A high-bandwidth mobile Bluetooth ad-hoc network for the purpose of Music Streaming

Rick Watertor (11250550), Wouter Loeve (11330198),
Okke van Eck (11302968), Xavier van Dommelen (11321458)

Abstract— **An high-bandwidth mobile Bluetooth ad-hoc network is presented. The challenge is to create a network that allows for bandwidth intensive applications, such as real-time audio streaming, while overcoming the bandwidth limitation of Bluetooth. Each device in the network contains the entire topology, such that minimal bandwidth is expended on maintaining the network and routing. Songs are split and sent as small packets to allow interleaving of network regulation packets, and to enable faster playback. Streaming results in a longer overall transmission time, but greatly decreases the time until playback. The network is performs well enough to allow concurrent streaming of at least two devices from one source. Additionally it supports at least two hops with acceptable latency.**

I. Introduction

In this paper possibilities of creating a high-bandwidth ad-hoc network using Bluetooth are investigated. This is tested by building a music streaming application. The creation of such a self regulating network brings along many complications as it hits the bandwidth capacity of Bluetooth.

This paper will discuss an implementation of a self regulating ad-hoc network. Multiple optimizations and future improvements are discussed as well.

A. Research project

The goal of this research project is to create, test and optimize a Bluetooth ad-hoc network for the purpose of assessing whether Bluetooth is a suitable technology for a wireless high-bandwidth ad-hoc network. To do this, an Android application is created, which streams audio by using the network. In the upcoming sections, the field of wireless ad-hoc networking will be explored and solutions for the challenges that were faced will be discussed.

II. Requirements

This section will explain in detail what the minimal requirements of the experiment are and how it was executed.

The Android ecosystem was used as the basis of the benchmark application. Android was chosen because of the availability of multiple mobile devices from different generations. Furthermore, it is likely that wireless ad-hoc networks' future lies in mobile computing as mobile devices are getting more common, more powerful and support a wide variety of wireless technologies. This includes the widely used Bluetooth.

A. Basic requirements

The network will need to be able to handle at least the following situations:

1) Automatically connect with other devices running the application using Bluetooth.
2) The network will be a self regulating ad-hoc network. Ergo, disconnecting nodes must not cause the network to fall apart, prevent the devices from functioning and the network should still be able to satisfy audio requests from others.
3) The application should be able to display all the audio tracks that are available in the network, and this list should be automatically updated when nodes join the network and disappear.
4) The application can request and retrieve a specified song by the user.
5) The song data should be playable. The time until playing and the receiving of the entire song shall be a benchmark.

A network was set up alongside its corresponding benchmark application. The benchmark was able to handle the specifications. Next, network improvements were constructed. The effect of these improvements will be measured and compared to gain further insights in the performance of these types of networks.

III. Bluetooth

This section will justify why Bluetooth was chosen as the underlying connection of the network. It will also explain how Bluetooth connections are established and how the protocols were improved to consume less bandwidth.

A. Bluetooth or Wi-Fi Direct

Wi-Fi Direct was taken into consideration for the underlying connections. Wi-Fi Direct has the benefit that it can achieve transfer times up to 250Mbps, while Bluetooth can achieve transfer times up to 25Mbps [1], although these are implementation dependent. Only taking the higher transfer time into account, it would be logical to choose for Wi-Fi Direct, but Wi-Fi Direct has one big downside. It works by using a master-slave network structure, in which it is not possible for a client to connect to multiple masters. Additionally, masters cannot connect to other masters. This structure was undesirable, since it is impossible to create a network that consists of more than two hops without having significant compromises in network structure, bandwidth and latency.

Lee, Park and Shah do provide some options to overcome such challenges [2]. However we thought it is interesting to see if it is possible to achieve music streaming with Bluetooth's limited bandwidth. The challenge for Wi-Fi direct is to set up such an ad-hoc network. The challenge for Bluetooth is to create a network that allows for audio streaming while overcoming the low speed of Bluetooth.

Bluetooth does not have the multi-hop problem since every device can both be a server and a client simultaneously. Then, all devices can thus connect to multiple peers, which makes a multi-hop network possible. Additionally, a disconnected device will normally not lead to all other connected devices getting disconnected from the network. This can only happen when all devices are connected trough a bridge device, and even then, the network would simply split into sub-networks. These sub-networks can easily reconnect whenever a new device can bridge the two networks. Bluetooth is able to do this, without having to renegotiate a group owner or a master.

It achieves an ad-hoc network structure natively, so it is easier to use and uses less power than Wi-Fi direct [3]. Therefore, at the moment, Bluetooth seems to be a more suitable technology for a multi-hop ad-hoc mobile network. Additionally, Bluetooth is more interesting to develop a high bandwidth application for, since Bluetooth's transfer time is limited.

1) Bluetooth Technologies

Besides *Classic Bluetooth* there is also the newer *Bluetooth Low Energy* (BLE). Classic Bluetooth was chosen for this research, since it is supported across a much wider range of mobile devices. Bluetooth Low Energy support was added to android in Android API level 18 [4]. The oldest devices used during this research have an API level of 16. Using BLE would lower the amount of test devices significantly.

B. Setting up a connection

Bluetooth's secure Rfcomm sockets require setting an *universally unique identifier* (UUID) that defines the service of the running application. Only sockets that are identified by the same UUID will be able to connect with each other, which prevents applications from connecting to the wrong socket, in the case a device runs multiple.

Before requesting a connection, a device needs to discover what other devices are available. To be able to be discovered by another device, the device also needs to become discoverable. As a naive implementation, we originally implemented the application to continuously attempt to discover devices and stay discoverable.

1) Service Discovery Protocol

Once a device has discovered nearby devices, it will start the Service Discovery Protocol. This protocol requests from a device which services it provides, in the form of a list of UUIDs. If a device provides the UUID of the service it looks for, it will attempt to connect.

The Service Discovery Protocol cannot be executed in parallel. This protocol queues the request, but also starts a timeout at the time of enqueuing. Because only a few requests could be finished, most requests would return an invalid result. They were never given the chance to discover the services in the first place.

Therefore, SDP was forced to be executed in series, by queuing any such requests.

2) Cache

The Service Discovery Protocol is a slow process and not necessarily needed. Since the Rfcomm sockets only allow connections with the corresponding UUIDs, devices could just attempt to connect with every device. Only proper connections would persist. The downside to this approach is that every device would get paired regardless of whether the connection establishment was successful. With Bluetooth's limit of 255 pairings, this is an unwanted side-effect.

With serial SDP, not every nearby device gets paired with the device that is searching for the network. It is now able to utilize the fact that the paired devices form a small group of devices that have a large likelihood of running our application. This behaviour is then used to try to connect to all the already paired devices that contain the cached service UUIDs, even before starting SDP to speedup the connection process. This results in a faster connection establishment even in an area where a lot of Bluetooth devices are available.

C. Improvements

Using the previously mentioned techniques, the application was improved by adding a time interval for searching for connections. However, the discovery and specifically the service discovery cost a lot of bandwidth and made the sending of packets across the network slower.

Therefore, an idle protocol was implemented. This method makes the device *idle* while it is sending or receiving data across the network. Idle in this context means that it will temporarily stop discovering and stop the SDP while it is in the idle stage. It is still able to receive new connections.

IV. Network topology

In this section the network topology will be explained. This includes how the graph is implemented, what kind of data is stored on each node and an explanation of how and why the Dijkstra algorithm was used.

A. Graph

The application uses a graph as a way of symbolizing the network. Each node represents a device that is linked to the network and an edge represents a connection between two nodes. Each device contains the entire topology, which means all known nodes and edges in the network.

The reason for storing the full topology on every device is that it minimizes the network traffic, partly because routing can be done locally. This minimization will further be explained in the Handshake subsection of the Network protocols section.

B. Node

Each node represents a device in the network. Therefore, the node should contain an identifier for the corresponding device. Bluetooth uses universally unique Bluetooth MAC-Addresses, so they serve as a great identifier for the device.

A node will also contain all the music metadata of all the songs that are found on the device. This metadata contains the song name, the artist, the song size and the path to the audio file on that device.

C. Dijkstra

The Dijkstra algorithm [5] was chosen as a method for determining the shortest path and for checking whether nodes or edges need to be deleted.

Dijkstra is used to calculate the shortest path to every other node starting from the node of the corresponding device.

These paths can be used for two aspects. Firstly, they can be used locally to determine the best node to send a packet to, given a destination. Secondly, the reachability of the nodes can be checked since disconnected nodes would not have a path anymore. When there is no path to a node, that node and its corresponding edges are removed from the graph. This automatically cleans up the network, once part of it is no longer reachable.

V. Network protocols

This section will describe the network protocols that are implemented and the reasoning behind them. These protocols are meant for establishing the self regulation of the network.

A. Handshake

A handshake will occur immediately when a connection is established between two devices. This handshake is meant to allow both parties to receive information of

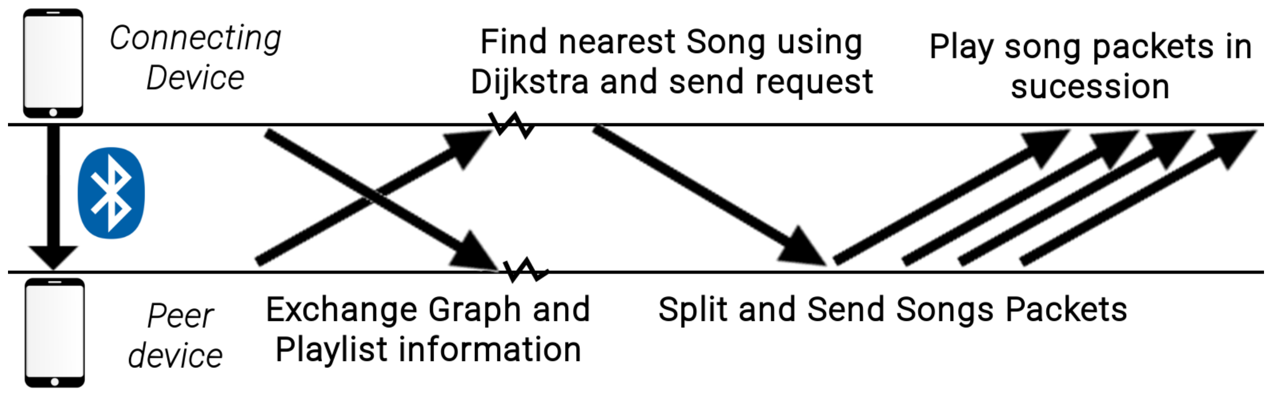

Connecting Device
Find nearest Song using Dijkstra and send request
Play song packets in sucession
Peer device
Exchange Graph and Playlist information
Split and Send Songs Packets

Fig. 1. The handshake and song protocol. First a device connects via Bluetooth, then a handshake is performed, exchanging graph information. Whenever the user wants, a song request can be sent. The target device then sends the song in split packets to the source, which plays it as a stream.

the whole network behind each of the other devices. Furthermore, this handshake will solve an important issue with MAC-address.

1) Graph sharing

After a Bluetooth connection has been successfully established, each of the two devices will send its current graph to the other device. Once the other receives the graph, it will calculate the difference between the received graph and its own graph. This difference contains the nodes and edges that appear in the received graph but are not present in its own graph. The difference also includes the now newly created edge between the two devices, as part of the handshake protocol.

The node will then apply the difference to its own graph, effectively updating the old graph to the new network. Additionally, it will send this difference across the network using the Network-Update protocol, which will be discussed in the next subsection. Both devices partaking in the handshake will perform the previously discussed steps.

2) MAC-address

An obstacle that was encountered during the development of the benchmark application for the experiment was regarding the device's own Bluetooth MAC-address. Newer android versions have blocked the possibility of requesting the devices own Bluetooth MAC-address. A couple workarounds were possible for some API levels starting from API 16, but these workarounds were encountered to be patched for later API levels [6].

An additional step in the handshake was implemented to overcome this problem. When a connection is established, each of the two devices is able to see the other device's Bluetooth MAC-address. If a device does not know its own MAC-address, it can adopt the destination address of the handshake as its MAC-address. However, the device could already have sent its graph with the faulty address. Since the receiving device already knows the MAC-address of the peer, the receiver can replace the faulty MAC-address with the actual address. This ensures that after the handshake both parties are aware of their own MAC-address, and both parties have the full graph with the valid addresses.

B. Network-Update

After a handshake is performed, the new network will consist of at least two nodes. To keep the network self regulated, it is important that each node will be kept up to date when another device joins or leaves the network. For these cases, the network update protocol was implemented. This protocol informs the entire network about these events. On receiving a network update packet, each device can evaluate and process the changes.

1) Flood

Each flood starts from one device and then propagates through the network. The flood protocol takes a payload of data and starts sending it to each of the peers from the source node. A flood message has an unique id. This id is created with the MAC-address from the source node and a counter that increases for each flood message that is sent from this node. Every node keeps track of every flood message counter it has received for every MAC-address. When a node receives a flood message, it checks if it has received this message before. If it sees that it has not received the MAC-address with corresponding counter before, it sends the flood message to every other peer and adds the MAC-address with counter to its list of seen ids. If the device has already seen the id of the flood, the message is not forwarded.

2) *Update*

There are two events that trigger a flood over the network. The first event happens after the graph difference calculation in the handshake protocol. The graph difference is flooded throughout the network and every node that receives this information will add the nodes and edges to its own graph, ignoring possible duplicates.

The second event is when a link between two nodes gets disrupted and disappears. Both nodes that were originally connected by this edge will notice and start broadcasting an update to the two remaining sub-networks. The broadcast informs the recipients that the edge needs to be removed.

The process of a network update is illustrated in the following pictures.

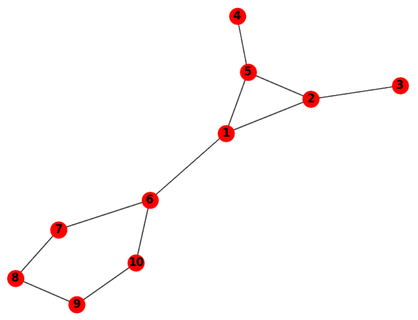


Fig. 2. An example network in which two sub networks can be found are connected via the route between node 1 and node 6.

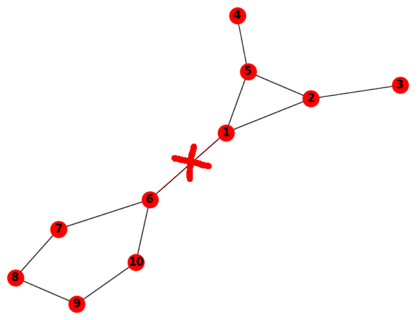


Fig. 3. The connection between the two sub networks is broken.

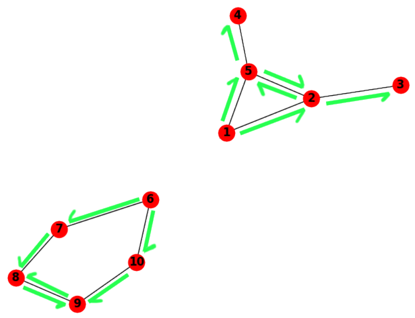


Fig. 4. As a result of the connection between the sub networks being broken, network updates are flooded trough the network to make sure all the devices contain an updated graph.

In the last picture the process of a flood is visualized. It is important to note here that network updates can cross over an edge twice. However, as part of the flood protocol, the duplicated broadcast will be ignored. It is also possible that the update does not cross over that edge twice. This can happen when for example node 1 sends an update to node 5 which in turns sends it to node 2. Meanwhile the network update of node 1 is also broadcast to node 2 directly. The packet that came from node 5 will be ignored if the update from node 1 arrives earlier. Additionally it will also not send the packet to node 5 as it already received a network update with the same flood id.

Since the songs of each device are linked to a node, not only the graph is updated because of this protocol, but also the list of songs. Thus when a node disappears, or is added, the song list can dynamically update.

C. *Considerations*

The protocol described above is the implemented protocol. A different protocol was also considered. This protocol was built upon the idea that it was better to not have the entire network topology saved on every device. This would make the ad-hoc network scale a lot better when enlarging the network, because not every device needs the full topology in memory.

It should be noted that everything in this section are our considerations when drafting a network-protocol. We were not able to test and compare the performance between the two protocols, so for now this section is purely theoretical and the testing will be added to the future works section.

This draft protocol works by finding the shortest route through the network by flooding packets through the network. As the flood traverses the network, every node adds itself to the list of traversed nodes, and then sends the updated packet through the neighbours that were not on the traversed nodes list. Then, when packets arrive at the destination, the shortest route would be chosen from the arrived packets and the sender can start streaming music over that route.

This method would incur a latency penalty as the destination node would have to wait until at least some of the route searching packets would have arrived. A consideration would have to be made to either wait until some packets have arrived or take the first packet that arrived and marking that as the fastest connection, even if the latter did not have the shortest path.

The implementation of the aforementioned protocol would likely have difficulties when routes would be broken off in succession. A new route would have to be negotiated, incurring a latency and bandwidth penalty. The latency penalty itself was not particularly problematic, but the assumption that the network would assume a static form for a prolonged amount of time is problematic, especially since an ad-hoc network was the goal. In a situation in which the network dynamically changes all the time, it would take a lot of traffic to renegotiate the route every time, possibly collapsing the network.

In short, our current more dynamic and reactive network protocol described in previous sections was implemented because we thought repeated flooding would hurt the performance of the network as a whole in case of a truly dynamic ad-hoc network.

1) Advantages and disadvantages

The considered advantages of using the current implementation versus the drafted protocol revolve around minimizing the network usage and CPU load on the devices. In the current implementation only network updates are flooded trough the network. The drafted protocol would lead to a network in which packets would be flooded constantly.

In the event that a device wants to start streaming audio and when an active route is broken, bandwidth intensive packets would flood large parts of the network.

The current implementation is less CPU-intensive compared to the draft as the devices only have to handle network-updates and do not have to respond to constant flooding. This in turn is beneficial to the power usage of the devices partaking in the network.

A disadvantage of the implemented protocol is that the entire network is saved on every device. In the test environment, which consisted of less than 10 devices, this was not a problem. However, when enlarging the network to hundreds of devices, there could be memory issues.

VI. Playing music

There are multiple ways music can be played with Android. This section will elaborate on the choices that were made regarding our system behind playing music. It will specifically discuss the created system for sharing entire songs at once. A system which streams the music has also been created, but that will be discussed in the next section.

A. Setting up the media player

Playing songs on Android can be done using media player objects [7]. Media players read byte arrays from files and convert them into playable music. The music bytes that were received from the network are stored in a temporary file. The media player can then start playing the music from the file.

B. Requesting and receiving songs

The next step is requesting and receiving songs through the network. The local graph contains information about where each song is stored. This graph can be used as a routing table when requesting songs, since the available songs are updated when the graph is updated. To request a song, a request packet is send to the node that contains the file. In the one-packet implementation, the node that receives this packet sends the entire song in one packet to the requesting node. The requesting node receives the whole song at once and stores it in a temporary file. The media player is then able to play the song.

C. Downsides

A big disadvantage to transmitting the entire song in one packet is that the requesting node has to wait for the whole song to be transmitted across the network. Moreover, if there are multiple hops, the song needs to be fully transmitted over each link, which greatly increases the waiting time.

Another downside is that the music is stored fully in memory. This is not a problem for regular sized songs (up to 10MB). However, larger files (with for example lossless compression) can cause nodes to run out of memory when receiving such songs.

VII. Streaming music over the network

In the previous section a basic way of sharing music over an ad-hoc network is described. This basic form of sharing songs has a couple of downsides that can be solved by implementing a form of active streaming. With active streaming, the song is split up in parts of a specified size and sent after each other across the network. This section will be dedicated to elaborating the implementation of active streaming.

A. *Split Packets*

The first step in implementing active streaming is splitting up the songs and sending them across the network. A system was created which splits up the songs into multiple packets and sends them to the receiving node. The receiving device stores each received song packet in a buffer. When all parts are received, it starts playing the song as normal. This system will be referred to as Split Packets. With this system, requesting songs is still the same as described in the section above.

B. *Streaming protocols*

Split Packets almost has the functionality of active streaming. The only thing that needs to be added is that the song parts are played in the form of a stream. Because there was no documentation of a media player having streaming capabilities, AudioTrack [8] was researched.

AudioTrack does have a streaming mode, but it does not provide automatic decoding of the music stream, while media player does. Since there are many different encodings, it was decided that it is out of the scope of this project to implement decoders for AudioTrack.

Instead, the media player supports a feature where it can start another media player right after the previous one completed. In order to stream the music, media players were chained together, each playing the small fragment of a song that is encoded in the received song packet.

As a consequence of Bluetooth being sequential, there is no issue of any song packets arriving in the wrong order. In other words, the packets arrive at the exact order they were sent, and each media player can be played right after the other.

C. *Buffering Streaming MediaPlayers*

Another issue that occurred was that starting a new media player sometimes delayed the music slightly, causing a slight cut in the sound. In most cases this was unnoticeable, but if the device was busy or did a lot of computation, it became more apparent.

This issue was minimized by buffering incoming song packets to the same file, and only making a media player when it is needed in the near future. This ensures that there are as little transitions between media players as possible, while keeping the benefit of streaming.

D. *Limitation*

There is an issue with streaming split-packets. Some music encodings have headers and footers. Since the packets are played in isolation, the media player is unable to understand the file, and therefore unable to play it. Luckily, in most major file formats this does not seem to be the case, so it was decided to continue using this form of streaming split-packets.

VIII. Extensions

We have applied extensions to the network protocol to improve latency and improve bandwidth usage.

A. *Priority Queue*

Splitting packets has the effect that music can be played earlier, but it also has the possibility to interleave packets. When a link-blocking song packet has been sent, another is immediately queued. With the normal setup, the queue is first come, first serve. Thus whenever more important packets are queued, they still have to wait until the entire song has been transmitted.

Instead, we implemented a priority queue, which assigns priorities to packets. Packets such as handshakes and network updates have a higher priority than packets that contain a song stream, such that the network is updated as soon as possible. This preserves the dynamic nature of the network while under heavy use.

B. Queue cleanup

Users of the application tend to skip through songs. A song could be wrongly requested, or they do not like the song. In the original setup, the old song would still be queued, even when a new request has been made.

While the new request traverses to its destination, all nodes clean up their queue to get rid of any song packets targeted to the requester.

There might be additional options to stop packet sending from all sources, and not only from those on the new path, for example by sending a *stop* packet. However, due to time constraints we were unable to implement this.

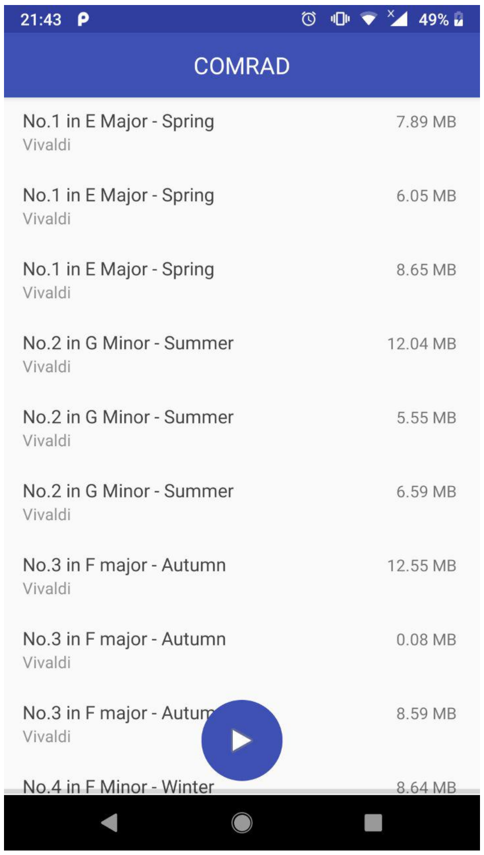
This implementation allows for faster pivoting between songs, which results in both a better user experience and a lessened bandwidth usage.

IX. Experiments

This section will will go over several experiments that were executed to test if the improvements have the desired result.

A. Comrad

The benchmark application was named Comrad. Comrad is built with a minimum Android API level of 16 and runs on Java 7. API level 16 was chosen to ensure all testing devices would be able to run the benchmark. The application is able to display all the songs that the application can find on its own device and of all the other devices that are connected to the same network. It has implemented all previously discussed network protocols. We will demonstrate experiments performed with different versions of the application, such that the network improvements can be compared.



No.1 in E Major - Spring Vivaldi	7.89 MB
No.1 in E Major - Spring Vivaldi	6.05 MB
No.1 in E Major - Spring Vivaldi	8.65 MB
No.2 in G Minor - Summer Vivaldi	12.04 MB
No.2 in G Minor - Summer Vivaldi	5.55 MB
No.2 in G Minor - Summer Vivaldi	6.59 MB
No.3 in F major - Autumn Vivaldi	12.55 MB
No.3 in F major - Autumn Vivaldi	0.08 MB
No.3 in F major - Autumn Vivaldi	8.59 MB
No.4 in F Minor - Winter	8.64 MB

Fig. 5. A screenshot of the Comrad application

B. Environment

There was no budget to buy a similar phone type multiple times. As a consequence, we were forced to execute the experiments with non-uniform device types, API levels and Bluetooth adapters. This influences the tests and should be taken into account with the results.

This situation does reflect a real-life implementation of the network, because many people use wildly varying devices, but it inhibits the ability to draw concrete conclusions out of the results. It was decided to use devices that were the *closest* to each other in terms of hardware, such that the hardware was an unlikely bottleneck.

A different factor that should be taken into account is the area in which the experiments were executed. The experiments were executed in *noisy* areas. Different passive and active Bluetooth devices were operating in the area. This was unavoidable in the working environment, so it was chosen that such an environment would become the standard for the experiments.

C. *Testing parameters, devices and files*

For the testing parameters we use a streaming package size of 256000 bytes, except in the package size test, where we test with different package sizes to determine their significance. This is also were we obtained the testing value of 256000 bytes.

Our testing network consists of the following devices: The Samsung Galaxy S7, Samsung Galaxy S5 Mini and the Samsung Galaxy S3 Neo and are arranged in the following configuration across all tests except for the package size test.

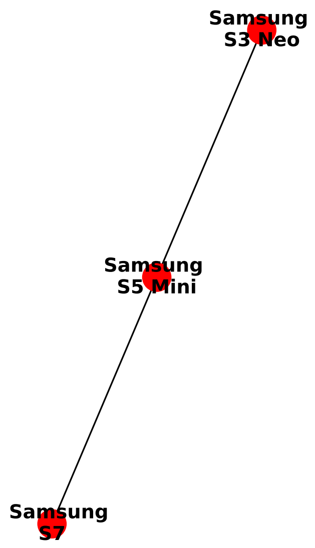


Fig. 6. Configuration used in the first two tests

The audio file used for testing is the song Africa by Toto in MP3 format with a bit rate of 320k bytes. The file is 6.29 MB large and is 4 min and 34 seconds long when played.

D. *Experiment details*

Three different experiments have been performed. All of the experiments we run measure the full transfer time of a song and the time until playing the song. The full transfer time of a song is the time when the last packet of the song has arrived. The time until playing the song is when the first packet has arrived. When running a test without streaming only the full transfer time is listed, as the entire file is sent at once.

The packet size test was conducted in order to find the optimum package size. This size is then used across the other tests.

1) *Packet size*

The audio file used is the Toto - Africa file. This test uses the following configuration: The S5 Mini requests a song from the S7 and does so three times for each package size that we test. The following package sizes were tested: 128000 bytes, 192000 bytes, 256000 bytes, 384000 bytes, and 512000 bytes.

2) *Playing music with and without streaming*

The audio file used is the Toto - Africa file.

This test consists of two configurations. The first configuration is the Samsung S5 Mini which requests a song from the S7. The second configuration is the same configuration described in the *Testing parameters, devices and files* section. There the Samsung S3 Neo requests a song from the S7 with the S5 mini acting as intermediary hop. This test is carried out with two versions of the applications. One that has no streaming and one that does. This test is to find the difference between streaming and no streaming over a single hop environment and a multi-hop one.

3) *Concurrent streaming*

This experiment is run with the configuration described in the *Testing parameters, devices and files* section. The audio file used is the Toto - Africa file.

In this test, two songs are requested concurrently: The S7 requests a song from the S5 Mini and the S3 Neo requests a song from the S5 Mini. This is meant to be a stress-test benchmark. For comparison, the transfer times of the same hops mentioned above are also listed but this time they have been run without concurrent streaming. This is done to make a fair comparison between the two devices since we do not have three devices with the same specifications.

X. Hypothesis

A. *Packet size*

We believe there is an optimal package size. Theoretically seen, if the package size is chosen too small, there will be a relatively larger overhead. If the package size is chosen too large, the time before the music can start playing takes too long (high latency) or the main memory of the device may be filled with packets, resulting the application to crash.

B. *Playing music with and without streaming*

We think that streaming is able to play the song faster since it only has to transfer a tiny amount of data (a single packet) before it can start playing. We also think that the second hop will increase latency to such an extent that it is noticeable for the user. In other words, the song might start buffering in the middle of playback.

C. *Concurrent streaming*

We do not think that a device running the application is able to concurrently stream audio files to simultaneously to two devices without noticeable interruptions in the playback of such devices. We also believe that the newer devices will have a higher performance in this benchmark since it may force the devices to their limit in both Bluetooth adapter transfer time and processing speed.

XI. Results

A. *Packet size*

The third experiment investigates differences in play- and transfer times between different audio streaming package sizes. This test is carried out between two devices: the Samsung Galaxy S7 and the Samsung Galaxy S5 Mini. The audio file used is the same one used by the other tests. We sample the same performance metrics as used before; time until playing and full transfer time for different packet sizes. The packet sizes we tested for are: 128000 bytes, 192000 bytes, 256000 bytes, 384000 bytes, 512000 bytes.

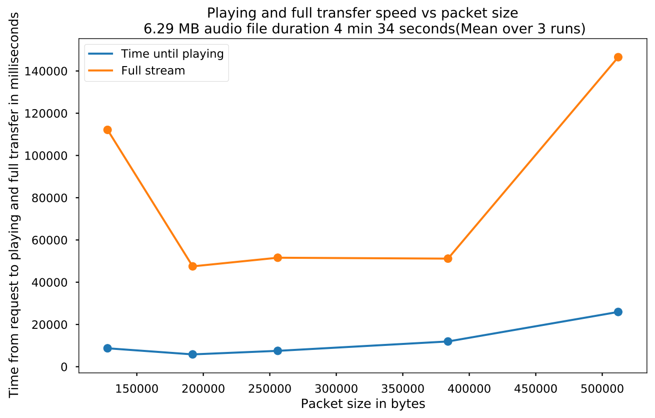


Fig. 7. Here, the orange line is the time it took for the full transfer of the audio file. The blue line is the time it took before the song starts playing (this is when the first packet arrives).

This graph was made to find the optimal parameter for the packet size. The time until playing shows a small upward trend while the full streaming time has a large discrepancy in the middle area.

B. *Playing music with and without streaming*

The first experiment compares streaming and non streaming across one and two hops. The amount of hops describes the amount of links a message has to cross before reaching its destination. The experiment will look into how long it will take before receiving the entire audio file. In the case of streaming, it will also indicate how long it will take before it is capable of playing the song.

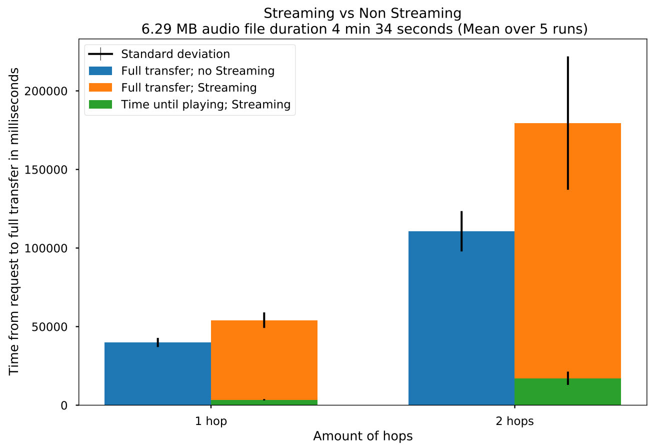


Fig. 8. Left, a distinction is made between transfer times by sending music in one packet versus streaming, over one hop. On the right, the same distinction is made over two hops. With the streaming variant, the time until playing is also displayed.

In the graph it can be seen that the full transfer when sending a file is faster when not streaming the song in chunks. However, when sending the full file, the device can only start playing when the full file is transferred. However, when streaming, the device can start playing when the first packet arrives. This is indicated by the green bar which is a part of the streaming bar (orange).

When comparing the results of the test with a single hop and two hops it can be seen that two hops takes more than twice as long in both situations. When streaming, this effect is stronger as the single hop has a larger percentual increase compared to the non-streaming results.

While streaming, no buffering or interruptions in the playing of the music was encountered.

C. Concurrent streaming

The second experiment investigates the differences between concurrent and non concurrent streaming. The concurrent streaming scenario that will be tested is when two devices request a song at the same time from the same device over separate directly connected edges. This experiment was conducted in a different environment than the previous experiment.

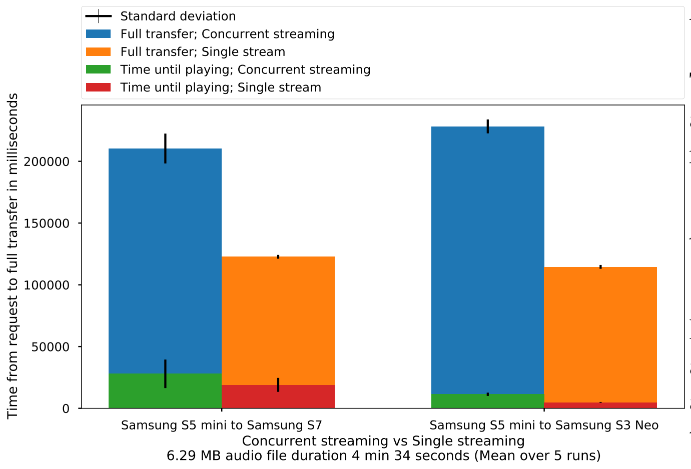


Fig. 9. Here, two one-hop channels are visible with their transfer times. For both channels the time until playing and time until full transfer are displayed.

This graph shows us two groups of bar plots. The left-most graph of each pair indicates the time until playing and the time until full transfer of the concurrent streaming. The right-most graph of each pair indicates the same hop (using the exact same device) when not concurrently streaming to give an insight into the performance of the application when a device is streaming to two requesting devices.

It can be seen that the performance of both hops is quite similar. The main difference is that for the hop indicated by the left pair consistently has a longer full transfer time and time until playing.

It is also important to note that the playing of the streamed song was not interrupted. In other words there was no noticeable buffering.

XII. Discussion

A. Packet Size

From the packet size graph it is clear we found a kind of optimum in the packet size in order to optimize the full transfer. This optimum starts at around 192000 bytes and ends at around 384000. This optimum is not very precisely defined by our testing since we only test for 5 different sizes.

This packet size does not seem to matter much for the time until playing. There is a slight upward trend as smaller packets do arrive earlier. The question is whether it is important to either have a fast first packet or a higher chance that the audio will not be interrupted by the slowness of successive packets.

The package size test corresponded to our hypothesis, an optimum package size can be found in the configuration we tested with.

B. Playing music with and without streaming and multi hop playback

Playing music by first fully buffering the song causes a long delay between requesting the song and being able to play the song. Especially when the request has to traverse multiple hops, the time seems to increase super-linearly. It would be expected that the latency would increase linearly, but because not every device has equal bandwidth and connection speeds, different hops introduce different latencies. Additionally, there is the additional overhead of loading a packet into memory and writing it out on another connection again.

Furthermore, overhead might also be involved in the two hops configuration when a device has loaded a packet into memory and attempts to send it to the next device while another streaming packet arrives from the source device. Bluetooth devices can only send one packet at a time because they have a single band. This might cause a higher latency depending on the timing of the incoming packet and processing time of a packet.

Streaming greatly lessens the amount of time before a song can be played, to about 20% of the fully buffered version. However, the total time of transmission is increased. Encapsulating each piece of the music into a packet is one of the causes of this overhead.

The overhead caused by the intermediary hop did not result in any stuttering or buffering issues in the playing of the music while streaming. This means the next packet was handled before the previous stopped playing. This was not in line with our hypothesis. We believed that bandwidth and latency of Bluetooth would be too low to facilitate a lack of stuttering or buffering.

We were able to send music over three hops, however due to our older devices being unstable it was not possible to extensively test this. A link would sometimes randomly break off, which interfered with the experiment, effectively making it impossible to produce these results within the time constraint.

C. Concurrent streaming

Concurrent streaming about doubles the time before being able to play and the time before the entire song is cached. This is a direct result of the fact that the hosting device needs to time-split the bandwidth to both devices. Half the time is spent sending packets to one device, and the other half is spent to send packets to the other. This causes the doubling of latencies on both devices.

It seems that transfer times of the two devices are very similar when not concurrently streaming. When streaming there is a slight difference in the full transfer times. A larger difference can be observed in the time until playing, or the time until the first packet is received. This has to do with that the source device has to send stream the packets sequentially, so the S7 seems to consistently be slower. This can be due to either the device's slower processing of the packet or that the source device (The S5 Mini) consistently sends the packet to the S3 Neo first. This is an interesting situation since one would think the newer device (The S7) would perform better than the S3 Neo.

It is quite an achievement that the application was able to stream to two devices concurrently without any buffering issues or interruptions, contrary to our hypothesis.

If many songs are requested from one device, the bandwidth would be over saturated, causing great latencies between packets. Some solutions are discussed in Future Research.

XIII. Future research

The choices made in the current implementation of our test application were founded upon theoretical considerations, such as the network protocol. The network protocol we implemented was only tested on a relatively small network in very specific lab-circumstances. It would be in the interest of future research within ad-hoc networking to test alternate protocols and implementations in order to optimize this benchmark. In order to achieve that we believe it is necessary to test a wide variety of protocols and assess which one is better in what situation. Appealing both to lab-situations and real life scenarios. This research is out of scope of this proof of concept.

Another consideration that needs to be taken into account that our tests were not conducted in a interference-free environment. Furthermore, we could also not achieve homogeneity in our test devices. In order to compare the performance of this application to others it is necessary to run all our tests in an interference free environment and on the same devices, to reduce the number of changing variables.

For larger high density networks there might be possibilities in optimizing the amount of nodes you want to connect to. In our version you connect to all devices that are in range, however Bluetooth has a device limit [1] that is implementation dependent. In order to optimize the connections there are several things that can be tried such as: limiting the number of connections to minimize congestion, only connect to the devices based on bandwidth/latency metrics or a combination of the two.

In order to improve performance when streaming multiple streams of the same song to multiple devices it might be beneficial to build in a cache of some sorts, so the load can be distributed over multiple devices.

The size of aforementioned audio streaming packets was at first arbitrarily chosen as 260000 bytes. We performed some testing to find an optimal range of packet sizes. However, more testing is necessary to find the precise optimal size of this parameter for different song sizes. Having it too small creates a lot of overhead for the devices. Having the parameter set too large results in taking a long time before the packets arrive and start playing.

It is also possible to build a feature to monitor the transfer time and packet handling speed of the individual nodes and links between nodes. The information gathered from this monitoring could be used to update the weights of Dijkstra's algorithm. This information may contain for example the CPU power of the device or the performance of the Bluetooth adapter, or even network load. This may result in a more even network load, and hopefully better network performance.

We have only presented one benchmark, that of music streaming. The network can be utilized in many more manners than music streaming, however. For example, video can also be streamed over this network, or it can be used as a distributed file system. In any case, each different application will bring along a different requirement in bandwidth and latency. We have presented a network that performs quite well under music streaming, but there are many other improvements and optimizations possible to improve the performance of the network. We will leave these for further research.

XIV. Conclusion

We have presented a self regulating high-bandwidth mobile Bluetooth ad-hoc network. This network is capable of streaming music over at least two hops. Streaming over three hops did not reliably work with our set of devices.

Streaming music greatly decreases the time before playing, but increases the overall transfer time. The network still performs well, even when two devices are streaming from the same device, leaving no room for buffering or interruptions in the audio playback.

Even though the network performs well in the experiments that were run. There is no confirmation that this network will scale up and perform similarly. Many more optimizations and alternative algorithms can be implemented, all of which will need to be tested more thoroughly than has been done for this paper.

However, this paper and network can be considered a worthy framework or benchmark which can serve as a basis for future research in the high-bandwidth Bluetooth ad-hoc networking field.

References

[1] B. S. W. Groups, *Bluetooth core specification*, 2019.

[2] J. H. Lee, M. Park, and S. C. Shah, "Wi-fi direct based mobile ad hoc network", in *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*, 2017, pp. 116–120. DOI: 10.1109/CCOMS.2017.8075279.

[3] G. Kalic, I. Bojic, and M. Kusek, "Energy consumption in android phones when using wireless communication technologies", in *2012 Proceedings of the 35th International Convention MIPRO*, 2012, pp. 754–759.

[4] *Android developers resource*, https://developer.android.com/guide/topics/connectivity/bluetooth-le.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[6] *Android 6.0 changes*, https://developer.android.com/about/versions/marshmallow/android-6.0-changes#behavior-hardware-id.

[7] *Android reference for mediaplayer*, https://developer.android.com/reference/android/media/MediaPlayer.

[8] *Android reference for audiotrack*, https://developer.android.com/reference/android/media/AudioTrack.