Spring 2023

A Data Delivery Mechanism for Disconnected Mobile Applications

Shashank Hegde
San Jose State University

A Data Delivery Mechanism for Disconnected Mobile Applications

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

Shashank Hegde

May 2023

The Designated Project Committee Approves the Project Titled

A Data Delivery Mechanism for Disconnected Mobile Applications

by

Shashank Hegde

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

Dr. Ben Reed	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

ABSTRACT

A Data Delivery Mechanism for Disconnected Mobile Applications

by Shashank Hegde

Previous attempts to bring the data of the internet to environments that do not have continuous connectivity to the internet have made use of special hardware which requires additional expenditure on installation. We will develop a software-based infrastructure running on existing Android smartphones to exchange application data between a disconnected user's phone and corresponding application servers on the internet. The goal of this project is to implement client and server modules for this infrastructure to run on a disconnected phone and the internet respectively. These modules will multiplex application data to be sent into packages and distribute the data present in received packages to applications. This project will define and implement the package format and end-to-end delivery guarantees for the transmitted application data. In practice, the data will be transported physically by a mobile device like the phone of a bus driver traveling between disconnected and connected areas. Therefore, the guarantees will be based on the assumption of an opportunistic, high latency, and unreliable store-and-forward network between the client and server.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank all the people who have contributed to the success of this project. First, I would like to thank my project advisor Dr. Ben Reed for his support throughout the course of this project. I have learned immensely from him. His expertise, insights, and guidance have helped me navigate the challenges of this project. I would like to thank my committee members Dr. Robert Chun and Dr. Thomas Austin for their invaluable feedback on my work. I would also like to thank other members of our lab for fostering a collaborative environment and inspiring me through their hard work. Finally, I would like to thank my family and friends for their constant support and encouragement.

TABLE OF CONTENTS

CHAPTER

CHAPTER 1

Introduction

The world is heavily dependent on the internet today. Many routine aspects of life such as e-commerce, electronic payments, social media, email, navigation, etc. that we take for granted today were not so convenient before the internet. The general population, especially students, benefit a lot from the sheer abundance of information on the internet.

The indispensability of internet connectivity was especially evident during the COVID-19 pandemic shutdowns when in-person activities or events had to be conducted online if at all. However, about 37% of the world population i.e. 2.9 billion people have never been online, and out of the connected users, hundreds of millions of people had connectivity which was intermittent or had speeds that limited the usefulness of the connectivity [1]. This lack of internet connectivity across the world significantly limits the ability of people to contribute to the global economy. Hence, narrowing the connectivity gap can help reduce this under-utilization of human resources around the world.

We try to solve the problem of data disconnection by building a novel infrastructure to transfer data to/from disconnected areas. Section 1.1 describes the infrastructure and Section 1.2 describes the delivery guarantees provided by the infrastructure.

1.1 Disconnected Data Distribution (DDD)

In the absence of internet connectivity, it is desirable to have an infrastructure that can physically bring the data of the internet to a user in a disconnected area and vice versa. Previous attempts to set up such infrastructure have required special hardware which requires additional expenditure [2][3].

We focus on building such an infrastructure and we call it Disconnected Data

Distribution (DDD). DDD will be a software-only infrastructure as we leverage the ubiquity of Android phones rather than using any additional hardware. Figure 1 depicts the DDD infrastructure on a higher level. The applications that form our

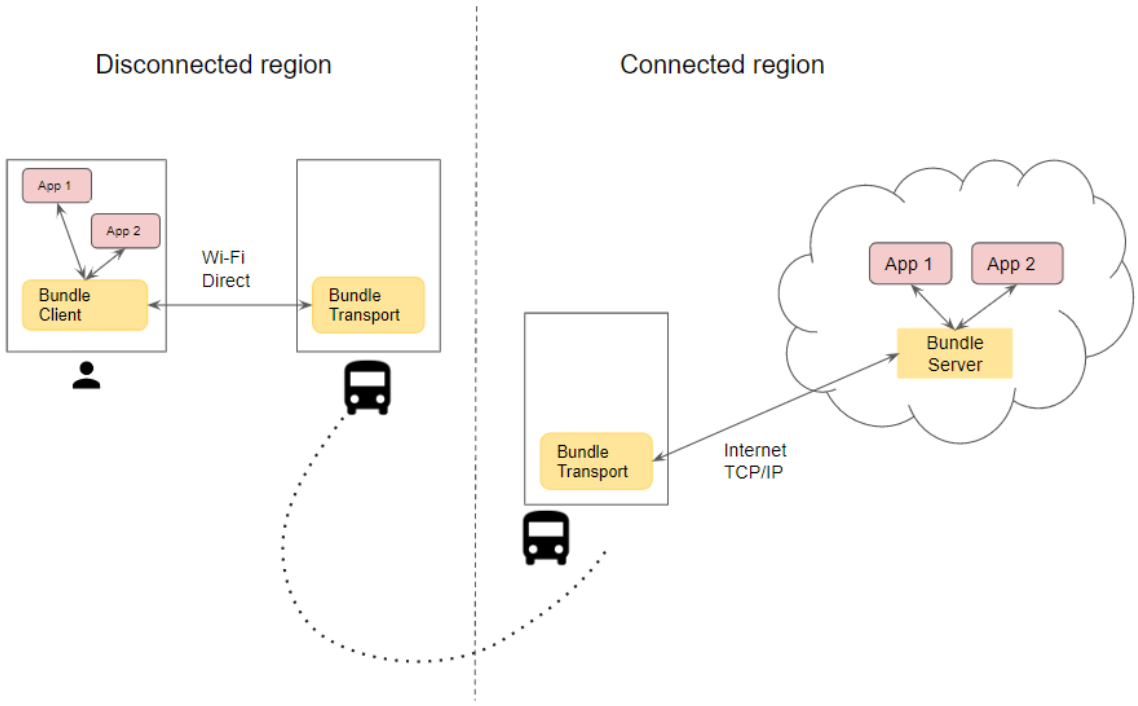

Figure 1: High-Level DDD Infrastructure

infrastructure are 1) Bundle Transport, 2) Bundle Client, and 3) Bundle Server. Bundle Transport is an Android application that will be installed in the phone of a potential carrier of bundles such as a bus driver, a kiosk vendor, delivery people, and others who normally travel between disconnected and connected areas. They only need to install an Android application from the Google Play store and do not need any other technical knowledge to be a part of DDD. They may need to pay for extended phone storage or internet services. They might have the incentive to be a part of DDD as they can use it as an opportunity to attract customers who want to use the services of DDD.

The Bundle Client is an Android application that will be installed by clients i.e.

disconnected users. It stores the data received from applications on the disconnected user's phone and it packages the application data. When a Bundle Client detects the presence of a Bundle Transport, it attempts to send the package to the latter using Wi-Fi Direct. It also receives packages that are sent to it by the Bundle Server from Bundle Transport. When Bundle Transport reaches an area with internet connectivity, it sends the packages to the Bundle Server over TCP/IP. The Bundle Server is a web service that unpacks the packages and routes the received application data to the application servers. It also packages application data for the client to send to Bundle Transport which can deliver it to the Bundle Client for which the data is intended.

1.2 Delivery Guarantees

DDD relies on a single hop of multiple physical untrusted carriers to carry data between disconnected and connected areas. We can abstract these links of transports connecting a disconnected region to the internet as a transmission network with the following characteristics:

1. End-to-end physical connection is not possible at any given point in time.
2. Is occasionally available to the sender or the receiver for a short duration of time.
3. Can introduce large unbounded delays in data transmission.
4. Unreliable and unsecure – data can be lost, corrupted, or duplicated.
5. High bandwidth.
6. Out-of-order delivery is possible.
7. Transmission succeeds with a low, non-zero probability – This means that if a sender keeps sending a data item indefinitely, it will be eventually delivered to

the receiver.

Since the transmission network is unpredictable, applications that use DDD need certain guarantees from the DDD infrastructure. DDD will provide the following guarantees to the applications under the constraints listed above:

1. Reliability – Guarantee that a unit of application data that is sent is delivered exactly once to the receiver while maintaining data integrity.

2. In-order delivery – Guarantee that the units of application data sent by the sender are received by the receiver in the order they were sent by the sender.

3. Deduplication – Guarantee that a receiver receives a unit of application data exactly once even though duplicates can be received from the transmission network.

4. Secure – Guarantee that the transmitted data is encrypted and can only be accessed by the sender and the receiver.

The Bundle Client and the Bundle Server can be broadly broken down into two parts: 1) The part which interfaces with the Android application and the application servers on the Bundle Client and Server respectively, 2) The part which packages the application data received from the first part on the sender, transports the packages and on the receiver end, unpacks the received package and delivers the received data to the first part. The focus of this project is defining how the second part can exchange data and communicate between the sender and receiver ends such that guarantees 1,2 and 3 above can be provided to the first part so that the first part can provide these guarantees to the applications. The security guarantee is not in the scope of this project.

CHAPTER 2

Related Work

This chapter discusses previous work that is related to this project to develop an understanding of previous attempts to provide delivery guarantees in a data transmission infrastructure over a high-delay and unreliable network. This discussion discusses important ideas that emerge from this related work. Section 2.1 covers the area of Delay Tolerant Networks (DTNs). Section 2.2 covers the Delay Tolerant Payload Conditioning (DTPC) protocol. Section 2.3 covers previous attempts to build village networks.

2.1 Delay Tolerant Networks (DTNs)

The internet which is based on the TCP/IP suite assumes the availability of end-to-end paths and delays in the order of milliseconds. Therefore, the data that switches and routers receive is not buffered in persistent storage before forwarding. As Delay-tolerant networks (DTNs) are characterized by intermittent connectivity, the assumptions of the internet mentioned previously do not hold in DTNs [4], and therefore, any given node cannot immediately forward data to another node. Therefore, DTNs use a store-and-forward approach to transmit data between two nodes in the network. Nodes in a store-and-forward network store data and forward it to other nodes in the network eventually delivering data to a destination. These store and forward routers have been called Data Mules [5] in DTN literature. Examples of DTNs include satellite networks where the motion of the earth and satellites do not allow continuous connectivity between them.

DDD infrastructure relies on mobile devices to physically carry data between a disconnected phone and the internet. The network of mobile phones such as the phone of a bus driver which connects disconnected users to an area with internet connectivity has high latency and the delay can be in the order of days rather than milliseconds.

In DDD, the transmission network consisting of mobile devices has intermittent connectivity and delays in the order of days. These are the same constraints present in a DTN. Therefore, similar to DTNs, DDD uses a store-and-forward technique to transport data. It is therefore useful to study the architecture and protocol widely used for data transmission in DTNs.

2.1.1 DTN Architecture and the Bundle Protocol

The DTN architecture [6] proposed an overlay network protocol called the Bundle Protocol running at the application layer bridging different heterogeneous networks. The architecture has the concepts of DTN regions and DTN gateways. A DTN region is characterized by similar network protocols and address families. A DTN gateway node connects two or more DTN regions. The architecture defined nodes running the Bundle Protocol as bundle nodes within different regions bridged together by nodes called DTN gateways. The Bundle Protocol also runs at the DTN gateway nodes forming an overlay network over the connected heterogeneous DTN regions. Within a given DTN region, the Bundle Protocol transmits data using the underlying protocol stack. For example, within region A with internet connectivity, the Bundle Protocol uses the TCP/IP stack for transmitting data. We could define region B as the nodes in space communicating using the Licklider Transmission Protocol (LTP) [7] instead of TCP. LTP is a point-to-point protocol designed to be used in deep-space links. If a node in region A needs to send data to a node in region B, the Bundle Protocol on the node in region A will use TCP to send data to a DTN gateway connecting regions A and B. Bundle Protocol running at the DTN gateway will use LTP for transmitting data to the node in region B.

If we were to apply the DTN architecture to DDD, we would have three DTN regions and corresponding gateways. The disconnected area, the area with internet

connectivity, and the area separating the disconnected and connected areas would be 3 DTN regions. The Bundle Client, Bundle Transport, and Bundle Server each would be DTN gateways running the Bundle Protocol. However, as we will see throughout this section, the DTN architecture including the Bundle Protocol does not completely fit our use case.

2.1.2 Bundle Format

The Bundle Protocol specification [8] describes the bundle format in DTNs. The protocol data unit of the Bundle Protocol is called a 'bundle'. Bundles should contain self-contained atomic chunks that an application can meaningfully process since we cannot rely on a conversational interchange [4]. These self-contained chunks of application data are called application data units (ADUs). A DTN bundle contains a single ADU.

DDD adopts the terms "bundle" and ADU from DTNs. A DDD bundle is also an encapsulation of ADUs. However, there are significant differences between a DTN bundle and a bundle in DDD. First, a DDD bundle is a Java Archive (JAR) file, unlike a DTN bundle which has a format specific to the Bundle Protocol. Second, a DDD bundle is an encapsulation of multiple ADUs from multiple applications while a DTN bundle contains ADUs from a single application. Third, the Bundle Protocol supports the fragmentation of ADUs while this project does not i.e. In the Bundle Protocol, any ADU can be divided into multiple fragments, each of which can be delivered through multiple bundles. The Bundle Protocol combines fragments received through multiple bundles into an ADU which is delivered to applications after assembly. In DDD the transmission network has a high delay and can lose data, it can take a lot of time before a fragmented ADU is assembled and delivered to an application. A receiver needs to wait for all fragments of any given ADU to be received even if all

other ADUs sent before and after have been completely received. This limits the ability of DDD to provide an in-order delivery guarantee to applications. Thus, DDD does not support the fragmentation of ADUs in favour of a simple design and in-order delivery guarantee.

2.1.3 Delivery Guarantees

The underlying protocol used by the bundle protocol in each region (for example, TCP and LTP) is called a convergence layer protocol. Since the bundle protocol needs to support a variety of convergence layers, there exists an adapter protocol corresponding to each convergence layer. This protocol is called a convergence layer adapter (CLA). Some examples of CLAs are TCPCL [9], UDPCL [10] and LTCPCL [11], etc. corresponding to TCP, UDP and LTP respectively for sending and receiving data within a region. The Bundle Protocol relies on the reliable delivery guarantees provided by CLAs within each region. However, since each region has a different CLA, the Bundle Protocol guarantees reliable delivery on a best-effort basis. The Bundle Protocol provides two optional features to increase the likelihood of reliable delivery of bundles – i. End-to-end acknowledgement and ii. custody transfer

Similar to the Automatic Repeat Request (ARQ) mechanism that is used in the TCP/IP suite for reliable delivery, the source of a bundle can request an end-to-end acknowledgement from the bundle destination in the bundle protocol and retransmit the bundle after a retransmission timeout (RTO). Custody transfer [12] is a mechanism provided by the bundle protocol where a bundle node transfers the responsibility of reliable delivery of a bundle to a bundle node further along the path and deletes its copy of the bundle. The bundle node which takes the responsibility becomes the new custodian of the bundle. A custodian sends an acknowledgement of the successful custody transfer to the previous custodian. The previous custodian retransmits the

bundle requesting custody transfer if the custody transfer acknowledgement is not received within a timeout. The custody transfer mechanism provides a safety net when the underlying reliable transport protocols fail to transport a bundle. Since DDD has one-hop transmission, transferring custody from the sender to the receiver would essentially mean having an end-to-end acknowledgement mechanism. Therefore, the Bundle Transport application is the only candidate for transferring custody from the sender. However, since the DDD architecture assumes that the Bundle Transport is not reliable, we cannot transfer custody to the Bundle Transport. Bundle Layer End-to-End Reliability (BLER) [13] uses a combination of custody transfer with end-to-end acknowledgement for reliable delivery of bundles. BLER attempts transmission with custody transfer enabled. After a bundle reaches its destination, the destination sends an end-to-end acknowledgement record to the sender.

All the approaches discussed above focus on reliable delivery for individual bundles. However, in this project, we focus on ensuring that any given ADU is reliably delivered even if individual bundles are not delivered. It is assumed that retransmission is the norm rather than the exception. Therefore, rather than triggering retransmission in the event of an RTO, each ADU is retransmitted in every bundle until the sender receives an acknowledgement for the ADUs the destination received.

A lot of research related to reliable delivery in DTNs focuses on estimating RTO timers in deep-space DTNs. BLER sets the RTO the same as the estimated round-trip time (RTT) using Contact Graph Routing (CGR). In [14], a method for estimating bundle delivery time in space inter-networking with scheduled contacts using CGR was proposed by Bezirgiannidis et al. Mechanisms to decide an optimal value for RTO have been proposed in [15], [16], and [17]. DDD's approach of continuous retransmission of ADUs eliminates the need for an RTO. Therefore, none of these techniques for estimating RTO is necessary for DDD.

2.2 Delay Tolerant Payload Conditioning (DTPC)

Delay Tolerant Payload Conditioning (DTPC) [18] provides a transport layer for DTNs over the Bundle Protocol and has its own protocol data unit called a DTPC item. A DTPC item is an aggregation of multiple ADUs. These DTPC items are then packed into a bundle and transmitted by the bundle protocol. As in DTPC, a DDD bundle is an aggregation of multiple ADUs from multiple applications. DTPC attempts to provide a transport layer on top of the bundle layer following the end-to-end argument [19]. Like in this project, DTPC provides a guarantee of end-to-end reliable delivery of application data. DTPC uses an end-to-end ARQ mechanism with positive acknowledgments like the previously described approaches. However, unlike a traditional approach to estimating RTO based on RTT, DTPC RTO is not a function of RTT. This is based on the observation that an end-to-end RTT cannot be accurately estimated in a DTN due to large variations in the constituent network delays. The reliable delivery mechanism used in this project is based on the same observation. However, unlike DTPC, the reliability mechanism in DDD does not require the use of RTOs as described in 2.1.3.

In DTPC, ADUs are delivered in the transmission order with the caveat that out-of-order items are delivered to the application if the missing data items are past their lifetime which is configured by the sender. In DDD, every bundle consists of a prefix of the sequence of ADUs that are not known to be delivered to the receiver. Thus, unlike DTPC, DDD always ensures that the ADUs are always delivered to applications in transmission order. Like DDD, DTPC also handles duplicates, delivering a single copy of an ADU to the application.

2.3 Village Networks

DakNet [2] was an ad-hoc wireless network used to bring asynchronous digital connectivity to rural areas. DakNet transmitted data between rural kiosks and portable storage devices called mobile access points (MAPs) using short point-to-point links. These MAPs were mounted on a vehicle like a bus or a bicycle. The operation of DakNet consisted of two steps: 1) As the MAP-equipped vehicle came within range of a village WiFi-enabled kiosk, it uploaded and downloaded data once it sensed the wireless connection. 2) When a MAP-equipped vehicle came within range of an Internet access point, it used the Internet to automatically synchronize the data from all the rural kiosks. DDD is an asynchronous network like the DakNet. Like DakNet, DDD uses portable storage to transport data between a disconnected device to a machine with Internet connectivity. However, DakNet required additional expenditure on acquiring and maintaining special hardware such as the hub, Wi-Fi-enabled Kiosk, and MAP. DakNet also required human assistance for data transportation and distribution at various points. DDD avoids these problems by using existing Android smartphones and a software-based infrastructure. The paper does not focus on the format of delivered data and the delivery guarantees.

KioskNet [3] was also a village network for bringing data connectivity to rural areas. Like DakNet, it used vehicles to ferry data between village kiosks and Internet gateways in nearby urban areas. Like the Bundle Server in DDD, KioskNet used a disconnection-aware proxy server to reassemble data received from the data ferries and transmit them to legacy application servers.

CHAPTER 3

System Overview

The unit of data transmission between the Bundle Client and the Bundle Server is called a bundle. A bundle is an encapsulation of data from several applications, metadata, and some data about the state of the communication for a given client as a JAR file. In Figure 2, the messages being transmitted represent bundles. The central layer in green represents the layer that implements the scheme that will be defined as part of this project. When sending data, the layer packages the application data received from the upper yellow layer, metadata, and state data into bundles and passes it down to the layer below it. When receiving data, the layer unpacks the bundle received from the blue layer below and passes the application data to the layer above it to be delivered to the applications. We call the carrier of the bundle a 'transport'. The Bundle Transport application runs on a transport's Android phone. This project focuses on defining the bundle format and how the central layer in green on both ends can exchange data to provide the 3 delivery guarantees described in Chapter 1 to the layer above and by extension, the applications.

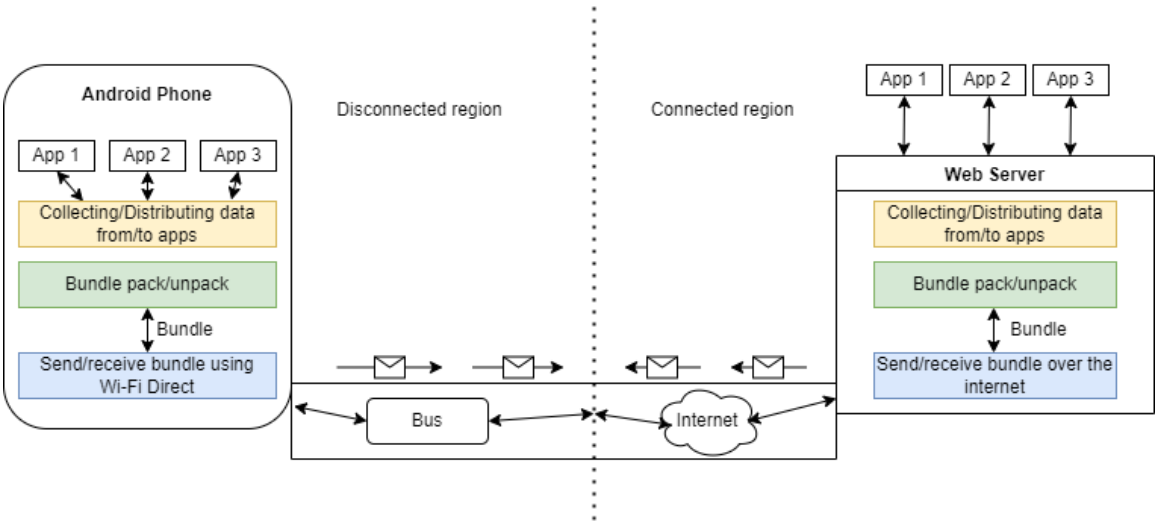

Figure 2: High-Level Architecture

Each bundle has a unique bundle id which is a combination of the client id – the client's unique identifier and a counter. A bundle consists of a header and a payload. The bundle header consists of the bundle id and information required for decrypting the bundle payload. The bundle payload consists of an acknowledgement record, a set of application data units (ADUs) of one or more applications, and some information related to routing. An ADU is a self-contained unit of data corresponding to an application. This might correspond to requests/responses from/to applications to/from their application servers. For this initial iteration of DDD, the bundle size limit is a constant and not dynamically determined. In the future, there might be models to determine an optimal value for the bundle size limit. Chapter 4 further elaborates on the structure of the bundle. As security and routing are not in the scope of this project, for the most part, we will ignore the presence of the bundle encryption and routing information in the bundle header and payload respectively. The details of bundle id generation are also out of the scope of this project.

The sender of a bundle maintains an application-specific id – a counter which is incremented and assigned to each ADU in the order ADUs are received from the application. The receiver of a bundle keeps track of the largest counter of an ADU received per application per sender. The server keeps a different mapping for each client while a client maintains just one mapping for the server. This mapping is used to detect ADUs already received so that the receiver can discard them rather than delivering them to the application. The receiver updates the mapping when the received bundle has ADUs with larger counters. The receiver also keeps track of the bundle id with the largest counter received so far per sender. This bundle id is used to discard any bundles having bundle ids with smaller counters. It is also included in all the subsequent acknowledgement records that are sent until a new bundle id is received.

Section 3.2 and Section 3.3 elaborate on the responsibilities of the Bundle Client and Bundle Server respectively. Section 3.4 talks about acknowledgements. Section 3.5 explains the working of the protocol with a few examples.

3.1 Bundle Client

When the bundle client detects the arrival of a transport, it establishes a connection with the Bundle Transport application running on the transport. The client then carries out two tasks in sequence: A) Receive bundles from the Bundle Transport, B) Send a bundle to the Bundle Transport. Section 3.1.1 and 3.1.2 cover flows A and B respectively.

3.1.1 Receiving Bundles from the Bundle Transport

The client maintains a receiver window of 10 (a voodoo constant) counters which correspond to the server's sender window for the client [1]. Therefore, even though the bundle ids for incoming bundles are generated by the bundle server, the client can calculate the bundle ids of those bundles using its client id and the window of 10 counters. The client requests Bundle Transport for bundles that were sent for it by providing these 10 bundle ids. Note that we have a window of size greater than 1 because we want to utilize the high-bandwidth network to get as many bundles as allowed by the network.

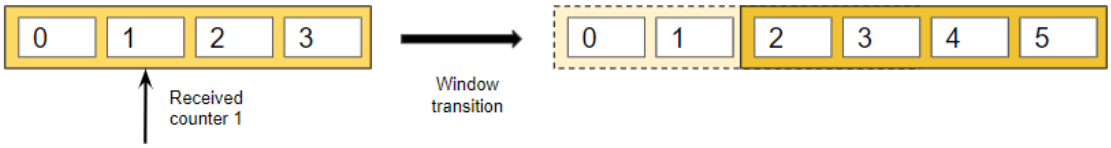

Figure 3: Receive Window

Consider the example shown in Figure 3. The client's receive window initially

[1]The constant was picked to balance the number of bundles that the Bundle Client expects with the ability to utilize the high bandwidth of the network and have multiple bundles in flight

consists of bundle ids with counters 0, 1, 2 and 3. In this example, we assume a window size of 4. The figure only shows the counters for clarity. After the client receives a bundle with a bundle id whose counter is 1, the window slides over 0 and 1 and new bundle ids with counters 4 and 5 are added to the window. Note that if a bundle whose id has a counter < 2 is received at this point, it will not be requested by the Bundle Client since they are not present in the receive window. The Bundle Client would not lose any data in doing so because that data has already been received. In general, when the client receives a bundle with counter x, it updates its receiver window to start from x+1, so that when the next transport arrives, the client asks for bundles with counter c such that x+1 <= c <= x+10. It is possible that Bundle Transport brings a bundle with counter $<$ x+1 after the window is updated. However, since such a bundle is a subset of the bundle with counter c, the client will not lose any data by not asking for the bundle with counter $<$ x+1.

If a received bundle has a larger counter than the bundle received so far, the client stores this bundle id. The client includes this bundle id in the acknowledgement record which it will pack in the next bundle it generates. The client decrypts the bundle and unpacks it. The received bundle has an acknowledgement record and ADUs. The ADUs are stored to be delivered to the applications asynchronously. The bundle id in the acknowledgement record is used to determine the ADUs to be sent in the next bundle. The acknowledgement record is explained in more detail in Section 3.4.

Although the implementation of the receiver window is not in the scope of this project, sliding the window is part of this project. Therefore, this project assumes an interface for manipulating the window without implementing it.

3.1.2 Sending a Bundle to the Bundle Transport

When a client detects the arrival of a Bundle Transport, it sends a bundle to the Bundle Transport. If the generated bundle has the same payload data i.e. same ADUs and acknowledgement record, as the last bundle sent, the bundle id of the bundle that was last sent to the server is assigned to this bundle signifying that this is a retransmission of the previous bundle. However, if this bundle has new data compared to the last sent bundle, a new bundle id is generated and assigned to this bundle. In any case, the generated bundle is then sent to Bundle Transport after encryption. The time spent on regenerating a previously sent bundle during retransmission could be saved by storing a generated bundle and sending it whenever a retransmission is required. However, since we want to minimize the storage overhead of the Bundle Client application, we trade off bundle generation time in favour of optimizing the app's storage requirements.

3.2 Bundle Server

When a transport arrives in an internet-connected region, the Bundle Transport application running on the transport connects to the bundle server running a gRPC server. The transport carries out two tasks in sequence. The tasks from the perspective of the bundle server are: A) Receive bundles from the bundle transport, B) Send bundles to the bundle transport. Section 3.2.1 and 3.2.2 cover flows A and B respectively.

3.2.1 Receiving Bundles from the Bundle Transport

Bundle Transport sends the bundles it collected from clients to the bundle server along with its transport id – a unique identifier for Bundle Transport. After delivering a bundle to the server, a transport deletes the bundle from its local storage.

Unlike the client, there is no receiver window on the server. The server accepts

all the bundles sent by Bundle Transport. The server identifies the client who sent a bundle from the bundle id. For each client, if a received bundle has a larger counter than the bundle received so far, the server stores this bundle id. The server includes this bundle id in the acknowledgement record which it will pack in the next bundle it generates for this client. The server decrypts the bundles and unpacks them.

Any received bundle has an acknowledgement record and might have ADUs. The ADUs are stored to be delivered to the application servers asynchronously. The bundle id in the acknowledgement record is used to determine the ADUs to be sent in the next bundle to be sent to the same client and to slide the server's sender window for the same client.

3.2.2 Sending Bundles to the Bundle Transport

Upon getting access to the internet, a transport requests the server for bundles that it can deliver to clients. Using the transport id provided by the transport in the request, the bundle server identifies the clients that the transport can reach. The server generates at most 1 bundle for each client that the transport can reach. The server maintains a sender window of 10 counters per client which it uses for the next 10 bundles it sends for a given client. When sending a bundle, the bundle server takes into consideration the bundles already present with Bundle Transport. A bundle that is already present on the Bundle Transport should not be sent to save on the bandwidth available to the transport. Therefore, the bundle server does not even generate such a bundle. If any bundle that the server would generate for the transport is a more recent version of a bundle already present on the transport, then the bundle on the transport is obsolete and needs to be deleted from the transport. For example; if the bundle on the transport for client C contains ADUs A1, and A2 and the bundle that the server generates next contains A1, A2 and A3, then the bundle

on the transport is obsolete. Consider another non-trivial example shown in Figure 4. The server sends a bundle B1 with A1, and A2 to the client through a transport T1. After B1 is received by the client, another transport T2 arrives which receives a bundle C1 with an acknowledgement for B1. When the transport T2 reaches the server, the server receives the acknowledgement for B1 first. When T1 arrives at the server, if the server has a new ADU A3 for client C, the new bundle B2 that it generates for the client will have A3 but not A1 and A2. B2 would still be considered a more recent version of B1 in this case making it necessary to delete B1 from T1 as shown in the Figure.

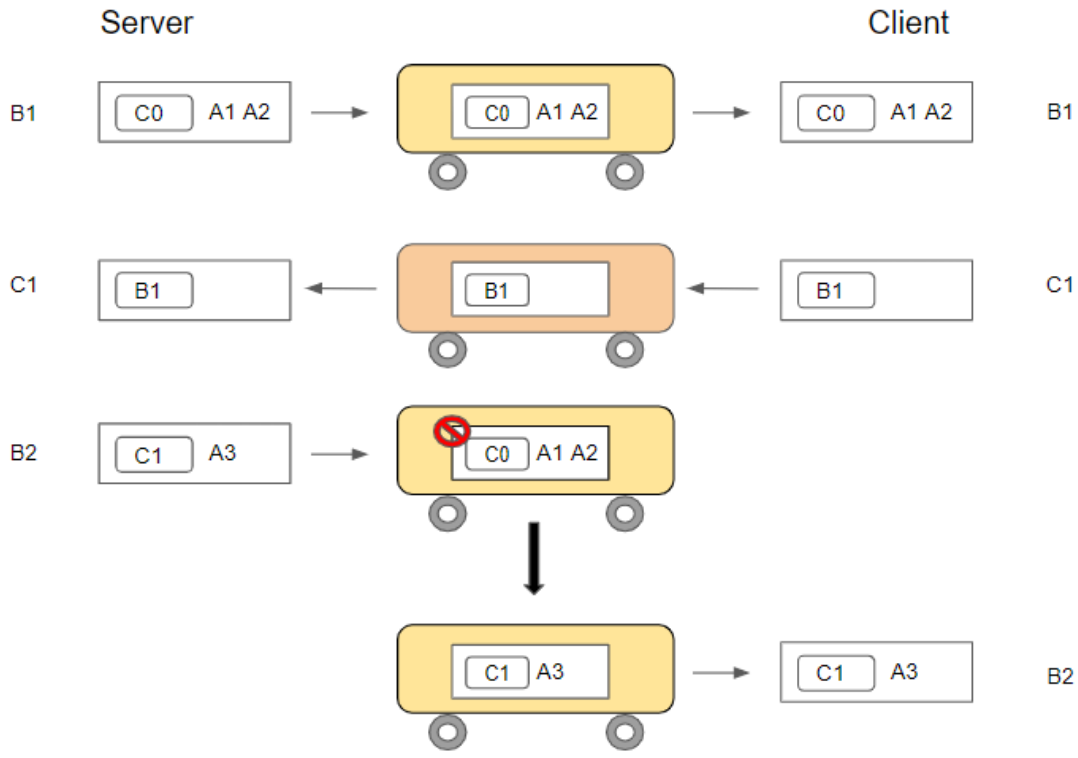

Figure 4: Bundle Deletion from Transport

For this purpose, when requesting bundles, bundle transport sends a list of bundle ids that it has stored locally besides the transport id. The bundle server sends a set of

new bundles along with a list consisting of bundle ids that the transport should delete from its storage. Let us call the list L. This list is a subset of the list of bundle ids that the transport sends to the server when requesting new bundles. Following is the process followed by the bundle server to decide whether a new bundle needs to be sent to the transport and which bundle id needs to be deleted from the transport when a transport makes a request to the bundle server to send bundles: Let us consider only 1 client for simplicity. In that case, L has bundle ids corresponding to just 1 client. The bundle server determines the bundle id of the next bundle it needs to send to the client. If the sender window is full for the client or if there is no new data to send as compared to the last bundle that was sent to the client, the server would need to do a retransmission of the last sent bundle if it is not already present on the transport. If the sender window is not full or there is new data to be sent to the client, the server would generate a bundle and assign a new bundle id to it. This new bundle id is added to the server's sender window. In all cases, the server adds all bundle ids in L to the deletion set. In the case of retransmission, the server checks if the bundle id of the bundle that was last sent to the client is present in L. If it is present, then the server does not regenerate the last sent bundle and deletes the last sent bundle's id from the deletion set. This same process is repeated for all clients. The final deletion set is the union of the deletion set corresponding to each client. In all cases, if a bundle is generated, the deletion set and the new bundles are returned in response to the transport's request. The transport deletes all local bundles whose id is present in the deletion set. Although the storage is not as big a concern as in the client, to keep the logic consistent with the client, we have the server regenerate the last sent bundle during retransmission rather than storing it.

As in the case of the client, the server only includes as much data as is allowed by the bundle size limit when generating a new bundle. A new bundle is always

encrypted before it is sent.

3.3 Acknowledgement Records

In the previous sections, the acknowledgement record has been mentioned several times. This section will elaborate on the nature and role of the acknowledgement record in greater detail.

An acknowledgement record contains the bundle id having the largest counter out of all bundles the receiver received from the sender. It is present in all bundles irrespective of the sender. The sender (whether a client or the server) uses an acknowledgement record to determine the next set of ADUs to be sent to the receiver. Consider the example shown in Figure 5. The rectangular boxes on the right of the figure represent bundles. The rectangles inside the bundles represent acknowledgement records. In step 1, the sender S sends a bundle B1 with ADUs denoted by A1 and A2 and an acknowledgement record with some value. When a bundle is sent, the sender maintains a mapping between the sent bundle id and the largest ADU counters for each application in the sent bundle. In this example, in step 2, the sender maintains a mapping between B1 and pair "A": 2 because the largest counter of ADUs of application A in bundle B1 was 2. The server maintains this mapping for each receiver (client). In step 3, the sender receives a bundle with an acknowledgement of the last sent bundle B1 from the receiver. Therefore, in step 4, the sender checks the payload content of acknowledged bundle B1 and updates the largest ADU id delivered value for application id 'A' to 2. The sender tries to generate a new bundle for the receiver. In order to determine the ADUs it should pack in the next bundle, the sender checks the largest ADU id successfully delivered for application 'A' and finds that it is 2. This means that all ADUs of application A having counter less than or equal to 2 have been received by the receiver. Therefore, in step 5, the sender does not pack ADU A1

and A2 in the next bundle B2 but instead packs ADUs with a counter greater than 2 such as A3. This shows the role of the acknowledgement record in determining the bundle content.

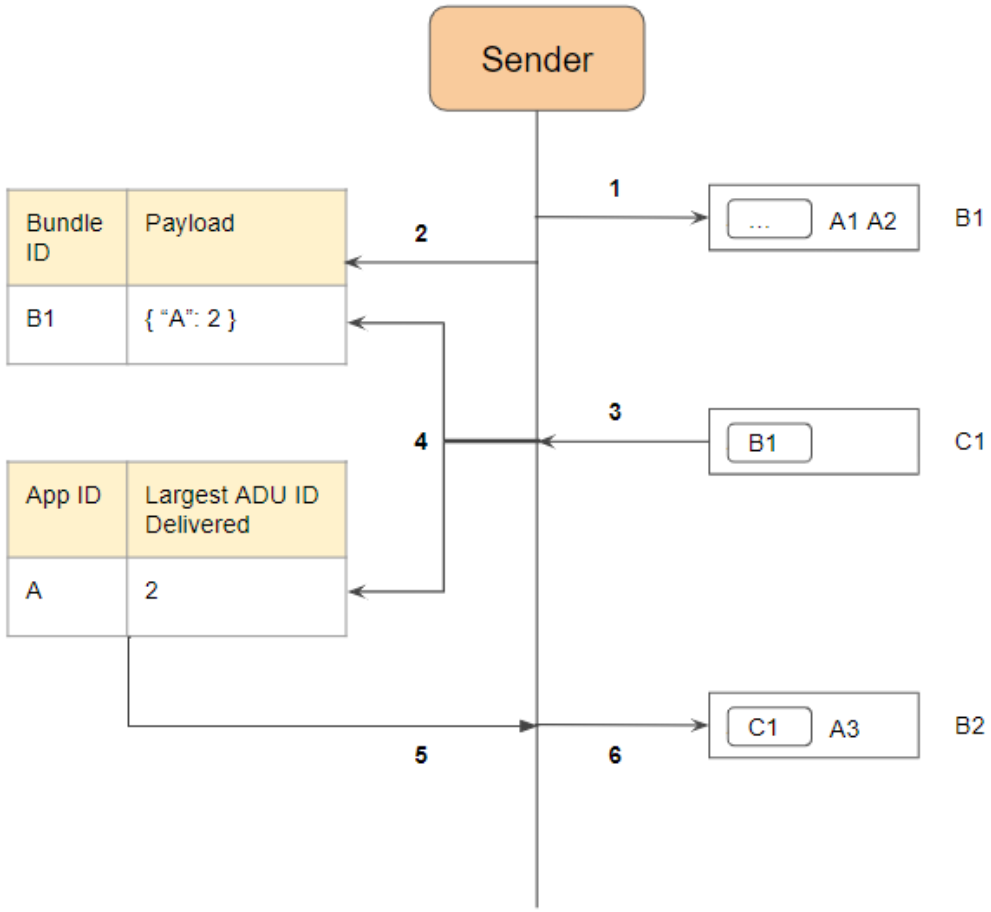

Figure 5: Acknowledgement Record Example

The acknowledgement record coming from a client also serves as a heartbeat to the server allowing the server to detect the presence and location (in terms of reachable transport) of the client. If the client did not receive any bundle from the server, this record will contain a constant 'heartbeat' message. Even if the client has no ADUs to send, the generated bundle will have an acknowledgement record containing a bundle id or a constant 'heartbeat' message thus ensuring that a heartbeat is sent by the

client any time a transport is available.

Apart from determining which ADUs to pack in a bundle, the server additionally uses the bundle id in the acknowledgement record to slide the sender window. When the server receives an acknowledgement for a bundle id with counter c, the server slides the window to start from c+1 till c+10. This update allows the server to add new bundles which did not overlap with the previous sender window to the new sender window.

3.4 Scenarios

With the help of some example scenarios, this section shows how the client and the server interoperate to deliver ADUs under the constraints and assumptions described in Section 1.3. In the scenarios below:

1. The interaction between a single bundle client and the bundle server is considered.

2. Just one application is considered without loss of generality i.e., any given bundle will have ADUs corresponding to a single application in the payload. The payload can include ADUs from multiple applications at the same time.

3. Application Data Unit (ADU) notation: ADUs sent by the client are named a1, a2, a3 and so on. ADUs sent by the server are named A1, A2, A3 and so on.

4. Bundle id notation: Client bundle ids are denoted as Ci where i is the bundle counter. Similarly, server-generated bundles have bundle ids denoted as Si where i is the bundle counter for the given client.

5. The rectangular boxes represent bundles. Only the content of the bundle payload is shown. An acknowledgment record is shown as a white box within the bundle.

6. The sender and receiver window size are 2.

7. Client and Server refer to Bundle Client and Bundle Server respectively.

3.4.1 Only client sends application data

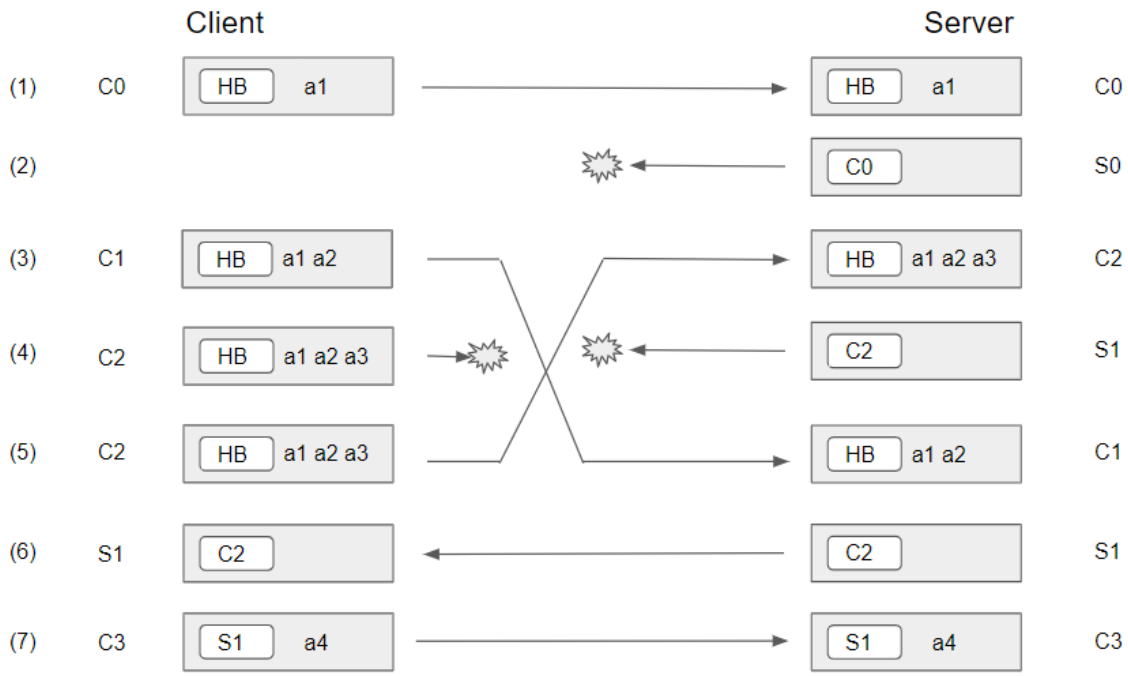

Figure 6: Only client sends ADUs

In the scenario shown in Figure 6, the client sends application data to the server and the server responds with acknowledgements and no additional data. Following are the explanations corresponding to line numbers (1) through (7): (1) The client sends a bundle with ADU a1 and an acknowledgement record containing a heartbeat message denoted by 'HB'. The bundle is successfully delivered to the server. At this step, the server unpacks the bundle and stores a1 to be delivered to the receiver application's server later. (2) The server sends a bundle with an acknowledgement record containing bundle id C0 which is the bundle id with the largest counter received so far. This bundle is lost midway. This represents situations where the transport delivers corrupted data to the client, loses data, or simply does not go back to the client again. (3) Since the client did not receive an acknowledgement for C0, it includes

a1 along with new ADU a2 and a heartbeat in the new bundle C1 which is generated when a transport arrives. (4) The client generates a new bundle with a heartbeat and a new ADU A3 along with previous ADUs A1 and A2. This bundle is lost. (5) At this point, the client does not have any new ADU to send. This can happen when either the application has not provided any new ADU, or the bundle size limit has been reached. The client sends the last sent bundle C2. When the server unpacks C2, it finds that the bundle has new ADUs a2 and a3 along with a previously received ADU A1. The server stores a2 and a3 to be delivered to the application's server and discards a1. As shown in (4), the server sends a new bundle S1 with an acknowledgement containing C2. This bundle is lost. At this point, the sender's window is full, and it will retransmit S1 until it receives an acknowledgement allowing it to slide the window. The server also received bundle C1 in (5). However, since C1 is smaller than C2 - which is the bundle id with the largest counter received so far, the server discards C1. This is how out-of-order bundles are discarded. C2 corresponds to the bundle id that will be sent in all acknowledgements from the client until a bundle with a bundle id with a bigger counter is received. (6) The server retransmits S1 with an acknowledgement record containing C2 which is received by the client. At this point, the client changes its receiver window from [S0, S1] to [S2, S3] because it received S1. Note that S1 is sufficient to slide the window even if S0 was not received. (7) At this point, the client has a new ADU a4 to send. On getting the acknowledgement for C2, the client also finds out that a1, a2 and a3 have been successfully delivered since C2 included A1, A2 and A3. Therefore, the client includes only a4 in the new bundle (bundle id C3) with an acknowledgement record containing S1 - the bundle id with the largest counter received so far from the server. On receiving the bundle, the server stores the new ADU a4. Since the server received the acknowledgement for S1, the server now slides the window from [S0, S1] to [S2, S3] which is the same as the

client's receiver window.

As we can see, ADUs sent by the client - a1, a2, a3, a4 were received by the server in order in spite of the loss of bundles and out-of-order bundle delivery.

3.4.2 Only server sends application data

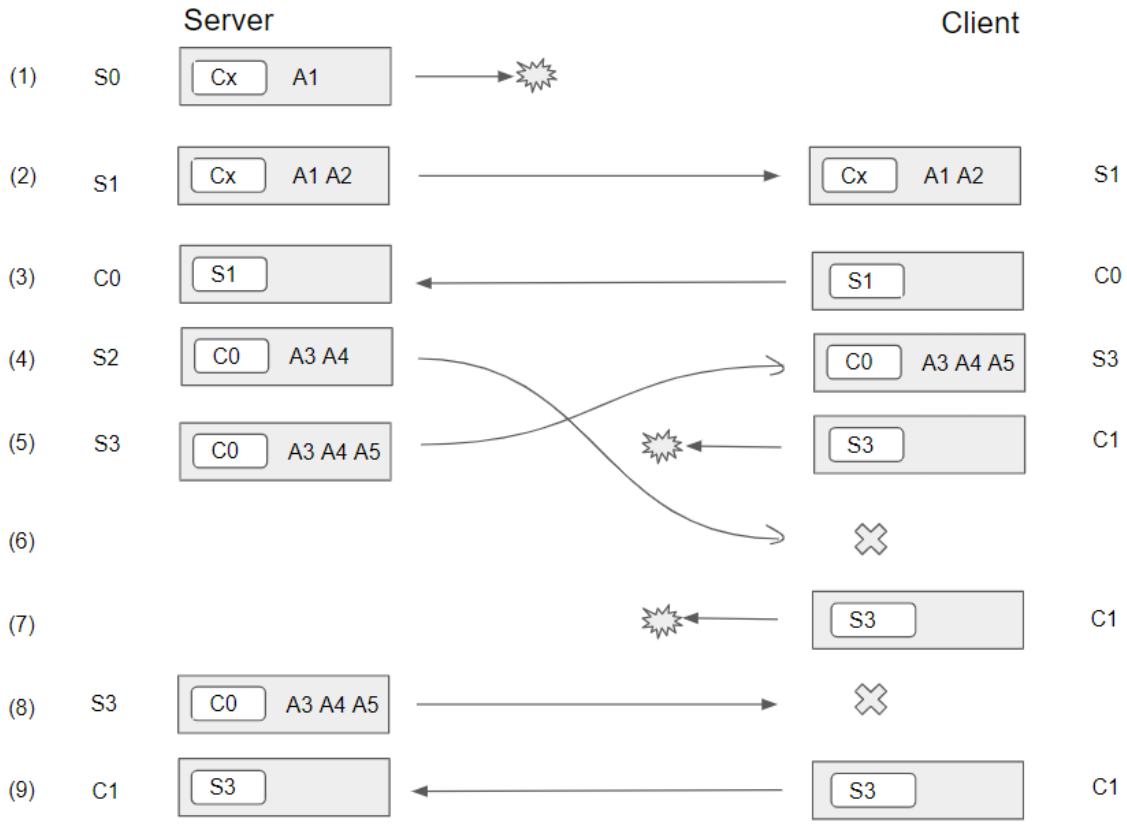

Figure 7: Only server sends ADUs

In the scenario shown in Figure 7, the server sends application data to the client and the client responds with acknowledgements and no additional data. Following are the explanations corresponding to line numbers (1) through (9): (1) The server sends a bundle S0 with ADU A1 and an acknowledgement recording containing Cx which corresponds to the last bundle sent by the client. The bundle is lost. (2) The server sends a bundle S1 with ADU A2, A1 and an acknowledgement recording containing

Cx. It also includes A1 since the server has not received any acknowledgement from the client. This bundle is successfully delivered to the client. The client stores A1 and A2 to be delivered to the application later. At this point, the server's sender window is [S0, S1] and full. Bundle S1 is successfully delivered to the client. The client's receiver window changes from [S0, S1] to [S2, S3]. At this point, the server's window and the client's window do not overlap at all. (3) Bundle C0 with an acknowledgement record containing S1 is successfully delivered to the server. The server's sender window slides forward from [S0, S1] to [S2, S3] allowing the server to send more bundles. (4) The server generates a bundle S2 with new ADUs A3 and A4. The server does not include A1 and A2, because it received an acknowledgement record for S1 which included A1 and A2. (5) When the next transport connects to the server, the server includes a new ADU A5 in bundle S3 along with ADU A3 and A4 and transfers it to the transport. At this point, the server's sender window is full. S3 arrives at the client before S2 gets a chance. The client unpacks S3 and stores new ADUs A3, A4 and A5 to be delivered to the application later. The client also sends a new bundle C1 with an acknowledgement record containing S3. However, this bundle is lost. The client's receiver window changes from [S2, S3] to [S4, S5] which does not overlap with the server's sender window [S2, S3]. (6) When the next transport arrives near the client, the client does not receive S2 because the client requests the transport only for the current window – S4 and S5. (7) The client again retransmits bundle C1 which is lost again. (8) The server retransmits bundle S3 since the sender window is full and it has not received any acknowledgements. However, the client does not ask for S3 since it is out of the client's receiver window. (9) The client again retransmits bundle C1. This time it is successfully delivered to the server. The acknowledgement causes the server to slide its sender window to [S4, S5] which overlaps with the client's receiver window.

As we can see, even if the server and client windows do not overlap, the client

will keep sending acknowledgement records which will cause the sender window to slide causing the sender and receiver window to eventually overlap. Also, ADUs sent by the server- A1, A2, A3, A4, A5 were received by the client in order in spite of the loss of bundles and out-of-order bundle delivery.

CHAPTER 4

Design and Implementation

In this chapter, the software design and implementation of the solution described in Chapter 3 have been discussed in detail to explain how the objective of this project was achieved. Section 4.1 covers the bundle format. Section 4.2 covers the software components of the bundle client and server. Section 4.3 and Section 4.4 elaborate on the design and implementation of the bundle client and server respectively.

4.1 Bundle Format

A bundle is implemented as a Java Archive (JAR) file with a few header files and another JAR for the payload. The header files consist of information that is necessary for the encryption/decryption of the payload. Since this project does not cover the security aspects, the structure of the header is not in the scope of this project. We will assume that it has a file containing the bundle id. The payload JAR file when extracted has a directory ADU storing the ADUs. The ADU directory will have a sub-directory per application named with the application's name. For example, ADUs corresponding to the application Signal will be in ADU/Signal/ inside the JAR file. Each ADU will have its own file named with the ADU id. Besides the ADU directory, the payload will have a file acknowledgement.txt for the acknowledgement record. Finally, the payload JAR file will have the JAR MANIFEST.MF file with an SHA-256 checksum of each file stored in the JAR.

A bundle has a size limit of 100 MB. Moreover, the total size of ADUs of any given application cannot exceed a fixed limit of 30 MB. For the first version of DDD, we fix the limits to arbitrary values. However, in the future, there might be algorithms to determine the appropriate size limits.

4.2 Components

Following are the components that are part of the high-level design of the bundle client and server.:

1. Bundle Delivery Agent (BDA): A client component responsible for detecting a transport and initiating the bundle generation process and requesting bundles from the transport.

2. Bundle Server Endpoint (BSE): A server component that listens for requests made by transport to send/receive bundles.

3. Bundle Security (BS): Component responsible for managing ids – client id and bundle id.

4. Window and Bundle Routing (WR): This is an abstraction that represents a combination of one or more server components that keeps track of which transports can reach which clients and maintain the receiver and sender windows on the client and server respectively.

5. Bundle Transmission (BT): A client and server component responsible for the process of bundle generation and handling of received bundles with the help of other components.

6. Application Data Manager (ADM): A client and server component which manages the data received from applications directly or through bundles. Responsible for storing ADUs and deciding which ADUs to send next to a given receiver. Following are the sub-components of ADM and their description:

   (a) ADM Coordinator (ADMC): Component of the ADM which implements the public interface of the ADM with the help of SM and DSA sub-components.

(b) State Manager (SM): Manages the state of communication between a client and server keeping track of the ADUs delivered or received per application. Responsible for deciding which ADUs are to be sent in a bundle based on the communication state.

(c) Data Store Adaptor (DSA): This is an abstraction for the part of the ADM that stores/fetches the actual ADUs from a data store. This is not an actual named module in the ADM.

BT, ADMC and SM sub-components of the ADM are in the scope of this project. Going forward, the abbreviations of the component names will be used.

4.3 Bundle Client

The components described above are the building blocks of the bundle client. To understand the design of the bundle client, we need to understand how these components fit and work together to execute the functions of the bundle client. This section covers the design and implementation of the bundle client. Sections 4.3.1 and 4.3.2 cover implementation details which are put in context when describing the execution flows in Section 4.3.3.

4.3.1 Dependencies

This section covers the APIs of the components that BT, ADMC, and SM rely on as well as the public APIs of BT that are used by the BDA to trigger the receive and send flows.

4.3.1.1 Bundle Security

```
public int compareBundleIDs(String id1, String id2, boolean direction)
public String decrypt(String bundlePath, String decryptedPath)
public String encrypt(String toBeEncPath, String encPath, String bundleID)
public String generateBundleID(boolean direction)
```

The function compareBundleIDs is called to compare two bundle ids by counters. This is used by BT to discard obsolete bundles. BT uses the function decrypt to decrypt the payload of a received bundle. The function encrypt is used by BT to encrypt the payload of a received bundle. The function generateBundleID is used by BT to generate a new bundle ID.

4.3.1.2 Window and Routing

```
public void bundleMetaData(String bundlePath)
public void updateWindow(String bundleId)
public void updateMetadata(String transportId)
```

The function bundleMetaData is called by BT to add routing metadata in bundle payload. The function updateWindow is used by BT to request WR to remove a bundle id from the receiver window The function updateMetadata is used by BT to indicate to WR that a bundle has been received from a given transport so that WR can use this information for routing.

4.3.1.3 Data Store Adaptor

```
public void persistADU(ADU adu)
public void deleteADUs(String appId, long aduIdEnd)
public List<ADU> fetchADUs(String appId, long aduIdStart)
```

The function persistADU is called by the ADM to request DSA to store an ADU when a bundle is received from the Bundle Server. The function deleteADU is called by the SM to request DSA to delete ADUs up to a certain ID. This is called when processing an acknowledgement. The function fetchADUs is called by the ADM to request DSA for ADUs starting from a certain ID for a given application when generating a new bundle.

4.3.1.4 Bundle Transmission

```
public void processReceivedBundles(String transportId, String
    bundlesLocation)
public BundleDTO generateBundleForTransmission()
```

The function processReceivedBundles and generateBundleForTransmission are called by BDA when receiving bundles from and sending a bundle to a Bundle Transport respectively.

4.3.2 Database

On the client, the state of the communication between the bundle client and the bundle server is stored in files. Files used for storing the state would be stored on the File System of the Android phone in the Bundle Client's storage when integrated. Following is the description of some important files:

1. LARGEST_ADU_ID_DELIVERED.json: This file stores a mapping between the application name and the largest ADU ID delivered to the server. For example, it stores the JSON string {"A": 1, "B": 3} if the largest ADU ids delivered to the server for applications A and B are 1 and 3 respectively.

2. LARGEST_ADU_ID_RECEIVED.json: This file stores a mapping between the application name and the largest ADU ID received from the server. For example, it stores the JSON string {"A": 1, "B": 2} if the largest ADU ids received from the server for applications A and B are 1 and 2 respectively.

3. LARGEST_BUNDLE_ID_RECEIVED.txt: This is a file that stores the bundle id of the last bundle received from the server.

4. LAST_BUNDLE_ID_SENT.txt: This file stores the bundle id of the last bundle sent to the server.

5. SENT_BUNDLE_DETAILS.json: SENT_BUNDLE_DETAILS stores information about the payload of the sent bundles. SENT_BUNDLE_DETAILS.json stores a mapping between bundle id of the sent bundle and a dictionary. For each sent bundle, the dictionary maintains a mapping between the application id and the largest counter of that application's ADUs which were present in the sent bundle's payload.

6. LAST_SENT_BUNDLE_STRUCTURE.json: A JSON file stores the structure of the last sent bundle payload which includes the range of ADU ids for each application, the bundle id and the value inside the acknowledgement record. Figure 8 shows an example. The "HB" acknowledgement suggests that the acknowledgement record of the last sent bundle contained a heartbeat message - "HB". The last sent bundle had ADUs for application A with ids ranging from 1 to 4. The bundle id of the last sent bundle in the example was "client0-0"

Figure 8: LAST_SENT_BUNDLE_STRUCTURE.json

4.3.3 Receiving Bundles from Bundle Transport

Figure 9 depicts the interaction of components for the flow when the client receives bundles from Bundle Transport. In step 1, Bundle Delivery Agent (BDA)

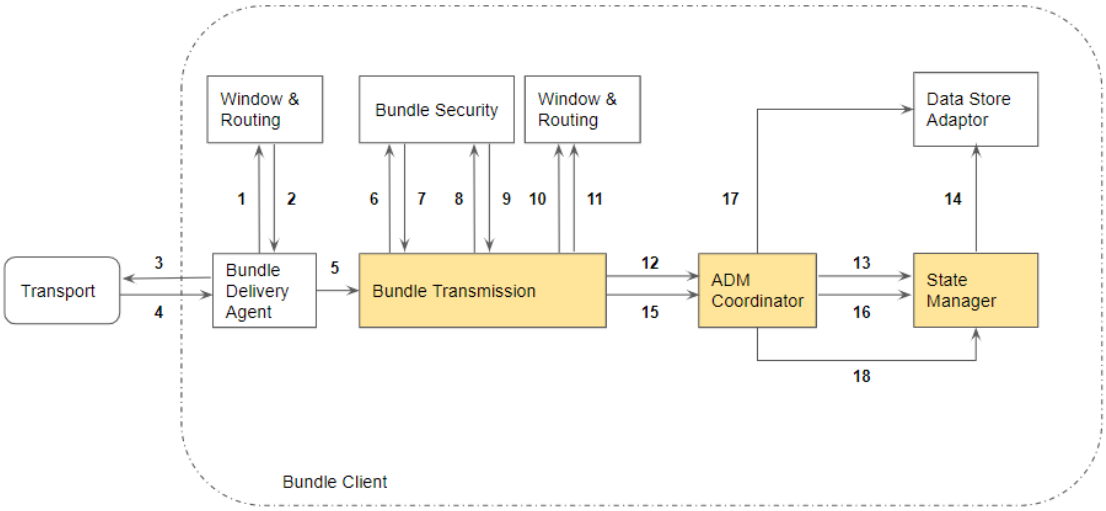

Figure 9: Bundle Client: Receive Bundles

upon detecting a transport requests Window & Routing (WR) for the bundle ids it should ask from the transport. In step 2, WR returns the window of 10 bundle ids to the BDA. In step 3, BDA requests for bundles corresponding to those 10 bundle ids from the transport. In step 4, The transport streams the corresponding bundle files back to the BDA. BDA requests Bundle Transmission (BT) to process the received bundles by providing the transport id and the location of the received bundles in step 5. For each bundle, BT carries out steps 6-18. BT fetches the bundle id with the largest counter received so far from the LARGEST_BUNDLE_ID_RECEIVED.txt file. In step 6, BT calls compareBundleIDs function of BS to compare this bundle id with the bundle id of the current bundle. In step 7, BT checks the value returned by compareBundleIDs call. If it is not 1, BT skips the next steps as the return value implies that this bundle has already been received. In step 8, BT requests Bundle Security (BS) to decrypt the bundle contents by calling the decrypt function and providing the location of the directory to place the decrypted payload. After BS returns in step 9, the path provided by BT has the decrypted payload. BT calls the

updateWindow function on WR by providing the bundle id for sliding the receiver window in step 10. In step 11, BT calls the updateMetaData function on WR by providing the transport id. After extracting the payload JAR file, BT calls ADM Coordinator (ADMC) to process the acknowledged bundle id in step 12. In step 13, ADMC further calls State Manager (SM) to process the acknowledged bundle id. In step 14, SM uses the acknowledged bundle id to determine the ADUs successfully delivered to the server and deletes those ADUs by calling deleteADUs function on DSA. SM also updates the LARGEST_ADU_ID_DELIVERED.json entries for each application whose ADU was present in the acknowledged bundle id. In step 15, BT requests ADMC to store the ADUs in the received bundle by providing the location of the ADUs and the bundle id. In step 16, before proceeding to store the ADUs, ADMC calls SM to: 1) Update the LARGEST_BUNDLE_ID_RECEIVED.txt with the current bundle id, 2) Get the largest ADU received per application so far from LARGEST_ADU_ID_RECEIVED.json so that the redundant ADUs are not stored again. In step 17, ADMC then calls DSA's persistADU function for each received ADU to store it. Finally, ADMC calls SM to update LARGEST_ADU_ID_RECEIVED.json in step 18.

4.3.4 Sending a Bundle to Bundle Transport

Figure 10 depicts the interaction of components for the flow when the client generates bundles for sending to a Bundle Transport. In step 1, Bundle Delivery Agent (BDA) upon detecting a transport calls the generateBundleForTransmission method of Bundle Transmission (BT) to trigger the bundle generation process. In step 2, BT requests ADM Coordinator (ADMC) to get the structure of the payload of the bundle last sent to the server. ADMC, in turn, calls State Manager (SM) which returns the structure of the last sent bundle in step 4, which ADMC returns to BT in

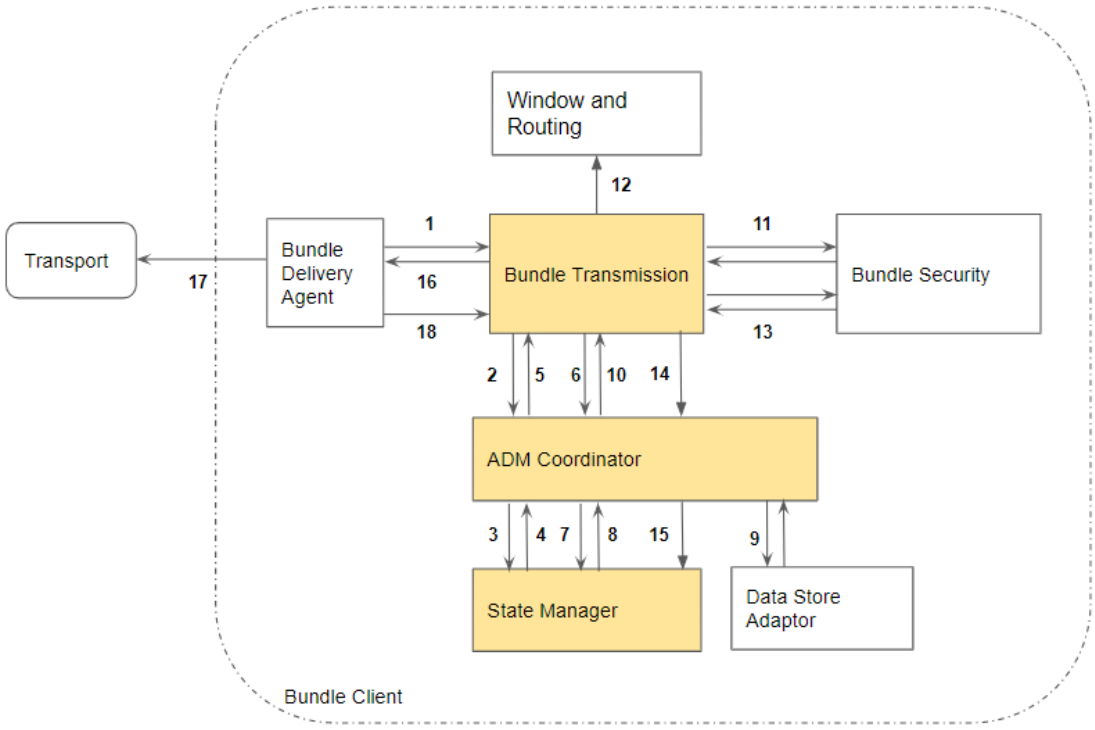

Figure 10: Bundle Client: Send Bundle

step 5. In steps 6-10, BT gets the structure of the new bundle that would be generated in memory. BT then checks if the new bundle payload would be the same as the last sent bundle payload. If the new bundle would have new data as compared to the last sent bundle, then BT generates a new bundle id in step 11. In step 12, BT calls WR's bundleMetaData function to add routing metadata to the bundle payload. BT then creates a payload JAR file and gets it encrypted from BS in step 13. During this step, the payload JAR is placed in a directory which is again compressed to a JAR by BT. In steps 14-15, BT notifies SM through ADMC about this newly generated bundle. In step 16, BT's generateBundleForTransmission execution ends notifying BDA of bundle generation. BDA then sends the bundles to the transport in step 17. After any given bundle is sent to the transport, BDA notifies BT about it at which point BT deletes the bundle from storage.

4.4 Bundle Server

The components described in section 4.2 are the building blocks of the bundle server. To understand the design of the bundle server, we need to understand how these components fit and work together to execute the functions of the bundle server. This section covers the design and implementation of the bundle server. Sections 4.4.1 and 4.4.2 cover implementation details which are put in context when describing the execution flows in Section 4.4.3.

4.4.1 Dependencies

This section covers the APIs of the components that BT, ADMC and SM interact with as well as the public APIs of BT that are used by the BSE to trigger the receive and send flows. The APIs of BS on the server are the same as the client-side APIs of BS covered in section 4.3.1.1.

4.4.1.1 Window and Routing

```
public void addClient(String clientId, int windowLength)
public void processClientMetaData(String clientMetaDataPath, String
    clientID)
public void updateClientWindow(String clientID, String bundleID)
public boolean isSenderWindowFull(String clientID)
```

The function addClient is called by BT to create a window for a given client the first time a bundle is generated for the client. The function processClientMetaData is called by BT to process the routing metadata in the received bundle. The function updateClientWindow is called by BT to add a bundle id to the server's sender window. The function isSenderWindowFull is used by BT to decide whether a new bundle id needs to be generated.

4.4.1.2 Data Store Adaptor

```
public void deleteADUs(String clientId, String appId, Long aduIdEnd)
public void persistADU(String clientId, ADU adu)
public List<ADU> fetchADUs(String clientId, String appId, Long aduIdStart)
```

The function deleteADUs is called by SM when processing an acknowledgement to request DSA delete ADUs upto a certain ID for a certain client and application. The function persistADU is called by ADM to request DSA store a ADU for a certain client when processing a received bundle. The function fetchADUs is called by ADM when generating a bundle for a client to request DSA for ADUs starting with a certain ID for that client and a specific application.

4.4.1.3 Bundle Transmission

```
public void processReceivedBundles()
public void notifyBundleSent(Bundle bundle)
public BundleTransferDTO generateBundlesForTransmission(String transportId,
    Set<String> bundleIdsPresent)
public List<Bundle> getBundlesForTransmission(String transportId)
```

The function processReceivedBundles is called by BSE when bundles are received from a Bundle Transport. The function notifyBundleSent is called by BSE to notify that a bundle has been sent to a Bundle Transport so that the bundles can be deleted from the server. The function generateBundlesForTransmission is called by BSE to trigger the generation of bundles for a given transport given the bundle ids already present. This function generates bundles for reachable clients and returns a wrapper object containing a set of bundle ids to be deleted from the transport. The function getBundlesForTransmission is called by BSE after calling the generateBundlesForTransmission function to get the list of bundles already generated for a

transport.

4.4.2 Database

On the server, the state of the communication between the bundle client and the server is stored in a MySQL database. Figure 11 shows the schema of the important tables. Following is the description of each of the tables:

```
-- RECEIVE FLOW TABLES
CREATE TABLE LARGEST_ADU_ID_RECEIVED (
    id VARCHAR(36),
    CLIENT_ID VARCHAR(100) NOT NULL,
    APP_ID varchar(100) NOT NULL,
    ADU_ID INT UNSIGNED NOT NULL,
    PRIMARY KEY(id)
);

CREATE TABLE LARGEST_ADU_ID_DELIVERED (
    ID VARCHAR(36),
    CLIENT_ID VARCHAR(100) NOT NULL,
    APP_ID varchar(100) NOT NULL,
    ADU_ID INT UNSIGNED NOT NULL,
    PRIMARY KEY(id)
);

CREATE TABLE LARGEST_BUNDLE_ID_RECEIVED (
    CLIENT_ID VARCHAR(100),
    BUNDLE_ID VARCHAR(100) NOT NULL,
    PRIMARY KEY(CLIENT_ID)
);
```

```
-- SEND FLOW TABLES
CREATE TABLE LAST_BUNDLE_ID_SENT (
    CLIENT_ID VARCHAR(100),
    BUNDLE_ID VARCHAR(100) NOT NULL,
    PRIMARY KEY(CLIENT_ID)
);

CREATE TABLE SENT_BUNDLE_DETAILS (
    BUNDLE_ID VARCHAR(100),
    CLIENT_ID VARCHAR(100),
    ACKED_BUNDLE_ID VARCHAR(100) NOT NULL,
    PRIMARY KEY(BUNDLE_ID)
);

CREATE TABLE SENT_ADU_DETAILS (
    ID VARCHAR(36),
    BUNDLE_ID VARCHAR(100) NOT NULL,
    APP_ID VARCHAR(100) NOT NULL,
    ADU_ID_RANGE_START INT UNSIGNED NOT NULL,
    ADU_ID_RANGE_END INT UNSIGNED NOT NULL,
    PRIMARY KEY(ID),
    CONSTRAINT fk_sent_bundle_details
    FOREIGN KEY (BUNDLE_ID)
        REFERENCES SENT_BUNDLE_DETAILS(BUNDLE_ID)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

Figure 11: Bundle Server Database Tables

1. LARGEST_ADU_ID_DELIVERED: Each record represents the largest ADU id delivered to a given client per application.

2. LARGEST_ADU_ID_RECEIVED: Each record represents the largest ADU id received from a given client per application.

3. LARGEST_BUNDLE_ID_RECEIVED: Each record represents the bundle id with the largest counter received from a given client.

4. LAST_BUNDLE_ID_SENT: Each record represents the bundle id of the last bundle sent to a given client.

5. SENT_BUNDLE_DETAILS and SENT_ADU_DETAILS: SENT_BUNDLE_DETAILS and SENT_ADU_DETAILS together store information about the payload of the sent bundles. Figure 11 shows the schema of these tables. SENT_BUNDLE_DETAILS stores the bundle id and the bundle id in the acknowledgement record inside the bundle. SENT_ADU_DETAILS stores the range of ADU ids for each application within a sent bundle. There is a one-to-many relationship between a record of SENT_BUNDLE_DETAILS and SENT_ADU_DETAILS. Therefore, there is a foreign key from the bundle id in SENT_ADU_DETAILS to SENT_BUNDLE_DETAILS.

4.4.3 Receiving Bundles from Bundle Transport

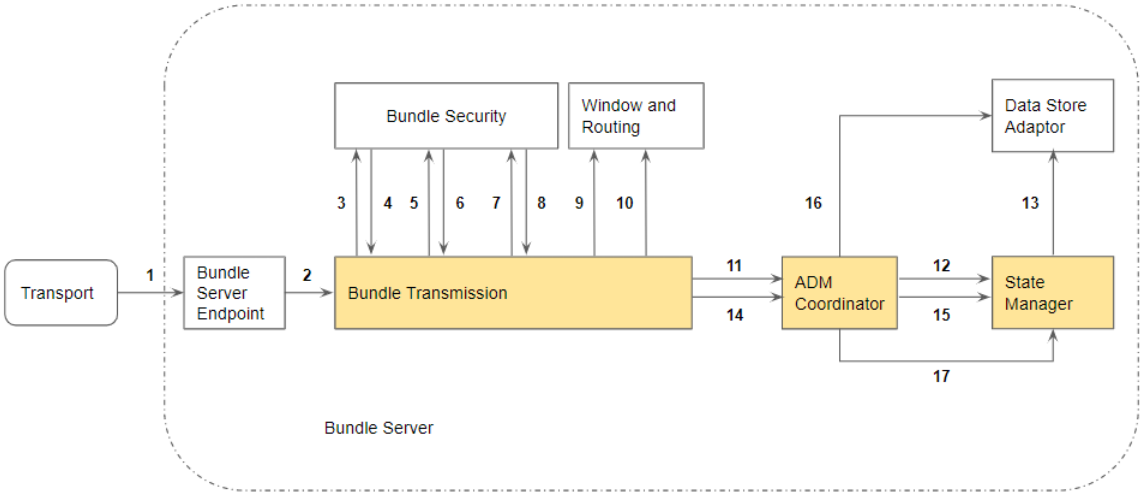

Figure 12: Bundle Server: Receive Bundles

Figure 12 depicts the interaction of components for the flow when the server receives bundles from the Bundle Transport. In step 1, A transport sends all the bundles it carries to Bundle Server Endpoint (BSE) which are sent to Bundle Transmission (BT) in step 2. In step 3, BT requests the client id for the received bundle id from BS which returns the client id in step 4. BT fetches the bundle id with the largest counter

received so far from this client from the LARGEST_BUNDLE_ID_RECEIVED table. In step 5, BT calls compareBundleIDs function of BS to compare this bundle id with the bundle id of the current bundle. In step 6, BT checks the value returned by compareBundleIDs call. If it is not 1, BT skips the next steps as the return value implies that this bundle has already been received. In step 7, BT requests Bundle Security (BS) to decrypt the bundle contents by calling the decrypt function. After step 8, BT expects the decrypted payload at the location it passed to the decrypt function. In step 9, BT sends the bundle id in the acknowledgement of the received bundle to Window & Routing (WR) for sliding the sender window. In step 10, BT calls the processMetadata function on WR to process the routing metadata in the bundle. In step 11, After extracting the payload JAR file, BT calls ADM Coordinator (ADMC) to process the acknowledged bundle id. In step 12, ADMC further calls State Manager (SM) to process the acknowledged bundle id. In step 13, SM uses the acknowledged bundle id to determine the ADUs successfully delivered to the client and deletes those ADUs by calling deleteADUs function on DSA. SM also updates all the entries for the client in the LARGEST_ADU_ID_RECEIVED table for each application whose ADU was in the acknowledged bundle id. In step 14, BT requests ADMC to store the ADUs in the received bundle by providing the location of the ADUs and the bundle id. In step 15, before proceeding to store the ADUs, ADMC calls SM to: 1) Update the LARGEST_BUNDLE_ID_RECEIVED table for the client with the current bundle id, 2) Get the largest ADU received from this client so far from the LARGEST_ADU_ID_RECEIVED table so that the redundant ADUs are not stored again. In step 16, ADMC then calls DSA's persistADU function for each received ADU to store it. Finally, ADMC calls SM to update the LARGEST_ADU_ID_RECEIVED table in step 17.

4.4.4 Sending Bundles to Bundle Transport

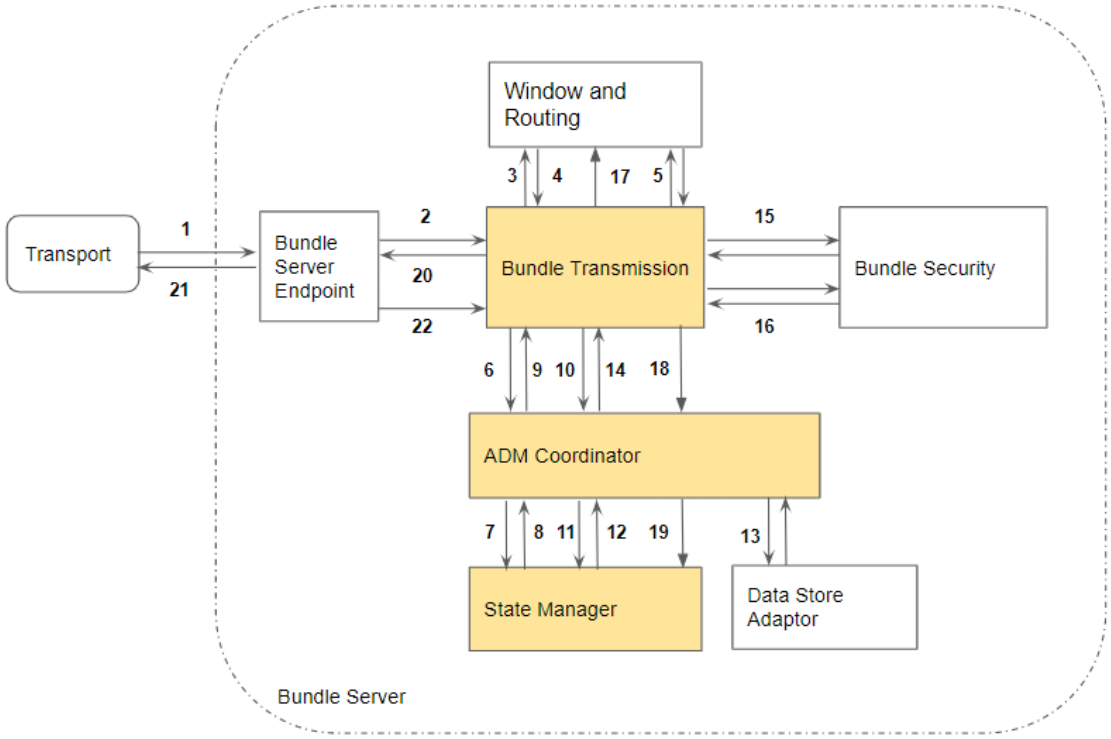

Figure 13: Bundle Server: Send Bundles

Figure 13 depicts the interaction of components for the flow when the server generates bundles for sending to a Bundle Transport. Bundle Server Endpoint (BSE) upon receiving a request for bundles from a transport in step 1 calls the generateBundleForTransmission method of Bundle Transmission (BT) to trigger the bundle generation process in step 2. BT gets the list of clients reachable from the transport in step 4 by calling the getClients method on Window & Routing (WR) and passing in the transport id in step 3. Steps 5-19 are carried out for each of those clients. In step 5, BT queries the isSenderWindowFull method of WR to check if the sender window for a client is full. In step 6, BT requests ADM Coordinator (ADMC) to get the structure of the payload of the bundle last sent to the client. ADMC, in turn, calls State Manager (SM) in step 7 which returns the structure of the last sent

bundle in step 8, which ADMC returns to BT in step 9. In steps 10-14, BT gets the structure of the new bundle that would be generated in memory. BT then checks if the new bundle payload would be the same as the last sent bundle payload. If the new bundle would have new data as compared to the last sent bundle, then BT generates a new bundle id in step 15. BT then creates a payload JAR file and gets it encrypted from BS in step 16. During this step, the payload JAR is placed in a directory which is again compressed to a JAR by BT. In step 17, BT calls the updateClientWindow method in WR by passing in the bundle id so that the current bundle id is added to the sender window. In steps 18-19, BT then notifies SM through ADMC about this newly generated bundle. In step 20, BT‘s generateBundleForTransmission execution ends notifying BSE of bundle generation. BSE then sends the bundles to the transport in step 21. After any given bundle is sent to the transport, BSE notifies BT about it in step 22, at which point BT deletes the bundle from storage.

CHAPTER 5

Experiments

Implementation of the data delivery mechanism described in this project was tested to verify that it works as expected under various scenarios. This chapter describes the details of some of the experiments that were conducted to test the implementation.

5.1 Experiment setup

A bundle client and bundle server running on the command line was implemented by creating mock implementations of all modules other than BT, SM, and ADMC. Although the implementation has been tested for multiple applications, multiple clients, and multiple transport ids, the results considering only a single application with the name 'A' and a single client with client id – client0 are presented for ease of understanding. The arrival of transport at the bundle client and server is simulated by calling the processReceivedBundles and generateBundleForTransmission functions of BT. The mock implementation of the DSA stores ADUs in a folder structure as

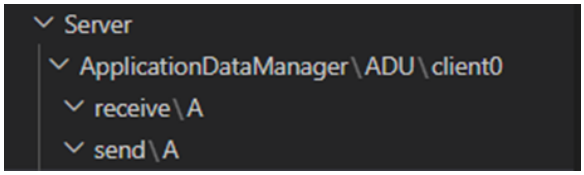

Figure 14: Server folder structure for storing ADUs.

shown in Figure 14 on the server. As we can see in Figure 14, there is a folder per client in Server/ApplicationDataManager/ADU. Each client folder has a 'receive' and 'send' folder to represent storage for ADUs received from and ADUs to be sent to an Android application running on the disconnected client's phone respectively. Both folders have a sub-folder per application (A in this case). The mock implementation of DSA on the client side has a similar structure except that it does not have a client

directory in Client/ApplicationDataManager/ADU.

5.2 Reliable Delivery

Consider the following sequence of events:

1. The bundle client sends a bundle containing one ADU A-1 to the server. The bundle id of the sent bundle is client0-0.

2. The bundle does not make it to the server. Therefore, the receive folder on the server for client0 and application A will be empty. Figure 15 shows the receive folder at the start of the experiment. It is empty to simulate the scenario.

3. The bundle client receives ADUs A-2 and A-3.

4. The arrival of a transport at the bundle client is simulated by triggering a bundle generation at the client.

5. The bundle client generates a bundle containing A-1, A-2, and A-3 and assigns it a new bundle id client0-1 since it has new data. A-1 is still in the bundle since no acknowledgement was received for the bundle client0-0 containing A-1.

6. The transport arrives at the bundle server and delivers the bundle client0-1 to the server.

7. The bundle server unpacks the bundle payload and ADUs are stored in the mock DSA's storage. Figure 16 shows the result of this experiment. As we can see in the figure, all ADUs A-1, A-2, and A-3 are present in the receive/A folder which is as expected.

We can see that the ADU A-1 was successfully delivered even though client0-0 was lost because it is sent in the subsequent bundle till an acknowledgement for client0-0 is received. In general, as an ADU is retransmitted in all the bundles till an

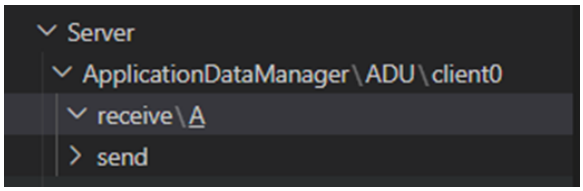

Figure 15: Server's receive directory before the experiment

acknowledgement is received and it is assumed that some bundle will eventually be delivered, an ADU is always reliably delivered to the receiver.

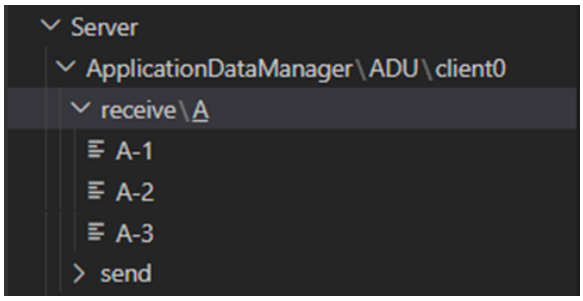

Figure 16: Server's receive directory after the experiment

5.3 In-order Delivery

Consider the following sequence of events:

1. A transport T1 arrives at the bundle server and requests for bundles.

2. The bundle server packs in A-1 and A-2 in a bundle with bundle id client0#0 and sends it to the transport.

3. Another transport T2 arrives at the bundle server. The bundle server generates a bundle containing A-1, A-2, and A-3 and assigns it a new bundle id client0#1 since it has new data. A-1 is still in the bundle since no acknowledgement was received for the bundle client0#0 containing A-1.

4. T2 arrives at the bundle client. At this point, the bundle client's receive directory is empty. The bundle client's flow for receiving bundles from the transport is

triggered. The bundle client receives client0#1 and stores the ADUs A-1, A-2, and A-3 as shown in Figure 17.

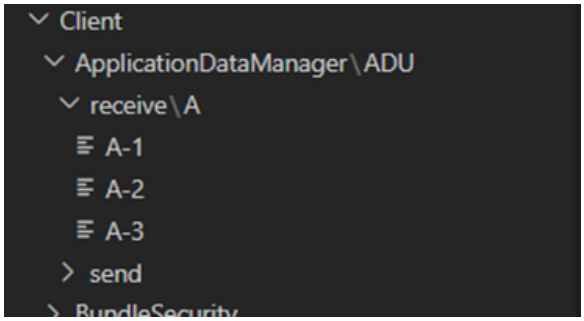

Figure 17: Client's Receive directory after receiving client0#1

5. T1 arrives at the bundle client. The bundle client's flow for receiving bundles from the transport is triggered. As shown in Figure 18, the bundle client discards the bundle with bundle id client0#0 received from T1 which is as expected.

Even though the bundles sent by the bundle server were received out of order by the bundle client, the ADUs A-1, A-2, and A-3 are received in order as expected. If the bundle client0#1 did not have ADUs present in the bundle client0#0, the ADUs would have been received at the client out of order making it difficult to deliver the ADUs to A in the order sent by the server. The in-order delivery of ADUs is thus made possible because at all times a prefix of undelivered ADUs is sent in each bundle.

```
Starting Bundle Client!
[BDA] Starting Bundle Delivery Agent
[BS] Largest bundle id received so far: client0#1
[BT] Skipping bundle with bundle id = client0#0 This is not the latest bundle sent by the server
```

Figure 18: Client discards the outdated bundle

5.4 Deduplication

Consider the following sequence of events:

1. The bundle client generates a bundle with bundle id client0-0 containing ADU A-1 which is received by the server via a transport. As shown in Figure 19, the server's receive folder for the client contains A-1.

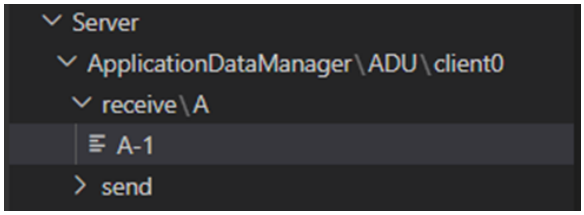

Figure 19: Server's receive folder before the experiment

2. The bundle server also inserts a record to the LARGEST_ADU_ID_RECEIVED table as shown in Figure 20.

```
mysql> SELECT *  FROM LARGEST_ADU_ID_RECEIVED;
+--------------------------------------+-----------+--------+--------+
| id                                   | CLIENT_ID | APP_ID | ADU_ID |
+--------------------------------------+-----------+--------+--------+
| 99b74b9e-49e2-4816-8e9c-3b7004137547 | client0   | A      |      1 |
+--------------------------------------+-----------+--------+--------+
1 row in set (0.00 sec)
```

Figure 20: LARGEST_ADU_ID_RECEIVED table before the experiment

3. After receiving a new ADU A-2 from the application, the bundle client also sends a new bundle client0-1 which is successfully received by the bundle server. When processing the ADUs in the bundle, the bundle server checks the LARGEST_ADU_ID_RECEIVED and notices that the ADU A-1 has already been received. Therefore, the bundle server skips storing the ADU A-1 as shown in the server logs in Figure 21.

```
[ADM] Skipping ADU of app = A, id = 1from client = client0 as it has been already received
[DSA] Stored ADUs for application A for client client0 at C:\Masters\CS 297-298\CS 298\Implementation\
2023-04-09T16:26:46.955-07:00  INFO 4856 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource
2023-04-09T16:26:46.959-07:00  INFO 4856 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource
```

Figure 21: Server skips processing the ADU A-1

4. The bundle server updates LARGEST_ADU_ID_RECEIVED entry as we can see in Figure 22.

```
mysql> SELECT *  FROM LARGEST_ADU_ID_RECEIVED;
+--------------------------------------+-----------+--------+--------+
| id                                   | CLIENT_ID | APP_ID | ADU_ID |
+--------------------------------------+-----------+--------+--------+
| 99b74b9e-49e2-4816-8e9c-3b7004137547 | client0   | A      |      2 |
+--------------------------------------+-----------+--------+--------+
1 row in set (0.00 sec)
```

Figure 22: LARGEST_ADU_ID_RECEIVED table after the experiment

5. The bundle server stores the remaining ADU A-2 in the send folder as we can see in Figure 23. As we can see, the receive folder does not have a copy of A-1, which is as expected in the deduplication guarantee.

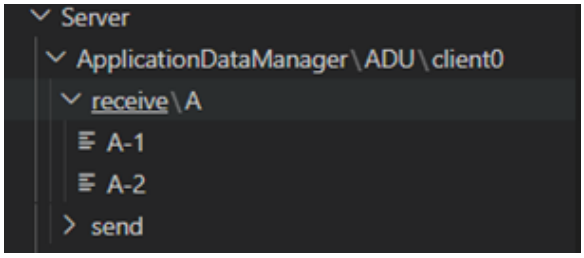

Figure 23: Server's receive folder after the experiment

5.5 Bundle Deletion from Bundle Transport

Consider the following sequence of events:

1. The bundle server sends a bundle client0#0 to the bundle client via transport0. Details of the bundle are shown in Figure 24.

```
mysql> select * from sent_adu_details;
+--------------------------------------+-----------+--------+--------------------+--------------------+
| ID                                   | BUNDLE_ID | APP_ID | ADU_ID_RANGE_START | ADU_ID_RANGE_END   |
+--------------------------------------+-----------+--------+--------------------+--------------------+
| d3620cb7-a126-4b1a-8ff5-2ecfe857cf59 | client0#0 | A      |                  1 |                  2 |
+--------------------------------------+-----------+--------+--------------------+--------------------+
1 row in set (0.00 sec)
```

Figure 24: Details of the bundle sent before the delete flow

2. After sending client0#0, the server gets another ADU from A's application server. The send directory of the bundle server is shown in Figure 25.

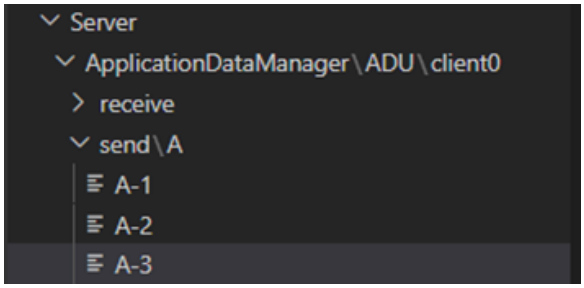

Figure 25: Server's send folder before the delete flow

3. The transport arrives at the bundle server and triggers the generateBundlesFor-Transmission function by passing the set of bundle ids it already has. This set contains client0#0.

4. The transport did not go to the bundle client with the previous bundle. Therefore, it does not bring any acknowledgement for client0#0 from the bundle client to the bundle server. Therefore, the new bundle generated by the server with A-3 has A-1 and A-2. Since this is a more recent version of client0#0, it needs to be deleted

5. The bundle server must add client0#0 to the deletion set that is returned to the transport and then send the bundle client0#1. As we can see in the server logs shown in Figure 26, the result is as expected.

```
[BundleServerApp] Following are the bundle ids that the transport transport-0 should delete
client0#0
[BundleServerApp] Following are the ids of the bundles to be sent to transport transport-0
client0#1
2023-04-09T19:16:41.541-07:00  INFO 32096 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
2023-04-09T19:16:41.553-07:00  INFO 32096 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

Figure 26: Server logs showing the bundle to be deleted

CHAPTER 6

Future Work

In the bundle client and bundle server, when a transport arrives and a bundle needs to be generated for transmission, currently a bundle is created locally, it is copied to the transport, and then deleted from the local storage of the sender upon successful transmission. In this case, we are storing copies of ADUs i.e., an ADU that is stored by the DSA is also stored in a bundle. However, it is important to optimize space as we want the bundle client to occupy limited space on the disconnected user's Android phone. This problem can be solved by creating a bundle file directly on the bundle transport on-the-fly with the help of streaming without creating the bundle locally on the client.

In this project, the size limit for a bundle is fixed to an arbitrary value of 100MB for all clients. Further experiments can be conducted to determine a suitable bundle size limit. The limit for the total size of all ADUs corresponding to an application that can be added to a bundle is also fixed to an arbitrary value of 30 MB. Moreover, the limit is the same for all applications. In the future, there should be an algorithm to decide a size limit for each application taking into consideration the fact that different applications have different ADU size distributions while also avoiding starvation of certain applications. For example, applications that need to send videos will have larger ADUs as compared to applications that send text messages in general.

Currently, the part of the bundle client in the scope of this project has been implemented in Java. Since this implementation will work with an Android Bundle Client application, there might be slight changes in the code to work with the Android SDK. Moreover, if the Bundle Client application needs to run on a platform without a Java-based SDK, the data delivery mechanism implemented as part of this project will have to be implemented in a different language that is supported by the platform.

CHAPTER 7

Conclusion

A huge part of the world is disconnected from the internet due to a lack of infrastructure. Parts of the world that have the necessary infrastructure might also lose connectivity due to disasters. The population in disconnected areas does not have access to the information available on the internet. Since the internet is an indispensable part of our lives today, it is important to bring the data of the internet to disconnected areas in the absence of end-to-end internet connectivity. DDD attempts to solve this problem by creating a software-only infrastructure for transporting data between the phones of users in disconnected areas and the internet. The infrastructure exchanges data between Android applications on a disconnected phone to the corresponding application servers on the internet through an intermediate carrier phone which travels between the disconnected area and an area with internet connectivity.

We can broadly break the DDD endpoints into two parts: 1) The part that interfaces with the applications and 2) The part that delivers the application data. The physical network carrying the data in DDD is assumed to be asynchronous, unreliable, high-latency, and high-bandwidth, unlike the internet. Although the network constraints are like those in a DTN, the DTN architecture is not suitable for providing delivery guarantees. The goal of this project was to implement the second part with a set of guarantees that the first part can build upon. Determining the set of guarantees and how they can be implemented so that the DDD infrastructure is deterministic from an application's point of view even though the underlying network is unpredictable was a challenging task. The mechanism for application data packaging and transmission developed in this project enables the transmission of application data between a disconnected client and application servers such that it is resilient to

loss, reordering, duplication, and corruption. The important idea that enabled the delivery guarantees is that the second part bundles an acknowledgement and a prefix of this sequence of undelivered ADUs. After overcoming all the roadblocks, the 3 delivery guarantees i.e. reliable, in-order, and deduplicated delivery of application data were successfully implemented in this project.

The solution developed as part of this project is central to the DDD infrastructure. Although there are potential enhancements to the implementation and more work to be done, this project is a huge first step in the full realization of DDD and bringing the world closer to full internet connectivity.

Bibliography

[1] https://www.un.org/en/delegate/itu-29-billion-people-still-offline Accessed: 2022-12-17

[2] A. Pentland, R. Fletcher, and A. Hasson, "DakNet: rethinking connectivity in developing nations," in Computer, vol. 37, no. 1, pp. 78-83, Jan. 2004, doi: 10.1109/MC.2004.1260729.

[3] S. Guo et al., "Design and implementation of the KioskNet system," vol. 55, no. 1, pp. 264–281, 2011, doi: 10.1016/j.comnet.2010.08.001.

[4] S. Burleigh et al., "Delay-tolerant networking: an approach to interplanetary Internet," vol. 41, no. 6, pp. 128–136, 2003, doi: 10.1109/MCOM.2003.1204759.

[5] R. C. Shah, S. Roy, S. Jain, and W. Brunette, "Data MULEs: modeling and analysis of a three-tier architecture for sparse sensor networks," vol. 1, no. 2, pp. 215–233, 2003, doi: 10.1016/S1570-8705(03)00003-9.

[6] K. Fall, "A Delay-Tolerant Network Architecture for Challenged Internets,"in Proc. ACM SIGCOMM '03. New York, NY, USA: ACM Press, 2003, pp. 27–34.

[7] M. Ramadas, S. C. Burleigh and S.Farrell. Licklider Transmission Protocol - Specification. RFC 5326, Sep. 2008.

[8] K. Scott and S. C. Burleigh. Bundle Protocol Specification. RFC 5050, Nov. 2007.

[9] M. Demmer and J. Ott, "Delay tolerant networking TCP convergence layer protocol", IETF DTNRG IRTF Research Group, <draft-irtf-dtnrgtcp-clayer-09.txt>, Mar. 2014, [Online]. Available: http://www.ietf.org/internet-drafts/draft-irtf-dtnrg-tcp-clayer-09.txt

[10] H. Kruse and S. Ostermann, “UDP convergence layers for the DTN bundle and LTP protocols”, IETF DTNRG IRTF Research Group, <draft-irtf-dtnrg-udpclayer-00.txt> (Work in Progress), Nov. 2008, [Online]. Available: http://tools.ietf.org/html/draft-irtf-dtnrg-udp-clayer-00

[11] S. Burleigh, “Delay-tolerant networking LTP convergence layer (LTPCL) adapter”, IETF Internet-Draft, <draft-burleigh-dtnrg-ltpcl-05>, Apr. 2013, [online]. Available: http://tools.ietf.org/pdf/draft-burleigh-dtnrg-ltpcl-05.pdf

[12] K. Fall, W. Hong, and S. Madden, "Custody Transfer for Reliable Delivery in Delay Tolerant Networks," 2003.

[13] E. Koutsogiannis, F. Tsapeli, and V. Tsaoussidis, “Bundle Layer End-to-end Retransmission Mechanism,” 2011, pp. 109–115, doi: 10.1109/BCFIC-RIGA.2011.5733233.

[14] N. Bezirgiannidis, S. Burleigh, and V. Tsaoussidis, “Delivery time estimation for space bundles”, IEEE Trans. Aerosp. Electron. Syst., vol. 49, no. 3, pp. 1897–1910, Jul. 2013.

[15] G. Yang, R. Wang, A. Sabbagh, K. Zhao, and X. Zhang, “Modeling Optimal Retransmission Timeout Interval for Bundle Protocol,” vol. 54, no. 5, pp. 2493–2508, 2018, doi: 10.1109/TAES.2018.2820398.

[16] R. Wang, A. Sabbagh, S. C. Burleigh, K. Zhao, and Y. Qian, “Proactive Retransmission in Delay/Disruption-Tolerant Networking for Reliable Deep-Space Vehicle Communications,” vol. 67, no. 10, pp. 9983–9994, 2018, doi: 10.1109/TVT.2018.2864292.

[17] R. Wang, M. Qiu, K. Zhao, and Y. Qian, "Optimal RTO Timer for Best Transmission Efficiency of DTN Protocol in Deep-Space Vehicle Communications," vol. 66, no. 3, pp. 2536–2550, 2017, doi: 10.1109/TVT.2016.2572079.

[18] G. Papastergiou, I. Alexiadis, S. Burleigh, and V. Tsaoussidis, "Delay Tolerant Payload Conditioning protocol," vol. 59, pp. 244–263, 2014, doi: 10.1016/j.bjp.2013.11.003.

[19] J.H. Saltzer, D.P. Reed, D. Clark, "End-to-end arguments in system design", ACM Transactions on Computer Systems (TOCS) 2 (4) (1984) 277–288.