Vehicle-based Disconnected Data Distribution

Aditya Singhania
San Jose State University

Vehicle-based Disconnected Data Distribution

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Aditya Singhania

May 2023

The Designated Project Committee Approves the Project Titled

Vehicle-based Disconnected Data Distribution

by

Aditya Singhania

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

ABSTRACT

Vehicle-based Disconnected Data Distribution

by Aditya Singhania

The world today is highly connected and there is an immense dependency on this connectivity to accomplish basic everyday tasks. However much of the world lacks connectivity. Even in well-connected locations, natural disasters can cause infrastructure disruption. To combat these situations, Delay Tolerant Networks (DTNs) employ to store and forward techniques along with intermittently connected transports to provide data connectivity. DTNs focus on intermittently connected networks however what if the regions are never connected? For example, Region A - is never connected to the internet, and Region B – has internet connectivity. Using a vehicle that travels between the two regions it is possible to transport the data from A to B and vice versa. Current vehicular-based DTN approaches make use of dedicated hardware infrastructures mounted on vehicles. Our research focuses to shift all this computing to personal smartphone devices. The users in disconnected areas will install the Disconnect Data Distribution(DDD Android application on their phones. This application will receive data from various applications on their phone via inter-process communication(IPC) and package the data into bundles using a DDD library. Vendors, transportation operators, delivery people, and others that regularly travel between connected and dis-connected areas can download the Bundle Transport Application on their Android phones. These devices will then communicate with each other using short-range wireless technology like WiFi-Direct in the disconnected region. The device on the transport can communicate with the Servers over the internet. For this project, we should assume that the Client and Server can provide data as desired and our focus is on the Transport.

ACKNOWLEDGMENTS

It is with profound gratitude that I express my appreciation to my esteemed thesis advisor, Dr. Ben Reed, for their invaluable guidance, unwavering support, and boundless patience throughout my research endeavors. Their exceptional expertise, insightful perspectives, and constructive feedback have been pivotal in shaping my research objectives and elevating the quality of my work. It has been an extraordinary privilege to have benefited from the mentorship of such a distinguished academic, and I shall always remain deeply grateful for their invaluable guidance.

I would also like to sincerely thank the esteemed members of my thesis committee, Dr. Thomas Austin and Mr. David Bui, for their discerning input and invaluable contributions to my research. Their erudite advice and rigorous critique have been instrumental in refining my scholarly pursuits and raising the caliber of my work. I am deeply grateful for their constant support and encouragement throughout my thesis journey.

Furthermore, I wish to express my heartfelt gratitude to the members of the DDD Lab for their generous support and collaborative efforts during my research. Shashank Hegde, Deepak Munagala, Abhishek Gaikwad, and Anirudh Kariyatil Chandakara have fostered a stimulating and supportive research milieu, and I have had the privilege of learning significantly from their domain expertise.

Lastly, I would like to extend my sincere appreciation to my family and friends for their unwavering support and encouragement throughout my academic journey. Their love, counsel, and empathy have been the driving force behind my achievements.

I am immensely grateful to all those who have rendered invaluable support and encouragement throughout my thesis journey, making this endeavor possible.

TABLE OF CONTENTS

CHAPTER 1

Introduction

The world today is highly connected and there is an immense dependency on this connectivity to accomplish basic everyday tasks. People can easily connect with each other over instant messaging, There is no need to carry multiple physical sources of information for example maps, videos, and information can be all streamed and downloaded as required. Digital connectivity fulfills social, economic, and even emotional needs. It has become the primary form of information dissemination. As such society has integrated connectivity into daily life and those who lack it are at a disadvantage. An article by International Monetary Fund talks about low internet connectivity leading to widening income inequality both within and between countries and less than half the population in developing countries has internet access [1]. Another report by United Nations International Children's Emergency Fund claimed that two-thirds of the world's children lack internet access and this negatively impacts their learning capability and access to education [2]. Over 15% of household lack access even in the United States [3].

There has been much research in trying to bring network connectivity to disconnected areas. Some of the approaches are using unmanned aerial vehicles (UAV), altitude platforms (AP), and low-Earth orbit (LEO) satellites [4]. However, all these involve technical capability, are costly to maintain, and are not feasible in all areas. Another approach, Delay Tolerant Networking(DTN) [5] has shown promise [6] by employing employ store and forward techniques along with intermittently connected transports to provide data connectivity. When we think about disconnected areas they are usually remote villages and areas with poor infrastructure.

Current approaches for DTN-based remote village networks involve transferring data to some customized hardware like a USB drive[7] or WiFi Radio transceivers[6]

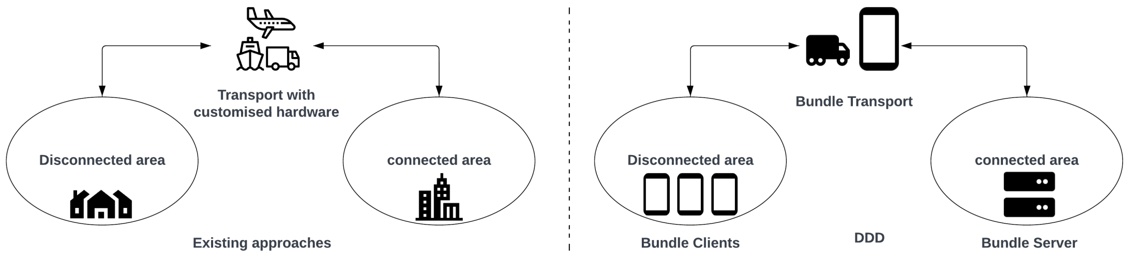


Figure 1: Overall System Architecture

and then carrying the data on a physical transport to a connected region where it can upload and download data to and from the internet or a central hub. Based on these approaches our infrastructure Disconnected Data Distribution (DDD) will use existing transportation infrastructure like delivery people, vendors, etc. that regularly travel between connected and disconnected regions. We want our infrastructure to be cheap, and simple to deploy therefore we aim to make our solution a software-focused approach that eliminates the need for customized hardware on transports. For this purpose, we make use of Android phones without any modifications. Although DDD is a form of Delay Tolerant Networking (DTN), we show how our internet data distribution approach simplifies the DTN architecture while providing stronger security guarantees compared to standard DTN approaches.

We call units of data as Bundles. Transport users can install a Bundle Transport Android application for the Play Store and require no further technical knowledge. They may need to acquire a micro SD card to enable their device to handle the extra storage requirements. The transport application users might have incentives in the form of attracting people to use their DDD-enabled services for example a bus driver attracting more riders. The users in the disconnected areas will require to download a Bundle Client Android application from the Play Store. The Bundle Server is fixed and on the internet. This infrastructure leverages the client-server architecture that most modern applications use today. A few desirable properties of the infrastructure are

1. Software only Approach - This avoids the deployment of customized hardware and thus there are no specialized maintenance issues.

2. Ease of integration with existing internet services -Existing applications like messaging, videos, etc. Should be able to integrate without the need for a lot of changes

3. Untrusted infrastructure - The transport infrastructure is untrusted and is unable to see the source and destination of the data. It is assumed that the data can be altered, intercepted, and collected by malicious agents. Anyone can act as the transport without affecting the security of the network.

As the Bundle Transport needs to communicate with the Bundle Clients without using the internet we look at short-range wireless methods using built-in WiFi Direct functionality. We prefer WiFi-Direct as multiple clients can connect wirelessly and transfer data simultaneously.

CHAPTER 2

Related Work

Initial research in the area of connecting rural and less connected areas aimed to increase the coverage of existing services without the need for new terrestrial towers. These options, therefore, provided highly reliable data rate services. Stratospheric Communication Platforms [8] [9] is a high-altitude platform approach that makes use of large high-altitude balloons to float platforms above the earth's weather and commercial air traffic. These platforms are equipped with necessary hardware like antennas that can beam signals to large areas below. The balloons can remain in the air for several months at a time. Another alternative to this approach is low-Earth orbit(LEO) satellites [10] which are smaller satellites that orbit much closer to the Earth when compared to traditional geostationary satellites. These satellites can communicate with ground-based terminals such as antennas placed in rural areas. These satellites are usually deployed in constellations consisting of a few hundred to a few thousand satellites. Starlink [11] is one of the recent satellite constellations in use to provide broadband internet access. However, a major limitation with these satellites is that the antenna should be pointing towards the satellite without any obstructions, Starlink tries to overcome this issue by introducing self-aligning antennas which align automatically based on satellite orientation [12]. However, these could never become popular because they demanded special maintenance and high setup costs. This prompted a shift towards a cheaper solution at the cost of increased delay and more delay-tolerant infrastructure.

In [6] A. Pentland, R. Fletcher, and A. Hasson, introduce DakNet which enables low-cost digital communication by transmitting data over short distances between kiosks located across villages using portable storage devices called Mobile Access Points(MAPs). These MAPs are powered by and mounted on buses, motorcycles, bicycles, etc., and make use of cheap WiFi radio transceivers for data transfer. The MAP, when in range, automatically connects to the kiosk and internet access points and performs a high bandwidth data synchronization thereby providing a high data throughput using a store and forward delay tolerant infrastructure. While the authors claim that MAPs are low-cost however they do involve customization and effort to set up the network. DDD aims to eliminate this customization by making using of already available Android Phones and shifting the focus to a software-based solution for village networks.

In [13] S. Guo et al introduce the KioskNet system which builds on the ideas of DakNet. In KioskNet common access points using shared terminals at kiosks in rural areas are the entry points for users. The kiosk is equipped with customized kiosk controller hardware and terminals which are made up of reused computer parts. While kiosks can communicate via a variety of connectivity options like long-range WiFi, towers, satellites, etc. KioskNet focuses on ferries that like MAPs in DakNet also require customized hardware in the form of a single-board computer with around 20-40GB of storage. The authors noted that they were able to rapidly deploy the system in 2 days in a rural area. The authors also note in the paper that it was a problem for them to convince drivers to have these single-board computers mounted below their dashboard which was one of the major motivations for transitioning to V-link [14]. The authors also note that the individual components of the network are cheap. However, the collective cost of setting up the network while cheaper is still high. All of these issues can be tackled by leveraging advancements in mobile technology which already has storage available and onboard hardware for creating access points. Since almost everyone travels with a smartphone these days there will be no convincing required.

The Wizzy Digital Courier project[7] a simple delay-tolerant approach to provide internet access to schools in Africa with no telephone lines involves using a USB stick to physically transport the data. The school composes an email message with topics for an internet search which is loaded onto a USB stick and is then carried by someone to a central computer that could be miles away. This person uploads the batch of queries and downloads the results of the previous queries and travels back. Thus the school now has access to data on the internet with a one-day delay. The project uses software that automates the process of reading queries off the USB Stick and transferring results to the USB. V-link [14] like the Wizzy Project also added USB stick functionality known as Key-Link and SMS functionality known as SMS-Link. V-link focused more on a software approach by reducing the dependence on customized hardware a problem faced by KioskNet. In V-link users just need to install the software and plugging in the USB stick is all that is required, the emails are automatically picked up and once this USB is connected to the central terminal it automatically sends the emails as intended. It can be noted that the USB may travel to multiple user terminals before connecting to the central terminal and the network is also tolerant of it. For small urgent messages, a mobile device connected to the central computer can instantly communicate with a mobile device connected to the user terminal and enable fast communication.

In [15] J.Burges et al shift focus towards a more software-focused approach to vehicle-based disruption-tolerant networks. They propose a new routing algorithm, MaxProp which aims to provide efficient and reliable communication in scenarios where the infrastructure is unreliable. The algorithm uses probabilistic techniques to determine the most efficient route for data transmission. The authors created the DieselNet test bed in order to demonstrate the efficacy of the algorithm. The test bed comprises of a network of 40 public transportation buses equipped with a Linux “brick” computer; USB 802.11b adapter, 802.11b AP, GPS receiver, and 40GB hard drive. All of this hardware is now available in Android phones in one way or another. Buses transfer data as they pass by each other and via available hot spots. MaxProp involves each node calculating the probability of its neighbors being the next hop in the network. It then selects the node with the highest probability, in a way it involves probabilistic flooding of data in the network. The authors go on to show that customized hardware is not necessary and that we can create disruption-based networks on the hardware of similar capabilities.

Based on the literature above [6, 13, 15] we need hardware capable of forming peer-to-peer connections and for intercommunication of data and around 30-40GB of storage to employ store and forward techniques all these requirements are satisfied by smartphones specifically Android phones using WiFi-Direct [16] and micro-SD cards.

2.1 Short-Range Wireless Technologies

Some commonly used short-range wireless technologies are Bluetooth [17], WiFi [18], and near-field communication(NFC) [19] implemented by RF circuit chips. NFC works on the principles of radio-frequency identification(RF-ID) technology and standards. It enables 2 electronic devices to communicate provided the distance is within 4 centimeters. NFC is bi-directional depending on the hardware and provides a low-frequency low-speed connection. NFC has been successfully deployed to mobile devices and current NFC applications include contactless transactions and initial data transfer for setting up more complex communication systems. One of the major advantages of NFC is that it provides for communication with unpowered equipment like cards, fobs, etc. by setting up an RF field that can power the passive devices. It also enables peer-to-peer communication in the case of powered-up devices. These properties make NFC a viable but less suitable solution for mobile devices to communicate in a disconnected setting. Bluetooth employs ultra-high frequency radio waves and is used for data transfer over short distances, with low bandwidth and low energy consumption. Bluetooth works on the data packet model and is best suited to transfer small quantities of data. Apple Inc.[20] has been doing some great research in Bluetooth low latency however it s hardware-specific and Apple hardware is expensive. Unlike NFC, Bluetooth does not require devices to be in the line of sight of each other. Bluetooth is still not suitable for transferring large files as it is slow and better alternatives exist if it is just the bi-directional packet transfer that we desire.

2.2 WiFi-Direct

WiFi-Direct technology makes use of the same WiFi radio frequencies and protocols used in traditional WiFi, however, it bypasses the need for access points and enables direct device-to-device communication. In traditional WiFi, a connection means a link between the source point and the destination point which is centrally controlled. Devices make use of access point terminals for example a WiFi router as a data exchange hub. In this case, each access point is also limited in the number of clients. Unlike this WiFi-Direct relies on peer-to-peer communication, which can be established when two devices need to discover each other which is a similar process to Bluetooth communication. The process is called Service Discovery and devices exchange Service Discovery request and response packets which help the devices determine if they can communicate directly with each other. After discovery, the devices establish a connection using WiFi Protected Setup (WPS) which is a standard method for connecting wireless devices. The two devices then exchange WPS Information Elements (IEs), which contain information about the network settings and security parameters. This information is used to establish a WiFi Direct connection allowing the devices to communicate using standard WiFi protocols without the need for access points. All Android devices support WiFi Direct starting version 4.0. Android provides a P2P framework [21] for managing WiFi Direct connections using APIs that can be used for service discovery, making connections, and building applications. Some added advantages are support for WPA2 encryption, a Service broadcasting mechanism for service discovery by potential peers, and dynamic group owner allocation based on device power management. WiFi Direct networks classify devices as Groups Owners or Peers. Group Owners are devices that create the network and Peers are other devices that connect to the network. The Group owner is responsible for managing the network. It assigns the IP address to each of the peers and acts as the central hub for communication. This means that if two peers want to exchange packets they must do so via the group owner. WiFi-Direct would be highly suitable for communication between Android devices in a disconnected setting because of its security feature, range, and speed. As seen in Fig. 2 the Transport applications will act as the group owners ensuring that no two peer devices can communicate directly and multiple peer devices can connect to the transport device and exchange data bundles.

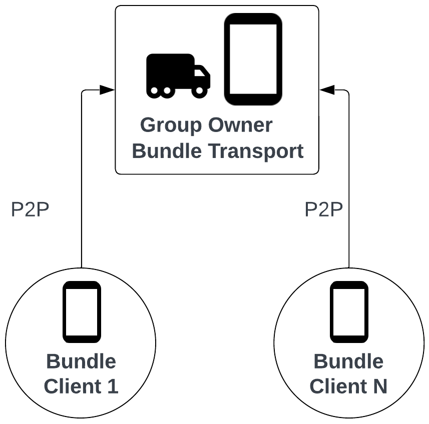


Figure 2: WiFi-Direct Infrastructure

Table 1 shows a quick performance comparison of different short-range wireless communication techniques [22]. As we can see WiFi-Direct provides the best speeds and range and is best suitable for less frequent data transfers as it consumes higher energy. This is ideal in case of delayed settings where we are expecting a large volume of data and longer intervals.

	NFC	Bluetooth	WiFi Direct
Speed	424 kbps	1-3 Mbps	$<$ 250 Mbps
Range	$<$10cm	$<$30m	46-100 m
Frequency	13.56 MHz	2.4 GHz	2.4 or 5.0 GHz
Energy Consumption	$<$200 mW	$<$1.0 W	2 - 20 W

Table 1: Performance comparison of different short-range wireless techniques [22]

2.3 gRPC

gRPC [23] is an open-source Remote Procedure Call (RPC) framework developed by Google. RPCs are a way of allowing machines to call subroutines on a remote machine while abstracting the details of how it is called and behaving as if they were executing a sub-routine call on the local machine. gRPC is a platform-agnostic and language-agnostic binary protocol. gRPC uses Protocol Buffers for defining the RPCs interface and serializing data between clients and servers. The clients and servers can be written in different languages. Several languages are supported including Java, C++, Go, Ruby, JavaScript, etc. gRPC supports four different kinds of RPC [] Unary RPCs where the client sends a single request and the server has a single response, Server Streaming RPCs where the client sends a single request and the server responds with a stream of messages, Client Streaming where the clients send a stream of messages to the server and lastly Bidirectional Streaming where streams of messages are exchanged between both client and server. Each type has different use cases. To stream data packets we will require to divide the data into chunks and we can use client-side streaming and server-side streaming for uploading and downloading files respectively. Fig. 3 shows a logical representation of what a file download or upload would look like.

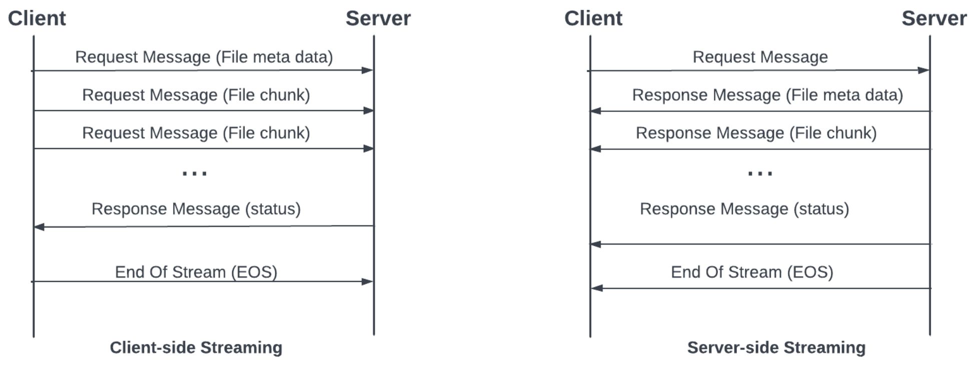

Client
Server
Request Message (File meta data)
Request Message (File chunk)
Request Message (File chunk)
...
Response Message (status)
End Of Stream (EOS)
Client-side Streaming
Client
Server
Request Message
Response Message (File meta data)
Response Message (File chunk)
...
Response Message (status)
End Of Stream (EOS)
Server-side Streaming

Figure 3: gRPC Streaming Example

For downloading a file in chunks the client would send a single request for a file and the server would respond with a stream of file chunks. Similarly for uploading a file, the client would stream multiple chunks and the server would respond with a single response.

CHAPTER 3

System Design

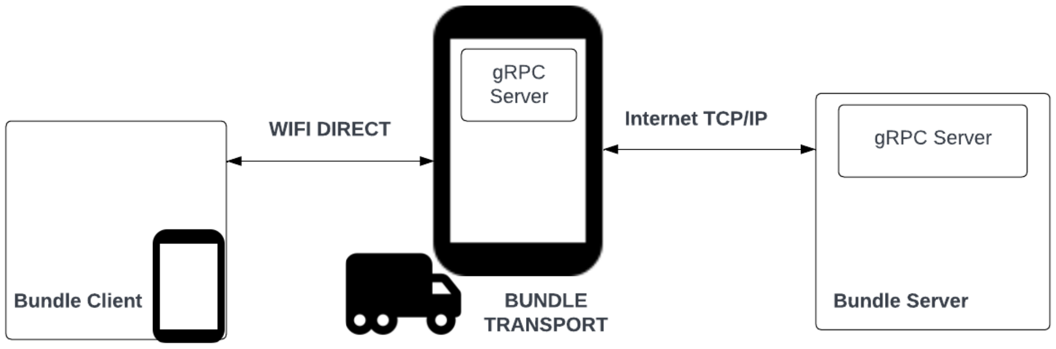


Figure 4: Overall System Architecture

The users in disconnected areas will install the DDD Android application on their phones. This application will receive data from various applications on their phone via inter-process communication(IPC) and package the data into bundles using the DDD library. Vendors, transportation operators, delivery people, and others that regularly travel between connected and dis-connected areas can download the Bundle Transport Application on their Android phones.

As we want the whole infrastructure easy to deploy and sustainable. All of it is built on a normal Android phone with no modifications to its operating system.

Fig. 4 shows the Bundle Client application which will be running on the phones of users in the disconnected region and will be responsible for communication between various applications which involves collecting data through the Android inter-process communication and packaging it into bundles. The Bundle Client application is also responsible for unpacking the bundles received via transport from the server and sending the data back to the application.

Fig. 4 shows the Bundle Transport application which resides on the data transport and physically moves the bundle between the connected and disconnected regions. It leverages WiFi-Direct to exchange data in the disconnected region.

Fig. 4 shows that in the connected region, a Bundle Server receives the bundles from the Bundle Transport over an internet connection such as mobile data or WiFi, unpacks them, and then forwards the application-specific data to the corresponding application servers. The Bundle server also packages data into bundles from the various applications and is responsible for routing the bundles through the correct transport to the clients.

For this project, we should assume that the Bundle Client and Bundle Server can provide bundles as desired and our focus is on the Bundle Transport. The Bundle transport is untrusted and works under the assumption that the data it carries can be altered, fabricated, replayed, and intercepted by malicious actors. The data transport cannot identify the source or destination of the data they are transporting to. Similarly, Bundle Clients do not need to authenticate data transports because data bundles themselves are opaque to the transport i.e. transports cannot differentiate valid bundles from random data as a result of end-to-end encryption using signatures and public keys by the Bundle Client and Bundle Server. Bundles are also incremental units of data and will grow in size until an acknowledgment is received from the server via the transport. We will see later that this property influences how Bundle Transport interacts with the Bundle Client and the Bundle Server. The identifier for a bundle or the Bundle Id is a pseudo-random unique combination of the client Id and the bundle counter. It is unique across all clients.

The Bundle transport has limited storage, and the bundle will also have some imposed maximum size to prevent the bundle from growing infinitely. We can expect the application to use at least 20GB of phone storage or an SD card storage chosen by the application user. The application will also use scoped storage [24] introduced in

Android 11 which prevents other applications from accessing application-specific data.

The Bundle Transport also generates a public/private key pair which will serve as a self-certified identifier when communicating with the Bundle Server. This id is also required for the Bundle Server to maintain a routing table and route bundles efficiently. While a malicious Bundle Transport may use this to delay data delivery, it will not cause the Bundle Client or Server to accept modified data.

Android devices have constrained storage when compared to dedicated hardware and it is crucial to delete inessential and redundant bundles for optimal space usage. The Bundle Transport deletes bundles received from a Bundle Client once they have been delivered to the Bundle Server. The bundles destined for the Clients are not deleted unless instructed by the Server. This is because the Transport does not know the destination Clients for the bundles and any Client can request bundles provided they know the Bundle Id for the request. If the transport deletes the bundle right after delivery then a malicious Bundle Client could consume all the bundles and deny service. However, a malicious Bundle Client is unable to decrypt the data without the private key for the destined Bundle Client.

A user with the DDD android application (Bundle Client App) will always first request bundles from a Bundle Transport it connects to. This enables the Bundle Client to process acknowledgments (ACKSs) to any previous bundles that the transport may have brought from the server and exclude those application data units which have been ACKed. This prevents the Bundles from growing infinitely. Once the ACKs are processed the Bundle client is now ready to create new bundles that must be sent to the server. Therefore the Bundle Client app now initiates a sending to the transport.

A Bundle Transport will never initiate any requests to any Bundle Client, this is again because the Transport is an untrusted entity in the network and any malicious transport could deny the client services.

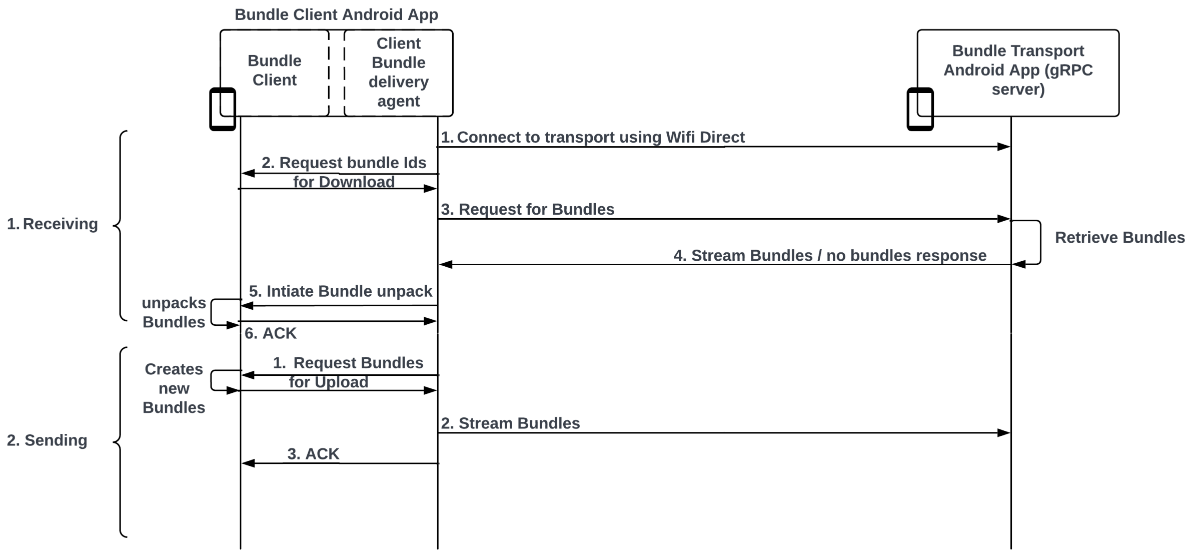

Bundle Client Android App
Bundle Client
Client Bundle delivery agent
Bundle Transport Android App (gRPC server)
1. Receiving
1. Connect to transport using Wifi Direct
2. Request bundle Ids for Download
3. Request for Bundles
Retrieve Bundles
4. Stream Bundles / no bundles response
5. Intiate Bundle unpack
unpacks Bundles
6. ACK
2. Sending
Creates new Bundles
1. Request Bundles for Upload
2. Stream Bundles
3. ACK

Figure 5: Client Bundle Delivery Agent

Fig. 5 above explains the interaction between a Bundle Client and Bundle Transport in a disconnected region. For Receiving

1. when a user is close to a Bundle Transport we expect the user to initiate a connection for example, by pressing a button on the Bundle Client App. The Bundle Transport will already have an open WiFi Direct group that the user's DDD application will be able to connect to.

2. Once the connection is established the bundle delivery agent on the Bundle Client app will internally place a request for the bundle Ids that it should request from the transport.

3. The bundle delivery agent will then use a gRPC call to send a list of bundles over to the Bundle Transport.

4. The Bundle Transport will start streaming those bundles to the bundle delivery agent. In case the Bundle Transport is not able to find any bundle with the

requested Ids it will send no bundles found as a response to the agent.

5. Once streaming is complete the bundle delivery agent will internally initiate the unpacking of bundles.

6. The Bundle Delivery Agent waits until the bundle unpacking is finished.

After this Sending

1. Sending is initiated by the bundle delivery agent by internally requesting bundles for upload.

2. This prompts the Bundle Client to create new bundles, the bundle delivery agent waits for new bundles to be created and then starts streaming those bundles up to the Bundle Transport via a gRPC call.

3. Once the streaming is finished the bundle delivery agent notifies the Bundle Client.

Once a Bundle Transport reaches a connected region it will upload bundles it carries from the Bundle Clients to the Bundle Server, i.e. the Bundle Server will first receive bundles. This serves a dual purpose, firstly as the Bundle Server is reliable the transport can go ahead and delete the bundles it carried from the Clients, therefore, freeing up some space for the new bundles. Secondly, any ACKs that the Client must have sent over can be processed by the Bundle Server and this again prevents the sever bundles from growing infinitely. Once the ACKs are processed the Bundle Server can create bundles to be sent over to the Bundle Clients.

Fig. 6 explains the interaction between Bundle Server and Bundle Transport in the connected region. For Receiving, the Bundle Server is always ready.

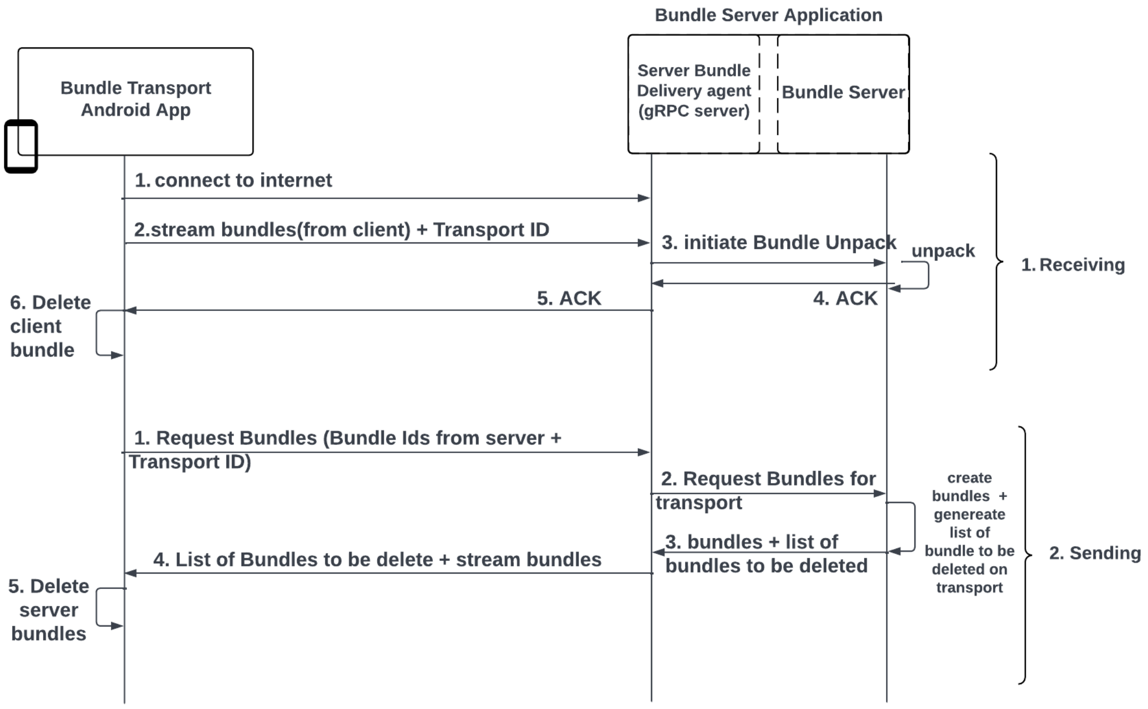

Bundle Server Application
Bundle Transport Android App
Server Bundle Delivery agent (gRPC server)
Bundle Server
1. connect to internet
2.stream bundles(from client) + Transport ID
3. initiate Bundle Unpack
unpack
1. Receiving
4. ACK
5. ACK
6. Delete client bundle
1. Request Bundles (Bundle Ids from server + Transport ID)
2. Request Bundles for transport
create bundles + genereate list of bundle to be deleted on transport
2. Sending
3. bundles + list of bundles to be deleted
4. List of Bundles to be delete + stream bundles
5. Delete server bundles

Figure 6: Server Bundle Delivery Agent

1. When the Bundle Transport is able to connect to the internet, it sends a list of Server Bundles that it has to the Bundle Server

2. It also starts streaming the client bundles to the Server Bundle Delivery Agent via a gRPC call.

3. The Bundle Server delivery agent forwards the list of bundles to the bundle server which it uses to decide which bundles must be deleted and also initiates bundle unpacking.

4. The Bundle Server ACKs bundle unpacking to the bundle delivery agent.

5. The bundle delivery agent ACKs this to the Bundle Transport.

6. Once the Bundle Transport receives the ACK it deletes the corresponding Client

Bundle.

For Sending

1. The Bundle Transport requests Server bundles by providing a transport Id.
2. The Server Bundle Delivery Agent requests the Bundle Server for bundles to be deleted list and the Server Bundles (if any) for the transport.
3. The Server Bundle Delivery agent receives these and initiates the streaming process.
4. The Bundle Delivery Agent streams the list of bundles to delete and the server bundles to the Bundle Transport over gRPC.
5. The Bundle Transport will receive the list of bundles to be deleted first and start bundle deletion in the background.

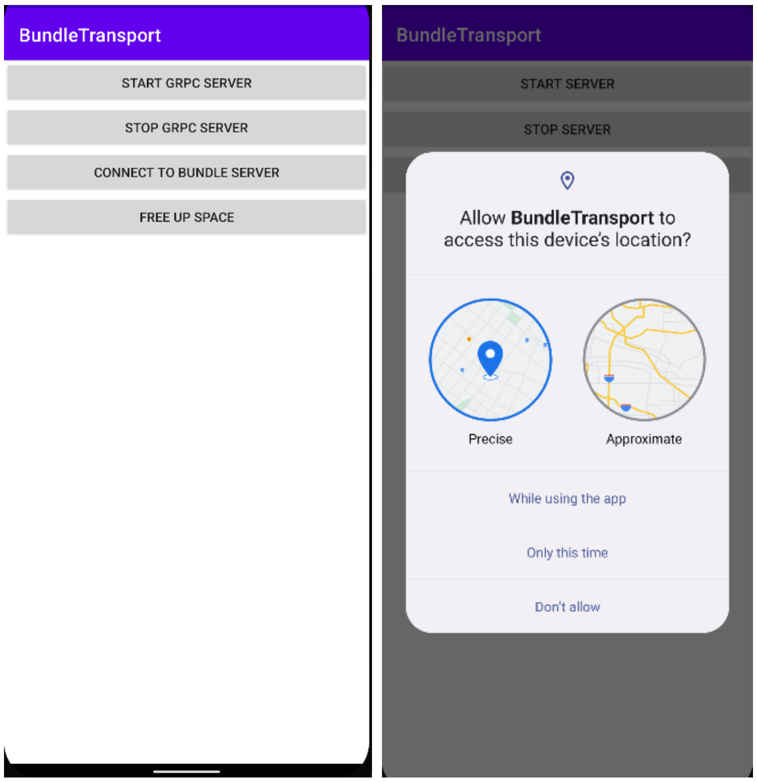
As discussed in the previous section, all bundles on the transport are deleted while interacting with the Bundle Server. First, the Client Bundles are deleted after they are uploaded to the Bundle Server to make more space for Server Bundles. Second, before the Transport starts receiving from the DDD Server it will delete the Server Bundles based on entries provided by the DDD Server. These bundles comprise Bundles that have been ACKed as well as bundles that have become redundant due to the incremental nature of bundles.

CHAPTER 4
Implementation

The implementations are currently written using Android SDK in Java for the Bundle Client and Transport applications and Java for Bundle Server. While we try to abstract and automate the process as much as possible, there are a few limitations with WiFi-Direct that will require the user to interact with our applications.

4.1 User Interaction

The bundle transport application must rename their devices to “ddd_transport”. This ensures identity is protected and bundle clients can find transports. We are currently unable to automate this step because device renaming in Android requires user intervention. One possible alternative we tried to explore is to overload the WiFi-Direct name via code but currently, there was no API exposed to do this. The Bundle transport application users are expected to manually start/stop the server from time to time using a button in the app. The Bundle Client application users will have to initiate the transfer to the transport application by pressing a button once they are in the range of the transport application.



Allow **BundleTransport** to access this device's location?

Figure 7: Bundle Transport UI

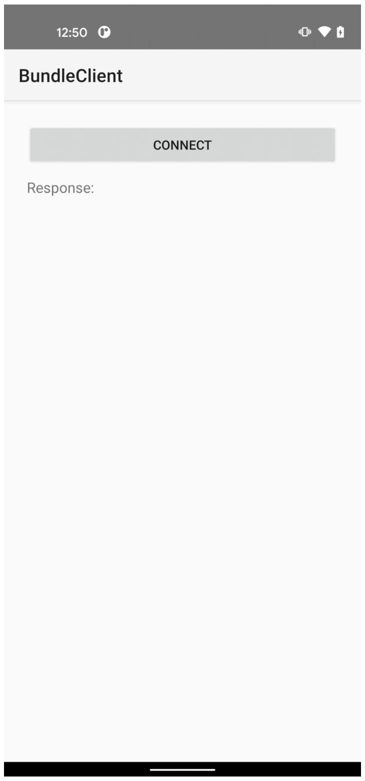


Figure 8: Bundle Client UI

4.2 Making connections

Clients will connect to the transport over WiFi-Direct using the Simple WiFi-Direct library[25].

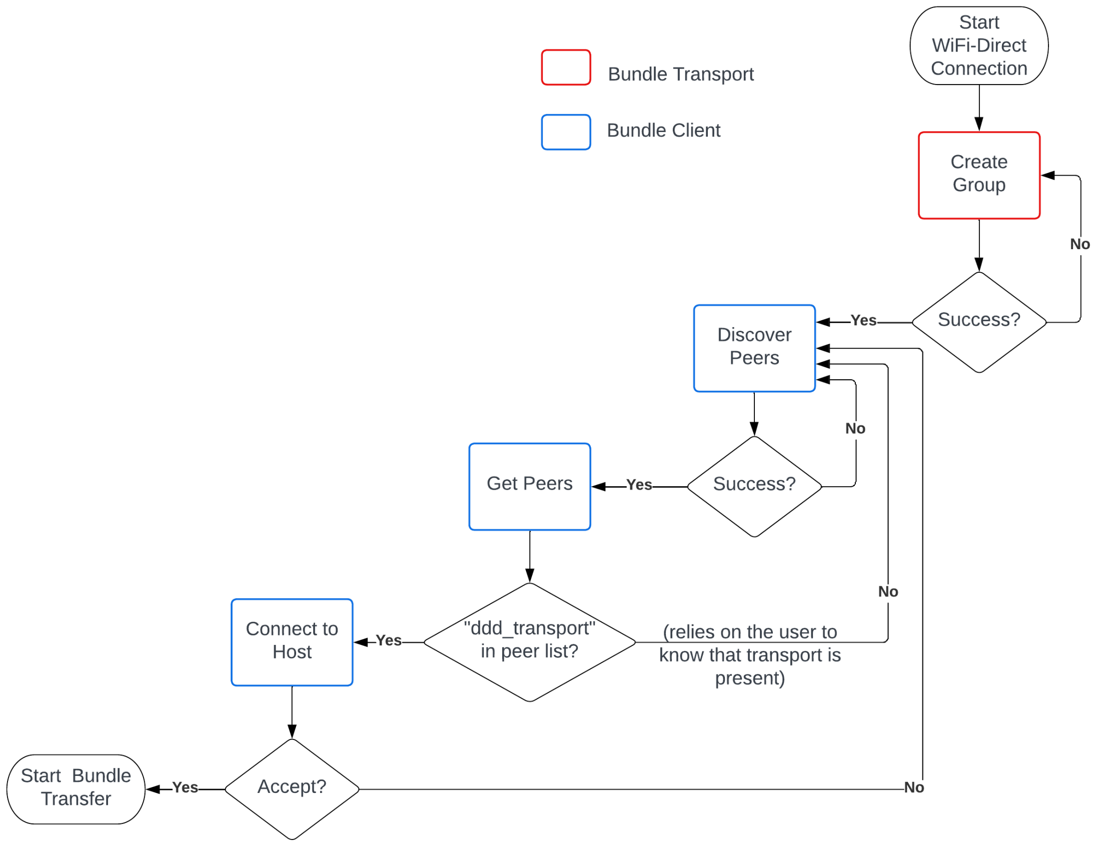


Figure 9: Making WiFi-Direct Connections Flow

Fig. 9 shows the flow in which devices will connect using WiFi-Direct on the Android platform. The create group method creates a WiFi-Direct peer-to-peer group for devices to join. This step is only required if legacy devices need to connect to the transport application for Android devices it is not required as all Android devices support WiFi Direct out of the box and the group is automatically created.

Once the Users manually initiate a connection the Bundle Client application will call `DiscoverPeers()` this will allow for service discovery for other devices. If successful we `GetPeers()` and check if the transport is present in the returned list. This can be done by looking for a specific device name in the list of peers. In our Case this name is "ddd_transport". Finally, if we find the desired group owner we `ConnectToHost()`. The group name is automatically created and handled by Android.

4.3 Managing connections

The Bundle transport is always the group owner so that clients cannot interact with each other. WiFi-Direct broadcasts intents [26] and we use a broadcast receiver in the Simple WiFi-Direct library to manage connections. An Intent is a message object that can be used to ask another app component on the same device to do a task. A broadcast is a message that can be received by any app. For example, in system events like system startup or device charging, the system sends out a variety of broadcasts. By supplying an Intent to `SendBroadcast()` or `sendOrderedBroadcast()`, you can send a broadcast to other apps. We implement a broadcast receiver that captures the following intents that are broadcasted by WiFi-Direct.

1. WIFI_P2P_CONNECTION_CHANGED_ACTION: report the state of the device's WiFi.

2. WIFI_P2P_PEERS_CHANGED_ACTION: when a client tries to discover peers this intent will be fired and used for making sure that the gRPC server is running on the transport.

3. WIFI_P2P_STATE_CHANGED_ACTION: This intent will let the applications know if WiFi is enabled on disabled on the device. WiFi should be enabled to use WiFi-Direct.

4. WIFI_P2P_THIS_DEVICE_CHANGED_ACTION: can be used to track if any changes happen in the device's details. This intent will be used to verify

that the transport application is named ddd_transport.

4.4 Storage Management

The bundles follow a store and forward scheme and are stored in the file system. We make use of Android Scoped storage [24] which was introduced for improved security since Android 11. `getExternalFilesDir()` [27] is an Android file system API that returns the path to an application directory created on the SD card if it exists. If the SD card does not exist it emulates SD card storage on the in-built storage of the device. Applications with Write permission can access the storage space but it is not visible to the user as media. Once the application is deleted its data is also deleted. This can be prevented by using the `getExternalStorageDirectory()` method which will return the root of the path to the SD card or emulated storage and the data will not be deleted when the application is deleted. Even though the data can be accessed by different applications on the transport the bundles by nature are opaque which means they are encrypted and immune to alterations by malicious agents. However, on the client applications, the data will be stored in scoped storage using the data directory of the application.

4.4.1 Client Storage

Fig. 10 show how the data is stored on the client device. Data is stored in internal storage and not an SD card because the data will be decrypted at some point while the bundle is being created and opened. "com.ddd.bundleclient" directory is created and managed by android. Only the to-send and received-bundles directories are exposed to the Bundle Transport Application. to-send is where the bundles to be sent to the Bundle Server are created and received-bundled is where the bundles which arrive from the Bundle Server are placed.

```
data
└─ data
   └─ com.ddd.bundleclient
      ├─ BundleTransmission
      │  └─ bundle-generation
      │     └─ to-send
      │        ├─ Bundle_id_1
      │        ├─ ...
      │        └─ Bundle_Id_n
      └─ Shared
         └─ received-bundles
            ├─ Bundle_Id_1
            ├─ ...
            └─ Bundle_Id_n
```

Figure 10: Client Storage

4.4.2 Transport Storage

On the transport since the bundles have been encrypted and are opaque we place them in the SD card directory.We can see this in Fig. 11. We are not concerned with any malicious agents altering the bundle. The directory names client and server denote the destination of the bundles stored there. The bundle are just stored directly without any organisation as this is not required.

On the transport, the users may want to free up storage space manually. The users have the ability to do so as well. This will have no effect on the service as we assume the transport to be unreliable.

```
sdcard
└── Android
    └── data
        └── com.ddd.bundletransport
            └── files
                └── BundleTransmission
                    ├── client
                    │   ├── Bundle_id_1
                    │   ├── ...
                    │   └── Bundle_Id_n
                    └── server
                        ├── Bundle_Id_1
                        ├── ...
                        └── Bundle_Id_n
```

Figure 11: Transport Storage

4.4.3 Server Storage

As shown in Fig. 12 the server the storage directories are created basis the transportID. The bundleTransmission directory is managed by the Bundle module which packs and unpacks bundles. The Shared directory will be accessed by both the transport and the Bundle module. However the Bundle Transmission directory will only be accessed by the Bundle Module.There are more directories that exist that the transport is not concerned with on the server. For example a staging area for the ADUs.

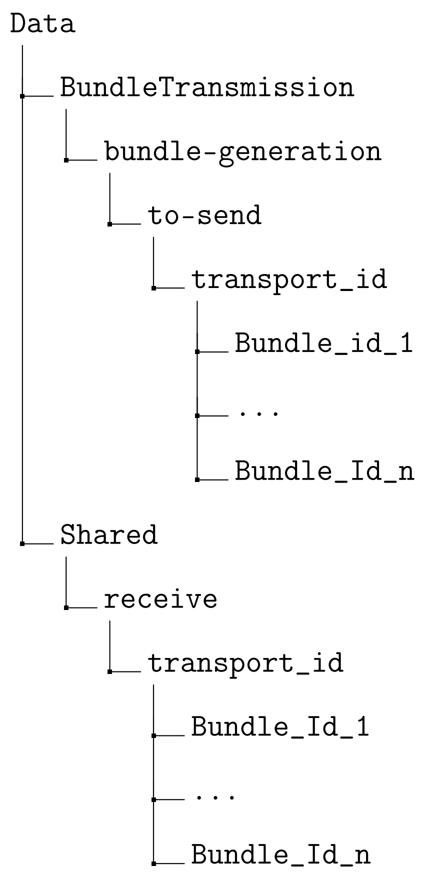


Figure 12: Server Storage

4.5 Streaming

We use gRPC on top of WiFi-Direct on the Bundle client to transport side and gRPC over the internet on transport to the Bundle Server side for communication.

The Bundle Client will first download the bundles that the transport is carrying. Refer to Listing 4.1, the Bundle Client to Bundle Transport Proto File the Bundle client will first call the `DownloadFile` method on the Bundle Transport once a WiFi-Direct connection is established on a background thread. The `ReqFilePath` message will be the Bundle Id that the client is expecting. The Bundle transport checks if the bundle exists and responds by streaming the bundle in chunks if found. Otherwise, the transport responds with an "UNAVAILABLE" status message. The calls for downloading bundles are made simultaneously and are independent of each other, which means we have a server streaming call for each bundle. However, the call for Uploading bundles is only made in the `postExecute()` method of the Download call thread, only once all the bundles are finished downloading and the thread has completed execution. This will allow the Bundle Client to factor in the newly received data while creating bundles for the upstream to the Bundle Server.

`UploadFile()` call works similarly to the `DownloadFile()` call except the roles are reversed and we have simultaneous client streaming calls. The Bundle Client will stream the MetaData and file chunks first and expect a single status response from the Bundle Client.

One might argue that gRPC provides bi-directional streaming to achieve the above functionality. However, using the bidirectional streaming approach introduces complexity to the solution. If the bundle were small enough to be transferred in a single call we could stream the requests but in our case, we would need a stream of streams because the bundle is also being split into chunks and we need multiple calls to transfer a single bundle. One major advantage of this approach is that we decouple the multiple bundle transfer for upload and download which makes the transfer opportunistic. While the bundle transfer is asynchronous and simultaneous the upload and download flow is sequential. This is because there are other modules that need to perform tasks that involve optimizing Bundle creation based on the data received from the transport. These modules will be discussed in a later section. The client is well aware of what bundles it is uploading and downloading which makes it possible to implement bi-directional streaming as the bundle Ids are known by the Bundle Client where the calls originate. However, the uni-directional approach makes it easier to manage transfer when the originator does not know what o expect i.e. in the case of the interaction between the Bundle Transport and the Bundle Server.

```
syntax = "proto3";
package protobuf;
option java_multiple_files = true;
option java_package = "com.ddd.bundletransport.service";
option java_outer_classname = "FileServiceEntry";
message ReqFilePath { string value = 1; }
message RespFileInfo {
  string name = 1;
  int64 size = 2;
}
enum Status {
  PENDING = 0;
  IN_PROGRESS = 1;
  SUCCESS = 2;
  FAILED = 3;
  UNAVAILABLE = 4;
}
message MetaData {
  string name = 1;
  string type = 2;
}
message File {
  bytes content = 1;
}
message FileUploadRequest {
  oneof request {
    MetaData metadata = 1;
    File file = 2;
  }
}
message FileUploadResponse {
  string name = 1;
  Status status = 2;
}
message Bytes { bytes value = 1; }
service FileService {
  rpc UploadFile(stream FileUploadRequest) returns (
      FileUploadResponse) {}
  rpc DownloadFile(ReqFilePath) returns (stream Bytes) {}
}
```

Listing 4.1: Bundle Client to Bundle Transport Proto File

The interaction between the Bundle Server and the Bundle Transport is very similar in the case of uploading and downloading files. Refer to Listing 4.2,The only difference is that the Metadata now also transmits the Transport Id which is used by the Bundle server for routing purposes. Additionally, the `DownloadBundle()` call first sends a list of downstream bundles that are already present on the Bundle Transport, the Bundle Server uses this list to generate a list of bundles to be deleted, generate new bundles, and keep track of the bundles already present on the Transport. When the Bundle Transport is downloading bundles it does not know the Bundle Ids of the bundles that are being downloaded. Hence this Id needs to be provided as a metadata field by the Bundle Server. This also means that in cases of failures the Bundle Transport, (where the calls originate) is incapable of keeping track of which bundle was lost or incomplete. In this case, the decoupling of bundle transfer ensures that this will not interfere with the other bundles and the failed bundle can be re-transmitted in its `onError()` implementation.

```
syntax = "proto3";
package protobuf;
option go_package = "./protobuf";
option java_multiple_files = true;
option java_package = "com.ddd.bundletransport.service";
option java_outer_classname = "BundleServerService";
option objc_class_prefix = "BSS";
import "FileService.proto";
message BundleMetaData {
  string bid = 1;
  string transportId = 2;
}
message BundleList{
  string transportId = 1;
  string bundleList = 2;
}
message BundleUploadRequest {
  oneof request {
    BundleMetaData metadata = 1;
    File file = 2;
  }
}
message BundleUploadResponse {
  string bid = 1;
  Status status = 2;
}
message BundleDownloadRequest {
  string bundleList = 1;
  string transportId = 2;
}
message BundleDownloadResponse{
  oneof request{
    BundleMetaData metadata = 1;
    BundleList bundleList = 2;
    File file = 3;
    Status status = 4;
  }
}
service BundleService {
  rpc UploadBundle(stream BundleUploadRequest) returns (
      BundleUploadResponse) {}
  rpc DownloadBundle(BundleDownloadRequest) returns (stream
      BundleDownloadResponse) {}
}
```

Listing 4.2: Bundle Transport to Bundle Server Proto File

4.6 Transport Identification

The transport Id is generated by generating a public-private key pair. The public part is Base64 encoded and stored on the transport as transportIdentity.pub. In order to get the transport Id the encoded key is then retrieved from the file decoded and hashed using SHA-1 after which it is encoded again to Base64 format. The Base64 encoding is performed for readability and also to reduce the length of the ID as it is being used for directory generation. Decreasing the length also makes it efficient to stream it over gRPC.

4.7 Bundle Client Integration

The Bundle Delivery Agent interacts with multiple Modules present in the Bundle Client Android application. These modules are considered a black box for this project and we assume that they are working as expected. Fig. 13 shows the sequence in which a bundle is converted to data and vice versa.

1. The Bundle Transmission module in the central module communicates with all other modules. Its primary purpose is to generate a Bundle from data provided by the Application Data manager.

2. The Application Data Manager modules will communicate with all other appli-

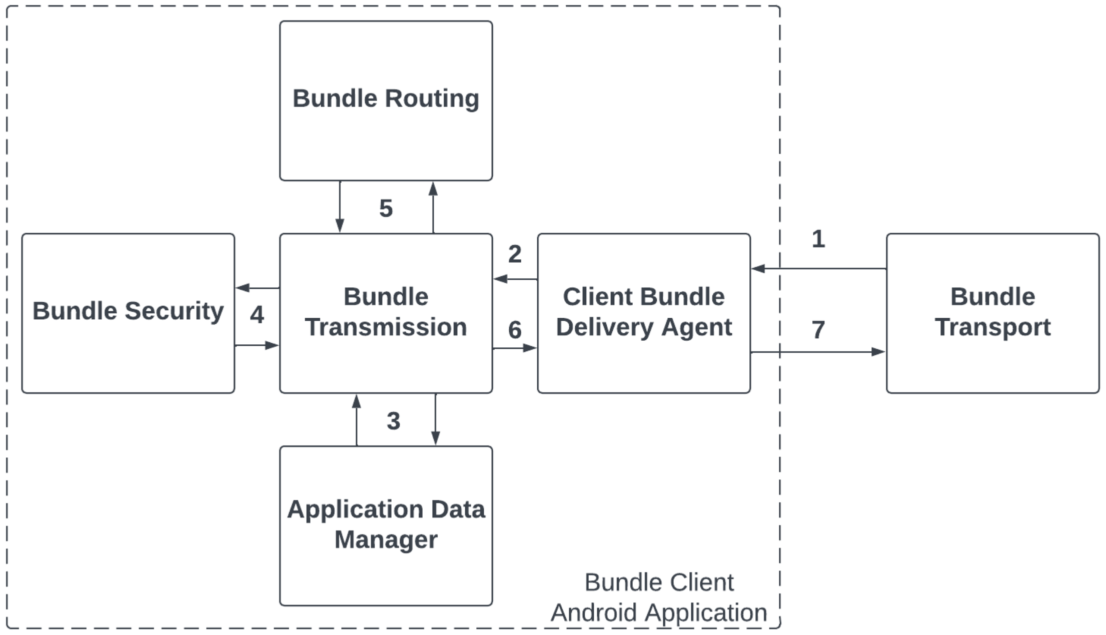


Figure 13: Bundle Client Modules Interaction Flow

cations on the device that have subscribed to the DDD service using Android inter-process communication (IPC). It will pull and push data from these applications and provides these Application Data Units (ADUs) to the Bundle Transmission module. The Application Data Manager also has a State Manager and Datastore which work together.

3. The Bundle Transmission module will use the Bundle Security module to perform encryption and decryption on the bundles. This module makes use of the Extended Triple Diffie-Hellman [28] algorithm to perform a key exchange and generate a shared secret which is used for encryption and decryption.

4. The Bundle Routing module generates the bundle and client Id. It also maintains a window of bundle counters which is in sync with the windows on the Server. This window enables the client to know which bundle ids are expected from a

transport. This module will make use of the Ratchet algorithm[29] to keep the windows in sync.

4.8 Bundle Server Integration

The Bundle server are similar to the counterparts on the Client and perform similar functions. A major difference is that the Application Data Manager must push and pull data from an Application Server Service. This service will interface between the actual application server and the Bundle Server. The Service will be registered with the DDD Bundle server which will provide a DDD Id to be used by the DDD network. Fig. 14 shows the sequence in which a bundle is converted to and from data and pushed to and from the service adapter.

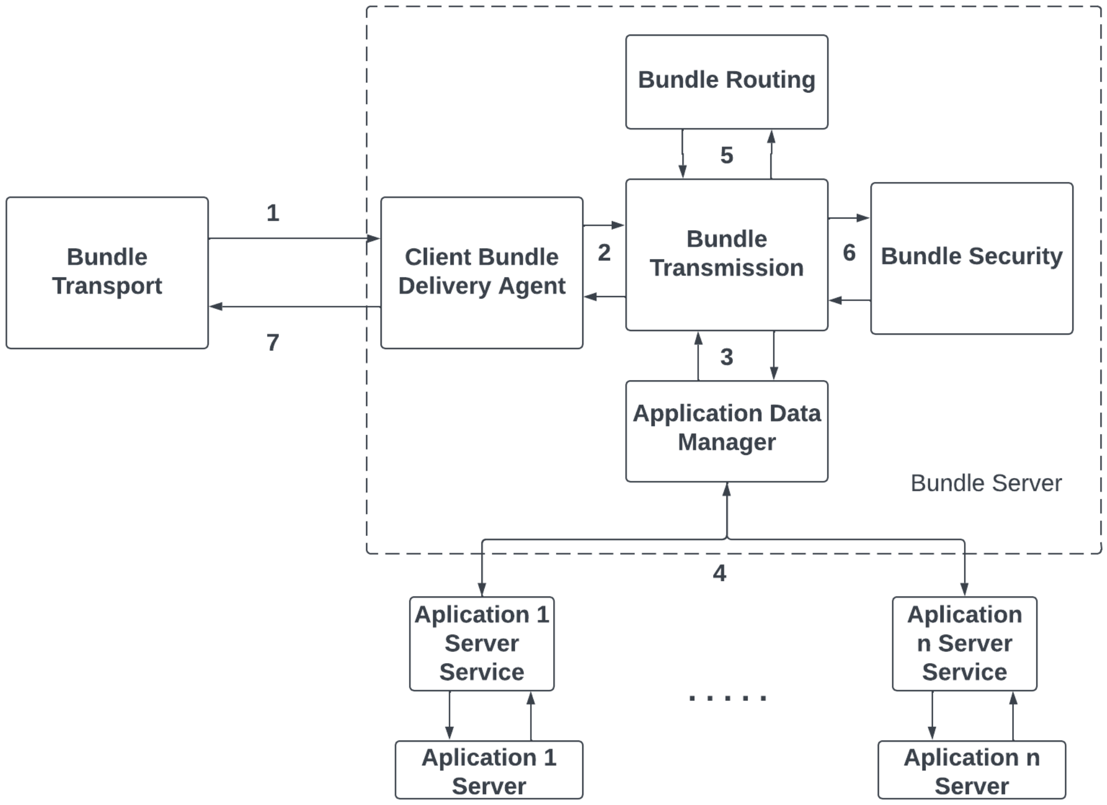


Figure 14: Bundle Server Modules Interaction Flow

CHAPTER 5
Experimentation

We will run experiments to check if file type, Phone hardware, and Message size will affect gRPC throughput. We use two different devices as clients: P1 – Samsung Galaxy S9 and P2 – Google Pixel 4A The server is running on a One Plus 6T device. P1 is a 2018 model, P2 is a 2020 model and the server device is a 2018 model. P1 is running Android 10, P2 is running Android 11 and the server phone is running Android 11. There is no internet connection on any of the devices. This setup will also help us demonstrate the effect on performance with different hardware.

5.1 gRPC throughput numbers over WiFi-Direct

gRPC loads all incoming messages in memory irrespective of if it is on the client or server. Therefore to prevent excessive resource consumption gRPC has message size limits set to 4 MB by default. There is no limit on outgoing messages. When I tried to send over a chunk of size 4MB got the following error: Errorio.grpc.StatusRuntimeException: CANCELLED: client cancelled this is because the server would not accept a message of size bigger than 4MB. The recommended chunk size for streamed messages appears to be 16-64KB.

5.1.1 Observations - SINGLE CLIENT

First, we connected a single client to the server device over WiFi-Direct and used gRPC to send a .flac file in chunks. The file is of size 70.2 MB. The chunk size is varied.

We observe from the plot 15 of throughput values vs Chunk sizes from table 2 that the throughput is affected by changes in chunk size as the chunk size increases the throughput improves. This is probably because with larger chunks the number of times that gRPC has to serialize and de-serialize messages is decreased.

Next, we change the file type to .zip which is a 75MB file consisting of (.flac, .txt,

Phone	Chunk Size	Time (seconds)	Rate (MB/sec)
P1	16 KB	2.83	24.8
P1	32 KB	2.46	28.5
P1	64 KB	2.54	2.76
P1	1 MB	2.07	33.9
P1	3 MB	2.11	33.2

Table 2: gRPC Throughput numbers over WiFi-Direct

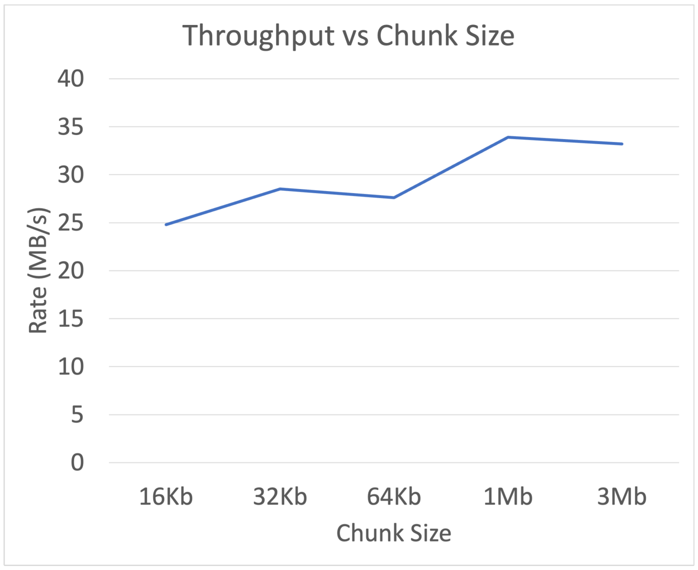


Figure 15: gRPC throughput for single client connection

.jpg) multiple file types. This is more in line with what our DDD network will be dealing with. This time we connect to devices one after the other.

Phone	Chunk Size	Time (seconds)	Rate (MB/sec)
P1	512 KB	2.85	26.3
P1	1 MB	2.52	29.7
P1	4 MB	2.19	34.25
P2	4 MB	1.6	46.8
P2	4 MB	1.43	52.44

Table 3: gRPC Throughput numbers over WiFi-Direct: single device connection

We observe from the throughput values in 3 that file type has no effect on throughput probably because we are streaming bytes. The device with newer hardware (P2) performs better probably due to a better WiFi chipset installed on it.

5.1.2 Observations - MULTIPLE CLIENTS

Phone	Chunk Size	Time (seconds)	Rate (MB/sec)
P1	512 KB	3.67	20.4
P2	512 KB	2.47	30.3
P1	1 MB	3.87	19.37
P2	1 MB	2.47	30.3
P2	4 MB	3.59	20.8
P2	4 MB	2.64	28.4

Table 4: gRPC Throughput numbers over WiFi-Direct: multiple device connection

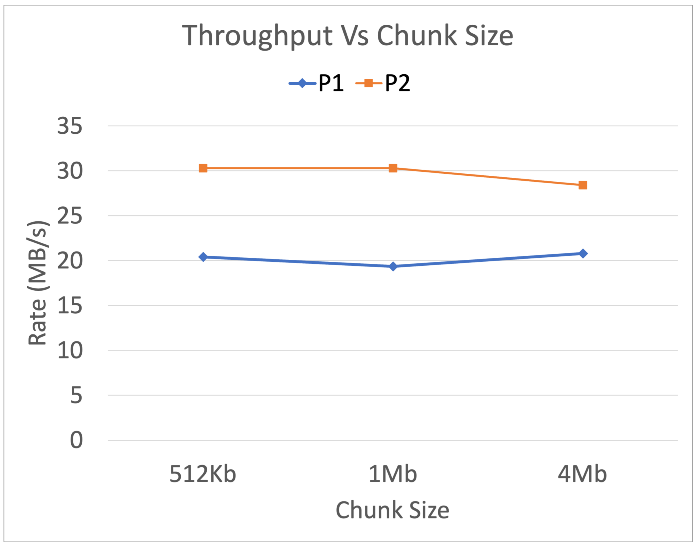


Figure 16: gRPC throughput for multiple connections

In this test, we connect both devices P1 and P2 simultaneously to the client. We send the same .zip file used in the previous run at the same time from both devices. We see from the plot 16 based on throughput values and chunk sizes in table 4 that the trend of P2 performing better than P1 is still observed. However, the overall throughput of both devices is lower.

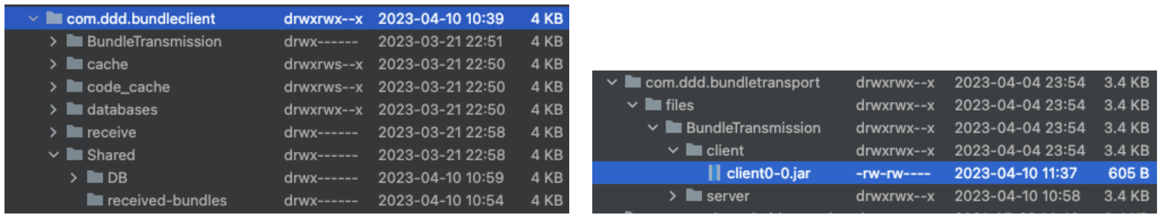
5.1.3 Conclusion

As more and more devices will connect to the network the throughput will go down. We can expect around 25 devices to perform transfers simultaneously. Devices with the latest hardware seem to give better performance and we observe the best performance with larger chunks as it decreases the overhead involved with serializing and deserializing messages.

5.2 Execution

We will run experiments to show how the system architecture translates into implementation. The Bundle Client Application is running on Phone P1. The Bundle Transport application is running on phone P2 and finally, the Bundle Server is running on a MacBook Pro Device with an M1 chipset. To simulate the connected and disconnected regions, P1 is never connected to the internet. P2 is not connected to the internet while interacting with P1 but is connected to the same WiFi access point as the Bundle Server device when interacting with the Bundle Server. Below we show the exchange of bundles between a Bundle Client and a Bundle Transport. There will be no deletion operations on the transport data. Once the Bundle Client connects to a Transport application via WiFi-Direct. The Client will first request for Bundles download. The Client uses a window and initiates a series of requests for all bundles expected from its window. The transport will fail bundle requests for bundles that are not available on the transport but stream the bundles that are present. We can see this behavior in the Fig. 17 wherein before receiving we see that in the first picture of client storage, the received-bundles directory is empty and the transport has a bundle "client0-0.jar". We can see that after the request is completed in Fig. 18 the client storage now has the bundle in its received-bundles directory.

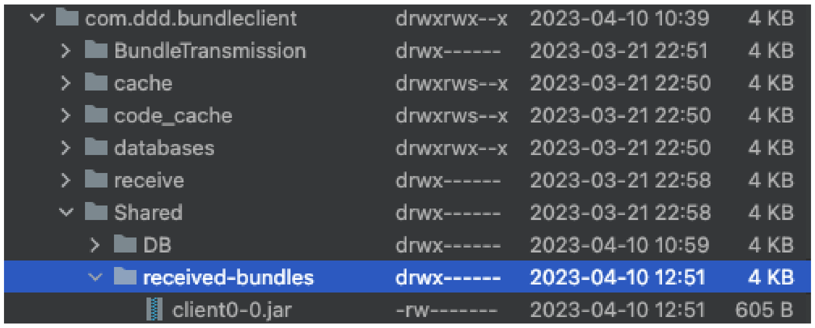
Similar to the previous example, the Client application will create a single



```
∨ com.ddd.bundleclient        drwxrwx--x  2023-04-10 10:39  4 KB
  > BundleTransmission        drwx------  2023-03-21 22:51  4 KB
  > cache                     drwxrws--x  2023-03-21 22:50  4 KB
  > code_cache                drwxrws--x  2023-03-21 22:50  4 KB
  > databases                 drwxrwx--x  2023-03-21 22:50  4 KB
  > receive                   drwx------  2023-03-21 22:58  4 KB
  ∨ Shared                    drwx------  2023-03-21 22:58  4 KB
    > DB                      drwx------  2023-04-10 10:59  4 KB
      received-bundles        drwx------  2023-04-10 10:54  4 KB
```

```
∨ com.ddd.bundletransport     drwxrwx--x  2023-04-04 23:54  3.4 KB
  ∨ files                     drwxrwx--x  2023-04-04 23:54  3.4 KB
    ∨ BundleTransmission      drwxrwx--x  2023-04-04 23:54  3.4 KB
      ∨ client                drwxrwx--x  2023-04-04 23:54  3.4 KB
          client0-0.jar       -rw-rw----  2023-04-10 11:37  605 B
      > server                drwxrwx--x  2023-04-10 10:58  3.4 KB
```

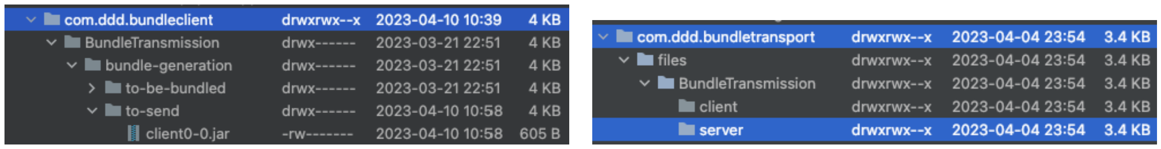
Figure 17: Before Receiving



```
∨ com.ddd.bundleclient        drwxrwx--x  2023-04-10 10:39  4 KB
  > BundleTransmission        drwx------  2023-03-21 22:51  4 KB
  > cache                     drwxrws--x  2023-03-21 22:50  4 KB
  > code_cache                drwxrws--x  2023-03-21 22:50  4 KB
  > databases                 drwxrwx--x  2023-03-21 22:50  4 KB
  > receive                   drwx------  2023-03-21 22:58  4 KB
  ∨ Shared                    drwx------  2023-03-21 22:58  4 KB
    > DB                      drwx------  2023-04-10 10:59  4 KB
    ∨ received-bundles        drwx------  2023-04-10 12:51  4 KB
        client0-0.jar         -rw-------  2023-04-10 12:51  605 B
```

Figure 18: After Receiving

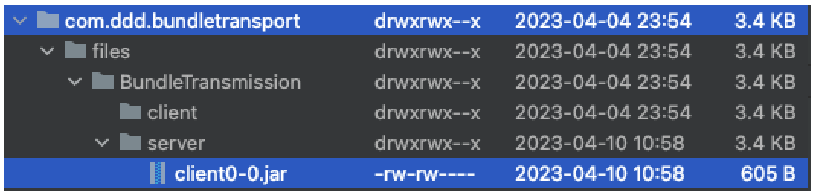
incremental bundle to be streamed to Transport. We can see this behavior in the fig. 19 where the client storage has a bundle "client0-0.jar" in its to-send directory while the transport storage does not have any bundles for the server. After the call is complete we can observe in Fig. 20 that the bundle "client0-0.jar" is now present on the transport storage in the server directory.



```
∨ com.ddd.bundleclient        drwxrwx--x  2023-04-10 10:39  4 KB
  ∨ BundleTransmission        drwx------  2023-03-21 22:51  4 KB
    ∨ bundle-generation       drwx------  2023-03-21 22:51  4 KB
      > to-be-bundled         drwx------  2023-03-21 22:51  4 KB
      ∨ to-send               drwx------  2023-04-10 10:58  4 KB
          client0-0.jar       -rw-------  2023-04-10 10:58  605 B
```

```
∨ com.ddd.bundletransport     drwxrwx--x  2023-04-04 23:54  3.4 KB
  ∨ files                     drwxrwx--x  2023-04-04 23:54  3.4 KB
    ∨ BundleTransmission      drwxrwx--x  2023-04-04 23:54  3.4 KB
        client                drwxrwx--x  2023-04-04 23:54  3.4 KB
        server                drwxrwx--x  2023-04-04 23:54  3.4 KB
```

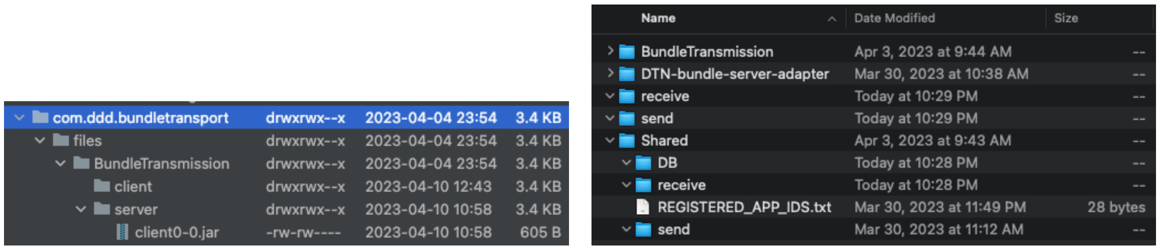
Figure 19: Before Sending



```
∨ com.ddd.bundletransport     drwxrwx--x  2023-04-04 23:54  3.4 KB
  ∨ files                     drwxrwx--x  2023-04-04 23:54  3.4 KB
    ∨ BundleTransmission      drwxrwx--x  2023-04-04 23:54  3.4 KB
        client                drwxrwx--x  2023-04-04 23:54  3.4 KB
      ∨ server                drwxrwx--x  2023-04-10 10:58  3.4 KB
          client0-0.jar       -rw-rw----  2023-04-10 10:58  605 B
```

Figure 20: After Sending

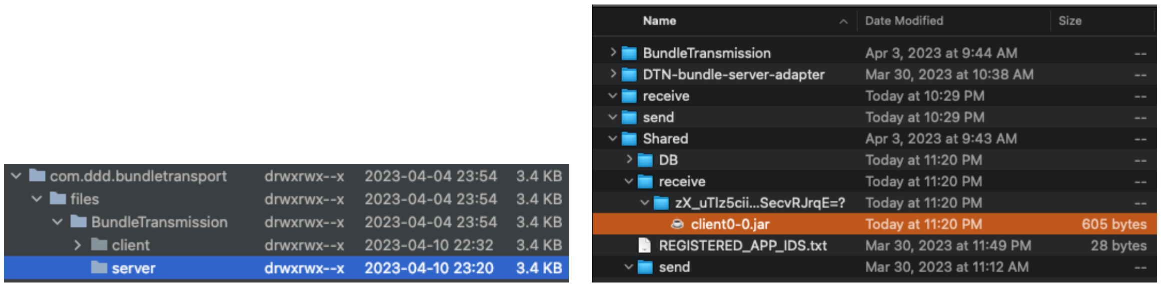
Once the Transport initiates a connection to the server using the Internet. Each file streaming has a deletion step associated with it. The Bundles from the Clients are uploaded to the server and deleted immediately once the streaming is finished. The behavior can be observed in Fig. 21 where we can see that the Transport has a bundle "client0-0.jar" for the server and the server storage is empty. After the call, we can see in Fig. 22 that the Server receive directory now has the bundle with the respective transport ID as the parent directory and the bundle has been deleted from the transport.



```
com.ddd.bundletransport      drwxrwx--x  2023-04-04 23:54  3.4 KB
  files                      drwxrwx--x  2023-04-04 23:54  3.4 KB
    BundleTransmission       drwxrwx--x  2023-04-04 23:54  3.4 KB
      client                 drwxrwx--x  2023-04-10 12:43  3.4 KB
      server                 drwxrwx--x  2023-04-10 10:58  3.4 KB
        client0-0.jar        -rw-rw----  2023-04-10 10:58  605 B
```

```
Name                         Date Modified               Size
BundleTransmission           Apr 3, 2023 at 9:44 AM      --
DTN-bundle-server-adapter    Mar 30, 2023 at 10:38 AM    --
receive                      Today at 10:29 PM           --
send                         Today at 10:29 PM           --
Shared                       Apr 3, 2023 at 9:43 AM      --
  DB                         Today at 10:28 PM           --
  receive                    Today at 10:28 PM           --
  REGISTERED_APP_IDS.txt     Mar 30, 2023 at 11:49 PM    28 bytes
  send                       Mar 30, 2023 at 11:12 AM    --
```

Figure 21: Before Sending

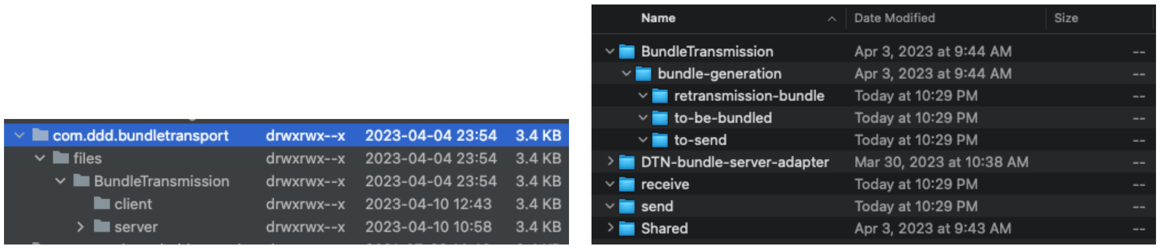


```
com.ddd.bundletransport      drwxrwx--x  2023-04-04 23:54  3.4 KB
  files                      drwxrwx--x  2023-04-04 23:54  3.4 KB
    BundleTransmission       drwxrwx--x  2023-04-04 23:54  3.4 KB
      client                 drwxrwx--x  2023-04-10 22:32  3.4 KB
      server                 drwxrwx--x  2023-04-10 23:20  3.4 KB
```

```
Name                           Date Modified               Size
BundleTransmission             Apr 3, 2023 at 9:44 AM      --
DTN-bundle-server-adapter      Mar 30, 2023 at 10:38 AM    --
receive                        Today at 10:29 PM           --
send                           Today at 10:29 PM           --
Shared                         Apr 3, 2023 at 9:43 AM      --
  DB                           Today at 11:20 PM           --
  receive                      Today at 11:20 PM           --
    zX_uTlz5cii...SecvRJrqE=?  Today at 11:20 PM           --
      client0-0.jar            Today at 11:20 PM           605 bytes
  REGISTERED_APP_IDS.txt       Mar 30, 2023 at 11:49 PM    28 bytes
  send                         Mar 30, 2023 at 11:12 AM    --
```

Figure 22: After Sending

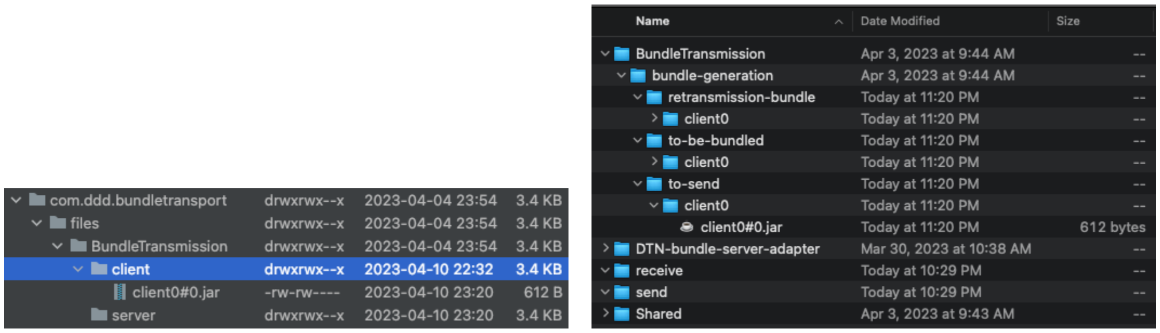
For each received request the transport sends a list of bundles already on the transport. This list enables the server to decide which bundles are redundant and should be deleted and which bundles to stream. The deletion steps occur before the streaming step to ensure we free up some space for bundles that are actually to be transported. We can observe in Fig 23 that the transport does not have any bundles for the clients in storage and there are no bundles to send to the Bundle Server. However, after the call, we can observe in Fig 24 that a bundle "client0#0.jar" was created on the server and sent over to the transport as seen in the client directory of the transport device.



```
com.ddd.bundletransport        drwxrwx--x  2023-04-04 23:54  3.4 KB
  files                        drwxrwx--x  2023-04-04 23:54  3.4 KB
    BundleTransmission         drwxrwx--x  2023-04-04 23:54  3.4 KB
      client                   drwxrwx--x  2023-04-10 12:43  3.4 KB
      server                   drwxrwx--x  2023-04-10 10:58  3.4 KB
```

```
Name                          Date Modified               Size
BundleTransmission            Apr 3, 2023 at 9:44 AM        --
  bundle-generation           Apr 3, 2023 at 9:44 AM        --
    retransmission-bundle     Today at 10:29 PM             --
    to-be-bundled             Today at 10:29 PM             --
    to-send                   Today at 10:29 PM             --
DTN-bundle-server-adapter     Mar 30, 2023 at 10:38 AM      --
receive                       Today at 10:29 PM             --
send                          Today at 10:29 PM             --
Shared                        Apr 3, 2023 at 9:43 AM        --
```

Figure 23: Before Receiving



```
com.ddd.bundletransport        drwxrwx--x  2023-04-04 23:54  3.4 KB
  files                        drwxrwx--x  2023-04-04 23:54  3.4 KB
    BundleTransmission         drwxrwx--x  2023-04-04 23:54  3.4 KB
      client                   drwxrwx--x  2023-04-10 22:32  3.4 KB
        client0#0.jar          -rw-rw----  2023-04-10 23:20   612 B
      server                   drwxrwx--x  2023-04-10 23:20  3.4 KB
```

```
Name                          Date Modified               Size
BundleTransmission            Apr 3, 2023 at 9:44 AM        --
  bundle-generation           Apr 3, 2023 at 9:44 AM        --
    retransmission-bundle     Today at 11:20 PM             --
      client0                 Today at 11:20 PM             --
    to-be-bundled             Today at 11:20 PM             --
      client0                 Today at 11:20 PM             --
    to-send                   Today at 11:20 PM             --
      client0                 Today at 11:20 PM             --
        client0#0.jar         Today at 11:20 PM      612 bytes
DTN-bundle-server-adapter     Mar 30, 2023 at 10:38 AM      --
receive                       Today at 10:29 PM             --
send                          Today at 10:29 PM             --
Shared                        Apr 3, 2023 at 9:43 AM        --
```

Figure 24: After Receiving

CHAPTER 6

Future Work

6.1 Multi-Hop Transport

The current implementation assumes that every transport will commute from a disconnected region to a connected one. However, it is also possible that some disconnected regions are unreachable by a single transport. In this case, we will require multiple hops for the bundle to reach the connected region. The transports will also have to talk to each other however with multiple hops there is no way of knowing how many hops will be required. One approach could be flooding the network with bundles but a major disadvantage is un-optimized storage management and denial of service. Another approach could be to introduce a routing table on the transports which would introduce complexity in terms of identity management, and denial of service attacks. The transport would have to necessarily generate and exchange metadata to update the routing tables and these changes could take very long to propagate through the network.

6.2 Transport Meta Data collection

The Transport application can collect metadata using GPS which may be used to identify areas without connectivity. This data can also be used to optimize routing. For example, if a transport follows a periodic route we may use that information to notify users, and we may use it to optimize bundle generation. This can also be used to score and optimize routes for bundles in the network. Meta-Data collection will still assume the transport as untrusted entities in the network and at the same time can help prioritize clients reachable from a particular transport.

6.3 Device coverage and Compatibility

Currently, our infrastructure only supports Android devices. The transport will continue to work primarily with Android devices however we can expand the client implementation to cover more devices. This is possible because the Group Owner in a WiFi-Direct network acts as a wireless access point and any device with WiFi capabilities can connect to it. This means that only Bundle Client implementation will be required to expand to IOS and PC devices. This will involve using the Proto file to build gRPC on different clients. We will also require the implementations to be rewritten in device-supported language, which should be possible and gRPC supports a wide variety of languages.

6.4 Transport Authentication

The transports have a public private key pair for identification. The transport Id is the public part of this pair.The transports are untrusted entities however if a malicious agent steals a transport's public key it can act as the transport. This is will have no effect on the end users as the data is opaque but it could still lead to a bad name for the transport and somewhere down the line deny service to its users.We need to implement preventive measures for this which can be done by taking advantage of gRPCs built-in auth mechanisms [30]. The transport can authenticate the channel or calls using gRPC and exchanging SSL data. Another approach could be to have a shared secret using the private key of the server or client and the public key of the transport. These can be used to generate a challenge for the transport which can be authenticated if it responds accurately to the challenge.

CHAPTER 7

Conclusion

There have been multiple attempts to bring connectivity to disconnected areas. Most approaches that work are geared towards emergencies where it is acceptable to bear set-up costs over delays. Very few approaches focus on cheaper solutions where delays in packet transmission are acceptable however most of them are built on top of customized hardware requirements. One of the major reasons why delay-tolerant networks were not widely adopted is the need to maintain and customize hardware in such networks. This project showcases how advances in mobile device technologies can be leveraged to eliminate the need for costly and highly customized software. We have demonstrated that Android mobile devices which are cheaply and easily available can be used as a carrier of data bundles from disconnected to connected areas. We have leveraged the use of technologies like gRPC and WiFi-Direct which are both platform-independent. The clients can be implemented on any device as gRPC uses proto files which are language-independent. Any device with WiFi capabilities can connect using WiFi-Direct to the Bundle Transport device. Multiple difficulties were faced in order to achieve this platform's independence. There has been no recorded use of gRPC over WiFi-Direct and running a gRPC server on Android devices is not officially supported.

Most approaches before have left aside the aspect of security and the user data is easily exposed to malicious agents which is unacceptable today. Our approach of propagating data as opaque bundles making the Bundle Transport an untrusted entity in the networks protects data privacy and security. This builds on top of the assumptions of existing solutions that at least some transport will reach the server.

Bibliography

[1] M. García-Escribano, “Low internet access driving inequality,” Jun 2020. [Online]. Available: https://www.imf.org/en/Blogs/Articles/2020/06/29/low-internet-access-driving-inequality

[2] “Two thirds of the world’s school-age children have no internet access at home, new unicef-itu report says,” Nov 2020. [Online]. Available: https://www.unicef.org/press-releases/two-thirds-worlds-school-age-children-have-no-internet-access-home-new-unicef-itu

[3] “The inclusive internet index,” 2022. [Online]. Available: https://impact.economist.com/projects/inclusive-internet-index/2022/availability?country=United-States

[4] D. F. Cabrera-Castellanos, A. Aragón-Zavala, and G. Castañón-Ávila, “Closing connectivity gap: An overview of mobile coverage solutions for not-spots in rural zones,” *Sensors*, vol. 21, no. 23, p. 8037, 2021.

[5] K. Fall, “A delay-tolerant network architecture for challenged internets,” *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003.

[6] A. Pentland, R. Fletcher, and A. Hasson, “Daknet: Rethinking connectivity in developing nations,” *Computer*, vol. 37, pp. 78--83, 02 2004.

[7] “Wizzy.org.za.” [Online]. Available: http://www.wizzy.org.za./

[8] J.-M. Kelif, ‘‘Coverage and performance of stratospheric balloons wireless networks,’’ in *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, 2016, pp. 1--6.

[9] S. Ilcev and A. Singh, ‘‘Development of stratospheric communication platforms (scp) for rural applications,’’ in *2004 IEEE Africon. 7th Africon Conference in Africa (IEEE Cat. No.04CH37590)*, vol. 1, 2004, pp. 233--238 Vol.1.

[10] S. D. Ilcev, ‘‘Low earth orbits (leo),’’ in *2010 20th International Crimean Conference "Microwave & Telecommunication Technology"*, 2010, pp. 406--408.

[11] [Online]. Available: https://www.starlink.com/technology

[12] T. Farrar, ‘‘Starlink’s reach won’t be enough to solve rural broadband dilemma - farrar,’’ Jan 2022. [Online]. Available: https://www.fiercewireless.com/tech/starlinks-reach-wont-be-enough-solve-rural-broadband-dilemma-farrar

[13] S. Guo, M. Derakhshani, M. Falaki, U. Ismail, R. Luk, E. Oliver, S. U. Rahman, A. Seth, M. Zaharia, S. Keshav, and et al., ‘‘Design and implementation of the kiosknet system,’’ *Computer Networks*, vol. 55, no. 1, p. 264–281, 2011.

[14] ‘‘Vlink.’’ [Online]. Available: http://blizzard.cs.uwaterloo.ca/tetherless/index.php/VLink

[15] J. Burgess, B. Gallagher, D. Jensen, and B. N. Levine, ‘‘Maxprop: Routing for vehicle-based disruption-tolerant networks,’’ in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, 2006, pp. 1--11.

[16] Jul 2013. [Online]. Available: https://www.wi-fi.org/discover-wi-fi/wi-fi-direct

[17] J. C. Haartsen, ‘‘Bluetooth—the universal radio interface for ad hoc, wireless connectivity,’’ 1998.

[18] ‘‘Discover wi-fi.’’ [Online]. Available: https://www.wi-fi.org/discover-wi-fi

[19] ‘‘Nfc.’’ [Online]. Available: https://nfc-forum.org/

[20] ‘‘Ultra-low latency audio over bluetooth.’’ [Online]. Available: https://www.freepatentsonline.com/y2019/0104424.html

[21] ‘‘Create p2p connections with wi-fi direct; android developers.’’ [Online]. Available: https://developer.android.com/training/connect-devices-wirelessly/wifi-direct

[22] J. Xu and W. Villarroel, ‘‘Wi-fi direct multi-group communication: Connect different wi-fi direct groups with access point,’’ Ph.D. dissertation, 2017.

[23] ‘‘Grpc.’’ [Online]. Available: https://grpc.io/

[24] ‘‘Scoped storage : Android open source project.’’ [Online]. Available: https://source.android.com/docs/core/storage/scoped

[25] S.-C. systems group, ‘‘Sjsu-cs-systems-group/simplewifidirect,’’ Oct 2021. [Online]. Available: https://github.com/SJSU-CS-systems-group/SimpleWifiDirect

[26] ‘‘Intents and intent filters: android developers.’’ [Online]. Available: https://developer.android.com/guide/components/intents-filters

[27] ‘‘Data and file storage overview: android developers.’’ [Online]. Available: https://developer.android.com/training/data-storage

[28] ‘‘Specifications - the x3dh key agreement protocol.’’ [Online]. Available: https://www.signal.org/docs/specifications/x3dh/

[29] “Specifications - the double ratchet algorithm.” [Online]. Available: https://signal.org/docs/specifications/doubleratchet/

[30] Oct 2022. [Online]. Available: https://grpc.io/docs/guides/auth/