Advanced Distributed Systems
Inter-Domain Handoff Between Wi-Fi and 3G

Steve Ajemian
Bhuwan Agarwal

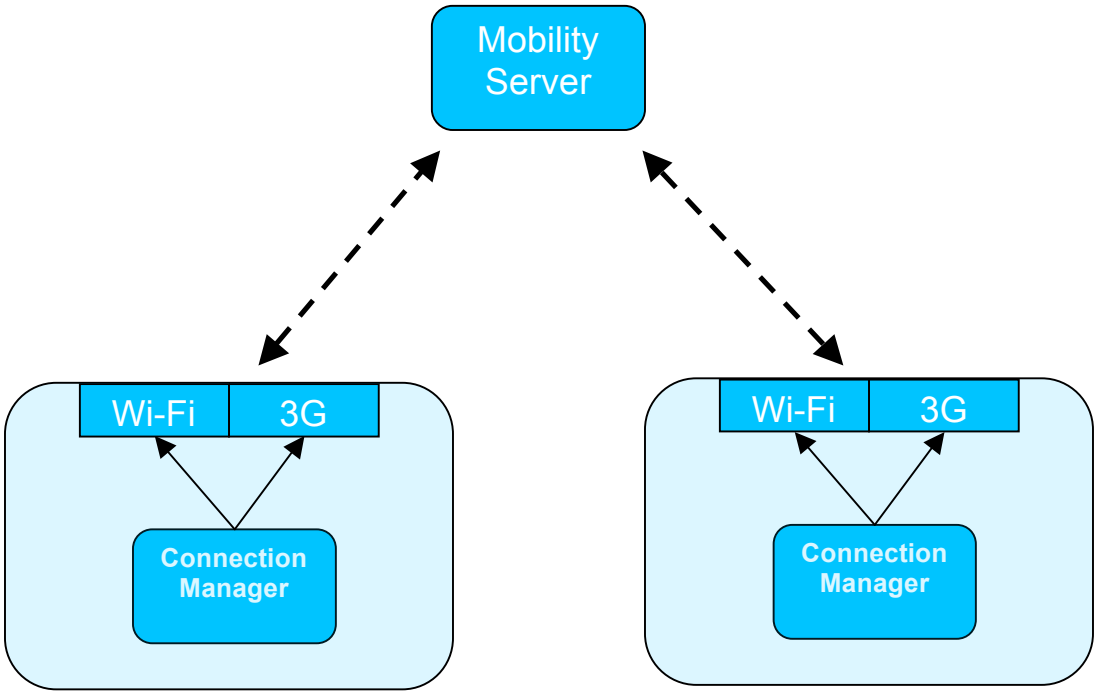
Project Overview

The goals of this project were to examine the performance of inter-domain handoff between Wi-Fi and 3G networks on smartphones. An increasing number of mobile devices are providing internet access from over both Wi-Fi and 3G. This trend has led to interest in looking at optimum ways to use these networks for a variety of needs. In this project, we wanted to examine the feasibility of using handoff for a VOIP application. We considered the case of two users having internet access on both of these interfaces, with the need to conduct VOIP phone calls. An additional requirement is to provide this capability while preferring Wi-Fi for the session if it is available. The desire to move data services to the Wi-Fi domain from 3G is due to the user wanting to minimize the use of data services over 3G.

Platforms

For our implementation, we used two HTC Tilt smartphones that are Wi-Fi enabled. These phones also provide data services over AT & T's 3G network using High-Speed Downlink Packet Access (HSDPA). This service provides data rates of 1.8, 3.6, 7.2 and 14.4 Mbit/s. The smartphones use Windows Mobile 6.1 for the operating system.

Communication between these devices is mediated by a server in the middle. For the server application, a linux machine was used running Ubuntu. This server also has a known public IP address that can receive TCP and UDP packets from the participating clients. A block diagram of the configuration is shown below:



Protocol

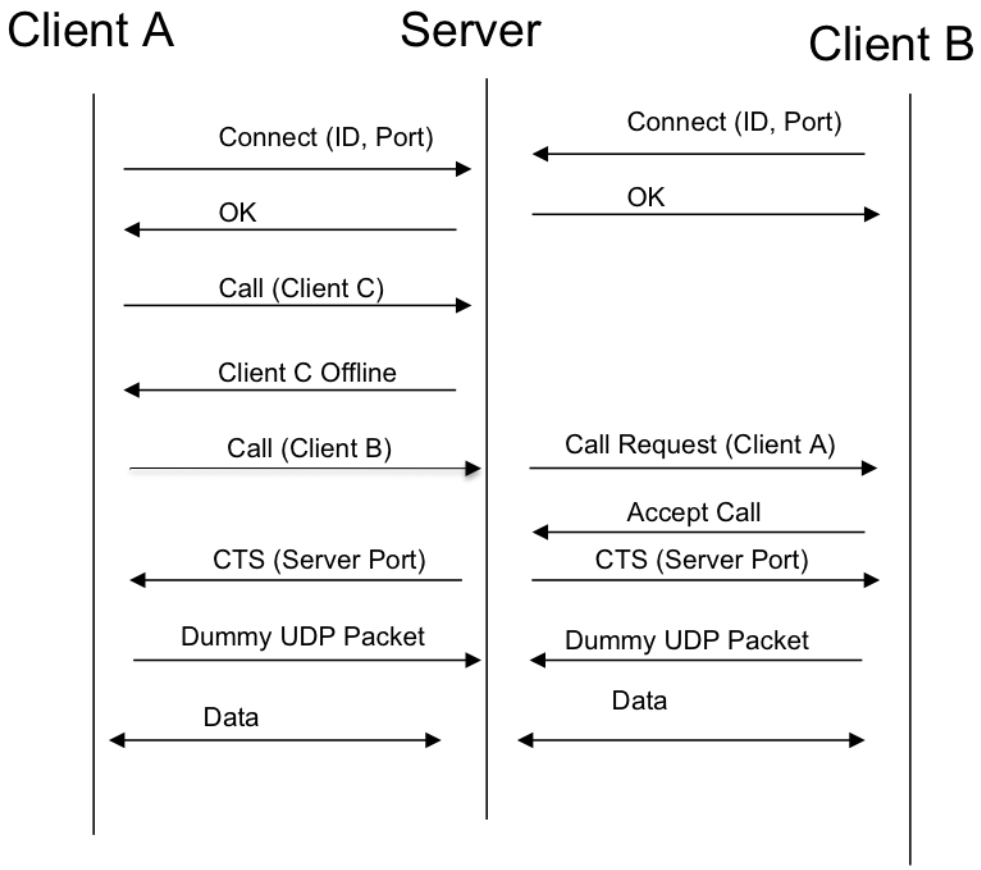
In order for two users to conduct a VOIP session using the server in the middle, three major stages are required. These stages are **Start Up**, **Handoff**, and **Termination**. The client and server applications provide the following capability:

1. Each party will continually send packets every 20 ms to simulate a VOIP packet to the Mobility Server, which relays packets to each party.
2. If one party detects a Wi-Fi access point, it will join AP and begin sending/receiving packets over the Wi-Fi interface.
3. The Mobility Server will terminate the 3G data link.

Note: The above capability is also provided when transitioning from Wi-Fi to 3G in the event of a loss of Wi-Fi access.

- **Start Up**

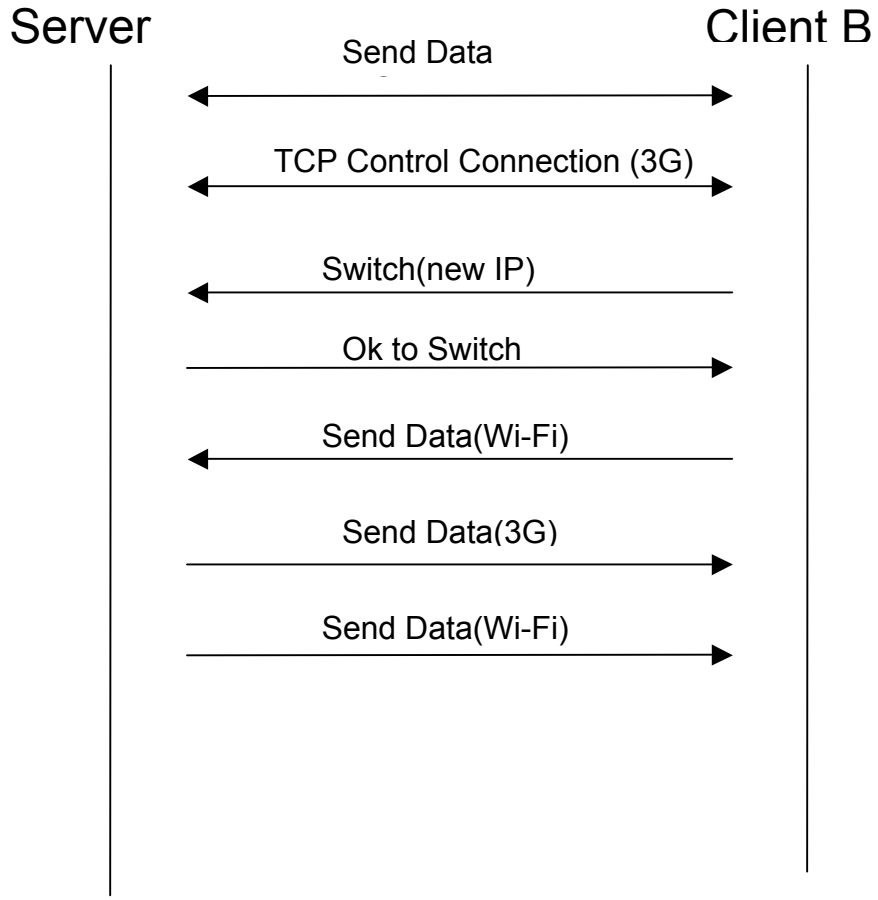
  The first stage, Start Up, requires each client to conduct a handshake with the server. The handshake with the server involves setting up a TCP connection for control messaging and a UDP port assignment. The server provides each client with the port to use for sending UDP packets for the VOIP traffic. Furthermore, each client provides the server with its port for receiving the VOIP traffic. The server obtains each client's IP address from the TCP connection. Once two clients have connected to the client, the control channel remains open until the data session begins. In order for a "call" to start, one client must make a call request which must be accepted by the client receiver. The server will provide the UDP port for receiving traffic once the call is accepted. The port assignment message from the server implicit provides a "clear to send" to both clients. A sequence diagram of this Start Up protocol is illustrated below.



In the above sequence diagram, the following exchange occurs:

1. Client A opens a control connection over TCP and sends start message over TCP Control Connection, sent to the well-known port on Server. Client B also opens a control connection. In this message both client A and B also sends their port numbers on which they will be listening for the UDP traffic.
2. Client A sends Call request to a client B, which must be online.
3. Depending upon the response of client B (Accept/Reject) server will either send a CTS message to both clients A and B, or a reject message to client A.
4. The CTS message will contain the server port number to send UDP packets to. The server will begin listening for data packets prior to sending the CTS message. After receiving the CTS message both clients will begin sending/receiving data.
5. Each client must send a "dummy" packet to the server to enable it to receive UDP datagrams on its port. This is because the NAT policy of the router forwarding packets may drop UDP packets destined for the client, unless it has previously sent traffic to the server.
6. Once the data session begins, each client will terminate the TCP control connection.

- **Handoff**
  Handoff from 3G to Wi-Fi is implemented in the same manner as a handoff from Wi-Fi to 3G. In each case, the client will open a new control connection over TCP on the interface to which it is switching. This enables the server to obtain the IP address of the new interface. A switch message is sent to the server, followed by an acknowledgement from the server. At this point, the client may begin sending over the new interface. However, the server must continue to listen for packets that may have been in transit prior to the handoff for a short period of time. These packets will also be forwarded until the timeout is met. The sequence diagram is shown below:

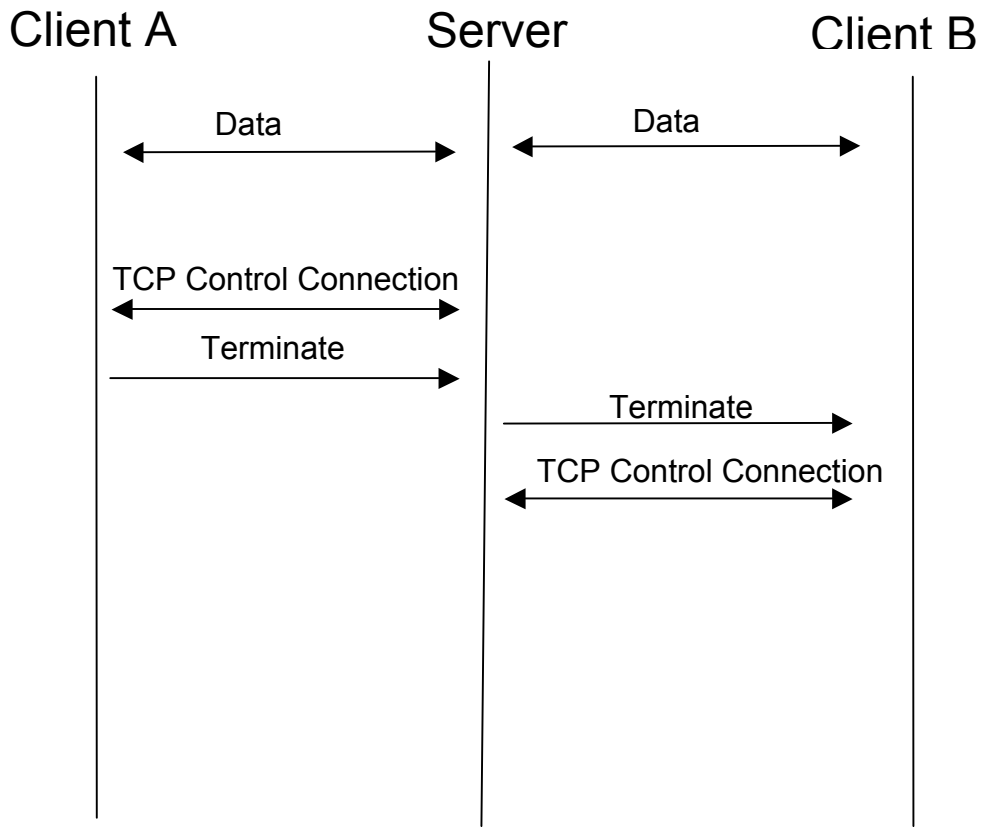


1. Client B enters Wi-Fi.
2. Opens TCP Control Connection over existing interface.
3. Client B sends switch message to server containing new IP.
4. The server receives the Switch message with the new IP address. The server continues sending data over 3G. The server will continue receiving data over 3G socket until the first UDP packet is received over the new interface. At this point it will start sending over Wi-Fi connection.

5. The client will receive packets over the new IP address after the server receives the first UDP packet. The client will maintain the 3G socket for some time t for delayed packets (timeout), then closes connection.

Note: The handoff protocol is the same from Wi-Fi to 3G.

- **Termination**
  The termination sequence requires each user to initiate a new control connection over the current interface in use. The control connection is used to simulate a client being "online" so that it may accept a call or make a call request. This sequence simply requires each client to send a terminate message to the server to indicate the end of the session. The sequence diagram is shown below:



1. Client will open a TCP connection to initiate tear-down by sending a Terminate message to the server.
2. The Server will send a terminate message to the other client over UDP (sent 4 times). The Server will set a timer while waiting for a TCP connection to be opened.

3. After each Client opens the TCP Control Connection, packets received on the UDP ports at each Client will be dropped, and the session will end.

Implementation

- **Client**
  The clients were implemented on Windows Dot Net framework in C#. The following is the architecture of client program:
  - **Form Class:** Implements the form interface and has buttons related to **Start**, **Switch** and **End** events which basically interact with the Handoff class.

  - **Handoff Class:** This class is the main entry point of the application and has two methods in it namely start() and show(). The start() method has all the logic in it and the show() method just writes output to the textbox on the phone. The user can interact with the application by pressing the above mentioned buttons:
    - **Start:** This button starts a thread on the start method of Handoff class, which is the main entry point of the program. It does the following:
      1. It opens a TCP Control Connection on the currently selected interface (3G/Wi-Fi) and sends a connect message to the server which has the client's ID and local port on which the client will listen for the data packets.
      2. It then waits for an incoming call message, which is sent to it by the mobility server and has the remote client's ID in it.
      3. The client can accept or reject the call and sends its response to the server
      4. If it accepted the call then it will receive a server port message which has the port on which the client can send his packets to the server which will be forwarded to the other client
      5. After getting the server port it starts a new Data Connection thread, which basically opens a UDP socket on the currently selected interface and starts sending on it. The Data Connection thread itself starts a new receive thread on this socket so that it can send and receive in parallel.
      6. It then terminates the TCP Control Connection
      7. After that it just waits for some event (Switch/End) to occur

    - **Switch:** This button sets the switch flag of the Handoff class, which is basically the notification to do a switch from the current interface (3G/Wi-Fi) to the other interface. It does the following:
      1. It detects what interface we are currently on (say 3G).

2. It then opens a new TCP connection on the new interface on which we want to switch (say Wi-Fi).
3. It sends a Switch message to the server with its client ID and the new port on which it will listen for the UDP packets.
4. The server on getting this message changes the current IP Address of this client to the new IP Address from which it got this message and also changes the current port to the new port that it got in the Switch message. After this the server send an OK message back to the client.
5. After receiving an OK from the server the client starts a new Data Connection thread on the new interface (Wi-Fi), which opens a new UDP socket and starts sending on that. It also starts a new receive thread also.
6. It then signals the previous Data Connection thread (3G thread) to stop sending.
7. When the server starts receiving packets from the new IP Address of the client, it stops sending over the old IP Address and the port and it starts forwarding the packets on the new IP Address and the Port (the one it got in Switch message).
8. Because the receive thread over the old socket which was opened on the previous interface (3G) won't receive anything, we eventually timeout and close the socket and return from the receive thread and the corresponding Data Connection thread and allocated resources are freed.
9. Then the TCP connection is closed

So each time we press Switch the above sequence of actions occur.

- **End:** Clicking on this button will set the terminate flag and it will wait for any pending Data Connection threads to return. After all the threads return the client application is terminated. (We can also terminate the whole application on press of this button instead of waiting for pending threads)

- **Data Connection Class:** This class takes care of the data session and handles sending and receiving of UDP packets. It has one method openDataConnection() which is the entry point of the Data Connection thread that is created in the start() method of Handoff class as described above. It does the following:
  1. It opens the UDP socket on the currently selected interface and starts sending over that.
  2. It also binds the above created UDP socket to the local port (sent to the server in the Connect/Switch messages), starts a receive thread and starts receiving the packets on the local port forwarded to it by the server

3. It will break of the sending loop if, either in the middle of sending it is signaled by the Handoff thread to stop sending as described above, or it has finished sending its packets.
4. After breaking of the sending loop it just waits for the receive thread to return which it created in step 2.

- **Control Connection Class:** This class is basically a wrapper for the TCP control connection. It just opens the TCP connection over the specified interface and has methods for sending and receiving messages over the opened TCP connection.

- **Receiver Class:** This class is just for receiving the UDP packets over the specified UDP socket. It has just one method named receive() which is the entry point of the Receiver thread which is created by the openDataConnection() method of the Data Connection class as described above.

- **MyTimeout Class:** This class implements the timeout mechanism. It has one method named timerCallBack(). This class does the following:

  1. Each time before a receive call in the receive() method of Receiver class we start a timer by making an instance of Thread.Timer class.
  2. If we receive something then the call to receive is successful and we dispose the timer created above.
  3. However if we don't receive anything, then timer expires eventually and the timerCallBack() function gets called. This function is passed the receiving socket as an argument. This function then closes the socket
  4. Because we close an active socket it throws SocketClosed Exception. We catch that exception in the Receive thread, dispose off the timer and return from the Receive thread.

- **Server**

  The Server application was run on the Commedia server within the Department of Computer Science. This server has a static IP address, which is known by all clients running the client application. The server is responsible for forwarding data packets to the proper client, depending on the session. Messages between server and client can occur over both TCP (for the control messages) and UDP (data messages). Therefore, the server must listen for connection offers over TCP. For UDP datagrams, the server will assign a port number for it to listen on, and will send this to the clients after a call is accepted. The implementation has a main loop consisting of a select statement that can accept new TCP connections, while also listening for TCP and UDP traffic. The server is also responsible for creating and maintaining the session between new clients, bridging the connections the two (if the receiver accepts the call). For each session, two relevant data structures were needed to maintain all the information necessary for

data exchange between each client. This essentially requires knowing the current IP address and the previous IP address (prior to the handoff). This stucture was called the client table:

```
struct client_table {
        int current_addr;
        int client_id;
        int tcp_socket;
        struct ip_entry ip1;
        struct ip_entry ip2;
} client_table;

struct ip_entry {
        int port;
        struct sockaddr_in addr;
        int socket;
} ip_entry;
```

A client_table entry is instantiated for every client connecting to the server. When a client connects, the tcp socket for the control connection is saved, as well as the current IP address. The "current_addr" field simply indicates which IP entry is currently active. The previous IP address will always be the other IP (not being currently used). However, the server may still receive packets from this address. This is because messages can be in transit during the handoff, and must be forwarded for a certain period of time. The ip_entry struct is simply a port, IP, and a UDP socket file descriptor.

The flow table, shown below, is used for a session between two clients. This contains the socket file descriptors for sending/receiving UDP packets that are forwarded from the server. Currently, the server receives data from each client on the same port, however, the structure provides for each client to have a unique port assignment.

```
struct flow_table {
        int flow_id;
        int client_0_id;
        int client_0_sock;
        int client_1_id;
        int client_1_sock;
}
```

Message Set

Included is a list of the message existing on both the server and client applications.

```
/*Message Type 0*/
struct client_start_message {
	int message_type;
	int client_id;			/*ID of client sender*/
	int client_port;		/*UDP port for client to receive packets on*/
} client_start_message;

/*Message Type 1*/
struct client_start_reply {
	int message_type;
	int server_port;
	int status;			/*0 = wait to for receiver to join, 1 = ok to send*/
} client_start_reply;

/*Message Type 2*/
struct data_packet {
	int message_type;		/*Type indicates what type of packet is being
sent/received*/
	int seq;			/*Sequence number of message being sent*/
	int payload_size;
} data_packet;

/*Message Type 3*/
struct switch_request {
	int message_type;
	int sender_id;
	int port;
} switch_request;

/*Message Type 4*/
struct switch_request_reply {
	int message_type;
	int status;			/*0 = wait to send on new interface, 1 = ok to send
on new interface*/
} switch_request_reply;

/*Message Type 5*/
struct client_call {
	int message_type;
	int dest_id;
} client_call;
```

```
/*Message Type 6*/
struct client_call_reply {
	int message_type;
	int dest_id;
	int status;			/*0 = Client not online, 1 = Clear to send*/
} client_call_reply;

/*Message Type 7*/
struct server_port {
	int message_type;
	int port;
	  } server_port;
```

Discussion

The following points summarize some design decisions and the issues and problems encountered during the project:

- **Mobility Server:** We used the mobility server as the man in the middle instead of having the clients directly talk with each other because almost all of the times the clients would be behind a NAT, whether they are on 3G or Wi-Fi, which implies that they won't have a public IP which the clients can use to communicate. Therefore we decided to do it with man-in-middle approach which will have a public IP and both of the clients can talk to each other via mobility server.

- **NAT issues:** Each client has to initiate at-least one UDP packet from the port that it sent to the server in Connect/Switch message, on which it intend to receive the UDP traffic, otherwise the NAT router might just drop the packet due to some policies which might have been set to drop packets to a port if no traffic is initialized from inside on that port.

- **Connection Manager:** Since we have multiple interfaces available over which we can open Sockets and connect to Internet, Windows operating system has a mechanism of calculating the best interface available, in terms of cost, speed etc., to use for connecting to Internet and gives Socket over that interface only. However for our application we needed a way to bypass this mechanism and specify a particular interface viz. 3G or Wi-Fi. Therefore we used Connection Manager API which lets you bypass the operating systems calculations and specify the interface on which we want to open the Sockets and hence communicate over that interface only.

- **UDPClient Class:** Initially we were using this class, which is a wrapper around UDP Socket, instead of using UDP Sockets directly. However we noticed that if we use the class in following way then we get really bad performance of 5-6 packets/sec:

UDPClient client = new UDPClient(localPort);

client.send(message, message.length, “aa.bb.cc.dd”, remotePort);
or
client.send(message, message.length, remoteEndPoint);

However if we use it in a following way then we get the expected performance of 47-48 packets/sec:

UDPClient client = new UDPClient(“aa.bb.cc.dd”, remotePort);

client.send(message, message.length,);

One probable reason behind this behavior might be that it might be doing DNS query for every packet when we try to use it in the first way, while in the second form it might have already stored that DNS response.

However after this inconsistent behavior we decide to work with Sockets directly.

- **Receive Timeout Option:** There is no option on receive() method call on the Sockets, on windows mobile, for setting the timeout value after which the receive should timeout in case it doesn’t receive anything. So we decided to use our own timer whose functionality is described above.

Results and Performance

The following table summarizes the tests scenarios, initial conditions of the clients and the results:

S.No.	Test Name	Test Description	Transfer Time (sec)		Throughput (Kbps)		Average Loss Rate %
			Min	Max	Min	Max	
1.	3G-No-Handoff	Both clients on 3G and no handoff	68.27	89.38	42.51	54.48	1.23
2.	Wi-Fi-No-Handoff	Both Client on Wi-Fi (Client 0 on jhuacm and Client1 on DSN-N) and no handoff	71.24	89.25	43.01	53.6	0.15
3.	Handoff-3G-Wi-Fi	Client A on 3G, Client B starts from 3G, switches to Wi-Fi (DSN-N) after receiving 1500 packets	69.04	99.03	38.76	53.25	2.08
4.	Handoff-Wi-Fi-3G	Client 0 on 3g, Client1 starts from Wi-Fi (DSN-N) switches to 3g after sending 1500 packets	69.98	94.52	40.13	54.87	1.26

In each of the test we sent 160 byte packets at the rate of 50 packets/sec, which is the rate required for VOIP. In examining the loss rates and throughputs, there is no significant difference between the handoff setting and non-handoff setting. The data we obtained supports the notion that our protocol can support VOIP data rates in a handoff environment. Graphs for inter-arrival times for the different test scenarios are shown below:

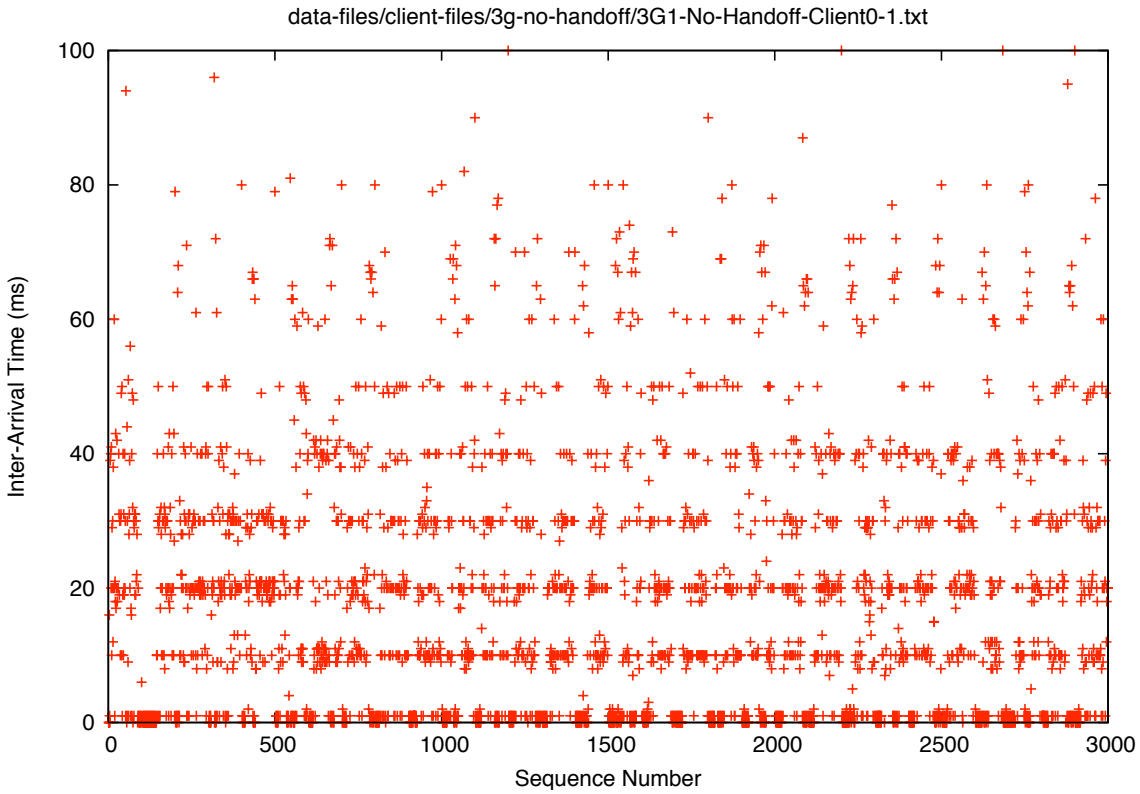


Figure 1 3G-No-Handoff

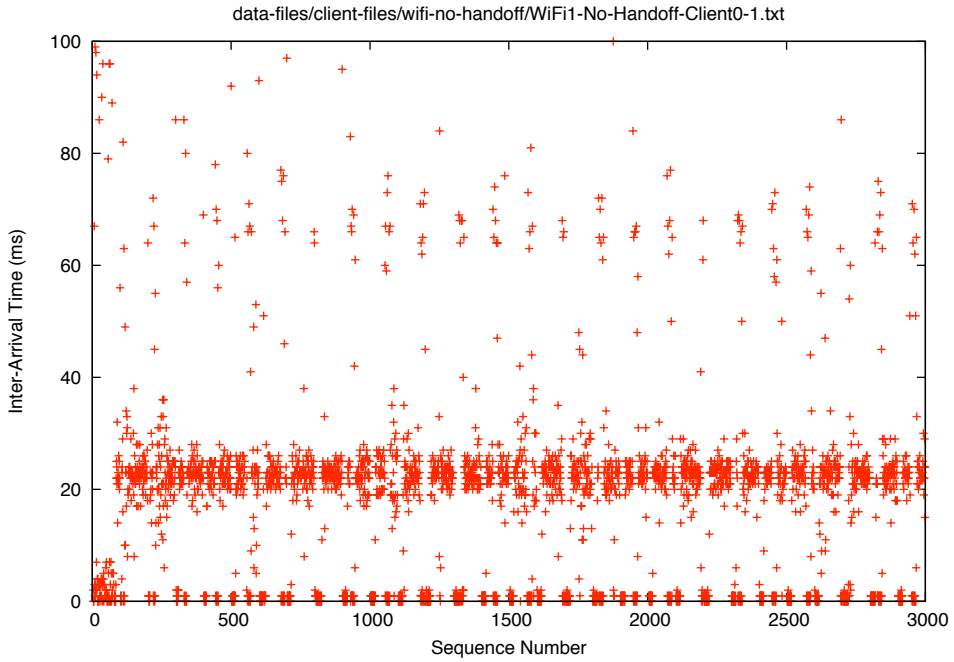


Figure 2 Wi-Fi-No-Handoff

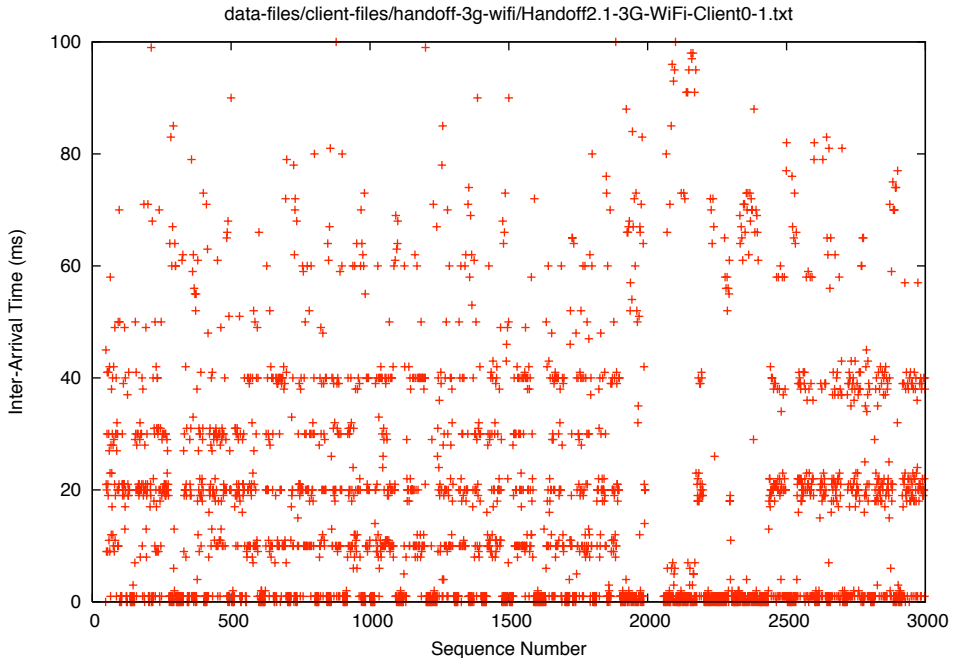


Figure 3 Handoff-3G-Wi-Fi

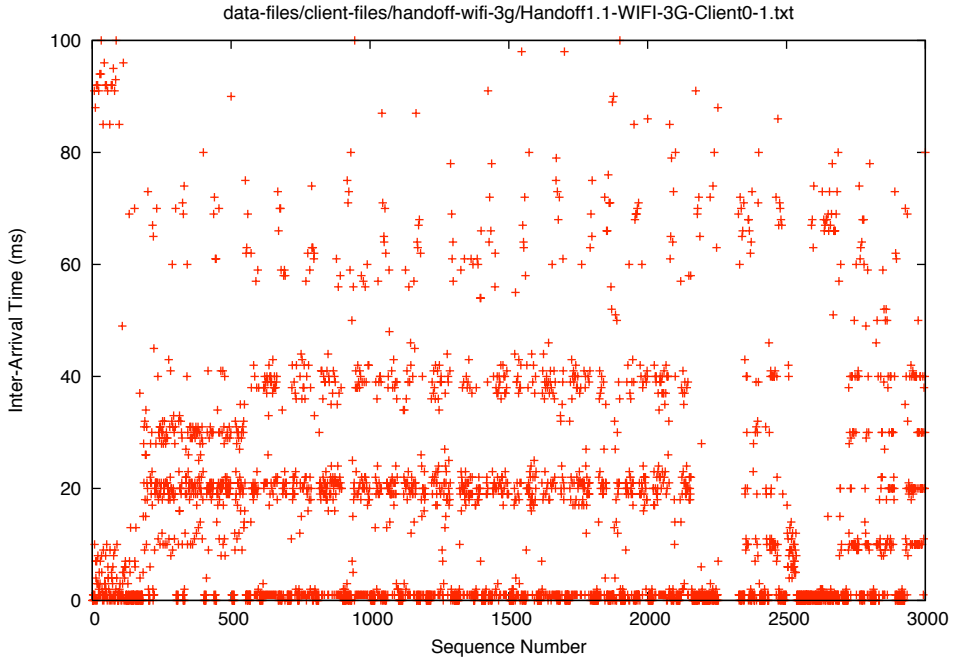


Figure 4 Handoff-Wi-Fi-3G

It can be seen that in case of 3G the inter-arrival times are range from 20 ms – 100 ms clustering at 20 ms intervals. However in the case of Wi-Fi the packets are clustered at 20 ms with relatively smaller variance in their inter-arrival times.

It can be observed that in Figure 3 and 4 there is a gap at around 2000 and 2200 respectively. This is the point where the handoff is occurring from one interface to the other. While there is a delay in packets during the handoff is taking place, however the delay remains within the threshold of 100 ms.

Conclusion

In this project we examined the possibility of conducting handoff of data sessions between 3G and Wi-Fi on smart-phones, while primarily focusing on a possible VOIP application. From the data rates that we obtained we observed, it is feasible to perform a handoff in a VOIP application in a manner imperceptible to the user.