University of Amsterdam

Individual Report

COMRAD

January 28, 2019

Student:
Rick Watertor (11250550)

Course:
Netcentric Computing

1 Introduction

Streaming music has become a lot more prevalent with the recent explosion of music streaming services. However, not all music is provided by these platforms, nor is there always a cellular or Wi-Fi connection. To still be able to listen to a large music library, we have designed an application that combines all music libraries of the participants in the network, and opens it for anyone in the network to stream and listen.

At the core of our application lies a multi-hop ad-hoc Bluetooth network, which can dynamically reshape and regulate itself. All devices partaking in the network will immediately know when a new song comes available or disappears.

We investigated different approaches of implementing this network. Since music streaming is a taxing task, we have performed a series of optimizations on our protocols, such that as much bandwidth for the application is available as possible.

The goal of our application is to research the network we have setup by deploying it for the purpose of shared-library music streaming.

2 Connectivity

2.1 Wi-fi Direct vs Bluetooth

There are some clear-cut advantages of Wi-Fi Direct over Bluetooth. Wi-Fi direct is 10 times as fast as Bluetooth, and is a more stable network. It also has a larger operating radius than Bluetooth.

However, for our application Wi-Fi Direct was not the way to go. (Android) clients are unable to connect with more than one server, thus making our endeavor of creating a multi-hop network impossible. It is not possible to have an android client or server device partake in multiple groups. Bluetooth does support multiple concurrent one-to-one channels, which allows us to create the network we wanted to create; one that spans multiple hops.

2.2 Service Discovery Protocol

The first step for a new device joining the network is actually learning who is in the network.

A Bluetooth protocol that helps here is the Service Discovery Protocol (SDP). SDP allows devices to detect what services nearby devices are running. Sadly, SDP in Android did not seem to work properly out of the box, as it would often return the wrong result.

We tried a naive approach, where we just tried to connect with everyone. This was possible because of the Secure Rfcomm sockets. These secure sockets only allow connections of the devices trying to connect with the supported service. The server's universally unique identifier (UUID) has to match the connecting device's UUID. This way, the application only succeeds connecting with devices also running our service.

While this seemed to work, all attempted connections would set up a pairing between the two devices, which was not beneficial. Our application would have the side effect of setting up pairings with other devices, which is unwanted for both parties, especially since there is a limit of 255 pairings.

Instead, we put more effort into making the SDP work properly. We figured out that detecting the services of all devices should be done in series to have a higher rate of success. This is because when trying to discover all the services at once, the discovery requests would be queued one after another. Because each query takes a while, the other queries would time out, returning an invalid result.

The discovery in series takes a much larger amount of time compared to the naive connection approach, but we can alleviate most of this by caching services of devices. When paired devices contain cached services we recognize, we immediately connect upon starting the app. During discovery we also look for cached services, where we can immediately connect as well. Only when the services were not cached would we have to launch the slow discovery protocol.

We decided this was the best we could achieve with the service discovery protocol, as it would only require the slow protocol whenever the device attempts to connect to new devices. Additionally, the slow SDP can always be done in the background, finding new peers as needed.

2.3 Connecting

Each device is both a server and a client. Upon starting the application, the device automatically starts a ServerThread. This thread listens for incoming connections, and establishes them if needed. It will always continue running, which allows our device to connect with all possible neighbours, up to seven. Additionally upon starting the app, a discovery sequence is initiated.

The discovery sequence first attempts to connect to paired devices that have the cached service, and then starts discovery. Every discovered device is checked for cached services. If they contain the service we provide, we also attempt to connect to the device. Otherwise the device is added to the list of devices we have to inspect with the service discovery protocol.

Once we find a device that provides the service, we start a ConnectThread that will connect with the respective ServerThread on that device. Upon connection, a handshake is performed, which is described later in the protocol section. Now a Bluetooth connection is established.

2.4 Flooding

We implement Flooding, or Broadcasting, by simply sending a message to all our neighbours. To prevent a message from looping and being sent twice, we keep a counter. Each broadcast message contains the sender's MAC Address and the wave identifier. Each recipient can then check whether they have already received this identifier from that address, and if so, decide not to forward the message. Additionally, broadcast messages are never forwarded to the previous hop.

2.5 Multi-hop

To provide a multi-hop connection, every node needs to know which other nodes are present in the network. To gain this knowledge, it would be possible to discover the topology with pings. A ping would inform a node that another node exists, and possibly provide the node with the information about the other node. Constant pinging would mean a constant strain on the network. Since the target application of the network involves a large amount of bandwidth usage, we decided it would be best to keep the bandwidth usage for maintaining the network to a minimum.

Instead, we opted for a more adaptive approach, where when a node connects with another node, it would inform the rest of the network in the form of a flood message. Similarly, if a node's peer falls away, the node would inform the rest of the network of this change in edges.

Extending this, it is easy for each node to understand the entire topology of the network. With this knowledge, each node can decide for themselves how an outgoing package should be sent. We opted for Dijkstra's Algorithm, which allows us to calculate the shortest path to all target nodes. We think the most optimal solution for network congestion is the least hop approach, where we prefer paths that take less hops. To accomplish this, we apply Dijkstra with a weight of 1 on all edges.

We did consider increasing the weight of an edge as the bandwidth on the link reduces, but this act would cause more bandwidth usage, as all nodes would have to be constantly updated on the state of all edges in the network. Additionally, there is no intuitive way to figure out the bandwidth of a given link, let alone the currently available bandwidth.

However, in order to improve the network, there may be a way to identify blocking links, and attempt to reroute messages around this link, such that all the edges in the network can be optimally utilized.

3 Protocol

To provide a common framework for message sending and receiving, we built a message layer on top of the Bluetooth sockets. The messages are put into a packet with a *target*, *type* and *payload*, which allows us to apply uniform actions on the messages, while easily encoding the payload for any message type. It also allows for each intermediate node to understand where the packet is supposed to go, and deliver it properly.

3.1 Handshake

When two devices connect, they perform a handshake. The handshake informs the other device of their entire current network. The packet contains the entire graph of the device. When the packet is received, the difference of their networks is calculated; that is, the device looks at their own network and sees which edges and nodes were added, and creates a *graph update*. Additionally, MAC Addresses are exchanged, if needed. The MAC Address exchange is further discussed in 5.1.

The *graph update* is applied locally to ensure a full view of the network locally.

3.2 Network

Additionally, the *graph update* obtained by the handshake is broadcasted throughout the devices' own network. This way, all devices in the network will be updated of the change in the network and will as a consequence receive all information about devices previously not seen.

Similarly, when a node falls away, the connection vanishes on a peer device. This device then informs the rest of the network, in the form of a broadcast, that the graph was updated.

After each graph update, the shortest paths to all the target nodes are completely recomputed. If there is no such path, the target node either does not exist or lies out of range.

Since every node in the network has the full topology of the network, we simply attach the list of songs to our *self node* (the node that represents our device). The songs can then be automatically serialized into the network packets, and as such the *handshake* and *update_network* packets would inform the recipients, and therefore the entire network, of our song list.

In other words, when a node is added or removed from the graph, it causes everyone's music list to update. Additionally, since the songs are contained in the nodes, all nodes are immediately aware of the source of the song, and can as thus request it from that respective node.

3.3 Songs

To facilitate requesting songs, a *Song* object is sent to the source of the song. The song is always requested from a source as close as possible, so if multiple devices host the same song, it will be requested from the host that has the shortest path to the requester. As a consequence, if the song is on our own device, we can immediately play it. If the song is on a different device, a targeted message is sent, along the path calculated with Dijkstra, to the target. The message contains the song's metadata as payload, such that the target can retrieve this song.

When the target device receives this packet, it loads the song's bytestream and sends it with the *send_song* packet over to the requesting device.

4 Streaming

The bytes received by the *send_song* packet were originally fully cached into memory, stored in a temporary file, and then read by android's media player. However, this has a few drawbacks.

Firstly, the connection between the source device and the target device would be fully saturated by the stream, allowing no network update packets to cross. This would sometimes cause an enormous delay in network updates. Additionally, each device that is partaking in the path would have to cache the entire song into memory. Some songs take up a lot of memory, resulting in a shortage of memory. Furthermore, the user would have to wait a long time before they can start playing the song.

We resolved these issues by streaming.

4.1 Splitting Packets

Instead of fully loading the song into memory, and transmitting it all at once, we separate the song into smaller packets. Each part of the song is split up into a byte array of 256000 bytes. This resolves the requirement for any device to fully load the song into memory, except the playing device. Additionally, it allows for interleaving important packets in between the sending of the song.

To interleave important packets between less important packets, we had to define a priority queue. Each link has a queue of messages of three importance rates: HIGH, MEDIUM, and LOW. When choosing what to send next, it will first tap the HIGH queue, then the MEDIUM queue and at last the LOW queue. Handshake packets are the highest priority, followed by update network packets, and the low priorities are the requests of songs and the actual sending of songs.

This way, the most important packets are sent before less important packets. The whole network will therefore be updated as soon as possible. Note that this approach would not have worked without splitting the packets, as there is no way to interleave packets into one packet.

4.2 Playing Music

Initially we opted for playing music once all packets arrived. We would simply store all incoming music packets into a buffer, wait until the buffer was filled and then play it with Android's MediaPlayer. Waiting for all packets to arrive would introduce a large delay before the player is able to play the song, even though it already had information ready to play.

Because there was no documentation of a MediaPlayer having streaming capabilities, we researched AudioTrack, which does have a streaming mode. However, AudioTrack does not provide automatic decoding of the music stream, while MediaPlayer does. Since we are supporting a wide range of music files, we decided it was too much effort to implement decoders for AudioTrack properly. Instead, MediaPlayer supports a feature where it can start a new MediaPlayer right after the previous one finishes. In order to stream the music, we simply chained a lot of MediaPlayers together, each playing the small chunk that is encoded in the packet.

There are some drawbacks to this approach. Some music encodings have headers and footers. The chained MediaPlayer is unable to figure out the format of the file, and therefore unable to play it. This seems to be of no concern with most major music files, but can be an issue with some formats. Additionally, on slower devices the starting of a new MediaPlayer may cause a small delay in the sound, resulting in a very brief jagged sound.

5 Miscellaneous

5.1 Retrieving MAC Addresses

The Android API has become more and more protective of programmatically gaining access to the Bluetooth MAC Address. As such, we had to create multiple ways to retrieve the address for different API versions.

For all API versions that support it, we get the MAC address from the Bluetooth handler. APIs 6.0 and above do not support this, and on some devices we could still access this through reflection. However, recently Android has also patched this loophole, and there is no way to programmatically get the MAC address anymore. Because our topology heavily relies on MAC Addresses, we had to come up with another way.

Neighbouring devices *can* find your mac address, thus we devised the following protocol: during the handshake, both devices set their source mac and their target mac. If our device's mac equals 02:00:00:00:00 (that is, undefined), we retrieve the destination mac from the incoming packet, update our local network and use that MAC address as our MAC address.

However, the device that could not get their MAC Address could already have sent their faulty graph. Thus, when the other device receives their handshake, they check the received graph for undefined addresses, and replaces any with the source MAC of the device of the incoming handshake. This establishes with certainty that the network on all devices is correct, and that all devices know the correct MAC addresses of the newly connected devices.

5.2 Idle State

A performance improvement that we made was the disabling of discovery and the Service Discovery Protocol during media streaming. We noticed that whenever SDP and discovery was running, all other Bluetooth traffic would be greatly delayed, especially when there were a lot of unknown devices around.

So instead of keeping SDP and discovery on at all times, we forcefully stop any discovery and SDP when a song request is sent, forwarded or received.

6 Conclusion

A lot has to be considered when setting up an ad-hoc network. Optimizing the network often requires thorough analysis and inspection of the network.

We have presented an implementation of an ad-hoc network using Bluetooth, which can dynamically adapt to changes. Because of our improvements to the network, it is robust and performant enough to facilitate music streaming.