. Terminate the socket connection using close() function.

Client:

1. socket() function to get socket identifier.
2. connect() to a server listening for requests at specific IP and port.
3. Once connection is established, exchange data using send() and recv() functions.
4. Terminate the socket connection using close() function.

Send data to the client:

Use the `OutputStream` associated with the `Socket` to send data to the client, for example:

```
OutputStream output = socket.getOutputStream();
```

As the `OutputStream` provides only low-level methods (writing data as a byte array), We can wrap it in a `PrintWriter` to send data in text format, for example:

```
PrintWriter writer = new PrintWriter(output, true);

writer.println("This is a message sent to the server");
```

The argument `true` indicates that the writer flushes the data after each method call (auto flush).

Close the client connection:

Invoke the `close()` method on the client `Socket` to terminate the connection with the client:

socket.close();

This method also closes the socket's `InputStream` and `OutputStream`, and it can throw `IOException` if an I/O error occurs when closing the socket.

Of course the server is still running, for serving other clients.

Terminate the server:

A server should be always running, waiting for incoming requests from clients. In case the server must be stopped for some reasons, call the `close()` method on the ServerSocket instance:

```
serverSocket.close();
```

When the server is stopped, all currently connected clients will be disconnected.

Implement a multi-threaded server:

So basically, the workflow of a server program is something like this:

```
ServerSocket serverSocket = new ServerSocket(port);

while (true) {

Socket socket = serverSocket.accept();

// read data from the client

// send data to the client  }
```

The `while(true)` loop is used to allow the server to run forever, always waiting for connections from clients. However, there's a problem: Once the first client is connected, the server may not be able to handle subsequent clients if it is busily serving the first client.Therefore, to solve this problem, threads are used: each client socket is handled by a new thread. The server's main thread is only responsible for listening and accepting new connections. Hence the workflow is updated to implement a multi-threaded server like this:

```
while (true) {

Socket socket = serverSocket.accept();

// create a new thread to handle client socket   }
```