Introduction

This article describes how to effectively communicate using sockets. There are a few well known methods to increase Java I/O performance. In this article we use them for communication between two Android devices connected via WiFi. We take into consideration various parameters such us sending times, packet size, resource and memory consumption.

Unbuffered data Streaming

The most basic data streams in Android are DataOutputStream and DataInputStream. When we use them data is written to the physical device as soon as possible after each write operation. All read and write request are handled directly by the underlying OS. Using unbuffered streams doesn’t require any additional resources for storing data. However frequent request to the OS often trigger operations that consume a lot of resources, which can degrade the performance of your program. Therefore using unbuffered data streams is effective only for small amounts of data.

To prove this concept we send data between two Android devices connected by WiFi. Results can vary according to many factors, including device model, software version and WiFi parameters. For analysis we use average results of a few tests. Files with 1kb, 512kb, 1mb, 4mb and 10mb data are sent and execution times are measured.

Client Buffered Data Streaming

We can use buffered streams on one side (client and server) to reduce execution times. In this article we buffer client input stream. Using buffers makes socket communication approximately 5-20% faster (depending on data size). This is possible because each read/write call uses data already stored in the buffer. The native API is called only when the buffer is empty. Most of the smaller calls can be done by accessing the buffer alone. There is some extra space required to hold buffer. However, this drawback is usually outweighed by general performance improvement.

Client and Server Buffered Data Streaming

To further reduce execution times we can use a BufferedOutputStream on the server side too. This leads to a significant performance improvement. When we use buffers only on one side of communication the overall performance is hampered by the unbuffered side. No matter how efficient is the buffering on client side we cannot receive data faster than it is send form the server.

In our test using buffers on both sides reduces execution times up to 85%. Improvement could be greater if we used different data amounts.

Test results

The table below shows test results for sending 1kB, 1MB and 10MB files using different I/O streams:

	1kB		1MB		10MB	
	Time (ms)	Gain (%)	Time (ms)	Gain (%)	Time (ms)	Gain (%)
Unbuffered	850	-	17800	-	121100	
Client Buffered	820	4	14200	20	95500	21
Client and Server Buffered	800	6	7440	58	17000	86

Table 1: Performance of Different Buffering Modes for 1kB, 1 MB and 10MB data.

These examples show that basic Java input/output streams work very inefficiently for large data payloads. For 10MB, data buffered streams give better results up to 85%. However, for reading/writing single bytes the normal DataInputStream and DataOutputStream classes are more efficient, because buffering could introduce too much processing overhead, and for a 1kB file performance gain is not significant. Chart 1 shows that it is very advisable to use buffering on both sides (server and client), as buffering on one side gives only minor performance improvement.

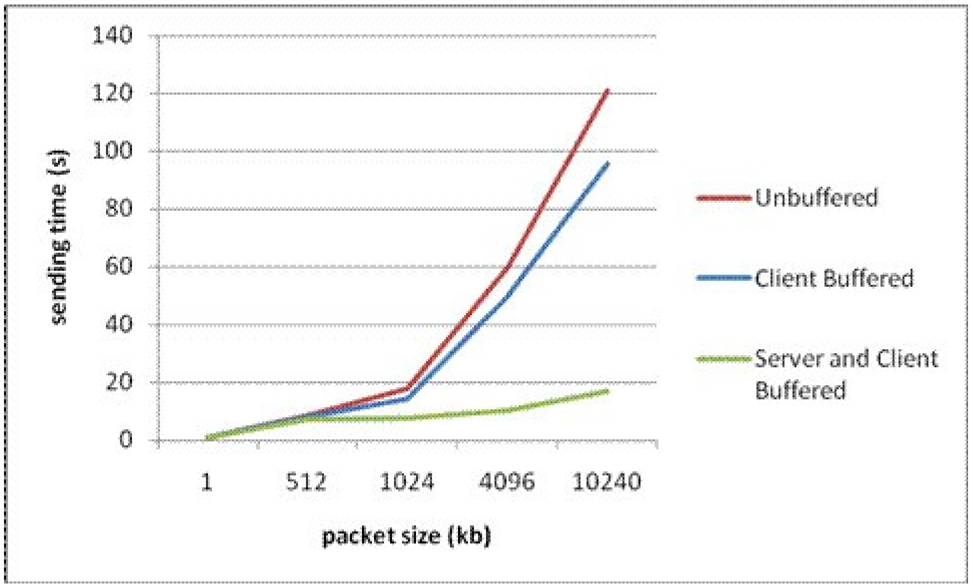

Figure 1: Packet size vs. sending time.

Figure 2 shows the percent performance improvement vs. packet size for buffering on both client and server side. These results confirm that buffering is more efficient for larger amounts of data. For small amounts of data improvement is much less significant. In this case buffering can introduce more processing and memory overhead than performance gain.

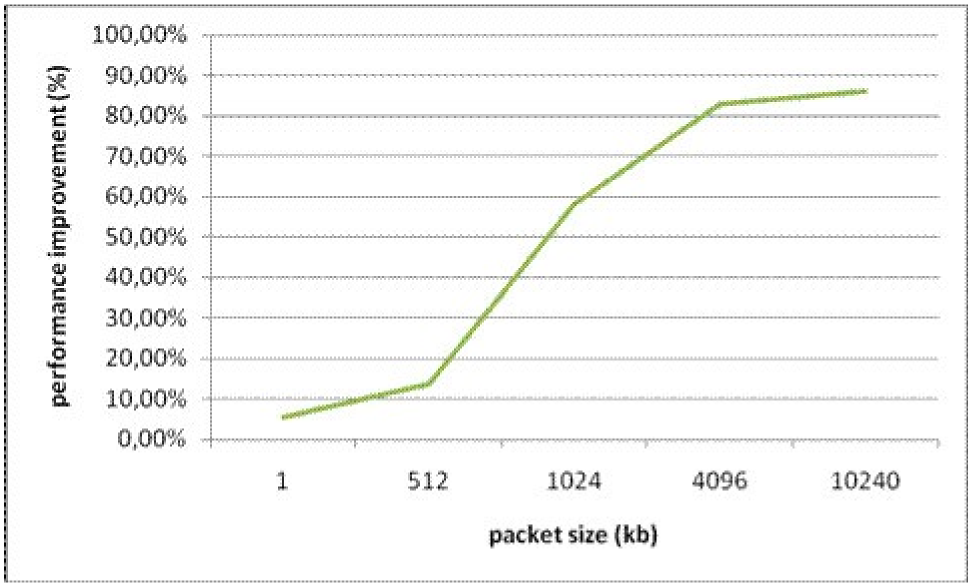

Figure 2:Performance improvement vs. packet size (Client and Server Buffering)

Code samples

Snippets of code used for testing buffered streaming are shown below. Full code of Client and Server application used for testing is in attachment to this article.

1. Unbuffered data streaming

Server:

```
try{

serverSocket = new ServerSocket(mPort);

Socket socket = serverSocket.accept();

DataOutputStream dataOutputStream = new

             DataOutputStream(socket.getOutputStream());



//read data for testing and send it to the client

File file = new File(mFilePath);

FileInputStream fis = new FileInputStream(file);

BufferedInputStream bis = new BufferedInputStream(fis);

             int ret = 0;

             while ((ret = bis.read()) != -1)

                          dataOutputStream.write(ret);

             Log.d("server", "finish sending");



             bis.close();

             fis.close();

} catch (IOException e){
```

```
             //handle socket and file IOExceptions

}
```

Code Sample 1

Client:

```
try{

Socket socket = new Socket(mDestinationIP,mDestinationPort);

DataInputStream dataInputStream = new

             DataInputStream(socket.getInputStream());

int data;



//read data from server

while ((data=dataInputStream.read()) != -1) {

             //do sth

}

} catch (IOException e){

             //handle socket IOExceptions

}
```

Code Sample 2

2. Client Buffered data streaming

Client:

```
try{

Socket socket = new Socket(mDestinationIP,mDestinationPort);

BufferedInputStream bufferedInputStream = new
```

```
             BufferedInputStream((socket.getInputStream());

int data;



//read date from server

while ((data= bufferedInputStream.read()) != -1) {

             //do sth

}

} catch (IOException e){

             //handle socket IOExceptions

}
```

Code Sample 3

3. Client and Server Buffered data streaming

Server:

```
try{

serverSocket = new ServerSocket(mPort);

Socket socket = serverSocket.accept();

BufferedOutputStream bufferedOutputStream = new

             BufferedOutputStream(socket.getOutputStream());



//read data for testing and send it to the client

File file = new File(mFilePath);

FileInputStream fis = new FileInputStream(file);

BufferedInputStream bis = new BufferedInputStream(fis);
```

```
            int ret = 0;

            while ((ret = bis.read()) != -1)

                        bufferedOutputStream.write(ret);



            fis.close();

            bis.close();} catch (IOException e){

            //handle socket and file IOExceptions

}
```

Code Sample 4

ref:http://developer.samsung.com/android/technical-docs/How-to-effectively-communicate-using-Sockets