Opportunistic Crowdsourcing for Collective Music Sharing

Zhaofei Chen

Degree project in
Electrical Engineering
Master of Science,
Stockholm, Sweden 2011

XR-EE-LCN 2011:009
Master Thesis Report

Opportunistic Crowdsourcing for Collective Music Sharing

Zhaofei Chen

Supervisor: Professor Gunnar Karlson

Date: 22 August 2011

School of Electrical Engineering
Kungliga Tekniska Högskolan
Stockholm, Sweden
Abstract

The fast spreading of smart mobile devices has changed the way people create and share multimedia content. Infrastructure networks, such as 802.11 and cellular, can provide users Internet access to disseminate this content, however their services may be limited due to insufficient deployment of infrastructure equipment, network congestion as downloading of large-size multimedia content becomes highly prevalent, or simply high prices that prevent users from utilizing these services. Middleware architectures have been designed and implemented on smart mobile device platforms to distribute content opportunistically when devices are within each other's communication range: mobile devices can either utilize connections to fixed infrastructure, such as public access points when in range, or distribute content to other devices directly in ad hoc fashion when in proximity.

In this work we have designed and implemented a real-time music-sharing application that utilizes a content-centric opportunistic networking middleware. The application provides an interface for users to publish audio files stored in their own mobile devices to a subscriber jukebox that is within their communication range. The jukebox then plays the audio file that is first in queue in its playlist by streaming it from the publisher user device. User devices are synchronized with the playlist in the jukebox so that users can observe the contents of the playlist on their own displays in real-time.
Content

1. Introduction ................................................................. 1
2. Background ............................................................... 3
   2.1 Mobile ad hoc networks (MANET) ............................... 3
   2.2 Delay tolerant networks ........................................... 4
   2.3 Content dissemination ............................................. 5
3. Related work ............................................................. 8
4. Podnet: Mobile P2P Opportunistic Content-centric Networking .......... 12
   4.1 Introduction .......................................................... 12
   4.2 Podnet architecture ................................................ 12
   4.3 Content exchange in Podnet ...................................... 13
   4.4 Implementation of Podnet system .............................. 14
5. Collective Music Sharing System ...................................... 16
   5.1 System overview .................................................. 16
   5.2 System design ...................................................... 17
      5.2.1 The jukebox application module ....................... 17
      5.2.2 Mobile application and UI design ..................... 20
      5.2.3 Sharing publish/subscribe system ..................... 22
      5.2.4 The role of Podnet service in the system ............. 23
      5.2.5 Playlist handling ........................................... 25
      5.2.6 Real-time streaming ....................................... 25
      5.2.7 Packing a MP3 audio file ............................... 29
      5.2.8 Scenarios considered for user experience and performance 29
   5.3 Implementation .................................................... 30
      5.3.1 Publishing a new audio file ............................. 30
      5.3.2 Receiving a published audio file ...................... 31
      5.3.3 Updating the playlist ..................................... 32
      5.3.4 Synchronization of the playlist ......................... 34
      5.3.5 Saving and loading history ............................. 35
      5.3.6 RTSP server ............................................... 37
   5.4 System test ......................................................... 38
6. Conclusion and Future Work ........................................ 40
   6.1 Conclusion ......................................................... 40
   6.2 Future work ....................................................... 40
7. References ............................................................... 42
1. Introduction

Wireless mobile networks are transforming the way we communicate and exchange contents. As smart mobile devices are becoming more popular, media content, such as images, audio files or videos, can be generated by anyone at any time and any place. Content is more commonly exchanged when people are on the move. Mobile devices can get access to internet via infrastructure networks provided by operators or available 802.11 access points. Although, 802.11 wireless access points can provide connectivity with relatively high data rates, their coverage is limited. The problem may partly be solved by deploying multiple access points to increase the coverage, yet this approach may still not be the best when network access is required for larger areas. Cellular networks, on the other hand, are capable of providing network access with larger coverage. However increasing number of users and quantity of shared contents can severely degrade the service quality, since limited capacity of a single base station may become a bottleneck. Deploying more infrastructure equipment and reducing the size of cells can increase the system capacity but it will also increase the costs and require more complicated network planning. Therefore, a complementary approach which allows mobile devices to communicate with each other without pre-built infrastructure should be considered.

![Diagram of mobile devices communicating without infrastructure](image)

Figure 1.1 Mobile devices communicate without infrastructure

This work focuses on content sharing in mobile ad hoc networks. We have designed a collective music sharing system which maintains a mobile ad hoc network topology among a central jukebox and a set of mobile devices. Mobile devices in the system can share music files that are stored in them with the jukebox by a publish-subscribe system and the jukebox can then play the music. Our system is designed to demonstrate that the concepts, such as mobile ad hoc network and publish-subscribe system, can be utilized to complement existing wireless content distribution approaches.

The rest of this work is organized as follows: Section 2 presents mobile ad hoc networks, delay-tolerant networks and publish-subscribe architecture. Section 3 contains the related work. Section 4 gives the Podnet architecture in detail. The
collective music sharing system which is designed and implemented in this work is presented in Section 5. Section 6 concludes the work and discusses future work.
2. Background

2.1 Mobile ad hoc networks (MANET)

A mobile ad hoc network (MANET) represents a system of wireless mobile devices that can be freely and dynamically self-organized into temporary topologies and it allows these devices to communicate with each other without a need for pre-built network infrastructure [1]. This characteristic allows MANETs to be established fast and with comparably lower cost. Moreover, failure of any single node usually does not have a severe influence on the performance of the whole network while any failure of central infrastructure in a traditional wireless network will cause all mobile nodes to be disconnected. With such advantages, MANETs can be used in applications of different areas, including [2]:

- Military communications which require fast-established networks.
- Emergency operations such as search and rescue. The operation team in the locale can establish MANETs for communication and information sharing.
- Wireless LAN conference. Temporary ad hoc WLAN can help people sharing information within an activity such as conference.
- Vehicular communications. MANETs can be used for communication among vehicles and between vehicles and roadside equipment.

To implement MANET we should first consider the characteristics of its topology. Since nodes in MANET are mobile and may join and leave the network at any time, the topology of MANET is dynamical. Thus a protocol is needed to sustain the connectivity between the participating nodes and to keep all the nodes accessible. An ideal protocol should not only maintain contacts between all participating nodes but also provide energy-efficient connections with relatively high throughputs.

Typically, mobile nodes in a MANET have limited communication range due to limited radio power and a node in MANET is usually not within direct communication range of all other nodes. Therefore a routing protocol is needed when the data needs to be transmitted between two nodes that are not within each other’s communication range. A considerable number of routing protocols in MANETs have been proposed so far and it is difficult to judge which one performs better with respect to different network scenarios. Routing protocols can generally be classified into three types [3].

Proactive routing protocols (table-driven protocols)

Proactive routing protocols require the node to maintain consistent and up-to-date routing information for every node in the network. When there is a change in network topology, routing information in each node should be updated [4]. Proactive routing protocols are also called table-driven protocols since the information is stored in the routing tables.
Reactive routing protocols (demand-driven protocols)

Reactive routing protocols are introduced to reduce the amount of overhead transmitted in the network since they establish routes only when nodes need to transmit. When a source node is ready to send a message to a destination, it invokes the route discovery mechanism to find the path to the destination [4]. The discovered route will remain valid until the transmission is over or the destination node becomes unreachable.

Hybrid routing protocols

Hybrid routing protocols are both proactive and reactive in nature. They allow the nodes to maintain detailed routing information of nodes that are nearby and to determine routes to nodes that are far away with the route discovery mechanism [3]. Hybrid routing protocols attempt to minimize the overhead transmitted in the network and choose the most suitable node for route discovery.

2.2 Delay tolerant networks

Traditional TCP/IP based Internet services provide end-to-end communications through a packet switching network model. A number of key assumptions are made in order to achieve this service [5]:

- An existing end-to-end path.
- Guaranteed link speed between every two routers.
- Low packet loss rate.

However, some challenged environments may violate one or more of these assumptions. For example, communication between a ground control center and an orbiting satellite can only be established during a specific period of time and the data transmission suffers from high delay and low link speed due to the long distance in between. The networks which lack continuous connectivity and suffer from frequent and long-time partitioning are called delay-tolerant networks (DTN).

In [5], the author proposed a framework for DTNs. Figure 2.1 illustrates the general scenario of a DTN given in [5]. The whole scenario is divided into four regions while network connectivity among the regions is not continuous. For example, region A can establish a network link with region B only when the bus in region B periodically passes near region A. The DTN gateways in the scenarios interconnect the regions and provide data forwarding service. In this architecture, an overlay, which works above the transport layer, is proposed as the bundle layer. The concept of bundle is introduced as a set of messages and the goal of this architecture is to deliver these bundles from source to destination. DTN gateways provide the services of storing and forwarding the bundle data from one region to another which is nearer to the destination.
The main characteristic of DTNs is lack of connectivity between the source and the destination, and this is also the major difference between a DTN and a MANET. When we consider sending messages, routing protocols proposed for a MANET are usually based on stable path or connected topology which a DTN cannot guarantee. To solve this problem, the routing protocols in a DTN must take a “store and forward” approach to guarantee a reliable data transmission. A large number of routing protocols have been proposed for DTNs. These routing protocols can be classified into two types: forwarding-based routing and replication based routing [6].

**Replication-based routing protocols**

In replication based routing protocols, the nodes replicate the message and send it to nodes that have been contacted in order to achieve higher delivery rates and lower latencies [7]. However this would lead to flooding the network with the replications of a message. In order to maintain a balance between high delivery rate and limited resource consumption, the protocol proposed in [8] introduced the spray-and-wait mechanism. With this protocol the nodes send replicated messages only to a portion of the nodes that have been contacted rather than all of them.

**Forwarding-based routing protocols**

The routing protocols that do not broadcast replicated messages are called forwarding-based routing protocols. Since only one copy of the message is routed, there will not be any wasted messages and the destination node does not need to provide feedback to help discarding the copies of the sent message. However, the delivery rate of forwarding-based routing protocols is usually relatively lower than for replication-based routing since messages may not have a high probability of delivery to their destinations in the network.

### 2.3 Content dissemination

With the recent developments in Web 2.0 technology and the increasing demand among people to share media content, Internet has become the most important platform for content sharing today. Applications, such as blogs, and other social
networking platforms, such as Facebook and Youtube, have provided people various ways to share contents.

In order to use network resources efficiently and to maximize the data rate that users experience, content distribution technologies such as content distribution network (CDN) and peer-to-peer networking have been developed. The concept of CDNs introduces a set of servers which store copies of contents that are usually stored in central servers. These servers are then distributed to various points in the network so that the content can be accessed by connecting to any of these servers, instead of connecting to the central servers. As a result, CDN technology lowers the probability of occurrence of bottlenecks close to a central server. However it requires high quality of load balancing control which increases the cost of deployment and management of the system. A peer-to-peer system also represents a similar content distribution concept to CDN. The difference is that content is distributed among end users who do the downloading.

Publish/subscribe system is another approach for content dissemination. To our knowledge, the earliest proposed publish/subscribe system is a news service in a software toolkit [9]. The subscriber who uses the service can subscribe to different news subjects and only receive the news published under these subjects. In the publish/subscribe system, the publisher does not need to know anything about the subscribers. Instead, it posts notifications, and the subscribers receive these notifications to download the newly published contents. The notifications do not need to be received by the subscribers immediately after publishing. The process can be asynchronous. According to [10] and [11], the publish/subscribe system can be described with three subscription models.

- **Topic-based.** The published notifications (contents) are grouped in topics (subjects). Any subscriber who subscribes to a specific topic will receive all the notifications related to this topic.

- **Content-based.** There is no pre-defined criterion for the contents. However the subscriber can have its individual criterion. Only the contents which match this criterion will be received by the subscribers.

- **Type-based.** It is similar to a topic-based system but instead of grouping notifications into topics, it uses the types in a programming language. Subscribers of a specific type will receive the instance of this type and its subtypes.

Publish/subscribe systems have been widely used in wired networks and generally the delivery of contents from the publisher to the subscribers is achieved by multicast. However, problems occur when implementing multicast in a DTN, since the network topology varies frequently and the publishers usually do not have connection to all subscribers. In order to achieve the delivery of content in publish/subscribe systems for DTNs, some solutions have been proposed.

SocialCast [12] is one of the proposed routing solutions that deliver content from publishers to subscribers in DTNs. First, the publisher picks up a fixed number of
neighboring nodes as carrier nodes and sends copies of the content to them. A carrier node delivers a copy of the content to a subscriber node that it has contacted or passes the content to another carrier which is more likely to contact the subscriber. Therefore the carrier node should have a buffer big enough to store the content.

In [13], a community detection algorithm divides the network topology into several communities. Within each community, a node is selected as the message broker. The publisher in the network delivers the content to the broker in its community and the content will be exchanged between the brokers of all the communities. Once a broker detects a subscriber in its community, the content is delivered.
3. Related work

Having Internet access is still a problem that needs to be solved for a considerable number of people living in the developing regions around the world. Therefore a social network service (SNS) with no Internet connectivity that can provide users with services such as search or resources sharing should be designed. The authors in [14] claim that even though the infrastructure is poorly distributed in some part of these regions, mobile phones have high penetration. Therefore, they proposed social network service called Goose. Goose is a SMS-style SNS that allows users to send voice or text messages over Bluetooth. In a region that lacks GSM coverage, messages are delivered via DTN which is established by human mobility. The application scenarios of Goose are given in Figure 3.1.

![Figure 3.1 Goose application scenarios [14]](image)

In a unicast scenario, a user can send a message to another user via Bluetooth social DTN or a GSM network if it is available. In a broadcast scenario, a user simply broadcasts a message to all other users within a community asking for help. The message in Bluetooth DTN is transmitted with the store-forward concept that is widely adopted.

The Goose architecture contains three manager modules. The contacts manager maintains an address book and takes care of users in the social network. A forwarding manager searches for neighboring users and implements an algorithm for storing and forwarding messages. Finally a network manager provides both Bluetooth and SMS interfaces and takes care of sending and receiving messages.

Goose system provides an alternative approach for sending messages with mobile phones. However its performance is still limited and it may only be complementary to a cellular network.
BitTorrent is a P2P content dissemination protocol which is widely used in Internet. The protocol requires each client, which is called a peer, to serve others by sharing some of its uploading bandwidth to increase the total downloading bandwidth in the system. A peer in the system will discover other peers by contacting a central node called tracker. The tracker stores IP address of all the peers in the system and maintain statistics of uploads and downloads of each peer. BitTorrent has become the most efficient content sharing protocol in Internet due to its good performance within wired networks that guarantees stable end-to-end communications. However, it will be a challenge for BitTorrent to perform well in a mobile wireless ad hoc network environment. Due to the dynamic topology of MANETs, the tracker in the system may be too far away or even invisible for some peers and it will also be hard for a peer to choose which peer to contact under such environment.

The authors in [15] proposed a trackerless approach and made changes to the original BitTorrent protocol. They implemented a peer discovery mechanism that can help peers to discover others without contacting the tracker. The mechanism uses periodic “Hello” messages to discover and connect to neighboring nodes. To share content, a peer needs to send “Offer” messages to neighboring nodes announcing the content pieces it has and the neighboring nodes may then choose to accept or reject the offer based on their preferences.

BitHoc is another application addressing the problem proposed in [16]. It is a trackerless BitTorrent-like application which is adapted to mobile wireless ad hoc networks. The system architecture is shown in Figure 3.2.

![Figure 3.2 Architecture of BitHoc [16]](image)

BitHoc architecture includes a content discovery service called BitHoc search engine, a membership management service called BitHoc tracker and a content sharing service called BitHoc client. Every time a user wishes to share contents, a meta-information file is generated by the BitHoc client and published to the local BitHoc tracker. Other nodes can search for this content by sending messages to the local tracker. If the content matches the search request, then the user can start downloading. The authors tested the system with several mobile devices and the results show that BitHoc achieves higher sharing ratio and less downloading time when compared with the traditional BitTorrent protocol.

Adaption of the BitTorrent protocol to MANETs is challenging due to intermediate
connectivity issues.

In [17], the authors proposed Haggle, an architecture for mobile devices which enables seamless network connectivity and application functionality in dynamic mobile environments. The main purpose of Haggle is to adapt various mobile applications to work seamlessly under different network connectivity scenarios. Figure 3.3 illustrates two applications that have been implemented with Haggle prototype.

![Figure 3.3 Haggle email and web application [17]](image)

Figure 3.3(a) shows the working process of Haggle Email application. When an email is sent by a user, it is delivered to Haggle’s core after being translated into corresponding data objects that can be managed by Haggle. Then Haggle decides what type of network connectivity is needed to send this email. If network access is available, email will be delivered following the standard procedure. In case it is not, a delay tolerant ad hoc link is used for delivery.

The web application is illustrated in Figure 3.3(b). When Haggle users browse the web with the Haggle web application, the downloaded data will be stored in an internal cache on their devices. This data can then be requested by another Haggle web user who would like to browse the same web content but does not have Internet access. Haggle web application allows users to have access to a web page not only through infrastructure networks as it is traditionally done, but also via peer-to-peer ad hoc link established with others.

Just-in-time binding [17] allows Haggle to adapt to different network environments by selecting the network connectivity interfaces only when transmission of data is required. This approach allows Haggle to operate over multiple interfaces, protocols and applications. Haggle can create metadata to be used in applications. This allows Haggle to support data-driven network operations. Moreover, Haggle manages the use of network resources centrally to coordinate the behaviors of applications to meet expectations of users. Any action that needs to access to resources is considered as a task and will be processed by a centralized module.

MobiClique [18] is a middleware architecture which allows users to contact others to exchange social network data opportunistically in a mobile environment. System
overview of MobiClique is shown in Figure 3.4

MobiClique users can update the social network information on their mobile devices either by connecting to a node in an infrastructure network, or to other mobile users opportunistically when they are within communication range. A three-phase periodical process is implemented on each device to exchange contents opportunistically. First, the MobiClique user discovers others within the communication range using Bluetooth or ad hoc Wi-Fi. Then when the link between two devices is established, the identification phase begins and social profiles on the devices are exchanged. Finally the content sharing phase allows the devices to exchange the social network contents to be shared.

MobiClique is implemented based on the Haggle architecture. A social network manager is added to the Haggle architecture to allow users to exchange social profiles and content. MobiClique also provides an API for further development. Although MobiClique is designed specifically for social network applications, it can also be used to disseminate various kinds of contents opportunistically.
4. Podnet: Mobile P2P Opportunistic Content-centric Networking

4.1 Introduction
Mobile devices are usually carried by the users and moved around according to users’ mobility. Therefore each two mobile devices can have a probability to stay within direct communication range in a certain public area which their users may get into. Media contents on the mobile devices then can be exchanged during these opportunistic short meetings. Instead of communication with network address and routing protocols, the mobile devices should contact each other based on the contents they have. During a meeting, one of the devices should check the content on the other one and decide which to request. This opportunistic content-centric networking concept, which is achieved with content-driven protocols and devices’ mobility, can be used for content dissemination among mobile devices. The Podnet middleware architecture is proposed according to such concept.

4.2 Podnet architecture

The Podnet system is a mobile P2P middleware architecture designed for opportunistic content dissemination. It allows mobile devices to exchange contents opportunistically without support from infrastructure network. Figure 4.1 illustrates the main architecture of this middleware.

![Figure 4.1 Architecture of Podnet middleware [19]](image)

Content needs to be organized hierarchically to be disseminated efficiently. It is divided into different feeds and each feed can have unlimited number of entries. For example, the pictures about yesterday’s football matches can be organized into a feed named “Yesterday Football”, and several entries can then be created under this feed while each of the entry contains pictures for a single match. A specific piece of
content, such as a picture in the football match, can be found as enclosure under an entry. This data structure has been implemented in both the local content database and the synchronization manager module, in order to make the search and synchronization services between all devices.

Podnet provides an API, which includes a publish/subscribe interface, to the application layer above. Applications can organize content that users would like to share according to the afore-mentioned data structure and publish them to the affiliated feeds. They can also receive contents from others by subscribing to the feeds that users are interested in. The discovery module is designed for devices to maintain connectivity with neighboring nodes. The transport module allows the middleware to divide large contents further into chunks.

The middleware implements a receiver-driven solicitation protocol, which allows devices to request contents from their neighbors based on the feeds that they subscribe to. Since the system is entirely content-centric, a traditional network layer may not be necessary and thus common unicast routing protocols are not needed.

4.3 Content exchange in Podnet

Content dissemination in Podnet system is achieved by device mobility. Publish/subscribe architecture and the solicitation protocol which are implemented on each mobile device. Figure 4.2 explains the procedure when two mobile devices contact opportunistically.
The discovery module in each device sends discovery beacons periodically while listening to beacons sent by other devices. In Figure 4.2 device B discovers a beacon sent by device A, it checks its contact history in the discovery module to find out whether device A is a newly discovered neighbor or a previously discovered neighbor with a newer version of contents. After confirming device A as a node, device B starts a new client service and sends a message to request a list of feeds from device A. When it receives the feed list reply from device A, device B checks if it is a subscriber of any feeds in the list; if it is, it sends a request message for the entry lists that belong to those feeds. When new entries are discovered in the entry lists sent by device A, device B sends a request to download the contents of these entries. The content are then downloaded via enclosures.

Since each device can send and receive beacons at the same time and they may have multiple client services simultaneously, the content exchange can be bidirectional and between multiple devices.

4.4 Implementation of Podnet system

The Podnet system is implemented in Java on Google Android OS platform. The
middleware is implemented as an Android service that runs in the background. When the service is started, the discovery module creates two threads to handle the beaconing process between user devices. A sender thread periodically sends out beacons while the listener thread receives beacons sent by neighboring devices. Each beacon message contains a globally unique ID to identify the source device and a content revision number which represents the current update version.

When the listener receives a beacon which indicates that a new neighboring device is discovered or content database of a previously contacted device is updated since last encounter, it creates a new client thread to communicate with this neighboring device.

When the middleware is started, a server socket is opened. This socket listens to the requests sent by neighboring devices and creates new server thread to serve them when necessary. The server can handle multiple clients at the same time.

The content database on each device is implemented as an Android Content Provider. Meta-data of feeds and entries are stored in this database and can be utilized by any application via Content Resolver in Android.
5. Collective Music Sharing System

5.1 System overview

The main motive of this system is to enable users to share contents, in this case music files stored in users’ own devices, in real-time without infringing on copyright laws. In contrary to other wireless ad hoc content sharing system, the concept of “sharing” here does not aim at exchanging the music files between neighboring nodes but streaming them to a central unit one at a time so that others can enjoy the music as long as they are in the vicinity. Typical venues to setup such a system can be public areas such as bars where people gather for entertainment. The system overview is illustrated in Figure 5.1.

![System overview diagram](image)

**Figure 5.1 System overview**

The system has two main actors: A computing device with the jukebox application which is connected to an audio system at a public venue, and a mobile device that is within communication range. The circle in Figure 5.1 indicates the communication range between the jukebox and mobiles devices. Every time a device is within the communication range, it can establish a direct link with the jukebox. Once the link is established, the mobile user can then play the music he/she wishes to share with others.

In order to manage and monitor the music shared by participating mobile devices, a playlist is created to display the songs published by the users. Mobile devices
synchronize their playlist with the jukebox.

5.2 System design

Figure 5.2 has illustrated the system modules in these applications.

![Figure 5.2 System modules](image)

The mobile application contains three main modules. Mobile Application module is basically an Android application which provides users a UI to publish songs and monitor the playlist. The Podnet middleware enables the contents exchange between user devices and the jukebox. Since shared songs are streamed in real-time, user devices need to have RTSP stream server modules.

The jukebox application has four modules. The Jukebox Application module provides users a UI to control the jukebox manually when needed. The RTSP Streaming Client module enables the streaming service in cooperation with RTSP streaming server. The Playlist Handler acts as the database of the Podnet service running on the jukebox and manages the playlist. Details of each module are introduced in following subsections.

5.2.1 The jukebox application module

Jajuk is a full-featured open-source jukebox application which is written in Java. It supports multiple OS platforms [20]. Various UI components and perspective designs in Jajuk are utilized in this work for a more convenient user experience. Figure 5.3 shows the main window when Jajuk is started.
Figure 5.3 Main UI window of Jajuk

The main display can be divided into five areas with different functionality. The menu bar on the top provides drag-down menus which are keys to call all functions in Jajuk. The perspective bar on the left provides several modifiable perspectives and gives the user multiple choices to browse and manage audio collections. The large field in the middle is used for displaying UI components in different perspectives. Jajuk provides users considerable number of dynamic UI components which can be used to build up unique customized perspectives for users. At the bottom of the window, the control panel and the playing status bar can be found. The user can give quick commands, such as “play”, “pause” or “stop”, to the media player by pressing the affiliated buttons.

Jajuk has a series of procedures to play a local audio file. This process is illustrated on Figure 5.4.
When a user wants to play an audio file using Jajuk’s UI he/she needs to either drag the file to the playlist or adds it to the playlist via Jajuk’s function menu. This dynamic playlist is created when Jajuk starts and can be saved into a file at anytime. Jajuk also provides several automatic playlists which can be generated according to user rates or bookmarks. All playlists can be monitored via a UI component called "Playlist View".

Regardless of which playlist is currently deployed, the files in a playlist are pushed into a final queue before they are played by the media player. The order of the files in this queue cannot be modified, which means only insert and remove operations are allowed. The status of this queue can be monitored via another UI component called "Queue View".

Finally, the media player is called to play the files in the queue. In Ubuntu OS platform, Jajuk collaborates with a media player application called “MPlayer”. Every time an audio file needs to be played, Jajuk generates a Linux command line to call MPlayer application to do the work. MPlayer provides feedback to Jajuk so that status information about the audio file being played can be displayed. MPlayer is an open-source movie player application which can run on multiple operation systems [21]. It can play various kinds of media formats and support a wide range of output drivers. Moreover, since it is open-source software, it can be customized using any library to satisfy user experience.

Jajuk UI has been modified to support the functions of our system. Figure 5.5 shows the screenshot of the modified UI.
We have added a new drag-down menu called “Podnet” to the Jajuk menu bar. This menu contains two functions to turn the Podnet service on and off and to reset it. When the Podnet service is reset, the content database and contact history will all be cleaned. The playlist in the juke box will also be cleaned. A Podnet perspective is also created with a single Queue View UI component. This perspective allows the user to monitor the audio files in the final queue of the playlist. There are four different columns in the playlist showing the title, the artist, the album and the recorded year of each audio file. We have replaced the album information with the device ID of the user device which published the corresponding audio file. Further details are introduced in the following sections.

5.2.2 Mobile application and UI design

We call the application that provides the interface for users to interact with the jukebox “PodMusic”. Figure 5.6 shows sample screenshots of this application.
The main screen in the PodMusic application has two tabs. The tab labeled as “Published” which is shown in Figure 5.6(a) contains the list of audio files that are shared by the device itself. The tab “Playlist” which is shown in Figure 5.6(b) displays the playlist in synchronization with the jukebox. Each row in the published list includes the title of the file and the device name of the mobile device that provides it. This lets users easily monitor all files that have been published by their device. A “PLAYING” tag is added next to the currently playing audio file shown in the playlist to inform the users.

When the “MENU” button is pressed, a system menu pops up at the bottom of the main screen. The menu functions, “Add Music” and “Setup”, are shown in Figure 5.7.

When the “Add Music” button is tapped, the application launches a new screen with a list of audio files available in the device. A sample screenshot is shown in Figure 5.8(a).
Each row in this list contains the title of the available audio files. When users choose any of these files, it is published and added to the published list in the main screen. Since this audio file list is provided by the media content provider built in the Android OS platform, all available files in device’s storage is accessible, regardless of where they are stored.

When “Setup” button in menu is tapped, the “Setup Preferences” screen is launched. Figure 5.8(b) shows a sample screenshot of the setup screen. PodMusic application allows users to set their device’s names.

### 5.2.3 Sharing publish/subscribe system

Sharing audio file information and playlist synchronization are achieved with the help of Podnet middleware in our system. The Podnet middleware provides a publish/subscribe interface via its API to the applications that bind to it.

Figure 5.9 illustrates the publish/subscribe architecture in our system. The system creates *music* and *playlist* feeds that enable all mobile devices to publish audio files and synchronize their playlists with the jukebox. The jukebox is subscribed to the *music* feed to serve the stream requests sent by users. It publishes the playlist to the *playlist* feed when it is updated so that user devices can synchronize their copies of the playlist. With this design, all participating devices can have a complete list of audio files that are published in the system.
5.2.4 The role of Podnet service in the system

The Podnet middleware has a Podnet Service Controller to allow users to turn on/off the Podnet service or reset the database and contact history. Figure 5.10 shows a sample screenshot from the controller.

Podnet service that runs on the jukebox needs to be modified since Ubuntu operating system does not have the Content Provider component as the Android operating system running on the user mobile devices have. We need to keep two lists of content entries on the jukebox subsystem for the two content feeds we have. Synchronization
module in the Podnet middleware also needs to be modified accordingly to support search, read and write functionalities into these lists.

**Entries in the content feeds**

The content feeds and entries in the Podnet middleware are defined with globally unique strings which are considered as IDs. Figure 5.11 shows the data structure of an entry.

![Figure 5.11 Structure of the entry](image)

When an entry is published for the first time, a globally unique ID will be generated and this will be the only identifier for this entry. This ID is never renewed regardless of how many times this entry is published or exchanged between different user devices. Information, such as the name of the author, a summary description and geographical location of the device when the entry is published can also be added to the entry, though it is not mandatory. The rest of the structure has information about the enclosure. An enclosure flag is set to indicate whether the entry contains an enclosure or not. If an enclosure exists, other information fields, such as enclosure name, type and size are added. Finally a local URI is set and the synchronization module is able to find the content via this identifier. An entry can be published without enclosure. In this case the enclosure flag is set to false and the rest of the enclosure data fields remain empty.

In our real-time audio streaming system, the audio files which are published into the music feed as entries do not need to enclose the actual music content to be played but only the metadata that is displayed. In fact, the actual music content does not need to be sent at all since the jukebox streams the published audio files in real-time. The jukebox publishes the playlist as an enclosed text file to an entry on the playlist feed. Details of the entries are shown in Table 5.1.
<table>
<thead>
<tr>
<th>ID</th>
<th>Music Entry</th>
<th>Playlist Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>Unique ID</td>
<td>Unique ID</td>
</tr>
<tr>
<td>Summary</td>
<td>Title, artist, album, year, music length, file size, device ID</td>
<td>Playlist version</td>
</tr>
<tr>
<td>Location Info</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Enclosure Flag</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Enclosure Name</td>
<td>N/A</td>
<td>Playlist file name</td>
</tr>
<tr>
<td>Enclosure Type</td>
<td>N/A</td>
<td>Text</td>
</tr>
<tr>
<td>Enclosure Size</td>
<td>N/A</td>
<td>Playlist file size</td>
</tr>
<tr>
<td>Local Uri</td>
<td>N/A</td>
<td>Playlist file path</td>
</tr>
</tbody>
</table>

Table 5.1 Music entry and Playlist entry

5.2.5 Playlist handling

The playlist handler module in the system provides multiple functions as follows:

**Database of the Podnet service**

As mentioned earlier in section 5.2.4, the database of the Podnet service running on the jukebox subsystem is replaced by two entry lists. These entry lists are created and stored in the playlist handler module. Every time a new audio file entry is received or a new playlist entry is published, the database is updated.

**Manager of the displayed playlist**

This module manages the playlist displayed on the jukebox subsystem. When a new audio file entry is received, the module calls the related function in the subsystem to update the playlist. Moreover, in order to prevent the playlist displaying all the songs that have been played, we limit the maximum number of played songs to five when designing the system. The playlist handler module takes care of this and updates the playlist when necessary.

**Publish playlist entries**

The playlist handler module is also responsible for updating the playlist and publishing new playlist entries. The playlist is updated when a streaming of an audio file ends, jukebox switch to the next audio file or a newly published audio file is received.

More improvements of the playlist handler module will be introduced in the future.

5.2.6 Real-time streaming
Real Time Streaming Protocol

The real-time streaming protocol (RTSP) is an application layer protocol which is used to control the delivery of data in real-time. It provides an extensible framework to enable controlled, on-demand delivery of real-time data, such as audio and video [22]. RTSP can help to establish and control multiple media sessions between endpoints. It supports VCR-like commands such as play or pause. However, streaming of data is not a task of RTSP. Usually RTSP works together with the real-time transport protocol (RTP) to achieve delivery of real-time data, and this is also what we have implemented in our system.

RTSP sessions are established and controlled with the commands sent between the RTSP server and the client. A typical command exchange process is illustrated in Figure 5.12.

![Figure 5.12 RTSP commands](image)

When a RTSP client requires a new RTSP session, it sends OPTIONS and
DESCRIBE commands to the RTSP server. The OPTIONS command requests the server to list the RTSP commands it supports whereas the DESCRIBE command requests the server to provide a detailed description for a specific media file. This description is formatted with respect to the session description protocol (SDP). When the description is received, the client sends out a SETUP command to establish a new session with the server and the server replies back with the REPLY command that contains the related RTP information. At this point both the client and the server are ready for real-time streaming. The client can control the session by sending the PLAY and the PAUSE commands. When the streaming is over, a TEARDOWN command is sent to terminate the session. All the RTSP commands are sent using TCP. The default TCP port for the RTSP server is 554 but the user can choose any other free port.

**Real-time Transport Protocol**

The real-time transport protocol (RTP) is widely used in communication and entertainment systems that streams media, such as online video and telephony services [23]. RTP is used in conjunction with the RTP control protocol (RTCP). RTCP monitors RTP transmission statistics, controls QoS and synchronizes multiple streaming sessions. RTP is designed to support a wide range of media formats. It is also extendable to support new formats. The Payload Type field in a RTP packet header indicates the media format of the payload data so that a corresponding decoder can be used to decode the media content. RTP packets are usually sent using UDP since it is more efficient than TCP when real-time applications are considered. In the streaming server implemented in our system, UDP is also chosen to send the RTP packets. The RTP packet header structure is given in Table 5.2.

<table>
<thead>
<tr>
<th>bit offset</th>
<th>0-1</th>
<th>2</th>
<th>3</th>
<th>4-7</th>
<th>8</th>
<th>9-15</th>
<th>16-31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Version</td>
<td>P</td>
<td>X</td>
<td>CC</td>
<td>M</td>
<td>PT</td>
<td>Sequence Number</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Timestamp</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SSRC identifier</td>
</tr>
<tr>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CSRC identifiers...</td>
</tr>
<tr>
<td>96+32×CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Profile-specific extension header ID</td>
</tr>
<tr>
<td>128+32×CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Extension header length</td>
</tr>
</tbody>
</table>

Table 5.2 RTP packet header [23]
The minimum length of a RTP packet header is 12 bytes, which means the SSRC identifier and all the fields before it are mandatory.

- **Version**: Indicates the version of the protocol.
- **P (Padding)**: A flag which is used to indicate whether there are extra bytes at the end of the packet or not. These extra bytes are used to fill up a block of certain size.
- **X (Extension)**: Indicates if there is extension header.
- **CC (CSRC count)**: The number of CSRC identifiers.
- **M (Marker)**: Used at the application level and defined by a profile. If it is set, it means that the current data has some special relevance for the application.
- **PT (Payload Type)**: Indicates the format of the payload. This is defined by a RTP profile.
- **Sequence number**: The sequence number is set randomly for the first RTP packet sent in a session. It is then increased by one after a new packet is sent. Sequence number is used to detect packet loss and restore packet sequence.
- **Timestamp**: Used to enable the receiver to play back the received samples at appropriate time intervals.
- **SSRC**: Synchronization source identifier uniquely identifies the source of a stream. The synchronization source within the same RTP session is unique.
- **CSRC**: Contributing source IDs enumerate contributing sources to a stream which has been generated from multiple sources.
- **Extension header**: The first 32-bit word contains a profile-specific identifier and a length field that indicates the length of the extension (EHL=extension header length) in 32-bit units, excluding the 32 bits of the extension header.

Since the streaming service implemented in our system only concerns single audio streaming from a RTSP server, the size of the RTP packet header is the minimum and no CSRC identifier and extension header are needed.

**RTSP client and server in the system**

The jukebox composes a Linux command to call an instance of the MPlayer when it needs to play an audio file. However, MPlayer does not support RTSP streaming. Hence it needs to be compiled with an appropriate media library that supports real-time streaming. In this work, we compile MPlayer with the Live555 Streaming Media libraries. The Live555 Streaming Media is a set of source-code libraries for standards-based RTP/RTCP/RTSP/SIP multimedia streaming [24]. We implemented and installed a RTSP server for each mobile device to provide the RTSP streaming service. After setting up a RTSP session, the server creates and sends RTP packets to the client. The RTSP server implemented in this work only supports the MP3 audio
file format at the moment. Other formats are considered to be added in the future.

### 5.2.7 Packing a MP3 audio file

The structure of MP3 files is shown in Figure 5.13. A MP3 file mainly contains a series of MP3 frames which store the encoded audio data. Optional ID3 tags, such as music title, artist name and so on, can be added in the beginning of a MP3 file.

Each MP3 frame contains an independent frame header that specifies several significant parameters of this frame. These parameters include MPEG version number, audio sampling frequency, bit rate, MPEG layer number and so on. The size of audio data within a single frame varies according to the bit rate of this frame. However, when the data in a frame is decoded, the playback time is the same. For example, for a layer 3 MP3 file, each frame contains 26 milliseconds of music, regardless of which bit rate is used.

In our system, a MP3 file is parsed during streaming service and every single frame is packed as payload data in a RTP packet and sent from the server to the jukebox. Since each MP3 frame can be independently played, the jukebox can start to play the streamed audio file immediately after receiving the first RTP packet. Moreover, though some of RTP packets may be lost during streaming; this does not affect the playback of other packets.

![Diagram of MP3 file structure](image)

Figure 5.13 Packing a MP3 file into RTP packets

### 5.2.8 Scenarios considered for user experience and performance

In this section we present some of the scenarios that we considered to design a robust system and address challenging use-case issues.

**Scenario A: Jukebox application is turned off or intentionally reset.**

When jukebox application is turned off unexpectedly due to a system crash or selecting the “Turn off Podnet Mode” function in the menu, the streaming stops if there exists one ongoing at the moment. The contact history in the discovery module
of the Podnet service is saved into a text file in order to keep the record of previously contacted devices. The entry lists under the music feed and playlist feed are also saved to a separate file. When Podnet service is turned on again, it loads the system using those files and restores the content history and received audio file entries. The jukebox then plays from the beginning of the last playing song when it was turned off.

If “Reset Podnet” function is selected in the jukebox menu, the Podnet service is stopped and no history record is saved. Moreover, the application deletes any history files that are generated in the past. When Podnet service is started again, it receives all published audio files entries from the mobile devices and creates a new playlist.

**Scenario B: Mobile application is turned off or intentionally reset**

When PodMusic application on a mobile device is turned off, the contact history and the database in the Podnet middleware is automatically saved. The jukebox does not play an audio file from this device if it published one earlier since the RTSP server is also turned off. If an audio file in the playlist cannot be played, the jukebox will automatically skip it to play the next one.

If the Podnet middleware on a mobile device is reset, all the contact history and related data are erased. When the application is started again, it immediately receives the playlist update as it is considered as new content for the device. If an audio file is previously published by a recently reset mobile device, it is still streamed if the mobile device is on and within communication range. However, the streamed audio file is not displayed on the published list since it is reset.

**5.3 Implementation**

All modules in this system are implemented using Java. The mobile application the Podnet middleware is supported by the Android libraries. Implementation details of some functions of our system are given in this section.

**5.3.1 Publishing a new audio file**

Three steps of operations are needed to successfully publish a new audio file. First, the user opens the system menu in the PodMusic application and selects “Add Music”. Then application launches a new screen which shows all the audio files stored in the mobile device. The user chooses one of the files and this is created as a new entry and published.

The pseudo code for publishing a new audio file is given below.

```
@Pseudo code 1: Method in PodMusic which is used to publish new music entry
1 publishNewMusicEntry (Intent intent) {
2     title = intent.getExtras().getString("title")
3     duration = intent.getExtras().getString("duration")
4     size = intent.getExtras().getString("size")
5     file name = intent.getExtras().getString("display")
6     artist = intent.getExtras().getString("artist")
```
album = intent.getExtras().getString("album")
year = intent.getExtras().getString("year")
entryId = Device name + Date
Entry entry = new Entry (entryId)
entry.setAuthor (Device name)
ip = getLocalIPAdress();
replace any ";" with " " in title, file name, artist, album
entry.setSummary
  (title+";"+duration+";"+size+";"+file name+";"+artist+";"+album+";"+year+";"+ip+";")
publish (entry, Music Feed)

5.3.2 Receiving a published audio file

The newly published audio files are received by the jukebox which then calls a method to handle them. After the metadata of the published audio file is downloaded, the application calls the playlist handler to update the playlist. The related pseudo codes are given below.

Pseudo code 2: Method in Podnet service in Jajuk which is used to handle newly received music entry

```java
receiveEntry (ReplyMessage reply) {
  if (reply.getStatus != OK) then return end if
  entry = reply.getEntry()
  if (entry has enclosure) then return end if //audio file entry does not have enclosure
  if (reply.getFeedID() is not the feed we want) then return end if
  if (entry.getID() is not the entry ID we want) then return end if
  if (the database has this entry already) then return end if
  if (feed ID is music feed) then
    PodnetPlaylistControl.addEntry(entry) //add this entry to the database
    ifChanged = true
  end if
}
```

Pseudo code 3: PlaylistControl-addEntry method in the jukebox that adds a new entry to the database

```java
PodnetPlaylistControl.addEntry (Entry e) {
  receivedMusicEntries.add(e) //add the new entry to the entry list
}
```

Pseudo code 4: Synchronization method in the jukebox which is called when downloading of all
new entries from a mobile device finishes

```java
doneSynchronizing (ReplyMessage msg) {
    peerID = msg.getNodeID()
    contentRevision = msg.getContentRevision()
    //update the contact history of this mobile device
    contactHistory.updateCache(peerID, contentRevision, 1)
    if (ifChanged) then
        //if there is any new entry, the playlist should be updated
        PodnetPlaylistControl.update(false)
        ifChanged = false
    end if
}
```

### 5.3.3 Updating the playlist

Maintaining and updating the playlist is the main task of the playlist handler module. Every time a new audio file metadata is downloaded or the playlist status in the jukebox is changed, the handler updates the playlist. Figure 5.14 illustrates how the playlist is created.

![Figure 5.14 Creating playlist from music entry list](image)

There are four parameters that need to be updated when the playlist is updated. The parameter of `firstEntryToDisplay` indicates the index of the first entry shown in the playlist displayed in the jukebox; `historyToDisplay` defines the maximum number of played songs displayed in the playlist. This parameter is set to 5 by default. The parameter of `indexOfPlayingEntry` indicates the index of the entry in the displayed
playlist. Note that this index is the index in the playlist but not in the music entry list. The parameter of oldEntryListSize is the one that indicates the size of the music entry list. This size should be updated every time a new entry is added.

Pseudo code 5: PlaylistControl-update method in jukebox which is called when the playlist is needed to be updated.

```java
PodnetPlaylistControl.update (boolean uiCalls) {
    if (uiCalls) {  //called by Jajuk, a song is finished or manually stopped
        index = index of currently played song in the displayed playlist in the jukebox
        if (final queue size is 0) then firstEntryToDisplay = oldEntryListSize
        else if (index > historyToDisplay) then
            remove elements in displayed playlist, from 0 to index-historyToDisplay-1
            firstEntryToDisplay += index – historyToDisplay
        end if
    else  // called because of newly downloaded entries
        if (receivedMusicEntries.size() > oldEntryListSize) then
            for each index between oldEntryListSize and receivedMusicEntries.size() do
                //convert the entry into a element which is recognizable by Jajuk
                StackItem s = new StackItem(createFileObject (receivedMusicEntries.get(index)))
                if (final queue size > 0) then
                    if (the player is stopped) then
                        add s to the displayed playlist and start the player
                    else
                        add s to the displayed playlist
                    end if
                else
                    add s to the displayed playlist and it will be started
                end if
            end for
            oldEntryListSize = receivedMusicEntries.size();  //update the parameter
        end if
    end if
    if (the player is not stopped) then
        indexOfPlayingEntry = index of currently played song in the displayed playlist
    else
        indexOfPlayingEntry = -1
    end if
    //publish this updated playlist
    publishNewPlaylist()
    // the playlist is updated, we have to increase the revision number
    increasePlaylistRevision()
}
```
Pseudo code 6: PlaylistControl-publishNewPlaylist method which is called when the playlist is needed to be published

```java
PodnetPlaylistControl.publishNewPlaylist () {
    entryId = DEVICEID + Date // DEVICEID = “Jajuk”
    entry = new Entry(entryId)
    entry.setAuthor (“laptop”)
    entry.setEnclosure (playlist revision number)
    rewritePlaylistFile() // update the playlist file which is used as the enclosure
    set enclosure information
    playlistEntries.add(entry) // new entry is published
}
```

Pseudo code 7: PlaylistControl-rewritePlaylistFile that updates the playlist file

```java
PodnetPlaylistControl.rewritePlaylistFile () {
    if (the playlist file exists) then
        delete the existed playlist file
    end if
    playlistFile = create a new empty file
    fw = FileWriter(playlistFile)
    bw = BufferedWriter(fw)
    //attach a time tag to compare which playlist file is newer
    bw.write (the number of milliseconds that have been passed since 1970+”;“)
    bw.newLine()
    //write the parameter of indexOfPlayingEntry, the mobile device will need it to find
    //where the “PLAYING” tag should be put
    bw.write(indexOfPlayingEntry+”;“)
    for each index between firstEntryToDisplay and receivedMusicEntries.size() do
        bw.newLine()
        bw.write(receivedMusicEntries.get(index).getSummary())
    end for
    bw.close()
    fw.close()
}
```

5.3.4 Synchronization of the playlist

After a new playlist entry is published by the Podnet service in the jukebox, all mobile devices update their local copy of the playlist once they receive it. The new playlist is not saved as a new file but replace the old one instead. When the playlist is updated multiple times in a very short time period, multiple playlist are published by the jukebox. If the mobile devices receive these updates not in chronological order, a newer playlist can be updated by an old one. The PodMusic application takes care of this problem by attached time tags to find out which playlist is newer as this should be
displayed. Implementation details are shown in pseudo code 8.

### Pseudo code 8: Process Published Playlist method to update the playlist to be displayed

```java
processPublishedPlaylist () {
    downloadPlaylistFile = Downloaded playlist file
    localPlaylistFile = Local playlist file
    fr1 = FileReader(downloadPlaylistFile)
    br1 = BufferedReader (fr1)
    fr2 = FileReader(localPlaylistFile)
    br2 = BufferedReader (fr2)
    downloadTime = br1.read(the time tag in the file)
    localTime = br2.read(the time tag in the file)
    if (downloadTime > localTime) then
        localPlaylistFile.delete()
        downloadPlaylistFile.renameTo(localPlaylistFile)
        return
    else
        return
    end if
}
```

5.3.5 Saving and loading history

If the jukebox is turned off when the system is running, it saves the contact history and the Podnet database, which includes those four parameters of the playlist, the whole audio file entry list and the latest playlist version number. When the jukebox is restarted, it reloads the state information from memory. Pseudo codes 9 and 10 present the implementation of saving and loading the database. The operations for the contact history are similar and thus not discussed here.

### Pseudo code 9: SaveInfo method to save the related database of Podnet service in the jukebox

```java
saveInfo () {
    if (infoFile exists) then
        delete existed file
    end if
    infoFile = create new empty file
    fw = FileWriter (infoFile)
    bw = BufferedWriter (fw)
    //write the playlist parameters in the first line
    bw.write(firstEntryToDisplay+";"+historyToDisplay+";"+indexOfPlayingEntry+";"+
            oldEntryListSize+";")
    bw.newLine()
}
```
// write the playlist revision number and the newest playlist entry ID in the second line
bw.write(playlist revision number + ";" + playlistEntries.get(playlistEntry.size()).getID + ";")

bw.newLine()
for each entry ∈ receivedMusicEntries do
    bw.write (entry ID + ";" + author + ";" + latitude + ";" + longitude + ";")
    bw.newLine()
    bw.write(summary)
    bw.newLine()
end for
bw.close()
fw.close()
}

---

**Pseudo code 10: LoadInfo method that loads the database of the Podnet service in jukebox**

```java
loadInfo () {
    if (infoFile not exists) then
        return
    end if
    receivedMusicEntries.clear()
    playlistEntries.clear()
    fr = FileReader(infoFile)
    br = BufferedReader(fr)
    line = br.readLine()
    restore the playlist parameters from line
    line = br.readLine()
    restore playlist revision number and entry ID from line
    entry = new Entry(entry ID)
    entry.setAuthor("laptop")
    entry.setSummary(playlist revision number)
    set enclosure information
    // republish the new playlist entry, only the newest one is restored
    playlistEntries.add(entry)
    while (line = br.readLine() != null) do
        restore the music entry’s ID, author, latitude, longitude from line
        br.readLine()
        restore entry summary from line
        receivedMusicEntries.add(restored entry)
    end while
    for each index between firstEntryToDisplay and receivedEntries.size() do
        StackItem s = new StackItem(createFileObject(receivedMusicEntries.get(index)))
        add s to the final queue
    end for
    set the final queue to play the song at indexOfPlayingEntry
```
5.3.6 RTSP server

The RTSP server can be seen as a separate module in the PodMusic application. Once it is started, it runs separately from other modules in the PodMusic application. When PodMusic is started, it opens a server listener socket on port 1234. The jukebox sends a request to this socket to stream an audio file. Every time the listener socket receives a request from the jukebox, it opens a new RTSP service thread and handles the RTSP commands received from the jukebox. The jukebox composes a Linux command line to call MPlayer to act as the RTSP client. This Linux command line is composed as shown in the example below:

```
  mplayer rtsp://(IP address):1234//(audio file title)
```

Usually the path to the audio file to be streamed is inserted at the end of the command line. However, we use the title of the audio file instead. This is because the file path on the device may be changed after the audio file is published. If the command still contains the old file path, the file cannot be found by the server. The title of the audio file however does not change even when the file is moved to new location on the mobile device. The server can still find the file via the Android media content provider. Pseudo code 11 shows the main method in a RTSP service thread.

<table>
<thead>
<tr>
<th>Pseudo code 11: Main method in a RTSP service thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 run () {</td>
</tr>
<tr>
<td>2 state = INIT</td>
</tr>
<tr>
<td>3 while (true) do</td>
</tr>
<tr>
<td>4 request_type = parse_RTSP_request() //blocking, waiting for commands from client</td>
</tr>
<tr>
<td>5 if (request_type is OPTIONS) then</td>
</tr>
<tr>
<td>6 send_RTSP_reponse_OPTIONS()</td>
</tr>
<tr>
<td>7 else if (request_type is DESCRIBE) then</td>
</tr>
<tr>
<td>8 send_RTSP_reponse_DESCRIBE()</td>
</tr>
<tr>
<td>9 else if (request_type is SETUP) then</td>
</tr>
<tr>
<td>10 rtpSocket = create new datagram socket</td>
</tr>
<tr>
<td>11 rtpLocalPort = rtpSocket.getLocalPort()</td>
</tr>
<tr>
<td>12 rtcpSocket = create new datagram socket</td>
</tr>
<tr>
<td>13 rtcpLocalPort = rtcpSocket.getLocalPort()</td>
</tr>
<tr>
<td>14 send_RTSP_reponse_SETUP()</td>
</tr>
<tr>
<td>15 state = READY</td>
</tr>
<tr>
<td>16 break</td>
</tr>
<tr>
<td>17 else</td>
</tr>
<tr>
<td>18 return</td>
</tr>
</tbody>
</table>


end if
end while // RTSP session is established
mRTPSource = create new MP3RTPSource thread with rtpSocket, audioFileName, IP addresses of server and client
mRTCP = create new RTCPRender thread with rtcpSocket
while (true) do
  request_type = parse_RTSP_request() // blocking, waiting for commands from client
  if (request_type is PLAY and state is READY) then
    Seq = mRTPSource.getSeq() // RTP sequence number
    time = mRTPSource.getTimeStamp() // RTP timestamp
    mRTPSource.ifPause = false
    send_RTSP_response_PLAY(seq, time)
    if (mRTCP is not running) then
      mRTCP.start()
    end if
    if (mRTPSource is not running)
      mRTPSource.start()
    end if
    state = PLAYING
    Vibrate the device for 500 milliseconds
  else if (request_type is PAUSE and state == PLAYING) then
    mRTPSource.ifPause = true
    send_RTSP_response_PAUSE()
    state = READY
  else if (request_type is TEARDOWN) then
    mRTPSource.ifStop = true
    mRTCP.ifStop = true
    mRTCP.RTCPocket.close() // interrupt to let it be closed
    send_RTSP_response_TEARDOWN()
    rtspSocket.close() // close this RTSP socket, ready to finish the thread
    break
  end if
end while

5.4 System test
The system is tested with a laptop and two Android smart phones. We have tested all of the system features, such as publishing audio files, synchronization of playlists and real-time streaming. Several challenging scenarios are also tested in order to judge the system’s robustness. According to our experience with running the system, the mobile device can easily publish audio files and the information is correctly exchanged between the jukebox and mobile devices. A synchronized playlist is successfully generated and updated continuously, as expected. In the challenging scenarios that are discussed in section 5.2.8, the system can handle those situations as designed, and the
system robustness is clearly demonstrated. However, a few problems still occur when the system is tested:

- When the jukebox has finished playing a song, it usually takes 5 to 10 seconds before the next song starts. This time gap is caused by sending and processing RTSP messages, which is necessary before an audio file can be streamed.

- When the mobile device is in a bad wireless environment, for example, at the end of its communication range, the connection between the jukebox and the device is not stable. The streaming quality is severely lowered under this situation and the music is played brokenly.

These problems influence the user experience, but they do not affect the main features of the system. The problems will be solved in future work as the system will be continuously updated.
6. Conclusion and Future Work

6.1 Conclusion

In this thesis, we studied mobile ad-hoc networks and publish/subscribe systems. Mobile ad-hoc networks allow devices to communicate when the network topology is dynamic and there is no fixed infrastructure. Publish/subscribe system is a content-centric approach developed for wired networks, yet it can also be applied in the wireless ad-hoc network domain to disseminate contents. Podnet is a middleware architecture which provides content dissemination service in mobile P2P opportunistic networks. It allows user devices to discover each other when they are within communication range and exchange their contents with respect to users’ interests.

We designed a remote collective music sharing system which utilizes the Podnet middleware. The system allows users to submit audio file metadata to a jukebox within communication range. Then the affiliated audio files, residing on users’ phones, are played in turn via real time streaming requested by the jukebox. Our system has two components: a jukebox application which can be run on a laptop/desktop and a mobile application that can be run on a set of Android smart phones. Both applications are designed and implemented to utilize the Podnet middleware. A synchronized playlist is provided to all participating devices to keep the users updated when the system is running.

In order to achieve real-time streaming of audio files, a RTSP streaming server is implemented on mobile devices. The server opens RTSP sessions to the jukebox and streams audio data using the RTP protocol. The server supports most of the RTSP commands, but currently only streaming of audio files in MP3 format is implemented.

6.2 Future work

Improve the performance of streaming service

Currently, there is a time gap between the songs that are played by the jukebox due to the delay caused by the setting up process before a new streaming session starts. It would be better if multiple threads can be implemented to solve this performance issue. When a currently streaming song is about to end, a new thread can be created in the jukebox to set up the streaming session for the next song in line. Hence, it can immediately be launched when the previous one is finished. Moreover, a more detailed analysis of MPlayer is needed to improve the streaming service quality when mobile devices occasionally lose connection for a short period of time due to either poor wireless environment or mobility.

Extend the supported audio formats

The RTSP server implemented in the mobile application only supports streaming of MP3 files at the moment. However, this server is designed to be extendable and thus
can support a large family of audio formats. Further inquiry of RTP standards for other audio formats is required.

**Develop similar applications inspired by the presented system**

A collaborative real time DJing application or a conference Q&A system, with which users can use their smart devices to ask questions via real time streaming to the public speaker system that runs the application, can be designed and implemented using a similar architecture.
7. References


