Spring 2023

Security and Routing in a Disconnected Delay Tolerant Network

Anirudh Kariyatil Chandakara
San Jose State University

Security and Routing in a Disconnected Delay Tolerant Network

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

Anirudh Kariyatil Chandakara

May 2023

The Designated Project Committee Approves the Project Titled

Security and Routing in a Disconnected Delay Tolerant Network

by

Anirudh Kariyatil Chandakara

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

Dr. Benjamin Reed	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science
Dr. Navrati Saxena	Department of Computer Science

ABSTRACT

Security and Routing in a Disconnected Delay Tolerant Network

by Anirudh Kariyatil Chandakara

Providing internet access in disaster-affected areas where there is little to no internet connectivity is extremely difficult. This paper proposes an architecture that utilizes existing hardware and mobile applications to enable users to access the Internet while maintaining a high level of security. The system comprises a client application, a transport application, and a server running on the cloud. The client combines data from all supported applications into a single bundle, which is encrypted using an end-to-end encryption technique and sent to the transport. The transport physically moves the bundles to a connected area and forwards them to the server. The server decrypts the bundles and forwards them to the respective application servers. The result is then returned to the original client application via the network of transports used previously. This solution provides a convenient way to establish connectivity in disconnected areas without additional hardware and can accommodate various application data. Furthermore, it ensures data integrity, confidentiality, and authentication by encrypting and validating the data during transmission.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Dr. Ben Reed for bringing this project to life. His support and guidance throughout the project have been invaluable. I am also grateful to Dr. Navrati Saxena and Dr. Thomas Austin for their valuable contributions and support in helping me write this paper. I would also like to thank my lab partners Abhishek, Shashank, Aditya, and Deepak for their support and encouragement throughout this journey. Everyone's guidance and support have been essential to the success of this project. Thank you for sharing your expertise, time, and knowledge with me.

TABLE OF CONTENTS

CHAPTER

CHAPTER 1

Introduction

The internet has become an integral part of modern life, connecting people all over the world and enabling access to a vast amount of information and resources. It has transformed the way we communicate, do business, and access entertainment. In addition, the internet has made it easier for people to access education, healthcare, and government services, and has helped to bridge the digital divide between those who have access to technology and those who do not. If an area experiences a loss of connectivity due to natural disasters or lacks access to the internet, the residents of that area will be unable to utilize these services. The COVID-19 pandemic highlighted the importance of internet access as a fundamental necessity and the challenges faced by those who do not have reliable internet connections. According to a United Nations report [1], the pandemic exacerbated existing inequalities and has disproportionately affected people living in rural and underserved areas, who have limited or no access to the internet.

Delay-tolerant networks (DTNs) are a type of networking technology that can be used to deliver internet connectivity to remote areas where traditional networking infrastructure is unavailable or unreliable. DTNs are designed to operate in these environments that cannot rely on continuous connectivity [2]. They use a store and forward mechanism; a node receiving data from a sender stores the data and waits for a connection to be established with the recipient. When a connection is established, the node forwards the data to the recipient. This process is repeated at each intermediate node along the path until the data reaches its final destination. This allows DTNs to function in scenarios where there is a high degree of network disruption as is the case in remote, disaster-stricken areas and space communications as well.

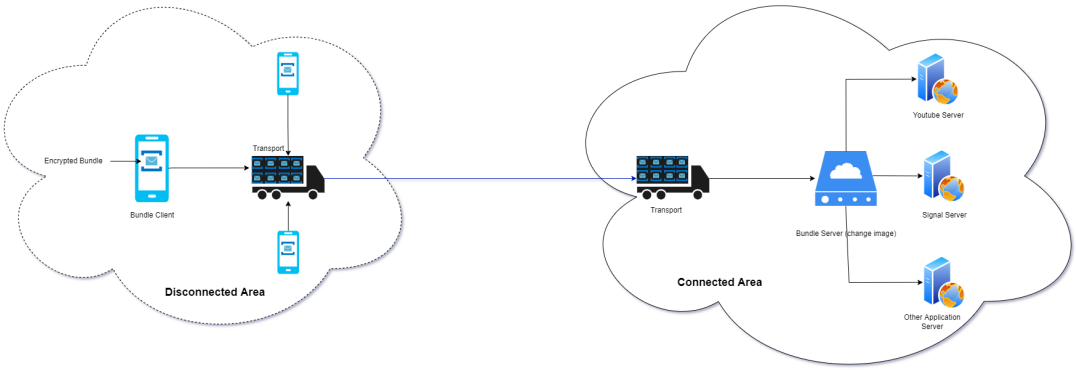

Figure 1: Project Overview

1.1 Project Overview

This project proposes an architecture that aims to utilize existing hardware to provide internet access to disaster-affected areas or regions with limited or no connectivity while providing end-to-end encryption. The users can use their phones to utilize this service by downloading the respective applications from the Google Play Store. When internet connectivity is no longer available, devices can use the *Bundle Client* application to access the internet for selected applications. This device will be referred to henceforth as the *Bundle Client* or simply as the *client*. Another mobile device that is capable of moving from a disconnected to a connected area will use the *Bundle Transport* application to store and forward bundles that it receives from clients it comes in contact with. This device will henceforth be referred to as the *Bundle Transport* or simply as the *tranport*.

When a transport comes in contact with a client, the client combines data from all supported applications into a single *Bundle* and encrypts it using an end-to-end encryption technique before sending it to the *Transport*. The transport stores the bundles and physically moves them to another location where it can connect to the internet. Figure 1 illustrates this scenario. Once the transport reaches a connected

area, it forwards all the bundles to the *Bundle Server*, which is a server that is running on the cloud that accepts and processes bundles. The Bundle Server then decrypts the bundles and forwards them to the respective application servers. The response is then sent back to the client based on the path selected by a routing table.

This solution offers a convenient way to establish connectivity in disconnected areas without the need for additional hardware while maintaining a high level of security. It can accommodate a wide range of application data, including those with longer half-lives. The term 'half-life' in this context refers to the amount of time it takes for the majority of the data to become irrelevant [3]. Intermediate nodes do not need to be trusted in this case as they simply act as a forwarding medium between the client and server. All data received from the transport at either end is not trusted until it can be decrypted and verified successfully which provides data integrity, confidentiality, and authentication.

CHAPTER 2

Related Work

This section describes the related work by various authors in the field of Delay Tolerant Networks (DTNs), routing in DTNs, and security strategies employed in DTNs.

2.1 Delay Tolerant Networks

Networks in areas hit by natural disasters or military ad-hoc networks violate one or more of the assumptions in the TCP/IP architecture as noted by K.Fall in [4]. They propose an architecture for these challenged networks, to operate on top of the existing OSI networking model and use a store and forward message-passing mechanism. The messages to be sent are aggregated into "bundles" and sit in the application layer of the networking stack [5]. The Bundle Protocol described in [6] consists of several concatenated blocks that can be extended if required. It provides the ability to cope with intermittent connectivity while also being able to take advantage of scheduled, predicted, and opportunistic connectivity. These features allow the creation of a software-only solution as it can be integrated into existing devices without the need for additional hardware infrastructure to deploy the network. The data to be sent can also be combined into a single bundle to maximize the available bandwidth and reduce the number of trips required. The bundle protocol also allows the use of digital signatures and message authentication codes to maintain the authenticity of the users and the integrity of the data between the endpoints [7].

2.2 Routing in DTNs

Routing in DTNs can be classified as follows:

(i) Flooding Based

(ii) Forwarding Based

(iii) Probability Based

Flooding-based routing techniques are one of the simplest DTN routing techniques of which Epidemic routing is an example. A. Vahdat and D. Becker explain in [8] how it forwards the message to all nodes it comes into contact with except the source. This increases the message delivery ratio in disrupted networks. The overhead however is significant as a node must have sufficient buffer space to store the packet. Mobile device battery life is a critical factor that must be taken into account when designing these systems. The high delivery ratio cannot be ignored and therefore multiple optimizations exist for epidemic-based routing. P. Garg et. all in [9] noted a 15% increase in delivery ratio and a 34% decrease in the overhead by implementing simple battery and buffer size constraints. Another optimization criterion proposed in [10] is to broadcast the current node's message list and require the neighbors to then request the messages they do not have to remove redundancy. This was shown to have the lowest average packet rate when compared to other epidemic routing techniques. As this optimization improves upon the existing epidemic routing protocol, it can be implemented in the disconnected setting of this project.

Forwarding-based routing techniques, on the other hand, find the best path for data transfer between nodes, to do this however they require some form of network topology information. The topology information can be decided from the source node as in Source Routing [11] or on a hop-by-hop basis known as Location-based routing [11]; here each hop simply stores the next physically closest connected hop. Other routing algorithms include Window aware adaptive replication (ORWAR) which exploits the context of mobile nodes such as their speed and direction and estimates the size of the contact window. It then selects messages based on their importance and size to minimize partially transmitted messages and optimize overall bandwidth [12]. These

routing protocols work effectively when information about the topology is known or can be detected. We can use a version of forwarding-based routing as we only have a single hop between the server and the client i.e., the transport. The transport, however, is not a trusted node in the network and therefore cannot be relied upon. Hence, we use the server to store a mapping between transports and clients once the bundles are verified.

Probability-based routing or history-based routing is proposed in [13] as an enhancement to epidemic routing and termed PROPHET (Probabilistic Routing Protocol using History of Encounters and Transitivity). The delivery probability is calculated as the likelihood of a node delivering a message to another node when they encounter each other. The more often they meet, the higher the probability and vice versa [14]. This predictability is also transitive. Nodes exchange messages only when the probability is high, they also update their probability along with the message. There have been multiple enhancements to PROPHET, such as incorporating the bundle protocols' hop count and the delivery probability when selecting the next hop to reduce delay and overhead [15]. PROPHET+ is another enhancement that uses a weighted function consisting of the nodes' buffer size, location, power, and popularity along with the delivery probability to maximize delivery rate and minimize transmission delay. This can be used in our project to rank the clients that the transports come in contact with. Each time a transport successfully delivers a bundle to the client, its score can be increased. The server can now use this score to rank the clients that each transport connects to, allowing clients that have a higher probability to be reached to be given priority over others with a lower probability.

2.3 Security in DTNs

DTNs use bundles as the message-passing protocol as such multiple security features are proposed as extension blocks in bundles. The bundle security protocol [7] and the newer BPSec (Bundle Protocol Security) [16] are two such proposals. Both use extension blocks in the bundle protocol to provide security features such as integrity, confidentiality, and authentication [16]. Each of which has its own block. An extension security block can also be used to encrypt the feature blocks mentioned previously [7].

Key management however is not considered in either of these protocols and is left to the implementation. A framework for key management is provided in [17] for DTNs that use the bundle security protocol which can be implemented. They propose using ESKTS (Efficient and Scalable Key Transport Scheme) which is based on public key cryptography and proxy signatures [18] as it provides hop-to-hop and end-to-end integrity along with authentication and confidentiality. This however relies on delegating the signing capability to the proxy entity (next hop in our case), as we do not trust the intermediate transport nodes in our implementation this scheme cannot be implemented.

Self-Certifying path names avoid key management machinery by using a locator that includes the remote server's public key [19]. The public key is embedded in the path name and used as the locator as well as for cryptography. It can therefore be used in a decentralized setting without a centralized key management authority. Z. Wilcox-O'Hearn and B. Warner, in their implementation, demonstrated that self-certifying path names also provide access control at a granular level for both users and files [20]. Self-certifying path names allow the masking of other client identifiers such as the layer 2 MAC addresses even from the transport using MAC address randomization to further enhance privacy [21]. This can be used while transmitting/receiving to and from the transport making it more difficult to identify a device or its location.

The encryption key generation can be done using the double ratchet algorithm as employed in Signal's messaging protocol [22]. The algorithm uses a Diffie-Helman ratchet for the given public-private key pair to generate a shared secret. This along with the result of a root key generated using the X3DH as described in [23] results in a symmetric key used to encrypt or decrypt the message. Exchanging the public keys and synchronizing the ratchets at the recipient's end, results in the generation of an identical symmetric key. Signal in their implementation rotate the ratchet, i.e. generate a new Diffie-Helman secret for every message to be sent and received. This is done to provide forward and backward secrecy and can be implemented in a disconnected setting as well, as long as the ratchets are synchronized. The double ratchet encryption is used in non-messaging applications as well and is shown in [24] where it is employed as a low-cost edge security gateway to protect legacy devices behind it and in [25] where it is used to provide security authentication for user equipment in 5G networks. To implement the algorithm in a disconnected environment, it is essential to ensure that the client has access to the server's key bundle; the key bundle is a collection of the server's public keys that a node would require to initiate communication with the server. Information regarding the ratchets' position while encrypting must also be sent to the receiver for it to synchronize its ratchets to match that of the sender to decrypt the message. Fortunately, in this project, the keys and the server's hostname can be baked into the client application, making the server's key bundle readily available to the client. The values used to synchronize the ratchet are also sent as part of every encrypted message. This enables the client and server to generate the same keys at either end and establish secure communication even in disconnected settings.

Along with privacy, authentication and integrity must also be maintained for each bundle. The digital signature is used to detect unauthorized modifications to

the data and to authenticate the identity of the signing entity as described in FIPS (Federal Information Processing Standards) by NIST (National Institute of Standards and Technology) [26]. The signature is generated by hashing the plain text and then encrypting the result. It is recommended to sign the plain text to be sent and not the encrypted data to verify the authenticity and integrity of the data as described by R. Kaur and A. Kaur in [27]. Multiple signature algorithms exist that implement hashing and encryption, the most common include RSA (Rivest–Shamir–Adleman), DSA (Digital Signature Algorithm), ECDSA (Elliptic Curve Digital Signature Algorithm), and EdDSA (Edwards-curve Digital Signature Algorithm) to name a few. Cendyne in [28] describes why Ed25519 an EdDSA signature that uses the SHA-512 algorithm to create the hash is more favorable. It requires relatively little computation while providing high-security levels (2^{128}) and taking up 64 Bytes of space. Signal also utilizes the Ed25519 algorithm in its implementation [22] and so will fit well with this solution.

CHAPTER 3

High-Level System Design

The system described in Section 1.1 includes multiple components to transfer data from the disconnected area to a connected one, including the Bundle Client(client henceforth), Bundle Transport (transport henceforth), and the Bundle Server (server henceforth). The scope of this paper however is to focus on the security and routing modules within these components, and how to effectively implement them within a disconnected setting.

The following sections, therefore, describe the creation of identifiers for both Bundles and Clients, as well as the routing mechanisms that are utilized to select the most viable client via a Transport. The final section of the chapter describes the Bundle format and illustrates the manner in which end-to-end encryption is achieved.

3.1 Bundle Identifier

Each bundle has its bundle identifier, which is used by the Client when requesting bundles from the Transport and by the Server to keep track of the bundles it has received. Each Bundle ID consists of the client's ID along with a counter. The counter is generated by the sender, so when the client sends a bundle to the server (upstream), it will increment it's sending counter, and similarly, when the server sends a bundle to the client (downstream), it will increment its sending counter. While there may seem to be a correlation between the two counter-values, they are entirely independent of each other as they are used for different purposes.

In the upstream direction, the server uses the counters to keep track of the order of the bundles when they are received out of order or lost. The client, therefore, generates counters incrementally. In the downstream direction, the client uses the counters to request bundles from the transport. The server, therefore, generates the counter values such that they are the same as the values the client requests from the

transport. This synchronization is maintained using the counter value and a window as explained in the Window section.

The counter is 8 Bytes and is added either before or after the Client ID based on the direction of the bundle. In the upstream direction, the counter is the 8 Most Significant Bytes (MSB) in the Bundle ID; in the downstream direction, it is the 8 Least Significant Bytes (LSB) as shown in Figure 2.

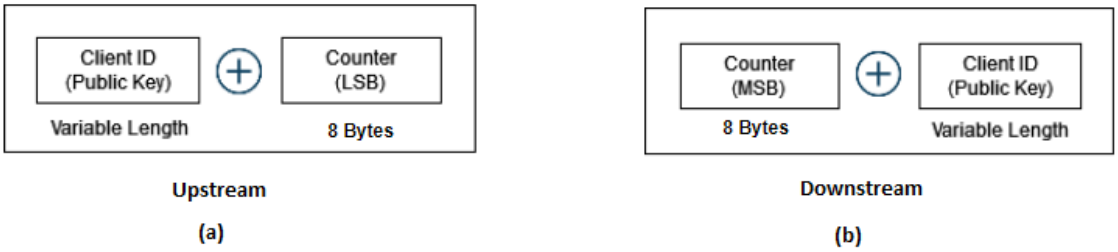

Figure 2: Bundle ID Generation

The Bundle ID will be exposed to the transport; it must therefore be encrypted to protect the client's bundles from being identified. As we have the server's public key, we can generate a Diffie Helman (DH) shared secret and use this as the shared secret for AES to encrypt the Bundle ID. The encryption flow is illustrated in Figure 3. The encrypted blob can then be sent to the transport. On the server end, it will receive the client's public key along with the bundle as part of its encryption header as defined in Section 3.4. It can now generate the same shared secret and decrypt the Bundle ID. The counter value can then be obtained using appropriate bitmasks.

3.2 Window

A window is maintained at the client and server of the same size (10 by default) to provide synchronization between them when transporting bundles. The server-side window is used to keep track of the bundles sent to the client and to make sure it does not send more bundles than the client can accommodate. The client-side window is used to request bundles from the transport. The server, therefore, maintains a

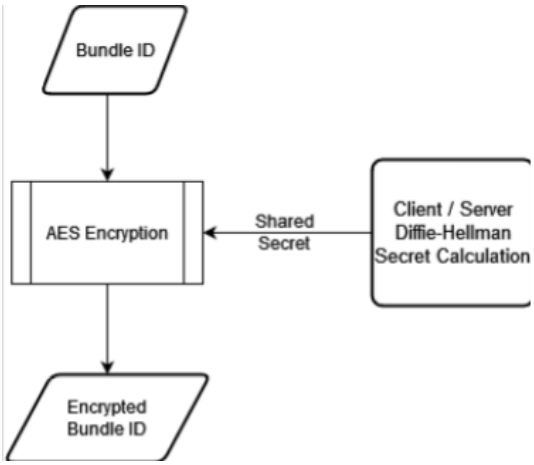

Figure 3: Bundle ID Encryption

window for each client. An underlying assumption here is that eventual delivery will occur. Meaning that even if bundles are lost during transit, eventually at least one of the retransmissions will result in a bundle reaching its destination.

When a bundle is to be sent downstream from the server to the client, the server checks the window for a slot. If a slot is available, a new bundle ID is generated and added to the window slot. If no slot is available, then the bundle in the last slot is retransmitted to the client. Only the last slot is to be retransmitted, given that every bundle is a superset of all outstanding or unacknowledged bundles. An acknowledgement is therefore always sent along with the bundle, if no bundle was received then a heartbeat message is present as an acknowledgment. The heartbeat message is sent only by the client to the server, this is so the server knows that the client is still reachable. When an ACK is received for a specific Bundle, it can be concluded that all previous Bundles have also been received by the client. Thus the window can be moved ahead to the next 10 slots starting from the received Bundle ID.

On the client side, the window is used to request bundles from the transport. When a transport becomes available, the client generates bundle IDs for all the slots

available in the window. These bundle IDs are then sent to the transport, and corresponding bundles if found on the transport, are transmitted back to the client. On receiving bundles from the transport, the window is moved ahead to the next 10 slots after the last bundle ID received from the transport. In cases where not all bundles requested are received, the window is moved to start from the last bundle ID received. The window is therefore able to ensure that only bundles that can be processed at the client are sent by the server.

3.3 Routing

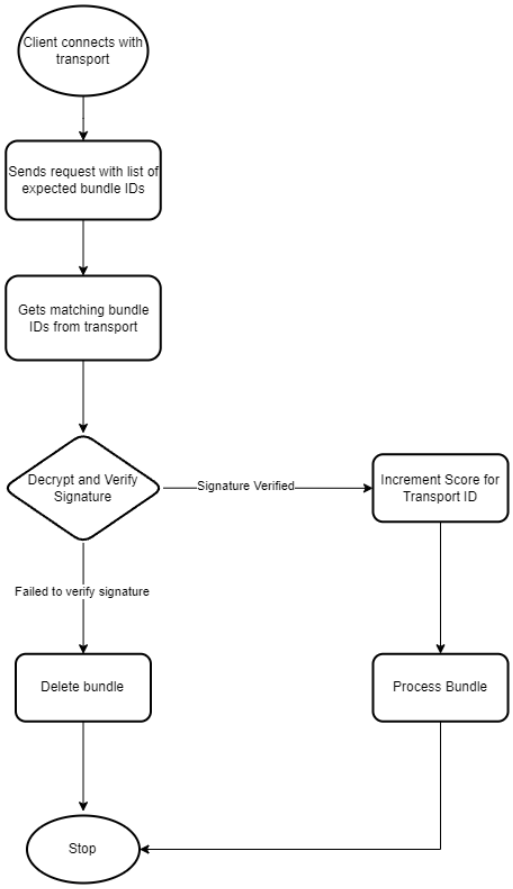

Figure 4: Client Receive Flow

The Bundle client maintains a table that maps the number of valid bundles (score) received by a transport. A bundle is said to be valid if it is decrypted and the

payload's signature is verified as illustrated in Figure 4. The client's routing table keeps track of the number of bundles received from all the transports the client comes into contact with. A snapshot of the transport table on the client is shown in Figure 5, here the client has received bundles from 3 different transports, T1, T2 and T3. The scores have been updated based on the number of valid bundles received from the respective transport. This table is sent to the Bundle server as part of the routing meta-data field in the bundle (see Section 3.4) to be used in the server's routing table.

Transport Table (Client)	
Transport ID	Score (long long)
T1	7
T2	3
T3	8

Figure 5: Transport Table (Client)

The Bundle Server maintains a routing table that maps the list of clients that are reachable via a particular Transport. This map is used to determine the following:

(a) The list of clients that are reachable via a Transport

(b) The reachability of a client via the transport

Routing Table (Server)	
Transport ID	Client Entry {Client ID, Score}
T3	{C2, 1}

(a)

Routing Table (Server)	
Transport ID	Client Entry {Client ID, Score}
T1	{C1, 7}
T2	{C1, 3}
T3	{C1, 8}, {C2, 1}

(b)

Figure 6: Routing Table (Server)

The tables shown in Figure 6 describes the routing table as stored on the server. The key is the Transport ID and the values are client entries. Each client entry stores the client ID and a score. The score denotes the number of valid bundles received by the client and is maintained by the client. It is sent to the server as part of the routing meta-data field of the bundle (Section 3.4).

When a bundle is received from a transport, the server checks if the routing table has a record for the current transport. If no record exists, it creates it with the respective Transport ID (TID) and adds a client entry using the client ID (CID) with a score of 0 (as we have not sent anything yet). If a record does exist, it searches for a client entry with the CID of the received bundle in the record. If the client entry is found, it replaces the scores with the score received from the client, otherwise, it creates an entry with the CID and a score of 0. This is repeated for all the transport scores sent by the client.

For example, when we receive the routing metadata from the client based on the values shown in Figure 5, we update search for the respective TIDs i.e. "T1", "T2" and "T3" in the server's routing table. The scores for the transport is then updated for the client from which the bundle was received from. Figures 6 (a) and (b) show the state of the table before and after the update respectively. The new transports "T1" and "T2" are added to the table while transport "T3" is updated with the new client's score. Note the order of the clients mapped to transport "T3" has also been changed and the client with the greater score (C1 with 8) is before C2.

The score is used to maintain the order of reachability for a particular transport, the list of client entries is therefore sorted in descending order based on the score, to give clients that are known to be reachable, priority over clients that are not as reachable. It is important to note that the priority does NOT mean that the client ID is skipped when sending the list of clients to the Bundle Transmission Module. It

simply means that it will be later on the list. The client entries are NEVER deleted from the routing table as in the worst-case scenario of most transports becoming unavailable, the server can still use a previously connected transport to send data to the client.

The score for a particular client is reset if the received score for the transport remains unchanged for a predetermined number of times. This threshold is denoted as *reset score* and can be adjusted based on connectivity parameters such as round trip time. *reset score* is initialized to a value of 10, but may be modified dynamically as needed.

3.3.1 Special Cases

1. **Client Moves from one Transport to another/missing:** This can occur if the client or transport no longer meet and a new transport is found.

   In this case, the client will create a new entry in its table and increment the respective score. On the server side, the new transport will be updated with the client entry. The client entry on the previous transport remains as is. When the same score is reported to the server *reset score* number of times, the score will be reset, resulting in a decreased client priority for the particular transport.

2. **Transport only delivers bundles in one direction:** This can occur when the client is only able to receive bundles from the server and all upstream bundles to the server are dropped by the malicious transport.

   In this case, the client score remains the same as the updates to the server are not sent from that transport. This case is an example of why deleting the client entry from the table could lead to it incorrectly losing connectivity.

3. **Client Receives same bundle repeatedly:** This can occur when the server

does not get the ACK for a sent bundle, and so retransmits it (Scenario 6 in the Scenario section)

In this case, the client has received the bundle before and so does not request the bundle from the transport. The bundle is therefore not sent to the client resulting in the score remaining the same. This case can only be handled at the transport and since it is not a reliable entity we choose the ignore incrementing the score counter in this scenario.

3.4 Bundle Format

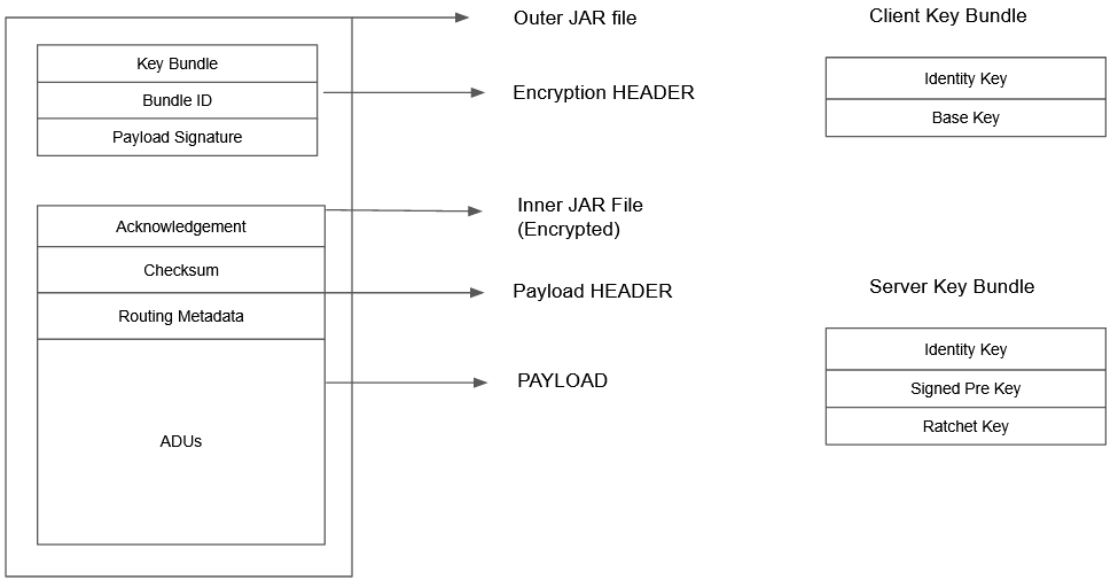

Figure 7: Bundle Format

The Bundle has a collection of fields that contains identifying and processing information along with the application data. The fields are as follows:

1. **Encryption Header:** This field consists of the parameters required to decrypt the bundle including the Key Bundle, bundle ID and the payload signature.

   (a) **Key Bundle:** The key bundle differs based on the direction of the bundle.

In the upstream direction, the client key bundle is used while the server key bundle is used in the downstream direction.

Client key bundle: Consists of the client's Identity and Base/Ephemeral public keys

Server key bundle: Consists of the server's Identity, Signed PreKey and Ratchet keys.

(b) **Bundle ID:** The generated bundle ID is stored here (This is the encrypted Bundle ID).

(c) **Payload Signature:** This stores the signature of the payload before it was encrypted.

2. **Payload Header:** This field includes the data required to parse the payload's header such as the header size, checksum, acknowledgments to be sent i.e., the Bundle ID to be acknowledged and the routing metadata.

Routing metadata: This field is filled only by the Bundle client and stores the client's Transport table as described in Routing Section.

3. **ADUs:** This field stores all of the application data units (ADUs) that need to be sent.

The payload and its associated header are encrypted using the key and ratchet specified in the encryption header. This encrypted data, along with the encryption header and bundle ID, are then compressed and packaged into a jar file.

3.5 Client ID Generation

The Identity public key of a client serves as its identity, which helps to keep other identifiers such as its layer 2 MAC address hidden from external entities. As the public key of the client may change later, the initial public key will be used throughout

as the client ID. This is done because changing the CID each time would result in tables that use the CID as the key on the server side being difficult to maintain and storing redundant and obsolete data.

3.6 Security Module

Security modules are implemented on the client and server side. An implementation of the double ratchet algorithm is used to exchange encrypted messages based on a shared secret. The shared secret is derived each time a message must be transmitted or received. There are 3 main components here:

1. Diffie-Hellman Ratchet
2. Root Key Generation
3. Symmetric Key Ratchet

3.6.0.1 Diffie-Helman Ratchet

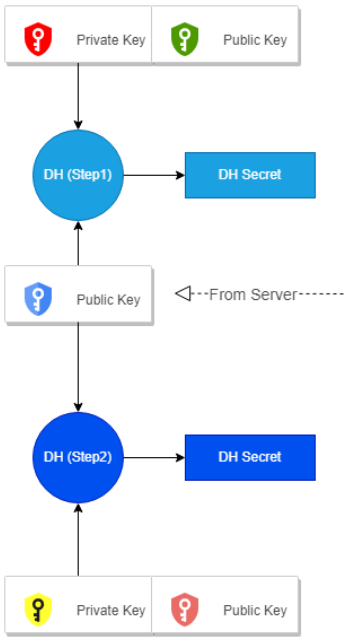

Figure 8: Diffie-Hellman Ratchet

The Diffie-Hellman (DH) ratchet as the name suggests uses the Diffie-Hellman algorithm to generate a shared secret. The Bundle client and server generate a public/private key pair. Initially, the public key of the server is hardcoded into the client allowing it to initiate communication, after which the public key of the sender will be sent to the receiver along with the bundle.

When a new public key is received, a ratchet step is performed; the receiver as shown in Figure 7 uses its current private key along with the received private key to generate a DH Secret to match the secret generated by the sender (DH step 1 of Figure 7). The receiver then generates a new public/private key pair and calculates a new DH secret (DH step 2 of Figure 7). This cycle is repeated every time a new public key is received. If any of the private keys in this exchange is compromised, it is isolated from all other secrets that are generated as each message will have a new secret. This is called forward secrecy wherein an intermediate key if compromised cannot be used to compromise later messages. Similarly, backward secrecy is also maintained as we cannot use the current key to compromise previous messages.

The DH secret that is generated is used as an input to a Key Derivation Function (KDF) to derive a sending and receiving chain key. The details of how the chain keys are derived will be explained in later sections, however, the use of the DH secret is as follows.

The DH secret generated in step 1 is used to generate a sending key on the client when it initiates communication. The server on its end will also generate a DH secret using its private key along with the client's public key that it receives. The derived secret is the same as it is generated using the DH exchange. This shared secret is passed to a KDF to derive a receiving and sending chain key on the client and server respectively to generate the same symmetric encryption key.

3.6.0.2 Root Key Generation

The Root Key is generated using the client's Identity and Ephemeral/Base Keys and the server's Identity and Signed Pr-Key for the first time, subsequent generations of the root key are derived using a Key Derivation Function (KDF). The derivation of the initial and subsequent root keys are explained in the following sections.

IK_C	Client's identity key
EK_C	Client's ephemeral key
IK_S	Server's identity key
SPK_S	Server's signed prekey

Figure 9: Key Table

3.6.0.3 Initial Root Key Generation

The root key is calculated using 3 Diffie-Hellman key exchanges as follows:

1. DH1 = IK_C + SPK_S
2. DH2 = EK_C + IK_S
3. DH3 = EK_C + SPK_S

The resulting DH secrets namely DH1, DH2 and DH3 are passed to a KDF, which returns the root key. Root Key = KDF(DH1, DH2, DH3) This is used as the initial root key.

3.6.0.4 Subsequent Root Key Generation

Once the initial root key is generated, all subsequent root keys are generated by the KDF with the inputs being the previous Root Key and the new Diffie-Hellman

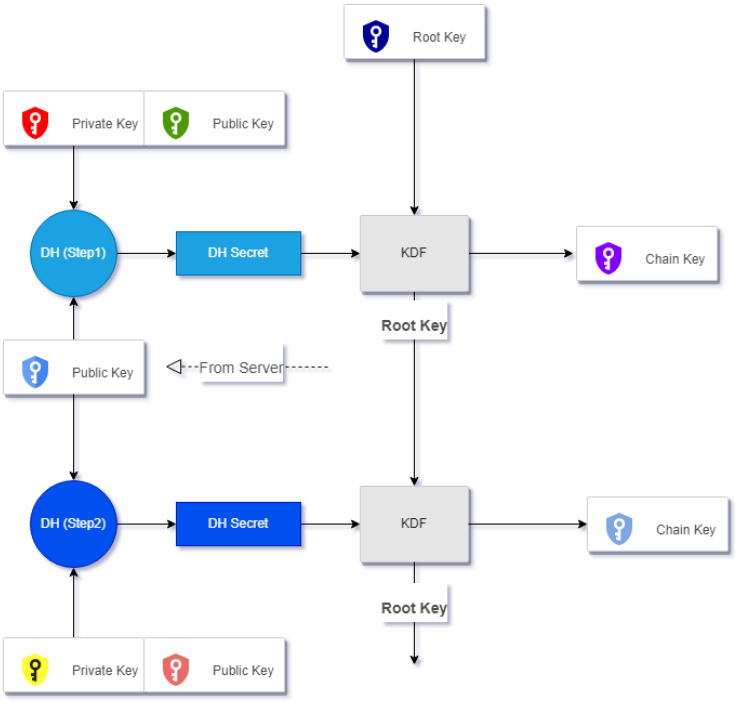

Figure 10: Root Key Generation

Secret as shown in Figure 9. The KDF also generates a Chain key. This is used to generate the sending or receiving chain keys and is described in the next section.

The requirement while selecting a KDF is that it must guarantee all outputs appear random, regardless of the inputs selected. The Hash-based message authentication code (HMAC) and HMAC-based key derivation function (HKDF) constructions, when instantiated with a secure hash algorithm, meet the KDF definition. KDFs therefore, provide forward security as the outputs appear random and have no relation to a previous or later output.

3.6.0.5 Symmetric Key Ratchet

The last ratchet in this algorithm is a symmetric key ratchet. These keys are also generated similarly using a KDF with the root key. The inputs to the KDF here are, the Chain key generated by the previous KDF and a constant value. The KDF

provides 2 outputs here as well; a chain key that will be used in the next ratchet step to generate further keys and a message key. The message key is used to encrypt or decrypt the message for the sending and receiving chains respectively. The Message keys can be stored to be used later on when handling out-of-order messages as they are not used to generate new keys. Figure 10 illustrates the entire flow that is triggered when a new public key is received from the server.

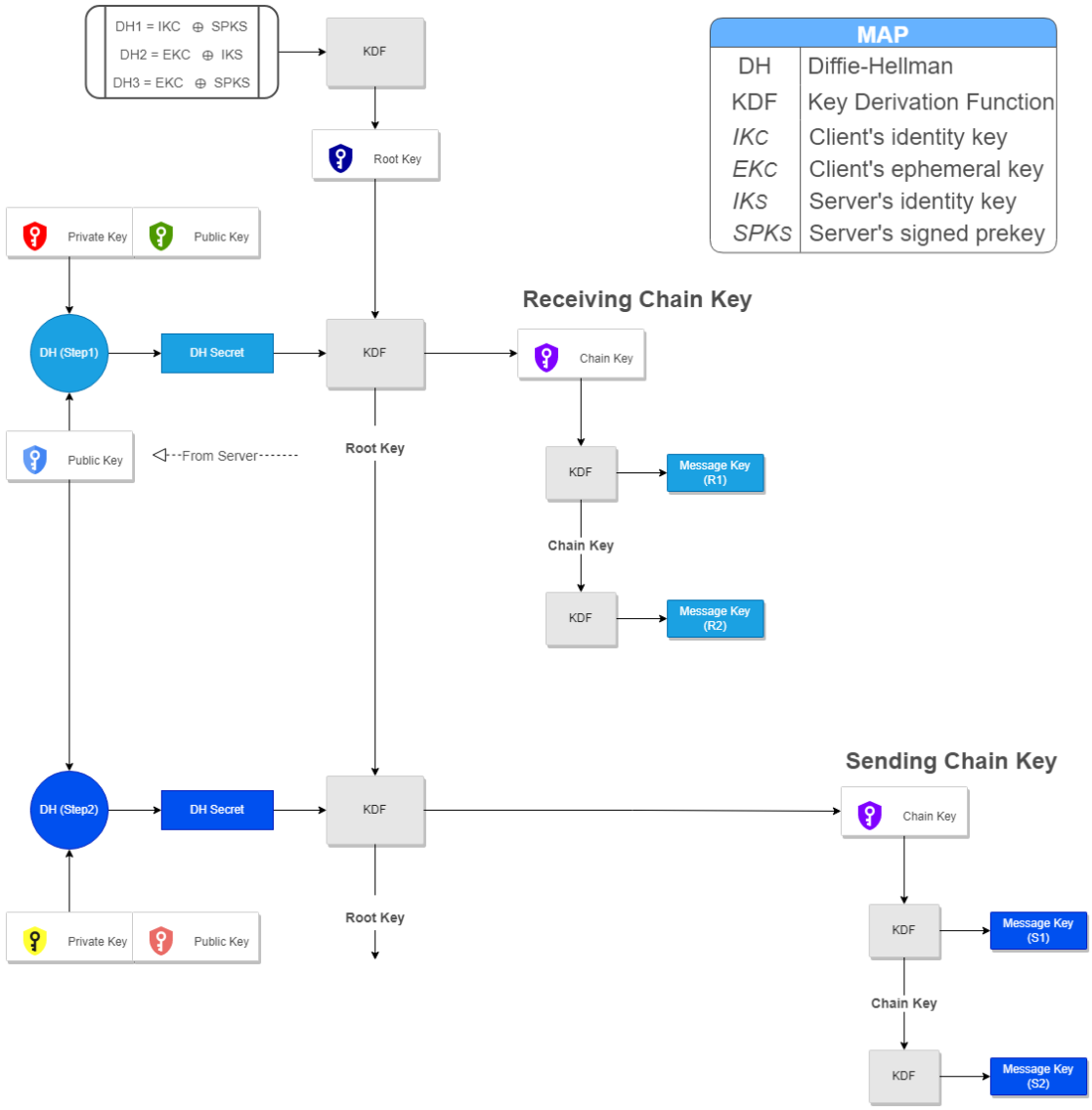

Figure 11: Security Overview

CHAPTER 4

Implementation Details

The implementation section of the paper includes a detailed description of the different modules that were developed based on the system design discussed previously. These modules include a security module for ensuring the confidentiality and integrity of data, key sharing mechanisms for secure key exchange between entities, cipher session management techniques to manage the state of the ratchets in the encryption and decryption algorithms, and an ID generation technique for generating unique identifiers for the bundle, client and transport. The window module was developed to manage the flow of data between the client and server, with separate modules for client and server-side window management. Finally, the routing module was designed for efficient routing of data between client and server, with separate modules at each end.

4.1 Security Module

The security module is responsible for generating and storing the keys necessary for encrypting and decrypting data. By restricting access to the keys to only the security module, the system can ensure that unauthorized parties cannot access and use the keys. The public keys are stored as files in the pem format and are required keys are sent as part of the encryption header in the bundle.

4.1.1 Key Sharing

Before communication can take place between the client and server, the server's keys must be generated and baked into the application. The public components of the server's Identity signed prekey and Ratchet Keys are then stored within the application, as illustrated in Figure 12. This allows the client to have access to the necessary keys to initiate communication.

In addition to obtaining the server's keys, the client must also send its own keys

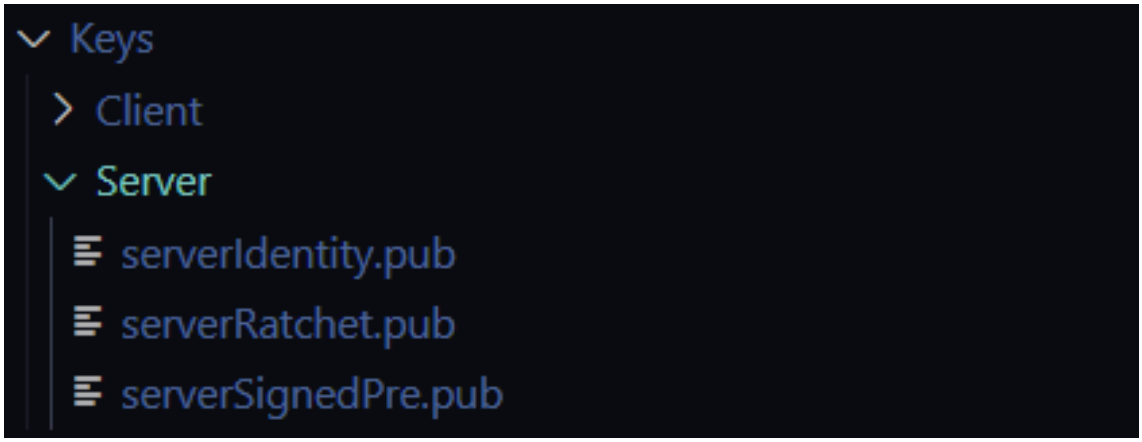

Figure 12: Server Keys on Client

to the server for the server to respond. These keys are sent as part of the bundle's encryption header, as depicted in Figure 13. The bundle ID is the name of the directory and stores the client's base and Identity key apart from the payload and its signature.

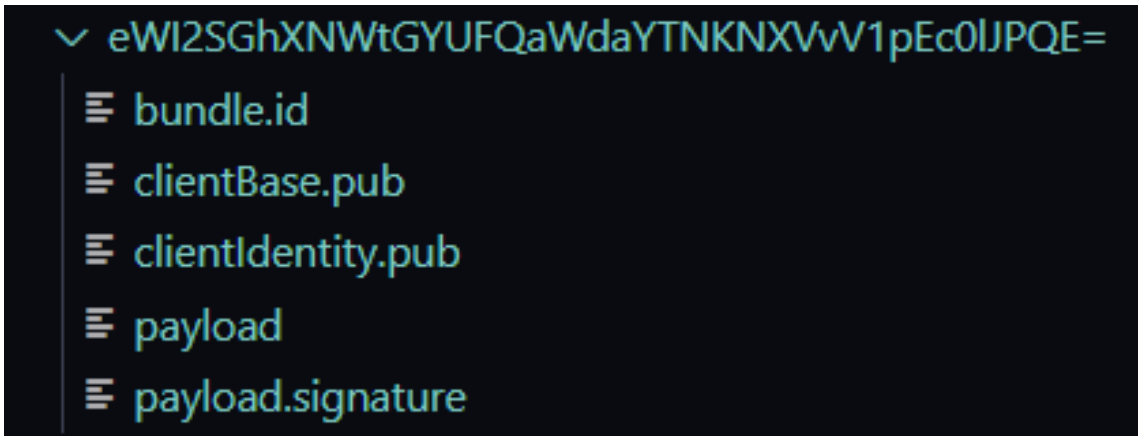

Figure 13: Final Bundle Structure

The signal protocol recommends the ratchet be rotated for each bundle that is sent. This means that a new Diffie Helman key pair is used to generate a shared secret, and the public component of this new ratchet key must be sent to the receiver. This public component is embedded into the data structure used for the encrypted message, as illustrated in Figure 14. The data structure stores the ciphertext and its

corresponding byte array format, as well as the current and previous counter values for the ratchet and ratchet keys. By employing these measures, the system can ensure the confidentiality, integrity, and authenticity of the communication between the client and server.

4.1.2 Cipher Management

As explained in section 3.6, the client's Identity and Ephemeral/Base, along with the server's Identity and Signed Prekey, are utilized to generate a Root key. Then, the server's ratchet key initializes the ratchet on the client, and the resulting secret and previously generated root key are used to create the sending and receiving chain keys.

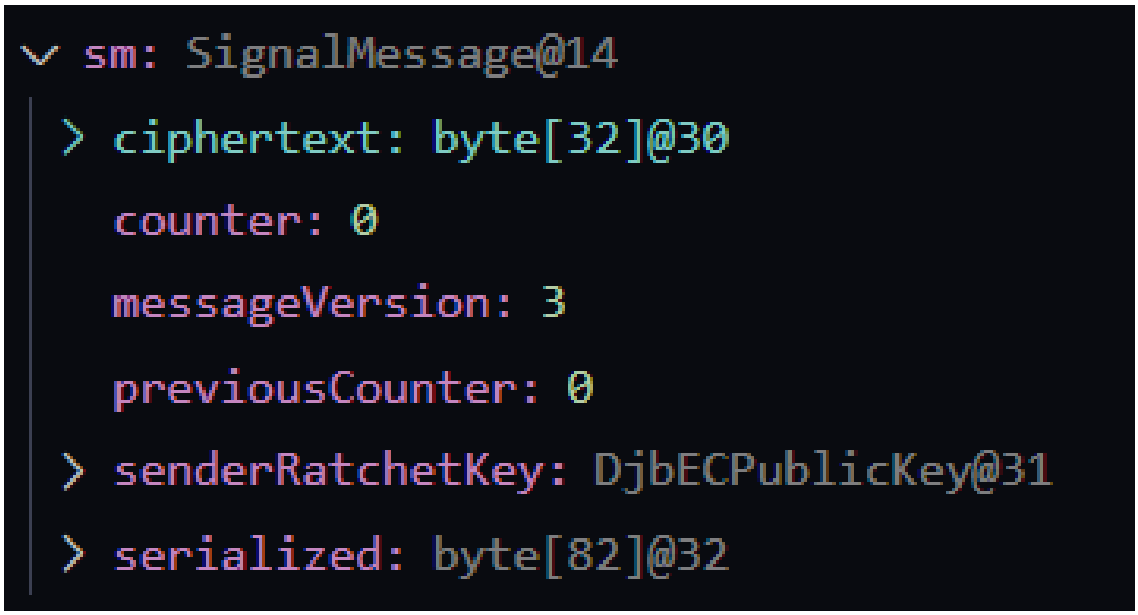

Figure 14: Encrypted Message Structure

The cipher session is then established, which is used to encrypt or decrypt data. It is also responsible for maintaining the ratchet synchronization by moving the respective counters. After performing the cryptographic operation, the cipher session returns either the decrypted byte array or the signal message object with the encrypted

data as shown in Figure 14. To handle multiple cipher sessions, the cipher is stored in an in-memory protocol store and is mapped to a protocol address consisting of the client ID and device ID, which helps in managing multiple cipher sessions, as is the case on the server that manages its clients.

The methods that can be used to initialize, encrypt, and decrypt bundles on the client are shown below in Figure 15:

```
/* Initialize or get previous client Security Instance */
public static synchronized ClientSecurity getInstance(int deviceID, String clientKeyPath, String serverKeyPath)

/* Encrypts File and creates signature for plain text */
public String encrypt(String toBeEncPath, String encPath, String bundleID)···

/* Decrypts the provided bundle and stores it in the specified path */
public void decrypt(String bundlePath, String decryptedPath)···
```

Figure 15: Client Security methods

The methods that can be used to initialize, encrypt, and decrypt bundles on the server are shown below in Figure 16:

```
/* Initialize or get previous server Security Instance */
public static synchronized ServerSecurity getInstance(String serverKeyPath)

/* Decrypts the provided bundle to the provided path···
public void decrypt(String bundlePath, String decryptedPath)···

/* Encrypts the data to the provided path for the specified client */
public void encrypt(String toBeEncPath, String encPath, String bundleID, String clientID)···
```

Figure 16: Server Security methods

4.2 ID Generation

There are 2 distinct types of IDs, those that identify a given node such as the client or the transport and those that identify a bundle.

4.2.1 Client & Transport ID Generation

The client and transport IDs are identified by their public keys to create a self-certifying identity. They are generated by creating a hash of their respective public

keys using the SHA-1 algorithm and encoding it to Base64. To ensure the IDs can be used safely as filenames, characters such as "/" and "+" are replaced with "_" and "-" respectively.

The methods that can be used to generate the client/transport ID are shown below in Figure 17:

```
/* Creates an ID based on the given public key file···
public static String generateID(String publicKeyPath) throws IOException, NoSuchAlgorithmException

/* Creates an ID based on the given public key byte array···
public static String generateID(byte[] publicKey) throws NoSuchAlgorithmException
```

Figure 17: Methods to generate an ID

The generateID() method is overloaded to support either the public key file or a byte array with the public key.

4.2.2 Bundle ID Generation

The bundle ID is a combination of the client ID and a counter value denoting the number of bundles sent from the node. The counter value is an unsigned long value and is 8 Bytes. It is added as a suffix to the client ID when the bundle is created in the upstream direction and as a prefix, in the downstream direction as described in Section 3.1 (Bundle Identifier). This combined value is then encoded to a URL-safe Base64 string following the same replacement technique as described in the previous section. This encoded value is then encrypted before being sent.

Bundle ID encryption is done using AES and the shared secret is created using the Diffie-Helman algorithm. The public Identity key of the receiver is used along with the private Identity key of the sender to generate this shared secret. This is used along with a salt value to encrypt the bundle ID. The resulting data is encoded to a URL safe Base64 string.

The methods that can be used to generate, compare and encrypt Bundle IDs are

shown below in Figure 18:

```
/* Generates bundleID···
public String generateBundleID(String clientKeyPath, boolean direction)···

/* Compares BundleIDs···
public static int compareBundleIDs(String id1, String id2, boolean direction)···

/* Encrypts the given bundleID···
public String encryptBundleID(String bundleID)···
```

Figure 18: Bundle ID methods

The direction field in the methods denotes the direction of the bundle, true to indicate Upstream and false to indicate Downstream.

4.3 Window Module

The window is used by the client and server to maintain synchronization between themselves. The server window is used as a transmission window while the client's window is used as a receiving window. It is implemented as a circular queue, the length of which must be the same at both ends. A default size of 10 is used if no size is provided during initialization. It is important to note that the bundle IDs in the window are not encrypted to make it easier to use within the module. However, prior to being shared outside the scope of the module, they are always encrypted.

4.3.1 Client Window

The client's window is initialized along with the client to the specified size. Bundle IDs are also generated for each slot in the window during initialization. When a transport is available and requests a list of bundle IDs, they are encrypted and the encrypted ID list is sent to the transport. Later, upon receiving bundles from the transport, the window is moved forward to the latest bundle ID received, and successive bundle IDs are then generated to fill in the new slots.

The methods that can be used to initialize, update, and retrieve the window are

shown below in Figure 19:

```
/* Allocate and Initialize Window with provided size···
public ClientWindow(int length, String clientKeyPath) throws InvalidLength, BufferOverflow···

/* Updates the window based on the Received bundleID···
public void processBundle(String bundlePath) throws IOException, RecievedOldACK, RecievedInvalidACK, InvalidLength, BufferOverflow···

/* Returns the entire window···
public String[] getWindow()···
```

Figure 19: Client Window methods

4.3.2 Server Window

The Server maintains a transmission window for each client. This is implemented as a hash table where the client ID points to the respective window. The window is the same length as sent by the client in its routing metadata. When a new bundle is to be sent, the window is checked to see if it is full; if yes, a new bundle cannot be sent, instead, the last bundle in the window is re-transmitted. If the window is not full, the bundle ID of the bundle to be sent is added to the window. When an ACK record is received from a client, its window is retrieved and moved ahead based on the received bundle ID, allowing new bundles to be sent later on.

The methods that can be used to add a client, update its window, retrieve the latest bundle ID, process an acknowledgment, and check the window's length are shown below in Figure 20:

```
/* Add a new client and initialize its window···
public void addClient(String clientID, int windowLength) throws InvalidLength···

/* Adds a new bundleID to the client's window···
public void updateClientWindow(String clientID, String bundleID) throws ClientNotFound, BufferOverflow···

/* Return the latest bundleID in the client's window···
public String getLatestClientBundle(String clientID) throws ClientNotFound···

/* Move window ahead based on the ACK received···
public void processACK(String clientID, String ackPath) throws ClientNotFound, InvalidLength, IOException···

/* Check if window is full···
public boolean isClientWindowFull(String clientID)···
```

Figure 20: Server Window methods

4.4 Routing Module

The routing module is used to map clients to transports through which they can be reached. The server stores this mapping and the client collects metadata to help maintain a score on the server. A reset is implemented to reset the score if the same score is received a set number of times (10 by default).

4.4.1 Client-side Routing

The client stores the transport and score mapping in a database to avoid data loss during crashes. The score denotes the number of times a transport has to deliver a valid bundle to the client, the validity of a bundle is determined by the security module. If a bundle has been decrypted successfully and had its signature verified, it is said to be a valid bundle. The transport score is incremented for each valid bundle it delivers. When a client transmits a bundle this mapping is exported as a JSON file and sent as part of the bundle's routing metadata field.

The methods that can be used to update the metadata and export it to a bundle are shown below in Figure 21:

```
/* Updates the score for the transport···
public void updateMetaData(String transportID) throws ClientMetaDataFileException···

/* Creates the metadata file in the bundle directory···
public void bundleMetaData(String bundlePath) throws ClientMetaDataFileException···
```

Figure 21: Client Routing methods

4.4.2 Server-side Routing

The routing table is stored here as a database with the transport ID, client ID, and score. The transport ID and the client ID together form the primary key to allow a client to be part of multiple transports. When a bundle is received, the transport and client ID pair are checked in the table. If it is not available, a new entry is added with the score added based on the routing metadata in the bundle. If an entry is

found, it is simply updated based on the value received from the client. If the received value is the same a set number of times (10 by default), it is reset to 0. When the clients for a respective transport is requested, the table is searched for the respective transport and client IDs, and the resulting client ID list is ordered by the score.

The methods that can be used to get the sorted list of clients and to process a client's metadata are shown below in Figure 22:

```
/* Returns the sorted list of clients accessible via the transport···
public List<String> getClients(String transportID) throws TransportIdNotFoundException···

/* Parses the client metadata and updates the respective scores···
public void processClientMetaData(String clientMetaDataPath, String clientID) throws ClientMetaDataFileException···
```

Figure 22: Server Routing methods

CHAPTER 5

Experiment

This section covers the experimental scenarios that can occur and how they are handled by the system. The following experiments were conducted in the following environment:

Java Version: 17.0.6

Android version: 13

Signal Library Version: 2.2.0

My SQL Version: 8.0.33

5.1 Client and Server Window

Figure 24 is an illustration describing the scenarios tested and the state of the window AFTER each transmission/reception with the following configurations:

(1) Window size of 3 on each side

(2) Server has received a bundle from the client

Note: The bundle IDs are represented as S0, S1 ... Sn for simplicity, the actual bundle IDs will be encrypted and encoded as described previously. Actual bundle IDs being sent to the transport are shown in Figure 23:

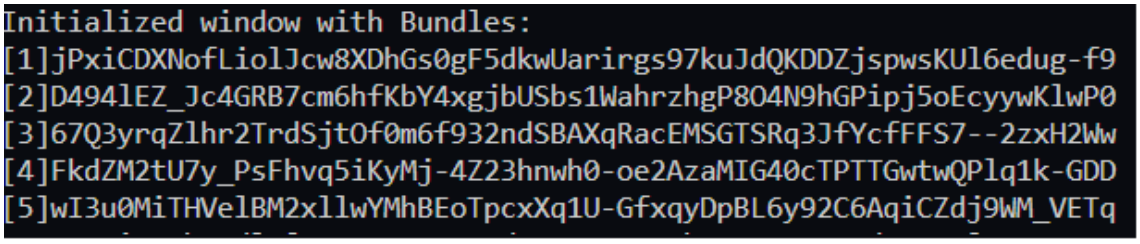

Figure 23: Encrypted Bundle IDs with their counter values

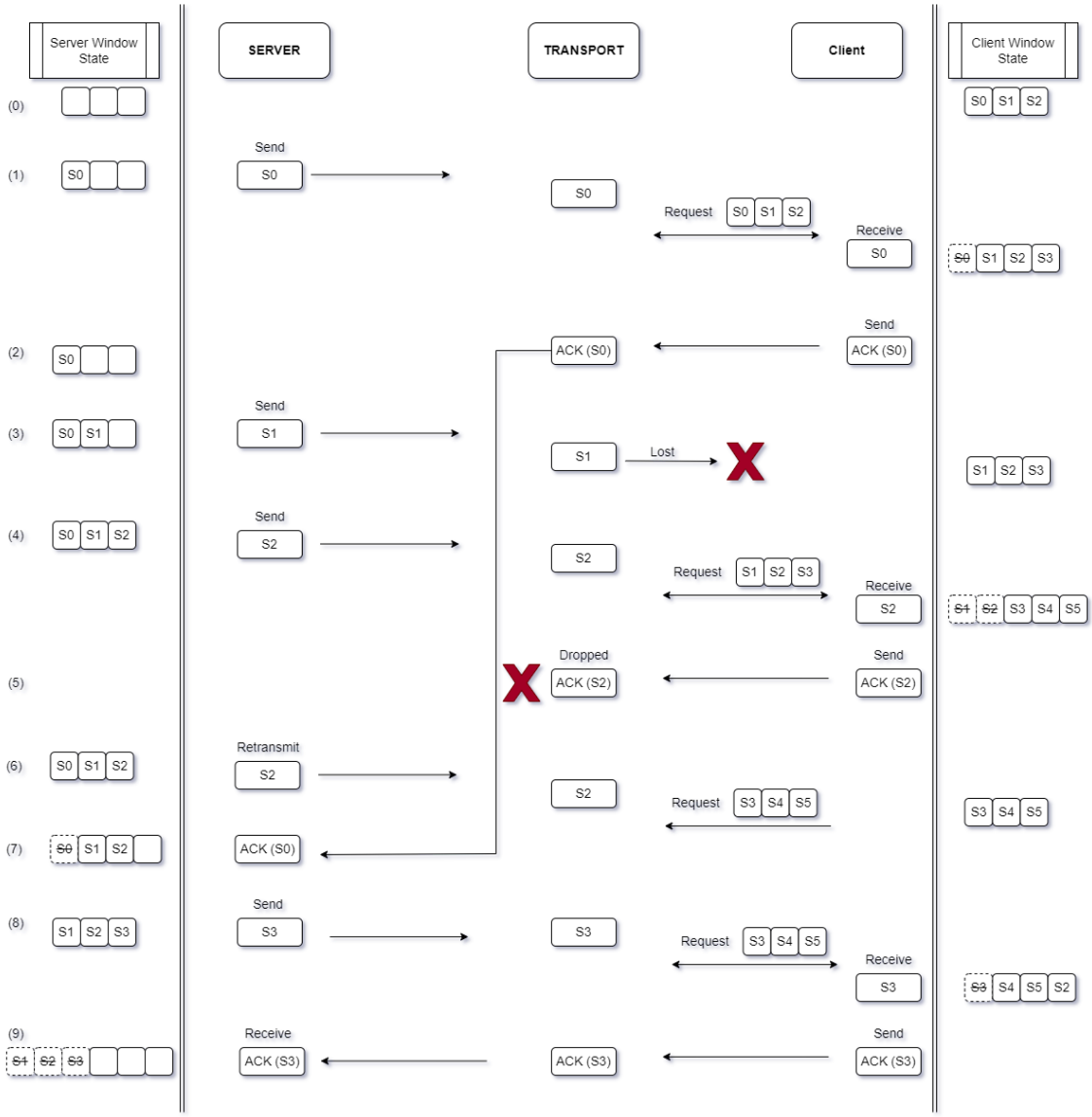

Figure 24: Scenarios for client and server window

5.1.1 Experimental Scenarios

Scenario 0: Initialize Window

Result (i) : Server creates a window of size 3 with empty slots

Server Window: (null, null, null)

Result (ii) : Client also creates a window of size 3 and fills Bundle IDs (S0, S1,

S2)

Client Window: (S0, S1, S2)

Scenario 1: Server tries to send S0 **(Successful Delivery)**

Result (i) : Server checks if the window is full, as it is not, it adds S0 to the window and sends S0 to the respective transport

Server Window: (S0, null, null)

Result (ii) : Client requests all bundle IDs in its window (S0, S1, S2) from the transport. The transport only sends S0 as it does not have S1 and S2. On receiving S0, the client window is moved ahead by 1 (as S0 was in position 1 in the window).

Client Window: (S0, S1, S2) $\rightarrow$ (S1, S2, S3)

Scenario 2: Client tries to send ACK for S0 **(Out of Order ACK)**

Result (i) : Client sends an ACK for the last received bundle ID (S0)

Result (ii) : Server does not receive anything at time 2, therefore does nothing.

Scenario 3: Server tries to send S1 **(Bundle Lost)**

Result (i) : Server checks if the window is full, as it is not, it adds S1 to the window and sends S1 to the respective transport

Server Window: (S0, S1, null)

Result (ii) : This is lost and therefore does not get to the client

Result (iii) : Client does not receive anything at time 3, therefore does nothing

Scenario 4: Server tries to send S2 **(Successful Delivery)**

Result (i) : Server checks if the window is full, as it is not, it adds S1 to the window and sends S1 to the respective transport

Server Window: (S0, S1, S2)

Result (ii) : Client requests all bundle IDs in its window (S1, S2, S3) from the transport. The transport only sends S2 as it does not have S1 and S3. On receiving S2, the client window is moved ahead by 2(as S2 was in position 2 in the window).

Client Window: (S1, S2, S3) $\rightarrow$ (S3, S4, S5)

Scenario 5: Client tries to send ACK for S2 **(Lost/Dropped ACK)**

Result (i) : Client sends an ACK for the last received bundle ID (S2)

Result (ii) : This is dropped by the transport and does not get to the server

Result (iii) : Server does not receive anything at time 5, therefore does nothing

Scenario 6: Server tries to send S3 **(Retransmission)**

Result (i) : Server checks if the window is full, it is. It cannot send the new bundle (S3).

Result (ii) : It now retransmits the last bundle in the window (S2)

Result (iii) : Client requests all bundle IDs in its window (S3, S4, S5) from the transport. The transport only has S2 for the client and so does not send anything to the client

Result (iv) : Client does not receive anything at time 6, therefore does nothing

Scenario 7: Server receives ACK **(Out of order ACK)**

Result (i) : Server now receives the ACK sent at time 2 by the client. The ACK received is for Bundle ID S0, the server can now move its window ahead by 1 (as S0 was the first ID in the bundle).

Server Window: (S0, S1, S2) → (S1, S2, null)

Scenario 8: Server tries send S3 again **(Post retransmission)**

Result (i) : Server can now send the bundle S3 as the window is not full after processing the ACK. It adds S3 to the window and sends it to the respective transport

Server Window: (S1, S2, null) → (S1, S2, S3)

Result (ii) : Client requests all bundle IDs in its window (S3, S4, S5)) from the transport. The transport only sends S3 as it does not have S4 and S5. On receiving S3, the client window is moved ahead by 1(as S3 was in position 1 in the window).

Client Window: (S3, S4, S5) → (S4, S5, S6)

Scenario 9: Client tries to send ACK **(Successful Delivery)**

Result (i) : Client sends an ACK for the last received bundle (S3)

Result (ii) : Server receives the ACK bundle and moves its window by 3 (as S3 is in position 3 in the window).

Server Window: (S1, S2, null) → (null, null, null)

5.2 Client and Server Security

Figure 25 describe the ratchet steps that take place to create encryption/decryption keys. Each bundle also consists of the N and PN values that describe the current and previous chain numbers and are used to synchronize the chains at either end.

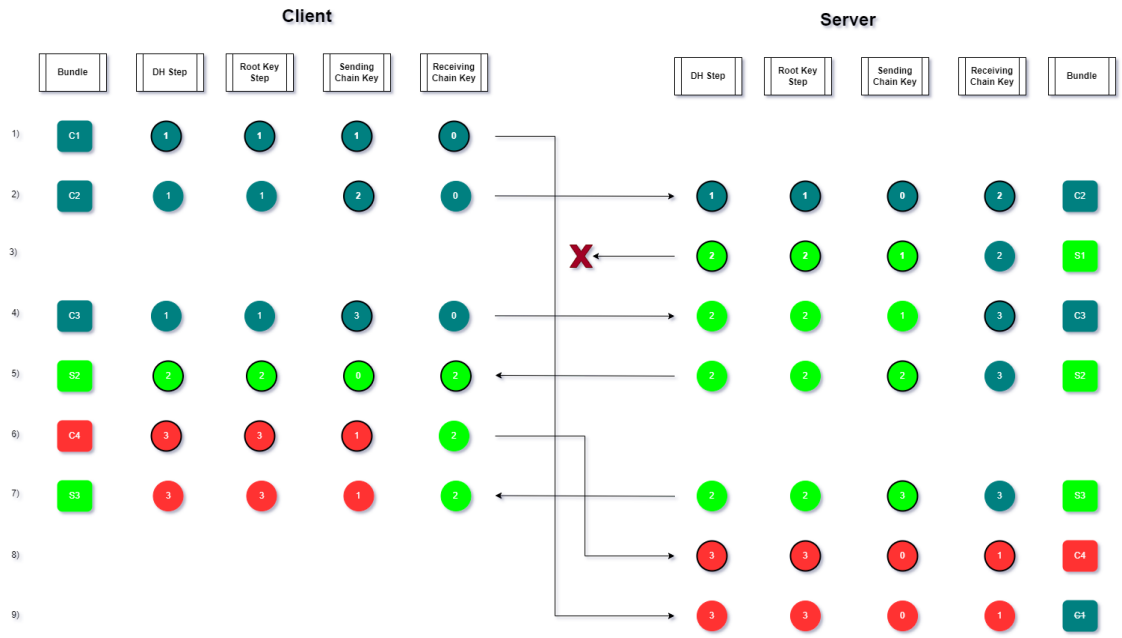

Figure 25: Ratchet Scenarios

Scenario 1: Client tries to send C1 **(Out of Order)**

Result (i) : Client sends bundle C1 with its first DH and sending chain key ratchet step

Result (ii) : Server does not receive it at time 1, therefore does nothing

Scenario 2: Client tries to send C2 **(Server Ratchet Initialization)**

Result (i) : Client moves its sending ratchet to 2 with all other ratchets remaining the same

Result (ii) : Server receives the bundle with a new public key, initializes its ratchets with the key and moves its receiving ratchet to the N value provided (2). The sending ratchet stays the same.

Scenario 3: Server tires to send S1 **(New Ratchet Step)**

Result (i) : The server moves its DH ratchet to 2, resulting in resetting the sending ratchet. It is now incremented and a key is generated using the new DH secret. The receiving ratchet is not reset as it has not received a new public key from the client.

Result (ii) : The client does not receive the bundle and therefore does nothing

Scenario 4: Client tries to send C3 **(Out of Order Delivery)**

Result (i) : Client moves its sending ratchet to 3 with all other ratchets remaining the same

Result (ii) : Server receives the bundle with the previous key, the receiving chain is, therefore, incremented to the N value of 3. All other ratchets remain the same.

Scenario 5: Server tries to send S2 **(Reset Client Ratchets)**

Result (i) : Server moves its sending ratchet to 2 with all the other ratchets remaining the same

Result (ii) : Client receives the bundle with the new public key and resets all of its ratchets. It then moves its receiving ratchet to the N value of 2.

Scenario 6: Client tries to send C4 **(Increment Client DH Ratchet)**

Result (i) : Client now increments its DH ratchet, resetting its sending key ratchet as well. The sending ratchet is then incremented. The receiving ratchet stays the same.

Result (ii) : Server does not receive anything and therefore does nothing

Scenario 7: Server tries to send S3 **(Successful Delivery)**

Result (i) : Server moves its sending ratchet to 3 with all the other ratchets remaining the same

Result (ii) : Client receives the bundle with the previous key, the receiving chain is, therefore, incremented to the N value of 3. All other ratchets remain the same.

Scenario 8: Server receives C4 **(Out of Order Delivery)**

Result (i) : Server receives bundle C4, it has a new public key, a new DH step is taken and all of the ratchets are reset. The receiving chain is then set to the N value of 1.

Scenario 9: Server receives C1 **(Out of Order Delivery)**

Result (i) : Server receives an older message C1, this is ignored as newer messages have already been received and the window is moved ahead (See window section)

CHAPTER 6

Future Work

In this section, we discuss potential areas for future work that can further improve the efficiency and performance of the proposed architecture. The highlighted areas have been scoped to have better solutions than the present implementation and can be improved with further research. The major area of improvement is the routing module as it has assumptions in the current implementation that may not be feasible in real world scenarios.

6.1 Improve Scoring Mechanism

The existing scoring mechanism for transports prioritizes clients based on their frequency of accessibility via a transport. However, the mechanism does not account for the decay of the score over time and instead resets the score if the same value is received a predetermined number of times. To improve the scoring mechanism, additional metrics such as round trip time could be incorporated, enabling the creation of a dynamic timeout for each entry in the scoring table. This approach would help to analyze the time taken for each bundle to be received by the client and dynamically time out the entry. By doing so, clients with shorter trip times could be prioritized, while also allowing for the detection of disconnections or disruptions in the network.

Incorporating a dynamic timeout mechanism into the scoring mechanism would therefore improve the overall efficiency and performance of the network by detecting and prioritizing clients with shorter trip times. Additionally, by detecting disconnections or disruptions in the network, the mechanism could enable more efficient retransmission of lost data, further enhancing the network's overall robustness and resilience.

6.2 Multi-hop Transports

At present, the number of transports is limited to a single hop between the client and the server. However, adding multiple hops to the network could enable a dynamic route to be calculated each time, thereby improving both the delivery time and the robustness of the network. The utilization of multiple hops would also allow transports to establish an ad-hoc network by utilizing principles of epidemic routing, wherein all bundles on a transport are forwarded to another transport when they come into contact with each other. This would enable the creation of a more efficient and resilient network, as it would reduce the likelihood of data loss and increase the overall connectivity of the network.

Another potential approach would be to have transports register themselves with the server, which would allow the server to create predetermined paths that bundles could take. This could reduce the number of copies in the network and provide a more efficient method of data transmission. By implementing this approach, it may be possible to optimize the network's routing and reduce the overall time required for data transmission.

6.3 Intermediate Servers

When deploying this architecture in regions with inadequate network infrastructure, such as rural or low-income neighborhoods, it may be advantageous to position a designated server closer to the source, in order to decrease the transport time required for connectivity. The intermediate server can have slower data connection speeds and only needs to be capable of connecting to the Bundle server, as all bundles can be forwarded from that location. This would increase the capital expenditure, as extra hardware would be required; however, state-of-the-art equipment capable of handling large data volumes would not be necessary. Instead, a relatively smaller device that

can connect to the primary connectivity provider over the air would suffice. This approach would enable improved internet access at a much lower cost than expanding the existing network infrastructure.

CHAPTER 7

Conclusion

The proposed architecture places a high emphasis on security, utilizing end-to-end encryption techniques to ensure data confidentiality, integrity, and authentication. All data received from the transport at either end is not trusted until it can be decrypted and verified successfully, preventing unauthorized access to sensitive information. Intermediate nodes do not need to be trusted in this case as they simply act as a forwarding medium between the client and server, further enhancing the overall security of the architecture.

Additionally, the architecture uses windows to regulate the bundles sent over the network to maintain synchronization between the client and server at all times. A routing table is also used to determine the path that the response from the server should take back to the client. The client collects and sends metadata to the server to improve the effectiveness of the routing table. This approach ensures that the response is routed back to the client in the most efficient way possible, reducing latency and enhancing overall network performance. Moreover, the proposed architecture eliminates the need for additional hardware, making it a cost-effective solution for providing internet access in areas where connectivity is limited.

However, there are some limitations to this architecture. For instance, the system relies on mobile devices to create the network, which can limit the range of the network. Additionally, the network may be slow due to the limited bandwidth of the mobile devices. The system also requires users to download and use specific applications to access the internet, which may not be feasible in some cases.

Despite these limitations, the proposed architecture has the potential to change lives and make a real difference in the world by providing internet access to disaster-affected areas or regions with limited connectivity. The system can be implemented

quickly and cost-effectively, making it a viable option for emergency situations. It has the potential to positively impact countless individuals and communities by providing reliable, secure and cost-effective internet access in challenging environments.

LIST OF REFERENCES

[1] Z. F. Yongwang Liu, ''The digital divide and covid-19,'' United Nations Economic and Social Commission for Asia and the Pacific, Tech. Rep., 2022.

[2] S. F. C. Caini, H. Cruickshank and M. Marchese, ''Delay and disruption tolerant networking (dtn): An alternative solution for future satellite networking applications,'' *Proceedings of the IEEE*, vol. 99, no. 11, pp. 1980--1997, Nov 2011.

[3] S. Todd, ''What is the half-life of data?'' https://www.linkedin.com/pulse/half-life-data-scottie-todd/, 2021, (Accessed on 12/17/2022).

[4] K. Fall, ''A delay-tolerant network architecture for challenged internets,'' *Special Interest Group on Data Communications (SIGCOMM)*, pp. 27--34, Aug 2003.

[5] M. Alfonzo, S. T. Integrados, J. A. Fraire, E. Kocian, and N. Alvarez, ''Development of a dtn bundle protocol convergence layer for spacewire,'' *2014 IEEE Biennial Congress of Argentina (ARGENCON)*, pp. 770--775, 2014.

[6] S. Burleigh, K. Fall, and E. Birrane, ''Rfc 9171: Bundle protocol version 7,'' <https://www.rfc-editor.org/info/rfc9171>, Jan 2022.

[7] S. Symington, S. Farrell, H. Weiss, and P. Lovell, ''Bundle security protocol specification,'' <https://www.rfc-editor.org/info/rfc6257>, Jan 2011.

[8] A. Vahdat and D. Becker, ''Epidemic routing for partially connected ad hoc networks,'' Duke University, Tech. Rep., 2000.

[9] P. Garg, H. Kumar, R. Johari, P. Gupta, and R. Bhatia, ''Enhanced epidemic routing protocol in delay tolerant networks,'' *5th International Conference on Signal Processing and Integrated Networks*, pp. 396--401, 2018.

[10] T. Choksatid, W. Narongkhachavana, and S. Prabhavat, ''An efficient spreading epidemic routing for delay-tolerant network,'' *13th IEEE Annual Consumer Communications and Networking Conference*, pp. 473--476, 2016.

[11] J. Shen, S. Moh, and I. Chung, ''Routing protocols in delay tolerant networks: A comparative survey,'' *International Conference on Circuits/Systems, Computers and Communications*, no. 23, 2008.

[12] G. Sandulescu and S. Tehrani, ''Opportunistic dtn routing with window-aware adaptive replication,'' *Asian Internet Engineering Conference*, no. 4, 2008.

[13] A. D. A. Lindgren and O. Schelen, ‘‘Probabilistic routing in intermittently connected networks,’’ *ACM SIGMOBILE Mobile Computing and CommunicationsReview*, vol. 7, no. 3, pp. 19--20, 2003.

[14] N. G. S. Pathak and N. Raja, ‘‘A survey on prophet based routing protocol in delay tolerant network,’’ *International Conference on Emerging Trends and Innovation in ICT*, pp. 110--115, 2017.

[15] S. Symington, S. Farrell, H. Weiss, and P. Lovell, ‘‘Rfc 6257: Bundle security protocol specification,’’ *Internet Engineering Task Force*, no. 6257, 2011.

[16] E. J. Birrane and K. McKeever, ‘‘Bundle protocol security (bpsec),’’ *Internet Engineering Task Force*, no. 9172, 2022.

[17] M. N. M. Bhutta, H. Cruickshank, and A. Nadeem, ‘‘A framework for key management architecture for dtn: Requirements and design,’’ *International Conference on Advances in the Emerging Computing Technologies*, pp. 1--4, 2019.

[18] M. N. M. Bhutta, H. Cruickshank, and Z. Sun, ‘‘An efficient, scalable key transport scheme (eskts) for delay/disruption tolerant networks,’’ *Springer*, pp. 1597--1609, 2014.

[19] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel, ‘‘Separating key management from file system security,’’ *ACM Symposium on Operating Systems Principles*, no. 17, pp. 124--139, 1999.

[20] Z. Wilcox-O’Hearn and B. Warner, ‘‘Tahoe – the least-authority filesystem,’’ Cryptology ePrint Archive, Paper 2012/524, 2012, https://eprint.iacr.org/2012/524. [Online]. Available: https://eprint.iacr.org/2012/524

[21] J. Martin, T. Mayberry, C. Donahue, L. Foppe, L. Brown, C. Riggins, E. C. Rye, and D. Brown, ‘‘study of mac address randomization in mobile devices and when it fails,’’ *Proceedings on Privacy Enhancing Technologies*, pp. 365--383, 2017.

[22] T. Perrin and M. Marlinspike, ‘‘The double ratchet algorithm,’’ https://signal.org/docs/specifications/doubleratchet/, Signal, Tech. Rep., 2016.

[23] M. Marlinspike, T. Perrin, and V. Stuart, ‘‘X3DH: Extensible messaging and presence protocol (xmpp) end-to-end key agreement and signature,’’ https://www.signal.org/docs/specifications/x3dh, July 2016, accessed on: Apr. 08, 2023. [Online]. Available: https://signal.org/docs/specifications/x3dh/

[24] R. Khan, K. McLaughlin, B. Kang, D. Laverty, and S. Sezer, ‘‘A novel edge security gateway for end-to-end protection in industrial internet of things,’’ *IEEE Power and Energy Society General Meeting*, pp. 1--5, 2021.

[25] Y. Zhu, Y. Zeng, J. Xu, Y. Xie, and J. Jiang, ‘‘A research on the authentication scheme for 5g network based on double ratchet algorithm,’’ *International Conference on Communication Engineering and Technology*, no. 5, pp. 49--53, 2022.

[26] N. I. of Standards and Technology, ‘‘Digital signature standard (dss),’’ Federal Information Processing Standards (FIPS) Publication 186-5, 2019. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf

[27] R. Kaur and A. Kaur, ‘‘Digital signature,’’ in *2012 International Conference on Computing Sciences*, 2012, pp. 295--301.

[28] Cendyne, ‘‘A deep dive into ed25519 signatures,’’ https://cendyne.dev/posts/2022-03-06-ed25519-signatures.html, Mar 2022.