A Mobile-First Disconnected Data Distribution Network

1st Shashank Hegde
Computer Science Department
San Jose State University
San Jose, California
shashank.hegde@sjsu.edu

2nd Deepak Munagala
Computer Science Department
San Jose State University
San Jose, California
deepak.munagala@sjsu.edu

3rd Aditya Singhania
Computer Science Department
San Jose State University
San Jose, California
aditya.singhania@sjsu.edu

4th Ben Reed
Computer Science Department
San Jose State University
San Jose, California
ben.reed@sjsu.edu

Abstract—**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 an additional expenditure on installation. Instead of acquiring special hardware, we take advantage of the ubiquity of Android smartphones and build a secure, software-only infrastructure on top of it called Disconnected Data Distribution (DDD) to exchange application data between disconnected Android phones and corresponding Internet services. Although DDD draws ideas from Delay Tolerant Networking (DTN), we show how our internet data distribution approach simplifies the DTN architecture while providing stronger guarantees than standard DTN approaches. We design DDD to be inexpensive to deploy and maintain, so it takes an entirely software-only approach to build the infrastructure. The data transporters and clients all run on unmodified Android phones.**

Index Terms—**mobile computing, village network, delay-tolerant network, Android, Wi-Fi Direct, network delivery guarantees**

I. Introduction

Over the past few decades, the Internet has become increasingly essential in our day-to-day tasks. The Internet is heavily used for information dissemination. Many routine aspects of life, such as e-commerce, electronic payments, social media, email, and navigation, depend on Internet connectivity. Digital connectivity also fulfills social, economic, and even emotional needs. 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. 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]. Another report by United Nations International Children's Emergency Fund (UNICEF) claimed that two-thirds of the world's children lack internet access, which negatively impacts their learning capability and access to education [2]. According to an article by the International Monetary Fund (IMF) [3], low internet connectivity has led to widening income inequality both within and between countries, and less than half the population in developing countries has internet access.

There have been efforts in the past to solve these problems with delay-resilient networks known as a **Delay Tolerant Network (DTN)** [4]. It allows devices in disconnected regions to gain access to information on the Internet. This is a store-and-forward approach that uses intermediate devices, known as transports, for physically moving data between the client device in the disconnected region and the source in the connected region containing information.

In this paper, we focus on building an infrastructure called **Disconnected Data Distribution (DDD)** for bringing data from the Internet to disconnected areas to enable connectivity in disconnected areas. DDD has the following goals:

- **Software-only solution:** This solution will be entirely software-based, using standard Android phones and cloud-based servers. We leverage the ubiquity of Android phones rather than using additional hardware.
- **Support for current internet services:** We want to bring the services that the rest of the world uses to disconnected areas. For example, we want to transparently integrate with existing messaging, video, and information services.
- **Delivery guarantees:** We assume that the data we send to and from disconnected areas can be lost, corrupted, or delivered out of order. To overcome this, applications that use the DDD architecture can expect certain delivery guarantees like deduplication, in-order delivery, and reliable delivery of application data.
- **Security:** We assume that the data we send to and from disconnected areas can be mutated, fabricated, replayed, and collected by malicious actors in the infrastructure. The DDD infrastructure will support end-to-end encryption of exchanged application data to preserve privacy. The devices carrying the data cannot see the source or destination of the data they are transporting.

In this paper, we describe DDD and how it implements its goals of a software-only solution, support for current internet services, and delivery guarantees. We will not go into the details of how DDD achieves the security goal.

In this paper, we test our DDD implementation by using the open-source Signal Android application, a highly popular messaging application with over 100 million downloads as of July 2023 [5]. Signal is an open-source multi-platform instant messaging service that strongly emphasizes end-to-end encryption. Our modifications to Signal Android to use DDD show the feasibility of this approach and provide a design that other applications can follow to enable connectivity in disconnected regions.

II. Related Works

Previous attempts to set up an infrastructure to enable data exchange between disconnected and connected areas have required special hardware which requires additional costs [6] [7]. DakNet [6] enables low-cost digital communication by transmitting data over short distances between kiosks located across villages using portable storage devices mounted on modes of transport and using cheap Wi-Fi radio transceivers for data transfer. When the device is near a kiosk, it performs a high bandwidth data synchronization thereby providing a high data throughput using a store and forward delay tolerant infrastructure.

S. Guo et al. introduced the KioskNet system [7] 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. KioskNet focuses on ferries that like MAPs in DakNet also require customized hardware in the form of a single-board computer with around 20-40 GB of storage. 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 [8]. 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, less convincing will be required.

For applications to use the Internet, a network should be able to provide good **delivery guarantees**. Bundle Protocol [9] is the protocol used in Delay-Tolerant Network (DTN) architectures. Custody transfer [10] is a mechanism where a bundle node transfers the responsibility of reliable delivery of data to a bundle node further along the path and deletes its copy of the data. Bundle Layer End-to-End Reliability (BLER) [11] uses a combination of custody transfer with end-to-end acknowledgment for reliable delivery of data. BLER attempts transmission with custody transfer enabled. After the data reaches its destination, the destination sends an end-to-end acknowledgment record to the sender. Delay Tolerant Payload Conditioning (DTPC) [12] provides a transport layer for DTNs over the Bundle Protocol providing certain delivery guarantees like reliable delivery, in-order delivery, and deduplication on its protocol data unit. Rather than using the bundle protocol and above mentioned delivery mechanisms that are built on top of it, we have developed a novel data delivery mechanism with certain delivery guarantees by building on the insights offered by the above approaches.

III. DDD Overview

DDD takes advantage of the structure of modern internet services. With the advent of NAT firewalls and mobile clients, internet services have evolved to a model of servers at well-known network locations and clients at arbitrary locations. Fig. 1 shows an example of such a service. The client may move to arbitrary locations on the network, but the mail server is always accessible from a fixed location.

We extend this model to the disconnected environment by extending the mobility of clients. We take advantage of people moving between connected and disconnected regions, called **transports**, to transport data to and from disconnected clients. Examples of such people include bus drivers, train conductors, truck drivers, commuters, postal workers, etc. To transport DDD data, transports need only to install a DDD transport app on their Android phone. They do not need any 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.

As the devices in the disconnected region need to communicate with each other without using the Internet, we look at short-range wireless methods using built-in **Wi-Fi Direct** functionality to establish temporary Wi-Fi connections between transports and disconnected users without requiring any additional hardware. We prefer Wi-Fi Direct as multiple clients can connect wirelessly and transfer data simultaneously. The only hardware required in disconnected areas is standard Android phones. The users and transports install our apps as standard Android applications which implement the delivery service offered by our infrastructure.

Data security is one of the goals of DDD. All data transmissions are end-to-end encrypted. A client identifies itself using a public key that it generates. This public key is the basis for end-to-end encryption for data sent from the client to the service on the Internet. This encryption, the details of which are not in the scope of this paper, allows us to preserve the confidentiality of the data that flows through the DDD architecture. We also keep the identity of clients hidden from the transport by not using persistent addresses for clients or servers. All data sent by a client will go to the server. Rather than send data to a particular client, the server will generate data with an ID in a pseudorandom sequence that the client expects. Because a particular client is always looking for a random id, transport will not be able to know that the client it interacts with in one exchange is the same client in another exchange.

IV. DDD Architecture

In DDD, the unit of data transmission between devices 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. Section IV-A talks about bundles in more detail. Fig. 2 depicts the DDD architecture.

The applications that form our infrastructure are: 1) Bundle Transport, 2) Bundle Client, and 3) Bundle Server. Bundle Transport is the Android application that will be installed by transports. Bundle Client is an Android application that clients,

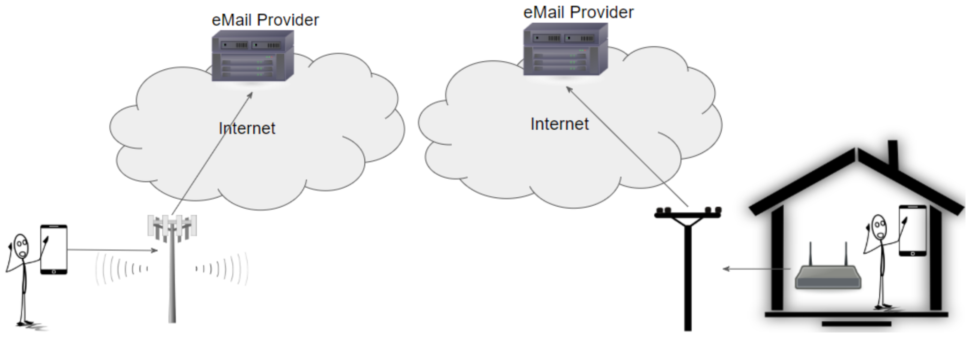


Fig. 1: The left figure shows a client using an email provider over the cellular network. On the right, the client uses home WiFi. The client will be using different IP addresses but still expects the same service.

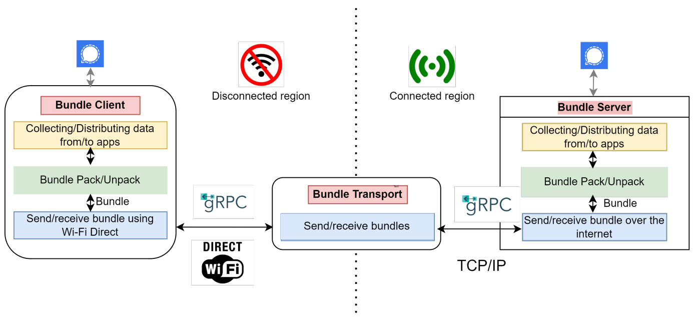


Fig. 2: DDD Architecture

i.e. disconnected users, will install. 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 and receive data to the latter using gRPC over Wi-Fi Direct. When Bundle Transport reaches an area with internet connectivity, it sends the data to the Bundle Server over TCP/IP. Bundle Server is a gRPC server that unpacks the packages and routes the received application data to the application servers. It also packages application data for clients to send to the Bundle Transport, which can deliver the packages to the Bundle Clients for which the data is intended.

A. Bundle

In DDD, we collect and store data from applications in chunks which we call **application data units (ADUs)**. A DDD ADU is a self-contained atomic chunk of application data. This might correspond to data sent between applications and their application servers. One or more ADUs from various applications are encapsulated in a packet which we call a **bundle**. DDD adopts the terms 'bundle' and 'ADU' from the Bundle Protocol [9]. A bundle has a unique **bundle ID** which is a combination of the client's identifier and a counter. It is unique across all clients. Note that given a bundle ID generated by a given Bundle Client, only the Bundle Server can get the client ID and counter from the bundle ID. Details of ID generation overlap with how DDD achieves the security goal which we will not cover.

A bundle is a zip file consisting of a few files which constitute the bundle header and another encrypted zip file for the payload. The bundle header consists of files for the bundle ID and information for decrypting the bundle payload. The bundle payload consists of an acknowledgment record, a set of ADUs of one or more applications, and some information required for routing in a zip file. Each ADU is assigned a unique identifier **ADU_ID**. An ADU also has an **application ID** linked to it, which is unique for each application on the Android OS.

Bundle Packing is the process of generating the bundle payload zip, encrypting it, and packing it with the bundle header into a zip file which is the bundle. **Bundle Unpacking** is the inverse of the Bundle Packing process. During Bundle Packing, all ADUs that are not known to be received by the receiver are included in the payload. Consider the following example: if a sender packs ADUs a1 and a2 of an application in a bundle with bundle ID B1. Next time a transport arrives, ADUs a1 and a2 will again be packed with any new ADU a3 in the next bundle with bundle ID B2. ADUs a1 and a2 will be packed in all new bundles till an acknowledgment record for bundle B1 is received. All subsequent bundles after receiving the acknowledgment will have ADUs starting from a3. This applies to ADUs of all applications. We cover acknowledgment records in more detail in section IV-E. Details of our approach for encryption/decryption of the bundle payload during Bundle Packing are beyond the scope of this paper. We place limits on the size of a bundle and the total size of ADUs of a given application. We fix the limits to arbitrary values for the first version of DDD. However, in the future, there might be strategies to determine the appropriate size limits.

B. 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 IV-B1 and IV-B2 cover flows A and B respectively.

1) Receiving Bundles from the Bundle Transport: All Bundle Clients in the disconnected region maintain a window of bundle IDs that they can receive from the Bundle Server, which is known as **receive window**. The Bundle Clients maintain a receive window of 10 counters which correspond to the Bundle Server's **sender window** for the client [1]. The Bundle Server maintains a sender window of 10 counters for each Bundle Client. In this way, 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 Bundle Client requests the Bundle Transport application for bundles sent for it by providing these 10 bundle IDs. Consider the example shown in

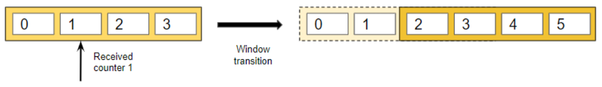


Fig. 3: Receive Window

Fig. 3. The client's receive window initially consists of bundle IDs with counters 0, 1, 2, and 3. In this example, we consider a window of size 4 for simplicity. 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 Bundle 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 $x+1 \leq c \leq x+10$. 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 $< x+1$.

When the Bundle Client receives a bundle, it does Bundle Unpacking to get an acknowledgment record and ADUs. ADUs are then pushed to the applications on the device with the help of Android **Intents**. In DDD, we send data from the Bundle Client to other applications by specifying the application ID, which is unique for each application on the Android OS.

Bundle Client uses ADU_ID to identify duplicate ADUs received and discard them. Moreover, if bundles are received out-of-order, i.e. a bundle having a bundle ID with a higher counter is received before a bundle having a bundle ID with a lower counter value, the receiver would have already received all ADUs present in the bundle with a lower counter value from the bundle with larger counter. Consider the example in IV-A. The bundle with bundle ID B2 has all the ADUs (a1 and a2) that were packed in the bundle with bundle ID B1. Therefore, if the bundle B2 is received before the bundle B1, the receiver can safely discard the latter. Thus, the incremental nature of bundle packing helps ensure that deduplication and in-order delivery of ADUs are guaranteed by DDD.

[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

2) Sending a Bundle to the Bundle Transport: When a DDD-enabled application wants to send some data to its server, and it detects that there is no internet connection, it will send the data to the Bundle Client with the help of a **content provider** [13] with a CRUD interface, which is hosted by the Bundle Client. The Bundle Client makes sure that any data that applications send does not cross a predetermined size limit. The Bundle Client then assigns an ADU_ID to each piece of data it receives. This ADU is then stored in the Bundle Client's storage. When the client detects the arrival of a Bundle Transport, it does Bundle Packing to send a bundle to the Bundle Transport. During Bundle Packing, it is possible that the payload data is the same as that of the last bundle sent. This could happen if no acknowledgment has been received from the Bundle Server after the last bundle was sent and one of the following three cases occurs: 1) there is no new ADU to send, or 2) the application size limit was reached for an application that has new ADUs to send, or 3) the bundle size limit was reached. In those cases, the bundle ID of the last sent bundle is reused. Otherwise, a new bundle ID is generated and assigned to this bundle. The generated bundle is then sent to Bundle Transport.

C. Bundle Transport

As shown in Fig. 4, the Bundle Transport application, which resides on the data transport device, physically moves the bundle between the connected and disconnected regions. It leverages gRPC over Wi-Fi Direct to exchange data in the disconnected region. A Bundle Transport acts both as a gRPC server for the Bundle Client and as a gRPC client for the Bundle Server (as we shall see in IV-D).

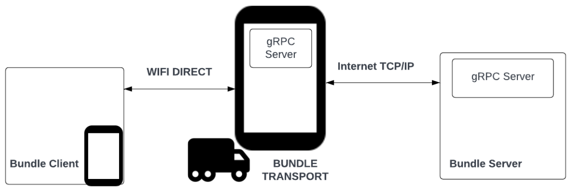


Fig. 4: Transport System Design

Following subsections describe some key considerations in Bundle Transport design.

1) Memory management in transport device: Since Bundle Transport stores bundles from multiple disconnected users, the user of this device can choose to store them in internal storage or SD card storage. The application will also use scoped storage [14] introduced in Android 11 which prevents other applications from accessing application-specific 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 Bundle 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 Bundle Transport deletes the bundle right after delivery then a malicious Bundle Client could receive all the bundles and deny service to the Bundle Client who should receive those bundles. Note that even if a malicious Bundle Client receives bundles, it will not be able to decrypt the data without the private key for the destined Bundle Client.

2) Security over Wi-Fi Direct: 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 that they are transporting. 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. The Bundle Transport generates a public/private key pair [15] which will serve as a self-certified identifier, i.e. the transport ID when communicating with the Bundle Server.

D. 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. Bundle 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. Bundle Transport provides its transport ID in all requests to the Bundle Server.

1) Receiving Bundles from the Bundle Transport: Bundle Transport sends the bundles it collected from clients to the Bundle Server. Unlike the Bundle Client, there is no receiver window on the Bundle Server. The Bundle Server accepts all the bundles sent by Bundle Transport. It 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 to include it in the acknowledgment record which it will pack in the next bundle it generates for this client. The server does Bundle Unpacking, after which it gets from the acknowledgment record, the most recent Bundle ID that the client has received from Bundle Server. This bundle ID 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.

Any received bundle has an acknowledgment record and might have ADUs. Bundle Server implements deduplication and in-order delivery guarantees on ADUs in the same way that Bundle Clients do (refer IV-B1). The ADUs are stored to be delivered to the application servers asynchronously. The application servers may generate some data in response to the data sent by the user. This data is stored with the application server until the transport device is ready to accept bundles to send to clients.

2) Sending Bundles to the Bundle Transport: Bundle Transport requests the Bundle Server for bundles to be delivered to Bundle Clients. Bundle Server maintains a **routing table** mapping transport ID to a list of client IDs that a Bundle Transport with a given transport ID can reach. Bundle Server updates the list of client IDs reachable from a Bundle Transport each time a bundle is received from a Bundle Transport. Using the transport ID provided by Bundle Transport in the request, Bundle Server identifies the Bundle Clients that the transport can reach using the routing table. The Bundle Server then generates at most one bundle for each client that the transport can reach. As mentioned earlier, the server maintains a sender window of 10 counters per client which it uses for the next ten bundles it sends for a given client. As in the case of the client, the server only includes as much data as is allowed by the bundle size limit during Bundle Packing.

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. The Bundle Server instructs the Bundle Transport to delete those bundles.

Note that the DDD delivery mechanism ensures that the sender and receiver windows overlap on at least one counter at all times. Moreover, the windows achieve complete overlap eventually when all ADUs have been successfully received by the receiver.

E. Acknowledgment Records

An acknowledgment 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. As mentioned in Section IV-A, the sender (whether a client or the server) uses an acknowledgment record to determine the next set of ADUs to be sent to the receiver. The acknowledgment 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 acknowledgment 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 acknowledgment record to slide the sender window. When the server receives an acknowledgment 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.

F. Demonstration of Delivery Guarantees

With the help of an example scenario, this section shows how a Bundle Client and the Bundle Server interoperate to deliver ADUs demonstrating delivery guarantees under the constraints of the underlying network. In this scenario, just one application has been considered without loss of generality i.e., any given bundle will have ADUs corresponding to a single application in the payload. In practice, the payload can include ADUs from multiple applications at the same time. 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. The sender and receiver window sizes are 2. Client and server refer to a Bundle Client and the Bundle Server respectively.

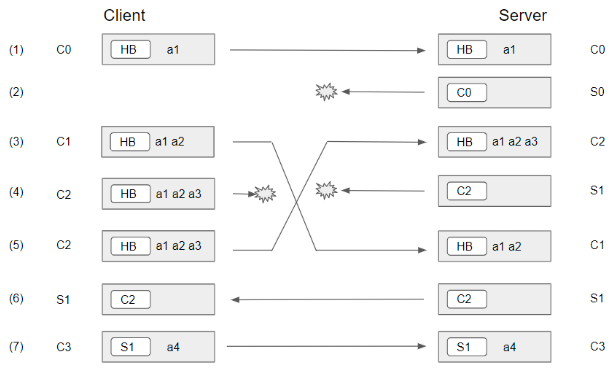


Fig. 5: Bundle Transmission Scenario

In the scenario shown in Fig. 5, the Bundle Client sends application data to the Bundle Server which responds with acknowledgments and no additional data. Note that in practice, the bundles sent by the Bundle Server will have ADUs but we have only considered 1-way ADU transfer in this example for ease of understanding. Following are the explanations corresponding to line numbers (1) through (7): (1) The Bundle client sends a bundle with ADU a1 and an acknowledgment record containing a heartbeat message denoted by 'HB'. The bundle is successfully delivered to the Bundle Server. At this step, the Bundle Server unpacks the bundle and stores a1 to be delivered to the receiver application's server later. (2) The Bundle Server sends a bundle with an acknowledgment 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 Bundle Transport carrying the bundle delivers corrupted data to the Bundle Client, loses data, or simply does not go back to the Bundle Client again. (3) Since the Bundle Client did not receive an acknowledgment for C0, it includes a1 along with new ADU a2 and a heartbeat in the new bundle C1 which is generated when a Bundle Transport arrives. (4) The Bundle 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 Bundle Client does not have any new ADU to send. This can happen when either the application has not provided any new ADU, or the application/bundle size limit has been reached. The Bundle Client sends the last bundle C2. When the Bundle Server unpacks C2, it finds that the bundle has new ADUs a2 and a3 along with a previously received ADU a1. The Bundle Server stores a2 and a3 to be delivered to the application's server and discards duplicate ADU a1. As shown in (4), the Bundle Server sends a new bundle S1 with an acknowledgment containing C2. This bundle is lost. At this point, Bundle Server's sender window is full, and it will retransmit S1 until it receives an acknowledgment allowing it to slide the window. The Bundle 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 Bundle Server discards C1. This is how out-of-order bundles are discarded. C2 corresponds to the bundle ID that will be sent in all acknowledgments from the client until a bundle with a bundle ID with a bigger counter is received. (6) The Bundle Server retransmits S1 with an acknowledgment record containing C2 which is received by the Bundle Client. At this point, the Bundle 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 acknowledgment for C2, the Bundle Client also finds out that a1, a2, and a3 have been successfully delivered since C2 included A1, A2, and A3. Therefore, the Bundle Client includes only a4 in the new bundle (bundle ID C3) with an acknowledgment record containing S1, the bundle ID with the largest counter received so far from the server. On receiving the bundle, the Bundle Server stores the new ADU a4. Since the Bundle Server received the acknowledgment for S1, the Bundle Server now slides the window from [S0, S1] to [S2, S3] which is the same as the Bundle Client's receiver window.

As we can see, ADUs sent by the client - a1, a2, a3, a4 were received by the Bundle Server in order in spite of the loss of bundles and out-of-order bundle delivery. The Bundle Server discarded an ADU a1 (in step (3)) that was already received thus demonstrating the deduplication guarantee.

V. Results

As part of DDD, we implemented a set of delivery guarantees. We conducted some experiments to test the delivery guarantees and the performance of our infrastructure in carrying out its goal of delivering data to disconnected Android applications. In this section, we describe how we tested our approach and summarize our results.

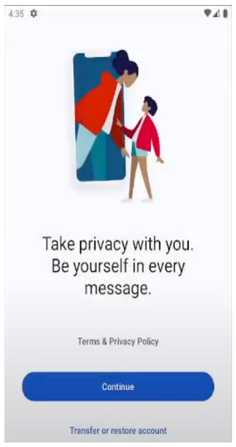
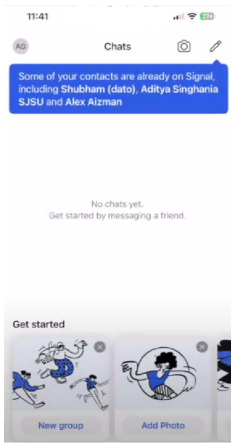

Take privacy with you. Be yourself in every message.

Fig. 6: Signal Registration Page


Some of your contacts are already on Signal, including Shubham (dato), Aditya Singhania SJSU and Alex Aizman

No chats yet.
Get started by messaging a friend.

Fig. 7: Signal Home Page

TABLE I
gRPC THROUGHPUT

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 II
gRPC THROUGHPUT: SINGLE DEVICE CONNECTION

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	1.51	49.6

A. *Implementation with Signal*

In order to test the infrastructure end-to-end, i.e. packing/unpacking of application data, the transmission of bundles over Wi-Fi Direct and TCP, and the delivery guarantees, we chose Signal, a real-world application for instant messaging. We installed Signal on an Android phone and switched off its connection to the Internet to simulate a disconnected user. We used another Android phone running the Bundle Transport application and had the Bundle Server running on a PC.

When the Signal Android application is downloaded, the first step after opening the Signal Android application is to start Signal's registration process. Here, we clicked on the ”Continue” button to proceed with registration, as shown in Fig. 6. On this button press, a part of the registration process was executed, generating some data unique to the device that the Signal's server requires for registering the user. The generated data was sent to the Bundle Client and stored as an ADU file. We brought the phone with Bundle Transport near the Bundle Client and initiated the delivery process. The Bundle Client packed the ADU into a bundle and transferred it to the Bundle Transport. When we connected the Bundle Transport to the Internet to simulate it entering a connected region, it transmitted the bundle to the Bundle Server. When this data reaches the Bundle Server via a transport device, the Bundle Server does Bundle Unpacking and forwards the received ADU to the Signal Server, which registers the user and generates a confirmation message as a response. When this response made its way back to the Signal application on the user's phone via Bundle Transport, we observed that the user was successfully registered and could then see the home page in the Signal application without ever connecting to the Internet, as shown in Fig. 7. If the Signal server received any data for this user, it could send the data to the Bundle Server. We found the registration to work even when the bundles were dropped by the Bundle Transport, sent twice to the Bundle Server, or delivered out of order to the Bundle Server thus confirming the delivery guarantees.

B. *Network Throughput over Wi-Fi Direct*

We ran 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 was 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.

Now, we connect a single client to the server device over Wi-Fi Direct and use gRPC to send a .flac file of size 70.2 MB with different chunk sizes. We observe from the table I that the throughput is affected by changes in chunk size, i.e. 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, .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. We observe from the throughput values in table II 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 Wi-Fi chipset installed on it.

In the DDD system, multiple devices can connect with the transport device using Wi-Fi Direct. 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 throughput values and chunk sizes in table III that the trend of P2 performing better than P1 is still observed. However, the overall throughput of both devices is lower.

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

TABLE III
gRPC Throughput: Multiple Device Connection

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	3.19	24.1

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.

Conclusion

We solve the problem of bringing data from the Internet to disconnected regions by creating Disconnected Data Distribution (DDD), 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. At the ends of the DDD infrastructure, Bundle Client and Bundle Server implement a mechanism for application data packaging and transmission of application data between a disconnected client and application servers such that it is resilient to loss, reordering, duplication, and corruption. DDD's infrastructure and functionality are built on a foundation of privacy: data transports are not trusted and do not have information about the users or services from the data they transport. Bundle Client and Bundle Server can see which services are used by the users but do not have access to the data payloads. Our software-only solution provides a path for quick provisioning and sustainable deployments. Rather than requiring services to be built for DDD, the DDD infrastructure works well with current internet services as demonstrated using the Signal application. DDD has the potential to bring the benefits of the internet to areas of the world that may never have internet connectivity.

References

[1] UN. (2023) Itu: 2.9 billion people still offline. [Online]. Available: https://www.un.org/en/delegate/itu-29-billion-people-still-offline

[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] 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

[4] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, and H. Weiss, "Delay-tolerant networking: an approach to interplanetary internet," *IEEE Communications Magazine*, vol. 41, no. 6, pp. 128–136, 2003.

[5] S. Foundation. (2023) Signal private messenger. [Online]. Available: play.google.com/store/apps/details? id=org.thoughtcrime.securesms

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

[7] 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.

[8] "Vlink." [Online]. Available: http://blizzard.cs.uwaterloo.ca/tetherless/index.php/VLink

[9] S. Burleigh, K. Fall, and E. J. Birrane, "Bundle Protocol Version 7," RFC 9171, Jan. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9171

[10] K. Fall, W. Hong, and S. Madden, "Custody transfer for reliable delivery in delay tolerant networks," *IRB-TR-03-030, July*, vol. 198, 2003.

[11] E. Koutsogiannis, F. Tsapeli, and V. Tsaoussidis, "Bundle layer end-to-end retransmission mechanism," in *2011 Baltic Congress on Future Internet and Communications*. IEEE, 2011, pp. 109–115.

[12] G. Papastergiou, I. Alexiadis, S. Burleigh, and V. Tsaoussidis, "Delay tolerant payload conditioning protocol," *Computer Networks*, vol. 59, pp. 244–263, 2014.

[13] "Content provider android," 2023. [Online]. Available: https://developer.android.com/guide/topics/providers/content-providers

[14] "Scoped storage : Android open source project." [Online]. Available: https://source.android.com/docs/core/storage/scoped

[15] A. Salomaa, "Public-key cryptography," 1996.