ABSTRACT

A hotspot is one of the solutions if a device without network access (referred to as End User Client in this paper) needs to be connected to the Internet. In hotspot the device without Internet (End User Client) uses the Internet services provided by other device which has Internet connectivity. The bandwidth of the hotspot is shared amongst both the devices. Even though there may be multiple device to connect to for hotspot, a device is allowed to be connected to only a single device. To improve the Internet connectivity and get a higher bandwidth, the End User Client is connected to multiple devices (referred to as peer devices) having Internet connectivity via Wi-Fi Direct and share bandwidth amongst them. The device can use the Internet connection of each of the devices for downloading and uploading data. However, servers would send data to the device only via a single route i.e. via the single device it is connected to. To make use of the bandwidth of multiple devices, the server would have to send subsets of the data via each of the peer devices which would then be combined at the End User Client. This solution is based on the Multipath TCP (MPTCP) protocol. In MPTCP a single device uses multiple adapters (for example Ethernet+Wi-Fi+4G) to achieve multiple connections to the Internet. Here a single device uses multiple mobiles connected over Wi-Fi Direct to achieve multiple connections to the Internet.

1. OVERVIEW

Currently in the smart phone era, it is uncommon for a cellphone user to not have Internet services on their phone. In fact, high speed Internet on a mobile device is the need of the hour. The speed of Internet on 4G network in most cases is around 20 Mbps as checked on speed test, potential speed are much higher. The speed of Wi-Fi Direct is 150 Mbps typical (depending on type of device it can go much higher)[1]. The question then, is how can the Internet speed be improved with this knowledge. The idea is to connect devices together using Wi-Fi Direct so that each user shares bandwidth allocated by each of the devices connected. Consider the example as shown in Figure 1, where each cell phone is connected using Wi-Fi Direct to each phone and simultaneously connected to the Internet via their respective wireless provider. Each cell phone can send and receive data via any of the cell phone connected with Wi-Fi Direct and also via their own wireless provider. Ideally the bandwidth available to each cell phone would be the sum of the bandwidth provided to each of the cell phone by the network provider. For example, if three cell phones have a bandwidth of 20Mbps connecting them via Wi-Fi Direct we could obtain a theoretical maximum bandwidth of 60Mbps.

![Figure 1: MPTCP model having server support MPTCP](image)

The problem with the model in Figure 1 is that with current TCP protocol each device can only create a single connection to the server as shown by the dotted line in Figure 1. Even though there are more available paths to transmit data, the mobile device can transmit data only through one of the paths to the server for a given connection. Simultaneously, even the server is capable of sending data only via one of the paths. To solve this issue, the MPTCP model derived from the MPTCP protocol[2] is used. Using the MPTCP model the device forms multiple TCP paths (depicted with dotted line in Figure 2) to the server informing the server that they have been originated from the same mobile client as shown in Figure 2. If the server needs to send data then it splits data proportionally and sends them to the device using each of the available paths. The mobile device would then receive data from multiple paths and combine them to form...
Figure 2: MPTCP model having proxy server in middle

Figure 3: End User Client

Figure 4: P2P Server

Figure 5: P2P Client

Figure 6: Proxy Server

Figure 7: Proxy Client

the original data. The additional requirement of this model is that the peer devices should support and be connected via Wi-Fi Direct and either an MPTCP capable proxy server or a webserver supporting MPTCP. The advantages are efficient use of bandwidth, avoidance of network congestion, and providing better download speeds for the end user.

The paper is organized as follows. Section 1.1 defines and explains some of the terms used in this paper. Each of the terms depicts its usage strictly only in the model explained in this paper. Section 2 Design Consideration, elaborates on the variety of schemes that were to be used for designing the system which eventually lead to the final design. This final design will be elaborated in Section 4, Architecture, where the design of the proxy server and the mobile device will be explained. Section 5, Implementation, describes the method used to implement the model. The method has 2 separate designs, one for the server and other for client. In Section 6 the results and its explanation have been mentioned. These results are based on strong and weak scaling. Finally in Section 7, Conclusion, we conclude based on the results of this experimental project.

1.1 Definition

End User Client: It is any device on the Wi-Fi Direct network which makes a new client connection to the proxy server. As shown in Figure 3, device A, B, C, D, E, F, G, and H i.e. all P2P devices are probable end user clients. There can be multiple end user clients in a Wi-Fi Direct network and each end user client can have multiple connections to the proxy server.

P2P Server: It is the device in a Wi-Fi Direct network to which all other P2P devices are connected as clients. As shown in Figure 4, devices A and E are P2P servers. Each P2P Server forms its own Wi-Fi Direct network. All other devices are connected to the P2P server in a Wi-Fi Direct network.

Proxy Server: Server which handles MPTCP connections and its sub-flows. In Figure 6, the dotted line depicts the proxy server.

Proxy Client: It is the devices in the Wi-Fi Direct network which is connected to the Proxy server. As shown in Figure 7 devices A, B, C are connected to the proxy server so they are proxy clients. Similarly, E, F, G are proxy clients.
Sub-flows: This term comes from the MPTCP protocol[2]. Every socket connection made from proxy client to proxy server is called a sub-flow as shown by the dotted arrow line in Figure 8. Each end user client may have multiple connections to the server, and each connection may have multiple sub-flows to the server. Each sub-flow belongs to a particular connection. Once the connection is closed the associated sub-flows also close. Consider this example, if an end user client wants to open google.com it would open one connection which would open multiple sub-flows for that particular connection to connect to google. Now if the same end user opens a new tab to open yahoo.com then a new connection would be established to yahoo server which would establish new sub-flows in it.

MPTCP_CAPABLE (term taken from RFC 6824)[2]: The proxy needs to know if the incoming connection is a new connection. Thus an end user client will first send an MPTCP_CAPABLE request to inform the proxy server it is a new connection.

MPTCP_JOIN (term taken from RFC 6824)[2]: If the end user adds new connection to a previously formed connection that was initiated via MPTCP_CAPABLE then it sends an MPTCP_JOIN to the proxy server.

2. DESIGN CONSIDERATIONS

2.1 Wi-Fi Direct

Wi-Fi Direct is technology which supports one-to-one and one-to-many connections without having a wireless access point. Wi-Fi Direct which was initially called Wi-Fi P2P is derived from Wi-Fi CERTIFIED Wi-Fi Direct which is certification for devices which support Wi-Fi P2P technology which can connect 2 devices directly without access point using the same frequency bands provided for Wi-Fi. The speed for data transfer can be as fast as Wi-Fi. In Wi-Fi Direct one device becomes the access point and the other becomes the client. The one behaving as the access point is the Wi-Fi Group owner while the other is the client in the Wi-Fi Direct network as shown in Figure 9. Also the most noticeable thing is that Wi-Fi P2P devices can be simultaneously connected to a Wi-Fi network or to a mobile data carrier while connected to other Wi-Fi P2P devices over Wi-Fi Direct.

Recently many devices are certified with Wi-Fi CERTIFIED Wi-Fi Direct having different hardware and software settings in the devices. The speed, number of peers connected, and behavior depends on the Operating system and the hardware of the device. The speed may vary if the device Wi-Fi supports a, b, g, or n and the speed may be as fast as 250Mbps[1]. Wi-Fi Direct works on both frequencies 2.4MHz and 5MHz based on the device hardware support[3]. The Wi-Fi direct device may support one or many devices to be connected to, therefore the P2P server must preferably be a device which supports more than one connection to itself[3]. Android has added Wi-Fi Direct framework
to its operating system, calling it Wi-Fi peer-to-peer (P2P) which is available for developer to make use of while building apps. This addition is present from API level 14. This new frameworks is in association with the Wi-Fi Direct alliance Wi-Fi Direct certification program [3]. Thus each android mobile using Android OS inclusive and later than API level 14 support Wi-Fi Direct, though the behavior of the devices that support Wi-Fi Direct vary from device to device. This may be due to an update in the framework made by android or in the device hardware. Similarly, Windows phones support Wi-Fi Direct and although the IPhone hardware support Wi-Fi Direct the OS has restricted its usage. Laptops/desktop previously had support for peer to peer connection and have changed their P2P connection by building support for Wi-Fi Direct. Windows 10 has a built-in API for using Wi-Fi Direct which can be written in C#, JavaScript and C++. To check if the particular laptop/desktop hardware supports Wi-Fi Direct the following needs to entered in command prompt console.

```
netsh wlan show wirelesscap
```

If the laptop/desktop supports Wi-Fi Direct then the output would show supported as shown below:

- Wi-Fi Direct Device supported
- Wi-Fi Direct Go supported
- Wi-Fi Direct Client supported

For simplicity in development we have only used Android in this paper.

### 2.2 Wi-Fi Direct supported network architecture

Originally the idea was to connect the devices in a cascade network. If this setting was possible, mobiles could be connected to to form a chain in a Wi-Fi Direct network as shown in Figure 10. This would not have a limit to the number of devices that could be attached to the network.

![Figure 9: Wi-Fi Direct Connection](image)

**Figure 9: Wi-Fi Direct Connection**

Wi-Fi Direct supports a client server model, where a server can have multiple clients, however, clients can have only a single server; An established client cannot form a new connection. This limitation restricts the number of devices that can be attached in a particular Wi-Fi Direct network. The network model formed by Wi-Fi Direct is as shown in Figure 11.

![Figure 11: Wi-Fi Direct Client Server network](image)

**Figure 11: Wi-Fi Direct Client Server network**

### 3. MPTCP PROTOCOL AND ITS DERIVED MODEL

Currently, for a single connection, TCP considers a single path from source to destination. RFC 6824 is a protocol which currently is still on experimental basis and not an Internet standard. The idea of RFC 6824 is to provide TCP with capabilities for using multiple paths to form a connection. Thus the name Multipath TCP (MPTCP). In MPTCP a typical user can use multiple modes to connect to the server simultaneously[2]. For example, the user can use Wi-Fi, Ethernet and mobile data carrier simultaneously. The advantage of this is that the server can send more data from the mode having more bandwidth and less congestion. It solves the problem on connectivity drop where if a user switches from Wi-Fi to Mobile data carrier, the previous connection breaks and a new connection is established[2]. If the user is watching a video or on data call then it causes disruption in the activity. The use of MPTCP would ensure that the new connection would need not be made after the previous one shuts down, instead it exists concurrently. Figure 12 shows the actual MPTCP protocol.

![Figure 12: MPTCP Protocol](image)

**Figure 12: MPTCP Protocol**

Based on this protocol an MPTCP model is designed where instead of using multiple adapters to connect to the web servers, multiple cell phones connected over Wi-Fi Direct are used. Each of the cell phones connected over Wi-Fi Direct have a separate connection to the servers. This design is shown in Figure 13. The additional effort required in this model is the connection to multiple devices via Wi-Fi
Direct which may produce a delay as compared to directly receiving data via multiple adapters.

Figure 13: MPTCP Protocol derived model

3.1 Design of the MPTCP model

In order to make a new connection with the server, the end user client generates a new RCLIENT, which is a random number unique in its Wi-Fi Direct network, then it selects a P2P device in its Wi-Fi Direct network and sends the RCLIENT and MPTCP_CAPABLE request to that device as shown in Figure 14 where device A sends to device B. Device B then forwards the MPTCP_CAPABLE request to the proxy server. The proxy server builds a new connection forming a sub-flow to device B and then generates a random number unique to the server and sends it to the proxy client. The proxy client then sends the RSERVER to the end user client. The end user can then add multiple sub-flows to the new connection formed. To do that, the end user client sends the RSERVER, RCLIENT and an MPTCP_JOIN request to the proxy server using other proxy clients as shown in Figure 14 where device A sends MPTCP_JOIN request to device C. Device C then sends the RSERVER and MPTCP_JOIN request to the proxy server. The proxy server accepts the MPTCP_JOIN and forms a new sub-flow. This new sub-flow is then added to the previously formed connection which is identified by the RSERVER number. Thus now there are 2 sub-flow through which device A can send or receive data from to the proxy server. Note that end user client can also use its own network to connect to the proxy server apart from other P2P devices, thus it can achieve a total of 3 sub-flows to be a part of the connection formed.

Figure 14: MPTCP Handshake

4. ARCHITECTURE

4.1 Design of Proxy Server

4.1.1 MPTCP_CAPABLE

The proxy server is to be capable of accepting multiple connections. The framework is a mimic of the current TCP socket programming code structure for the ease of writing server codes. The server is designed such that it data is fed sequentially which is sent to the end user client in parallel using multiple subflows. When the proxy server receives a new incoming request to create a new connection (MPTCP_CAPABLE), it generates a random number unique to the server. The server then creates a new connection for the new request and adds this new connection to a hash map clientInfo with the connection as the value and the RSERVER as the key. The server can get any connection using the RSERVER as its identifier. The server then adds the socket connection as a sub-flow to the sub-flow list in the new connection created. As shown in Figure 15, every connection has one sub-flow which is the socket connection which provided the MPTCP_CAPABLE request.

Figure 15: Hash Map of proxy server with client connection and its default Sub-flow

4.1.2 MPTCP_JOIN

When the proxy server receives an MPTCP_JOIN request it extracts the RSERVER and uses it to identify the Client connection. This is done by the server, where it extracts the client connection from the hash map clientInfo using the RSERVER provided as the key. The server then adds the socket connection as a sub-flow to the client connection returned from the hash map. Thus now there are two sub-flows in the sub-flow list of the client connection as shown in Figure 16.

Figure 16: Hash Map of Proxy Server with two Sub-flow

4.1.3 SENDING DATA

The proxy server needs to send data to the end user client by using each of the sub-flows available. To be efficient and scalable the proxy server sends more data through the sub-flow with higher speed or more bandwidth. Similar to TCP, the data is divided into subsets of data called packets and sent via each datagram. The sub-flow which sends data faster to the end user client would be sending more packets to the end user client. The design has to be such that the process of sending more data via a faster sub-flow should not be complex and should use minimum resources. To solve this problem the process is handled by the sub-flow thread by letting the sub-flow thread extract data from the client...
connection when it finish sending data to the user. For example, in Figure 17, the sub-flow through device B is twice as fast than sub-flow through device A, thus the server sends twice the amount of data through B than through A. This process is explained by the following example. Let sub-flow A get the first data and begin sending to the client, then sub-flow B gets the second packet and send it to the end user client. As the speed of sub-flow B is higher, it will receive the acknowledgment for the second packet while the sub-flow A would be still be waiting for the acknowledgment of its packet delivery. Sub-flow B then extracts the third packet and sends it to the end user client. Sub-flow B would receive the acknowledgment for the third packet with a slight delay as per when sub-flow A received acknowledgment for packet one. Eventually, twice as much data is sent through sub-flow B than through A.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP address of peer 1</td>
<td>Socket connection 1</td>
</tr>
<tr>
<td>IP address of peer 2</td>
<td>Socket connection 2</td>
</tr>
</tbody>
</table>

**Figure 18: Hash Map of Wi-Fi Direct peer sockets**

check if it is connected to the Internet. After calculating the time taken for the P2P client ping also called as round trip time (RTT), the client sends this data to the P2P server. The P2P server then adds this data to its routing table which consists of Client IP, IP of next mobile to reach there (usually the P2P server itself) and the ping/RTT time. Every time the server is updated it broadcasts its routing table to all P2P clients who then store the routing table and use it for routing, and to get information of devices in the Wi-Fi Direct network. This process of server requesting for RTT, P2P client finding the RTT, updating the P2P server routing table, and broadcast to all P2P client takes place every 30 seconds. A typical routing table is shown in Figure 19.

**Figure 19: Routing table of P2P Devices**

<table>
<thead>
<tr>
<th>IP address</th>
<th>Next IP</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.12.49</td>
<td>192.168.13.29</td>
<td>1234</td>
</tr>
<tr>
<td>192.168.13.29</td>
<td>192.168.13.29</td>
<td>1233</td>
</tr>
<tr>
<td>192.168.15.59</td>
<td>192.168.13.29</td>
<td>Inf</td>
</tr>
</tbody>
</table>

**Route Data**: Each device in the network acts as a router i.e. it first checks the IP address of the the packet it received; if the IP is not same as the IP of the device itself then it checks in the routing table for the route. There are only 2 type of data routing, one is where the the proxy server sends the data to the proxy clients and other when P2P devices send data to other P2P devices. When the server sends data to the proxy client it extracts the IP address of the destination from the data it stores along with RCLIENT. When the data is received from another P2P device, the P2P device extracts the destination given in the sub-destination of the datagram header. If the IP address is not the same as of the recipient IP address then the device searches for the IP in its routing table and obtains the next hop IP. In this model, the next hop IP can be the address of the final destination device or IP of the P2P server. The data then eventually is transferred to its recipient or to the P2P server which indirectly transfers it to the recipient. As shown in Figure 20 the P2P server gets a packet intended for other peers then it forwards it to the intended peer.

**4.2.2 Mobile as a MPTCP Initiator**

**Initiate MPTCP_CAPABLE handshake**: The end user client sends a new MPTCP_CAPABLE handshake request to the proxy server via the P2P device selected having the smallest RTT. It creates a new datagram which encloses the information like RCLIENT and the MPTCP_CAPABLE request and sends it to the selected device. The selected device then acts as proxy client and establishes a socket...
connection with the proxy server and stores that connection in a hash map with RCLIENT as key. When the proxy client receives data from the proxy server, it uses the same RCLIENT to get the IP of the end user client and then forwards the RSERVER to the end user client and thus forming a sub-flow for the new connection. This process is depicted in Figure 21.

![Figure 20: Device as a router](image)

![Figure 21: Device initiating an MPTCP_CAPABLE request](image)

**Initiate MPTCP_JOIN handshake:** After creating a new connection and adding the default sub-flow to it, the end user client then selects all the other P2P devices in the routing list having limited RTT which was tested during broadcast. The end user client creates a datagram having information of the RCLIENT, RSERVER, and MPTCP_JOIN request to the selected devices. The devices receive the request and then act as proxy client and establish a new socket connection to the proxy server. The proxy server then stores this connection in hash map with RCLIENT as key and then forwards the data to the proxy server. When the proxy client receives data from the proxy server it identifies the end user client IP address stored along RCLIENT and connection. The proxy client then transfers the acknowledgment to the end user client and thus forming a new sub-flow under a previously created connection. This process is depicted in Figure 22.

![Figure 22: Device initiating an MPTCP_JOIN request](image)

![Figure 23: Data from End User Client to Proxy Server via sub-flow](image)

**Receive Data:** Similar to request data, the proxy client accepts data from the proxy server. The listener of the proxy server stores the RCLIENT from which the IP address of the end user client can be retrieved. The device which acts as proxy client then acts as P2P device and forwards the data to the end user client. The end user client then responds with an acknowledgment to the data sequence to the proxy server via the same sub-flow. This process is depicted in Figure 24.

![Figure 24: Data from proxy server to End User Client via sub-flow](image)

**Request data:** End user client uses the sub-flows to send data to the proxy server. The data consists of request parameters which the end user client wants to download from the proxy servers. The end user client then selects one of the sub-flows to send data to the proxy server. The end user client adds the data sequence to the datagram and sends to the selected sub-flow. This end device in a sub-flow acts as a P2P device and as a proxy server. It is a mediator between the 2 networks the Internet and the Wi-Fi Direct network.

**4.2.3 Mobile as a Sub-flow**

**Receive Data:** Similar to request data, the proxy client accepts data from the proxy server. The listener of the proxy server stores the RCLIENT from which the IP address of the end user client can be retrieved. The device which acts as proxy client then acts as P2P device and forwards the data to the end user client. The end user client then responds with an acknowledgment to the data sequence to the proxy server via the same sub-flow. This process is depicted in Figure 24.

![Figure 23: Data from End User Client to Proxy Server via sub-flow](image)

![Figure 24: Data from proxy server to End User Client via sub-flow](image)

## 5. IMPLEMENTATION

This section provides a brief overview of techniques used to implement the design as mentioned earlier. The implementation is divided into 3 parts. First is the implementation of proxy server, second is implementation of mobile devices, and third is designs which are common to both mobile and server. Here we refer the P2P devices as Mobile device as our end goal is to test on Android device. As the implementation is done for Android, and Android API is written in java, java language was selected for the mobile code. To be consistent even the proxy server is written in java too. The implementation took place in 2 phases. First, the feasibility of the project was tested in Android with a
GUI. Then the project was developed, run and tested in a stable environment.

5.1 Phase 1: checking feasibility and working

5.1.1 Discovering Peers

In order to perform testing on Android, a simple GUI was built having a single task (on click of button) discover all peers in the vicinity[4]. The GUI appeared as shown in Figure 25 on which, if Discover Peers was clicked would show peers in the vicinity as shown in Figure 26.

Figure 25: Screen shot of Discover Peers GUI

For the above mentioned tasks, the following code was written. First in the onCreate method access to the device WiFi P2P API was requested[4].

```java
//code to set the WiFi P2P manager and receiver
mManager = (WifiP2pManager) getSystemService(Context.WIFI_P2P_SERVICE);
mChannel = mManager.initialize(this,
    getMainLooper(), null);
mReceiver = new WiFiDirectBroadcastReceiver
    (mManager, mChannel, this);
```

When the Discover Peers button is clicked the following code is run. The reason code as mentioned in the following is 0 if error in discovery, 1 Wi-Fi device is unavailable or P2P is unsupported, and 2 is busy[4]:

```java
mManager.discoverPeers(mChannel, actionListener);
```

Connecting tests were performed to check if 2 devices could be connected to the proxy server and to the P2P device simultaneously. The proxy server was setup with returning a String to the the proxy client, the proxy client then acts as P2P client and sends data to the P2P server which was received successfully and displayed using Toasts. The next test was to check if multiple mobiles could be connected simultaneously to the P2P server. The issue here is that by entering the code that follows the device selecting the button of the peer should encourage the other device to become the P2P server as provided in the documentation, however this is not the case. Any of the device may the be group Owner also called as P2P server[4].

```java
config.groupOwnerIntent = 15;
```

Once devices are connected then we show if the device is the P2P server by querying if the device is the group owner. The return of the query is displayed along with the name of the device connected. Connection of P2P Server to multiple devices is shown in Figure 27 and the P2P client connection to the P2P server is shown in Figure 28 where true means the device is the P2P server.

Figure 27: Screen shot of P2P Server

Figure 28: Screen shot of P2P Client

5.2 Phase 2: Implementation on stable environment

5.2.1 Implementation of Common Code

Datagram: In order to send data and information across a socket, a datagram has been designed as shown in Figure 29. The data and information is encapsulated into datagram to which a TCP Header is attached for point to point delivery. The datagram is designed such that the first four bytes hold the size of the remaining datagram. The remaining part of datagram can be as large as number of bytes
an integer can hold. Thus next 4 bytes hold the size of the information data in bytes. The information byte is a simple custom build JSON string. This JSON stores single key single value or single key multiple value pairs. The remaining part of datagram is the data that is to be transmitted. The main function of datagram info JSON is to carry information such as the IP, RCLIENT, RSERVER, query, query response, acknowledgments byte order and sequence information of data. The IP includes the IP of Sender, Receiver, or sub-destination i.e. IP of the end user client or the proxy server. The query contains query like, MPTCP_CAPABLE, MPTCP_JOIN requests and their response acknowledgments. The information in a datagram is

Figure 29: Design of Datagram

stored as follows:

```java
//JSON information in bytes
byte[] infobytesJson;

//data
byte[] data;

//the complete datagram after combining the JSON
//size+JSON bytes+data bytes
byte[] bytesUsedToTransfer;
```

Using the above data bytes, a single byte array is created which can be used to transfer the above data across the sockets.

```java
for (int j = 0; j < infoLengthInBytes.length; j++, i++) {
    bytesUsedToTransfer[i] = infoLengthInBytes[j];
}
for (int j = 0; j < infobytesJson.length; j++, i++) {
    bytesUsedToTransfer[i] = infobytesJson[j];
}
for (int j = 0; j < data.length; j++, i++) {
    bytesUsedToTransfer[i] = data[j];
}
```

Each time the application sends a datagram, it adds the datagram details via the JSON and calls the sendDatagram method as shown below:

```java
//sending datagram
datagram.sendDatagram(s.getOutputStream());
```

The sendDatagram method first sends the size of datagram and then sends the entire datagram bytes across the sockets as shown below:

```java
out.writeInt(bytesUsedToTransfer.length);
out.write(bytesUsedToTransfer);
```

On the receiver side, the data in datagram is extracted by calling the readNewDatagram method which extracts the JSON information and then again waits on the socket till a new datagram is received:

```java
//calling read datagram
Datagram datagram = read.readNewDatagram(s.getInputStream());
```

**Sequence:** The major problem in asynchronous data transfer is keeping track of data received. One way of doing it is to keep an index which would assign a sequence number to the byte received. An integer consumes 4 bytes, which would require storing additional 4 bytes per byte. This method is not space efficient. By using Sequence class, a list is maintained which stores the small and large integer which determines the range of data received. Data is sent in chunks, thus on the receiver side the start sequence number and the end sequence number is stored. If another data chunk received consists of sequence number which is adjacent to the previous chunk then instead of storing the new data sequence, the previous data sequence is modified to accommodate for the new chunk sequence. For example, if byte 0 to 10 is received first, then we store the a new object sequence whose small value is 0 and large value is 10. Now if the receiver, receives data chunk 11 to 20, then instead of adding a new object sequence, the previous object sequence is modified having small value as 0 and large value as 20. This indicates that values 0 to 20 has been received.

When the user extracts data the data byte and particular object sequence is also removed from the list. This makes storage of index space efficient.

### 5.2.2 Implementation on Proxy Server

The proxy server code is to be kept similar to how the SocketServer code exists.

ServerSocket server = new ServerSocket(Integer.parseInt(port));
while(true){
    new SocketSubflow(server.accept()).start();
}

The above code is similar to how the current ServerSocket calls a thread to handle a new request. The difference is that traditionally the new thread would have the socket passed in its constructor and it the thread would then handle the request, however, here we call SocketSubflow thread which is responsible for handling the new sub-flows for each connection. It manages the MPTCP_CAPABLE and
MPTCP_JOIN requests. The ClientThread receives the request and responds by reading from and writing to the ClientConnection (c) analogous to Socket in traditional TCP connection. The read and write are shown in below code where b is the bytes to be read to and the bytes to be sent to the end user client.

```java
int bytesread = connection.read(b);
connection.write(b);
```

When the proxy server receives a request it creates a sub-flow, the sub-flow then checks if it is an MPTCP_CAPABLE or MPTCP_JOIN request. If the request is MPTCP_CAPABLE then the proxy server has to start a new ClientConnection and a new ClientThread to handle the incoming request as shown below:

```java
// creating new client connection
client = new ClientConnection();

// finding unique random number rServer for new client
generatorServer(rServer);

// adding rServer to the ClientConnection
client.rServer = rServer;
client.rClient = dataRecieved.json .getValue("RCLIENT");

// Calling new thread to read from and write to socket
startReadAndWriteSocket();

// sending Rserver to client
Datagram sendRServer = new Datagram();
sendRServer.json.put("RSERVER", rServer);
sendRServer.json.put("ACTION", "MPTCP_HANDSHAKE_ACK");
sendRServer.sendDatagram(new DataOutputStream(s.getOutputStream()));

// calling ClientThread for executing the new ClientConnection
ClientThread clientThread = new ClientThread(client);
clientThread.start();
```

Similarly, if the incoming request was for MPTCP_JOIN then the proxy server, would extract the RSERVER from the request and would add the newly created sub-flow to the ClientConnection as returned from the hashmap with key as RSERVER as shown below:

```java
//identifying the client connection based on RSERVER
ClientConnection tempClient = ClientConnectedInfo .subflowManagerThreads.get(rServer);

//Calling new thread to read from socket
startReadAndWriteSocket();

// sending MPTCP_HANDSHAKE_ACK to client
Datagram sendRServer = new Datagram();
```
the random unique client number (RCLIENT) to the proxy server via the P2P device. Once the device establishes a connection it acts like a proxy client and is added to the sub-flow list of the particular client. The proxy client uses the RCLIENT to identify the end user client and the socket connection to the proxy server. The end user client uses the RCLIENT to store and retrieve the client connection from a hash map with RCLIENT as key and client connection as value. The client now sends the request data via this new sub-flow to the proxy server. In parallel, the client also queries the remaining P2P devices to request an MPTCP JOIN and form new sub-flows. These new sub-flows are used to transfer data in parallel to the end user client.

**Mobile socket Writer** When a socket connection is established by a P2P devices, it stores the socket connection of each device. The socket connection consists writer and listener thread. The writer thread consists of a buffer in which data to be sent is added. When buffer is empty the writer waits for the data to be added. Once the data is added the writer is notified and the writer then send the data using the socket established.

**Mobile socket listener** The listener of the mobile is the heart of the entire mobile application design. The listener performs various tasks like initiating new MPTCP JOIN, adding data to the writer, writing data to the buffer for the end user client to read, sending acknowledgment, sending negative acknowledgment and so on. On receiving data the receiver first checks if the destination IP matches the IP address of the P2P device. If it does not match then it looks up the next node to forward the datagram to so the data reaches the device with the IP same address. It then extracts the socket connection with that IP address and places the data in the writer list of the particular socket connection and notifies the writer to send. If the IP address of the datagram destination matches the IP address of the P2P device then the listener reads for the action to be performed over it.

If the action is MPTCP_CAPABLE or MPTCP_JOIN then the P2P device acts as a proxy client and then establishes new socket connection with the proxy server. This new socket connection will be stored in the hash map with RCLIENT as the key.

If the action is DATATOSERVER, NACK, or DATATOCLIENTACK then it reads the RCLIENT from the datagram info JSON and pulls up the appropriate socket connection to the proxy server and then places the data in the writer list of that socket that notifies the writer.

If the action is MPTCP_HANDSHAKE_ACK, DATATOSERVER_ACK, or DATATOCLIENT then the listener retrieve the RCLIENT from the datagram and extracts the client connection from the client connection hash map.

If it is an MPTCP_HANDSHAKE_ACK, then it adds the IP address of the P2P device sending the datagram to the sub-flow list.

If it is DATATOSERVER_ACK then the listener notifies the end user client that the data is sent to the proxy server successfully.

If it is DATATOCLIENT, then the listener checks if it the finish datagram or if it contains data. If the finish is false, then it adds the data to the data buffer and updates the sequence appropriately. If the finish is true, then it checks if all the data in the particular byte order has been received, by checking the sequence class for data. Then the listener will reset all the buffer and sequence and will increment the byte order to receive the next byte order. If the IP of the destination is null, then it means the data is arrived from the proxy server. The device then accessed the RCLIENT stored with that socket connection and forwards the datagram to the end user client associated with that RCLIENT.

### 6. RESULTS

**Scaling:** As the idea of increasing the number of mobiles is analogous to increasing the number of cores in parallel processing, the same metrics has been used to measure scalability as that of parallel processing. To measure scalability in parallel processing two metrics are followed in general. They are as follows[5]:

- **Strong Scaling:** Letting the size of work to be the same and increasing the work being done in parallel. The idea is to reduce the time consumed in processing the work. In our case,

- **Weak Scaling:** Increasing the work to do as we increase the processing being done in parallel. The idea is to keep time constant, however process more work in same time.

The Strong scaling results are shown in table 30 and the Weak scaling results are shown in table 31. The efficiency when adding a new mobile to download the data almost doubles, however, after that when more mobiles are added to form sub-flows, the efficiency increases but at the same a linear rate. The reason behind is that as the number is sub-flows increases, the sequential overhead at the sender and receiver, blocks new data to be removed and added respectively unless the previous threads finishes so. Thus it has to wait for the previous thread to finish and only then it can proceed.

<table>
<thead>
<tr>
<th>No of Mobiles</th>
<th>Data Size (Mb)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19</td>
<td>148</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>77</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>55</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>44</td>
</tr>
</tbody>
</table>

**Figure 30: Strong Scaling**

<table>
<thead>
<tr>
<th>No of Mobiles</th>
<th>Data Size (Mb)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>148</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>146</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>166</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>177</td>
</tr>
</tbody>
</table>

**Figure 31: Weak Scaling**

### 7. CONCLUSION

MPTCP model has successfully been used to create multiple multiple TCP connections across different mobiles. Using these simultaneous TCP connections data can be transmitted parallely. The data is split at the sender side and a
subset is sent over each of the connections formed. The data is then assembled at the receiver. In this process the total delay caused by network is reduced as data is sent in parallel and better internet speed and connectivity.

Based on the results received from the experiment, using multiple mobiles would provide much better result in terms of speed as the bandwidth is shared among each of the mobiles. Using multiple mobiles can also be used to increase connectivity as there are multiple subflows to the server, if one fails the other can be used. As the server choses to send more data from the path which is less congested, this method is very usefull in avoiding congestion, provided that each of the mobile use different paths to connect to the web server.

8. REFERENCES