

Secure Protocol for Courier-Dependent Communication



Chen Sun
Trinity College
University of Oxford

A thesis submitted for the degree of
Master of Science

Trinity 2014

Acknowledgements

Abstract

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Aims and Objectives	2
1.3	Dissertation Structure	3
2	Background and Related Work	5
2.1	Delay-Tolerant Networks (DTN)	5
2.2	Deniable Authentication	7
2.3	Bundle Security Protocol [3]	7
2.4	DTN Anonymity and Secure Architecture [1]	9
3	System Description	12
3.1	System Model	12
3.1.1	Overview	12
3.1.2	Initial Set-ups	12
3.1.3	Communication Model	14
3.1.4	Honest Entity Behaviour	15
3.2	Adversary Model	17
4	Protocol Specification	19
4.1	Notations	19
4.2	Submit Protocol	20
4.2.1	Preconditions	20
4.2.2	Postconditions	20
4.2.3	Message Sequencing Chart	21
4.2.4	Specification	21
4.3	Transmit Protocol	23
4.3.1	Preconditions	23
4.3.2	Postconditions	23

4.3.3	Message Sequencing Chart	23
4.3.4	Specification	24
5	Security Analysis	26
5.1	Protocol Properties	26
5.1.1	Submit protocol	26
5.1.2	Transmit protocol	27
5.1.3	On the whole	27
5.2	Property Proof	28
5.2.1	Primitives	28
5.2.2	Submit protocol	29
5.2.3	Transmit Protocol	31
5.2.4	On The Whole	31
6	Protocol Implementation	33
6.1	Application Overview	33
6.2	Software Design	36
6.2.1	Program Architecture	37
6.2.2	Delegate Model	39
6.2.3	Message Structure	40
6.3	Implementation Details	44
6.3.1	UML Class Diagram	44
6.3.2	Connection Establishment	46
6.3.3	Message Processing	47
6.3.4	Cryptographic operations	47
6.3.5	File Management	48
6.3.6	Error Handling	49
7	Test and Evaluation	51
7.1	Latency Test	51
7.1.1	Submit Protocol	51
7.1.2	Transmit Protocol	51
7.2	Scalability Test	51
7.2.1	Submit Protocol	51
7.2.2	Transmit Protocol	51

8	Conclusion and Future Work	52
8.1	Conclusion	52
8.2	Discussion and Future Work	53
A	Application Instruction	55
A.1	Run Submit Protocol	56
A.2	Run Transmit Protocol	58
	Bibliography	61

List of Figures

2.1	Overview of a Delay-Tolerant Network, cited from [6]	6
2.2	Message Format of Bundle Protocol [2]	9
2.3	Rural Area DTN [10]	10
3.1	Communication Model	14
6.1	GUI Overview	34
6.2	Use Case Diagram	34
6.3	Acceptor Architecture	39
6.4	Initiator Architecture	40
6.5	State Machine of \mathcal{A}	41
6.6	Structure of Meta Block	42
6.7	Structure of Message Block	42
6.8	Normal Message and Error Message	44
6.9	UML Diagram	44
A.1	GUI Overview	56
A.2	Running Submit Protocol	58
A.3	Running Transmit Protocol	60

Chapter 1

Introduction

1.1 Motivation

In modern world, Internet plays a significant role in everyone's daily life as it breaks the geographical boundary and provides services all over the world. Its major achievement is that it allows each pair of end-to-end entities to build stable and reliable connections between each other so that communication becomes no longer a problem. However, there are still many places out of the reach of Internet. A simple example would be a isolated village in the deep of a mountain where infrastructures have not been built, or some extreme environments like interplanetary or under deep ocean where stable and continuous connectivity can not be achieved. In these cases, Internet may fail to provide its service. Entities in such environments can be abstracted as off-line entities as Internet is no longer available to them.

One way to achieve communication between such off-line entities is using a portable device as a courier to deliver messages. Assuming there are two off-line entities Alice and Bob, and Alice wants to send Bob a message. What could happen is a portable device (maybe several) called Courier copies the message from Alice, physically carries the message to Bob and then delivers the message to Bob. Through this intuitive and practical method, the communication between is achieved.

However, despite the probable efficiency defect it may have, we concern about security issues of this kind of communication. Assuming the content of the message is highly confidential, how to prove the authenticity of the message? How to protect it from being revealed to other entities under the threat that the Courier might be intercepted and the data it carries can be examined during the transportation? Bearing those questions in mind, we construct a specific scenario where these security issues are taken into account:

Assuming there are some off-line entities $Alice_0, Alice_1, \dots, Alice_m$, they want to send secret messages to another off-line entity Bob by using portable devices as couriers to help delivering the message. However, these entities are separated by a border - $Alice_0, Alice_1, \dots, Alice_m$ are located on one side of the border while Bob is located on the other side of the border, and crossing the border not only is time consuming but also requires all data examined by the security guard. Thus, to make a successful delivery, a Courier first approaches $Alice_0, Alice_1, \dots, Alice_m$ one by one and get messages from them. Then it crosses the border carrying those messages and be examined. Finally it approaches Bob and delivers those messages to it. As each message sent by Alice is considered highly confidential, it is expected that the message content will never be leaked to a third party (including Courier).

This project is designated to design and implement a protocol for this specific scenario to allow this kind of courier-dependent connection established efficiently and securely between Alices, Bob and the Courier. The protocol name is called Courier-Dependent Security Protocol, or CDSP as abbreviation. Although CDSP is created under above specific settings, the application of it can be generic, as it can be used in many courier-dependent communication scenarios.

1.2 Aims and Objectives

As stated above, this project aims to create a protocol between off-line entities Alices, Bob and Courier to provide security assurance for the message content they exchanged. The detailed scenario has been briefly introduced and the basic requirements have been clarified. More concretely, the invented CDSP takes following properties into consideration:

- Authentication

As the cost of such communication will be high - the transportation of courier is costly, it requires that the Courier only delivers message for authenticated entity. Namely, if Courier wants to deliver message for $Alice_0$, $Alice_0$ must prove its identity before start submitting its message. Same authentication is required for Bob - Bob has to prove its identity before Courier transmits the message it demands.

- Authenticity of Origin

The authenticity of origin of messages should be preserved during the communication, which means when message recipient gets the message, he is able

to ensure the message creator. Meanwhile, it implies the integrity of message should be preserved.

- Confidentiality of Message Content

The message content carried by courier should be kept confidential to any entities except the message creator and recipient. Because there is possibility that the courier could be compromised and data it carries may be examined by third parties, courier should gain no knowledge of the actual content it carries.

- Deniability

As mentioned above, in secret delivery missions, it would be plausible if Alice is able to deny sending the message in such a way that those who got the message can not prove its authentication to any third parties.

- Efficiency

Due to the potential limitation of computing and storage capability of Courier and the high cost of such communication, protocol should be designed in such a way that it uses least number of messages and smallest message size to achieve the goal. Especially for cryptography operations, where overhead for encrypting/decrypting messages could be high.

At the end of the project, following objectives must be achieved:

1. A fully specified Courier-Dependent Security Protocol should be created and it should meet the requirements mentioned in the above list.
2. A Java library should be developed providing essential functions for implementing the protocol.
3. An application should be built to actually run this protocol.
4. A test of the application should be done to evaluate the performance of the protocol application.

1.3 Dissertation Structure

In the rest of the dissertation, the detailed work will be presented. In Chapter 2 some related works will be discussed, and the differences of this project with those works will be highlighted. Then Chapter 3 will thoroughly describe the system that the protocol serves, including all set-ups, assumptions and rules. It is designated to convey

a detailed picture of the scenario. Chapter 4 will fully illustrate the specification of CDSP with the help of message sequencing charts and afterwards, in Chapter 5, the security properties of it will be highlighted and verified. After that, the implementation of the protocol application will be demonstrated in Chapter 6. No concrete code but graphs and charts will be given to give a high level understanding of the work. Then it moves to Chapter 7, where the design of tests and evaluations of the application will be shown and some data will be analysed to show the performance of the protocol. Finally, Chapter 8 will summarise the achievements of the project and draw some reasonable conclusions by pointing out the current system limitation and potential improvements in future work.

Chapter 2

Background and Related Work

Since the Delay-Tolerant Network architecture is first introduced, its security concerns have drawn the attention of relative research groups. Soon, much effort has been put on analysing the practicality of security implementation in DTNs and many designs have been made. Here two of security protocols which considered similar to this project are introduced as references.

2.1 Delay-Tolerant Networks (DTN)

Comparing to Internet which is relatively stable and reliable, so called “challenged networks” are characterized by latency, bandwidth limitations, error probability, node longevity, or path stability that are substantially worse than is typical of today’s Internet [6]. The reason for causing the networks “challenged” could be various - from geographical distances, lacking infrastructure to extreme environment conditions. Examples of such networks have been given by Fall, such as exotic media network - like near-earth satellite communication, and military ad-hoc networks. In order to achieve successful communication within such networks, instead of using TCP/IP - which will behave disappointingly [7], each challenged network may introduce its own protocol suits and network architectures to meet their specific needs. However, the diversity of these various protocols and architectures prevents those networks to communicate with each other and it has been justified by Fall [6] that simple link-repair approaches is not sufficient to solve the whole problem.

Then the architecture of Delay-Tolerant Network (DTN) was introduced to tackle the problem. Basically it achieve communication between various disparate challenged networks with significantly different sets of physical and operational constraints (latency, stability, etc.) by adding another layer of protocol to the local protocol stack. The Figure 2.1 briefly illustrates the architecture of a delay-tolerant network.

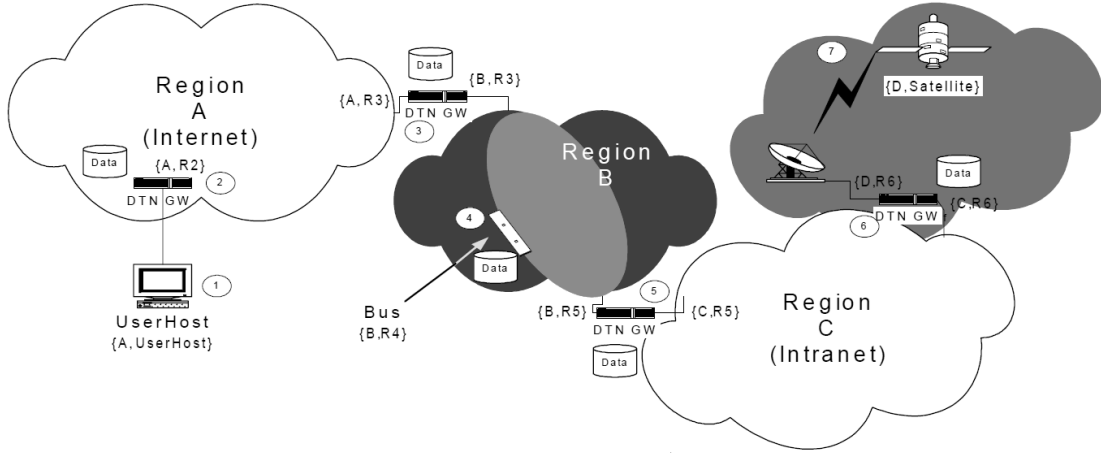


Figure 2.1: Overview of a Delay-Tolerant Network, cited from [6]

It separates different challenged networks into “regions” - within each region, nodes communicate under a same protocol, while different regions may run different protocols. As shown in the figure, region A uses Internet, region B uses a bus to deliver messages, region C uses its own Intranet and region C uses satellite to communicate. Different regions are connection by DTN gateways where messages from one region will be routed and forwarded to another region.

One of the key concept of DTN is custody transferring, which refers to the delivery of message from one DTN hop to another together with its reliable delivery responsibility - it’s very much like delivering packages through a postal service. And using message couriers to achieve such custody transfer has been accepted as one practical implementation to overcome the difficulties in DTN [20][5]. Although the security issues of DTN have been discussed and evaluated in the high level design of the architecture [8][2] (mainly to maintain efficient routing and prevent abuse of DTN), many security concerns still remain to be considered in the specific courier-dependent message delivery scenario (like what if the courier could be hijacked and examined?). Referencing the high level security framework pre-defined in the architecture, this project will dive into the detail of courier-dependent message delivery and create a practical and efficient security protocol for such scenario.

2.2 Deniable Authentication

Privacy protection has been taken much attention to since the communication through digital networks grows dramatically. Most network protocols require authentication as one of its essential steps to achieve secure communication. However, common authentications do not take privacy as one of its goals, thus they require revealing unique identifier of an entity to prove its identity to others - such like a digital signature or a piece of plaintext whose ciphertext can only be decrypted by the entity. As a matter of fact, it is this authentication mechanism disclose the identity of authenticator to third parties. Conversely, in many cases entities want to authenticate themselves to the target entities without revealing their identities to any third parties. This issues have been investigated and analysed by Abadi, who consequently introduced “private authentication” and presented protocols meet the requirements [13].

The concept of deniable authentication is created even earlier, by Dwork and Sahai [4]. It indicates the situation that an entity wishes to authenticate a message to the target entity while no any other entities can verify the authentication. Namely, the target entity can not prove the authenticity of the origin of the message to any other entities even itself is convinced by the authentication, thus the message creator can fully deny the existence of the message. It is extremely powerful in terms of privacy protection as combining with privacy authentication, no any private information will be leaked during the authentication. And its repudiation property is proved useful in applications like voting systems and commercial negotiation systems.

So far, many deniable authentication protocols have been invented, they can be sorted as two main classes - interactive [11] and non-interactive [14][15][9]. Interactive deniable authentication requires at least 2 messages exchanged during the protocol, one forward and one reply. While non-interactive deniable authentication can achieve the goal in just one message with the cost of heavier computation.

This feature is added in the CDSP to provide an extra level of security to the message being exchanged in order to against potential malicious operations of couriers and message recipients. It might be very important in circumstances like military network or secret message delivery missions.

2.3 Bundle Security Protocol [3]

Soon after the release of DTN architecture specification [8], a protocol called Bundle Protocol [2] is designed and documented by Scott et al., which formally defines the

format of messages (named bundles) exchanged between each end-to-end entities and abstracts the services provided by DTNs. Analogous to TCP/IP providing end-to-end connectivity specifying how data should be formatted, addressed, transmitted, routed and received between originator and destination in Internet, Bundle Protocol takes care of those issues for every DTN hops. Figure 2.2 shows the the basic block formats in bundle protocol. Basically, it defines that a single bundle should contain one primary bundle block and arbitrary number of bundle payload blocks. The primary bundle block is highly formatted, it describes the information about the whole bundle, while each bundle payload block provides different services that can be totally independent to each other and its format is relatively free to extend. This message structure is designed for extension as new functions could be easily added to the protocol by defining another new type of bundle payload block.

The Bundle Security Protocol is one of extensions of Bundle Protocol. It is designated to provide data integrity and confidentiality protection for the Bundle Protocol. The main contribution is defining 4 new types of bundle payload block - BundleAuthenticationBlock, PayloadIntegrityBlock, PayloadConfidentialityBlock and ExtensionSecurityBlock. Those new types of blocks - as their name stated, can be appended to any bundle to add an extra level of corresponding security protection to it. For example, if a bundle needs to be authenticated, the originator should add an extra bundle payload block to the original bundle, formatted as a BundleAuthenticationBlock. Thus the authentication of the bundle can be checked in every DTN hop during the transmission.

We can see that Bundle Security Protocol has defined the security issues in a high level, it ensures the secure message transmission between originator and receiver connected by hops. However, the detailed operations between hop to hop is not covered by this protocol. Referring to the architecture of DTN, every end-to-end nodes are linked by hops, and the message transmission from one hop to the next is called custody transfer. The implementation of custody transfer can be various, but a common method is courier-dependent transferring. In Bundle Security Protocol, such custody transferring is assumed to be accomplished securely and efficiently so that it only cares the higher level issues. Unfortunately, it is not always the case. Every custody transferring can be complicated and extra work should be done to keep it functioning. Thus this project will explore the specific scenario of courier-dependent custody transferring and create a protocol for it. Although the message format of Bundle Security Protocol is highly referential, it still not quite fit the scenario. The major problem is: as a general protocol it has a very heavy overhead to maintain the

Primary Bundle Block		
	Version	
	Proc. Flags (*)	
	Block length (*)	
	Destination scheme offset (*)	
	Destination SSP offset (*)	
	Source scheme offset (*)	
	Source SSP offset (*)	
	Report-to scheme offset (*)	
	Report-to SSP offset (*)	
	Custodian scheme offset (*)	
	Custodian SSP offset (*)	
	Creation Timestamp time (*)	
	Creation Timestamp sequence number (*)	
	Lifetime (*)	
	Dictionary length (*)	
	Dictionary byte array (variable)	
	[Fragment offset (*)]	
	[Total application data unit length (*)]	
Bundle Payload Block		
	Block type	
	Proc. Flags (*)	
	Block length(*)	
	Bundle Payload (variable)	

Figure 2.2: Message Format of Bundle Protocol [2]

consistency of different blocks and extensions, which seems redundant in the specific courier-dependent scenario. So comparing to the Bundle Security Protocol, the newly created protocol will use less-sized messages and be more problem specific.

2.4 DTN Anonymity and Secure Architecture [1]

The DTN Anonymity and Secure Architecture (DASA) is inspired by Seth et al. who tried to find a secure solution for helping rural areas to get continuous Internet access despite the long-period disconnection [10][12]. Although many different situations

may arise in rural areas, Seth et al.'s study focuses on a particular example of them: buses with wifi-based access and storage capability periodically drive past each villages and collect data from users in villages. Then buses carry the data to the nearest local Internet gateway and send out the data collected. Figure 2.3 roughly illustrates the scenario. Basically, Seth et al.'s architecture achieves the secure communication between each user and buses in such a way that every data delivery from user to bus must be mutually authenticated. Based on this achievement, Kate et al. abstracts the scenario such that it not only applies to rural area networks but also any generic DTNs. Furthermore, Kate et al. also adds two more security properties to the original architecture - data confidentiality and anonymity, forming the DTN Anonymity and Secure Architecture.

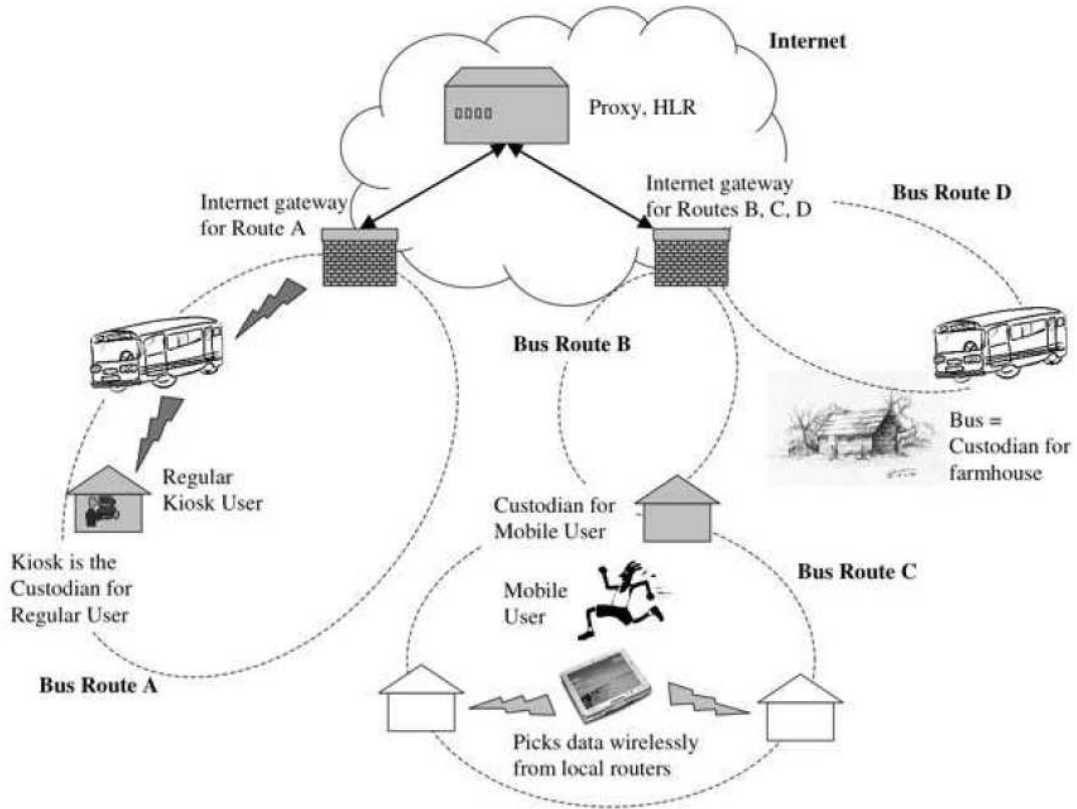


Figure 2.3: Rural Area DTN [10]

The DTN Anonymity and Secure Architecture is very close to the scenario of this project, as it takes care of the authentication processes between end users and couriers - here the buses, and conceals the identity of end users. However, there are still some points remain controversial: (1) It does not hide the message content

from the DTN router (courier), so the message content will be totally exposed once the DTN router is compromised. (2) It requires mutual authentication when end node transmit message to a DTN router (courier). Normally, mutual authentication is interactive or it demands for extra assumptions - such as Sakai-Ohgishi-Kasahara (SOK) Key Agreement Scheme [26], which leads to computational complexity, thus one way authentication might be a better choice. (3) Its anonymity mechanism highly relies on a trusted third party “private key generator” which may or may not be introduced in the system. (4) The message sent from end users is not fully deniable. According to the specification of its anonymity mechanism, it only conceals the sender identity behind a group of valid users identities. Although the courier does not know the true identity of sender, it at least can prove that one of the member in the group created the message to a third party.

Chapter 3

System Description

3.1 System Model

3.1.1 Overview

This protocol is typically used under the scenario that there is no reliable or stable networks existed between communication entities (called Alice and Bob). Entities under such conditions can be abstracted as off-line entities in the sense that their network is restricted and can not reach the others' networks. This system allows those off-line entities to communicate by using a Courier to deliver the message. Assuming Alice wants to send a message to Bob, the Courier firstly gets the message from Alice and stores it, then physically transport to Bob and send the message to Bob.

The goal of this protocol is to ensure the message from Alice to Bob is secure in the sense that no one can reveal its content but Bob. To achieve that, the Courier should never be trusted, which means the real content of the message should not be accessed by Courier and both Alice and Bob are able to deny the communication with Courier at any time. Furthermore, to prevent sensitive information from being leaked by malicious message recipient (Bob), message creator (Alice) should be able to deny the message content as well.

3.1.2 Initial Set-ups

According to the description above, totally 3 different types of entities are defined in the protocol - Alice, Bob and Courier. Following specification lists the notations and jobs of all 3 entities together with the information they should hold before running the protocol:

- Alice \mathcal{A} denotes a set of devices who create the message and waits it to be delivered, so it is also called “message creator”. It possesses an unique $\mathcal{ID} \in \{0, 1\}^*$, a secret key sk_A as part of its asymmetric key pair, and the message \mathcal{M} to be sent.
 \mathcal{A} ’s ID and public key should be known by at least one Courier so that it will be connected by Courier.
- Bob \mathcal{B} denotes a set of devices who wait incoming messages delivered by Courier, so it is also called “message recipient”. It possesses an unique $\mathcal{ID} \in \{0, 1\}^*$, and a secret key sk_B as part of its asymmetric key pair.
 Similarly, its ID and public key should also be known by at least one Courier.
- Courier \mathcal{C} denotes a set of devices who carry the message of Alice, physically transport from Alice to Bob, and deliver the message to Bob. Initially it only possesses an unique $\mathcal{ID} \in \{0, 1\}^*$ and at least a contact $\mathcal{ID} \in \{0, 1\}^*$ and its corresponding public key, specifies the entity it is going to contact.
 However, a single Courier will play two different roles in the protocol run - one receives message from Alice, one delivers message to Bob. They are denoted as \mathcal{CR} (Courier Receiver) and \mathcal{CS} (Courier Sender) in the protocol specification.
 In addition to \mathcal{CR} , \mathcal{CS} possesses some more information: the \mathcal{ID} of the message creator and the encrypted message received from \mathcal{A} .

Public Key Distribution The distribution of asymmetric key pairs used for authentication is out of the scope of this protocol, thus \mathcal{A} and \mathcal{B} are assumed to hold their asymmetric key pairs before running the protocol. Furthermore, all entities in the system are assumed to know each other’s public key (does not matter it is distributed with manufacture, authenticated by CA, or by key exchange), before running the protocol.

Devices v.s. Entities A device x denotes a physical object that runs the protocol. Differently, an entity denotes a particular role in the running of the protocol. It should be noted that any device can run multiple instances of this protocol with other devices simultaneously, thus a single device can be any three entities at the same time. The role it plays in different communications is defined by what information x holds and what sub-protocol it runs (will be described in Communication Model). However, an entity in the protocol can be associated with only one device.

3.1.3 Communication Model

Ultimately, every single run of this protocol achieves an abstract M-to-1 communication - a certain number of devices $a_0, a_1, \dots, a_M \in \mathcal{A}$ send message to a single device $b \in \mathcal{B}$ independently, using a device $c \in \mathcal{C}$ as media. As physical transportation is extremely slow and costly compare to network communication, the total number of physical transportation should be reduced to minimum. The optimized solution appeared to be separating the protocol into two main phases - Message Acquisition phase and Message Delivery phase. Assume off-line devices $a_1, a_2, \dots, a_M \in \mathcal{A}$ need to send message to the off-line device $b \in \mathcal{B}$. In Message Acquisition phase, a Courier will physically transport to every $a \in \mathcal{A}$, connect to it and get the message that is for the b . After the Courier collects all needed messages, it enters Message Delivery phase, where the Courier transports to b and transmit all acquired messages to it.

According to explanation above, the whole task can be divided into $M + 1$ individual communications. Every such individual communication happens between a Courier $c \in \mathcal{C}$ and one of $\mathcal{A} \cup \mathcal{B}$ after the Courier connects to the target. All these communications use one of typical network communication methods (e.g. cable, Wifi, Bluetooth, etc.) and apply its corresponding communication protocols (e.g. TCP/IP, UDP, Bluetooth protocol, etc.). The following figure shows how the communication is organized.

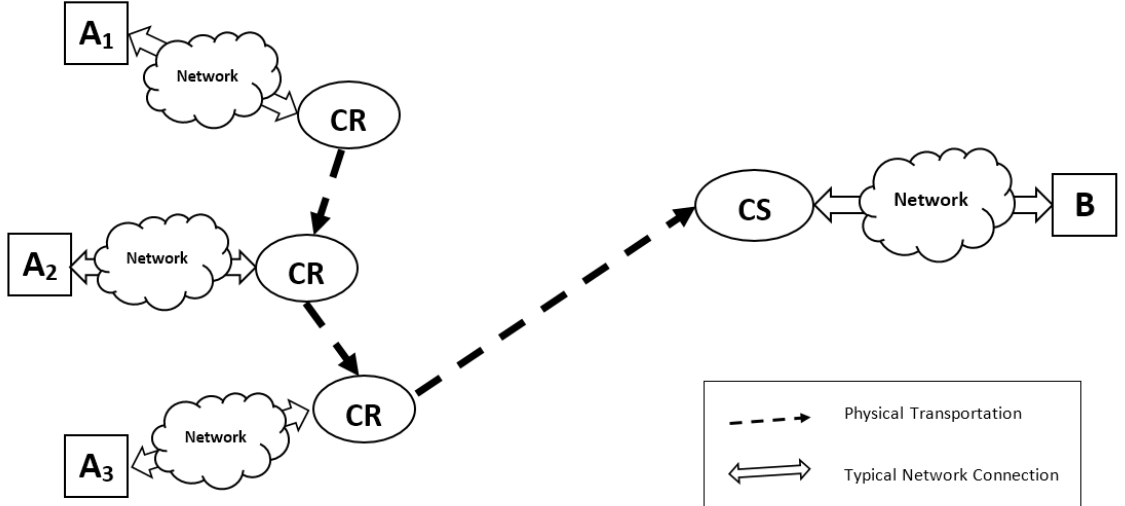


Figure 3.1: Communication Model

The figure "Communication Model" illustrates the procedure of a 3-to-1 communication. The Courier first connects to A_1 , be the role of \mathcal{CR} , and gets A_1 's message

for B through the connection, then disconnects and transports to A_2 . It should do the same thing to A_2 and then transport to A_3 . After it collects all data from A_1 , A_2 and A_3 , it transports to B and be the role of \mathcal{CS} to send all collected messages to B. The networks between Courier and As or B can be any kind of connection and do not to be the same, as long as both entities accept.

To define different kinds of communication, this protocol consists of 2 sub-protocols - a) Submit protocol: how message is submitted from \mathcal{A} to \mathcal{C} ; b) Transmit protocol: after \mathcal{C} gets connected with \mathcal{B} , how message is transmitted to \mathcal{B} . These two sub-protocols ensure the secure transmission of message from \mathcal{A} to \mathcal{B} and they will be analysed in the later part.

3.1.4 Honest Entity Behaviour

Generally, all 3 entities in this protocol act as devices who possess their own initial information (described above), take \mathcal{M} as input, and output \mathcal{M}' . All honest entities in this protocol should follow the following procedure:

1. If entity is \mathcal{C} , initiate the protocol by sending \mathcal{M}_0' , otherwise ignore this step.
2. Wait for \mathcal{M}_1
3. Receive \mathcal{M}_1 , decrypt all its ciphertexts and reveal their contents if applicable.
4. Check validity of the contents of \mathcal{M}_1 (e.g. check message format, check sender ID, receiver ID, digital signature or MAC, if applicable). If any check violates, immediately report “Protocol Error” and abort the session.
5. Process the message content (e.g. print the result, stores the contents in local storage).
6. Prepare and send message \mathcal{M}_1' . If no message to send, abort the session silently.
7. Back to step 2.

However, three entities have different behaviour patterns, they will be specified here, respectively:

Alice \mathcal{A}

- An entity $a \in \mathcal{A}$ create the message to send, and prepare its meta data. Then it continuously listening to its network, waiting for incoming Couriers.
- Alice will submit its message to any Courier who request for it. When submitting, the message recipient must be explicitly specified.
- Alice is allowed to submit any message arbitrary times, to single or multiple Couriers. Alice itself should not care who carries the message, nor how many Couriers carry the message.
- During an uncompleted session, if no response from Courier for a long time, Alice should be able to detect the timeout, cancel the effects of previous actions in this session and abort the session voluntarily.
- If the message is not successfully sent, Alice should wait for next Couriers to send this message.

Bob \mathcal{B}

- An entity $b \in \mathcal{B}$ must be continuously listening to its network, waiting for incoming messages.
- Bob will download all messages from any Courier who transmits. If any message from \mathcal{A} is invalid, Bob simply discards it.
- Bob will discard all duplicated messages. Duplicated message are defined as messages whose Meta Signature are exactly the same. Because message Meta contains its creator ID and timestamp, same Meta reflects those messages are created by the same entity at the exact same time.
- During an uncompleted session, if no response from Courier for a long time, Bob should be able to detect the timeout, cancel the effects of previous actions in this session and abort the session voluntarily.

Courier \mathcal{C}

- As Courier's physical transportation is a very complicated task, it is out of the scope of this protocol. It is assumed that Courier is carried by an intelligent agent (such like human) who always knows where the Courier should transport to.
- Courier should transport to every $a \in \mathcal{A}$ one by one and get their messages if there are any. Courier should be capable of storing those data for a long time.
- After collect messages from all a in the list, Courier should transport to every b that is the recipient of the Courier and deliver all messages to them. If the receipt from b violates the messages Courier just sent, Courier should resend the messages by restart the session again.
- Once a full protocol run has been completed, all relative data stored in Courier is expired and should not affect future run of protocol.

3.2 Adversary Model

The adversary model in this protocol is mostly derived from Dolev-Yao model which implies "adversary carries the message" [17]. Moreover, the adversary can also do something special in this protocol system. Specifically, adversary \mathcal{Z} has following capabilities:

- It supervises the whole network system, which means it knows when, where and how any two entities are communicating, and it knows which entity possesses what information.
- It can access/rewrite any message passing through the network.
- It is a legitimate user of the network and it can be any of 3 entities in this protocol.
- It can access to all the Courier's data at any stage of the protocol.
- Any $a \in \mathcal{A}$ or $b \in \mathcal{B}$ will always have the opportunity to be connected by any $c \in \mathcal{C}$, which means an adversary will always have opportunity to be contacted by any honest entities.

It should be noted that this protocol, same as all other network protocols, is vulnerable to DoS attacks. \mathcal{Z} can always prevent a message from being sent, thus it will not be covered in the security analysis.

Chapter 4

Protocol Specification

A successful run of CDSProtocol consists of 3 stages - (1) message creators submit their messages to a single courier. (2) courier physically transports to the message receiver. (3) courier transmits the messages to the message receiver. As controlling couriers and planning couriers' routes are out of the scope of this protocol, it is assumed that in this protocol, couriers eventually are able to approach the target. Therefore stage (2) will not be discussed here.

The protocol specification will focus on the other two stages - (1) and (3), we call them Message Acquisition phase and Message Delivery phase. To ensure secure communication in both two phases, two sub-protocols - Submit Protocol and Transmit Protocol, are defined for those two phases respectively. The Submit Protocol runs between Alice and Courier, while the Transmit Protocol runs between Bob and Courier. As those two sub-protocols can be run independently, they will be explained separately.

4.1 Notations

The detail entity operations and message sequences of two sub-protocols will be displayed in message sequencing charts separately. Before showing those charts the notations are introduced first.

Encryption Functions \mathcal{E}_k : This notation denotes an abstraction of all encryption functions, including both symmetric and asymmetric encryptions. The subscript k denotes the key used for the encryption processes, and it is used for differentiating symmetric and asymmetric encryption, such like if the key is specified as an asymmetric key, it indicates the function is an asymmetric encryption, while symmetric key indicates symmetric encryption.

Decryption Functions \mathcal{D}_k : Similar to encryption functions, this notation denotes symmetric and asymmetric decryption functions, and subscript k denotes the key used for decryption.

Message Authentication Code Function \mathcal{MAC}_k : It denotes an abstract MAC function, and the subscript k indicates the key used for the function.

Digital Signature Function $SIGN_E$: It denotes an abstract digital signature function, and the subscript E indicates the entity who creates this signature using a secret key sk_A . And it can only be verified under the corresponding public key pk_B .

Concatenation $||$: It denotes the operation that concatenates two pieces of data together. For example “ $A||B$ ” simply means appending B to A .

Accumulation $^+$: It represents many pieces of data which have same format accumulated together. For example “ ID^+ ” equals a sequence of IDs concatenated together, which can be “ ID_0 ” or “ $ID_0||ID_1$ ” or “ $ID_0||ID_1||...$ ” where it contains at least one ID and the content of IDs can be different.

4.2 Submit Protocol

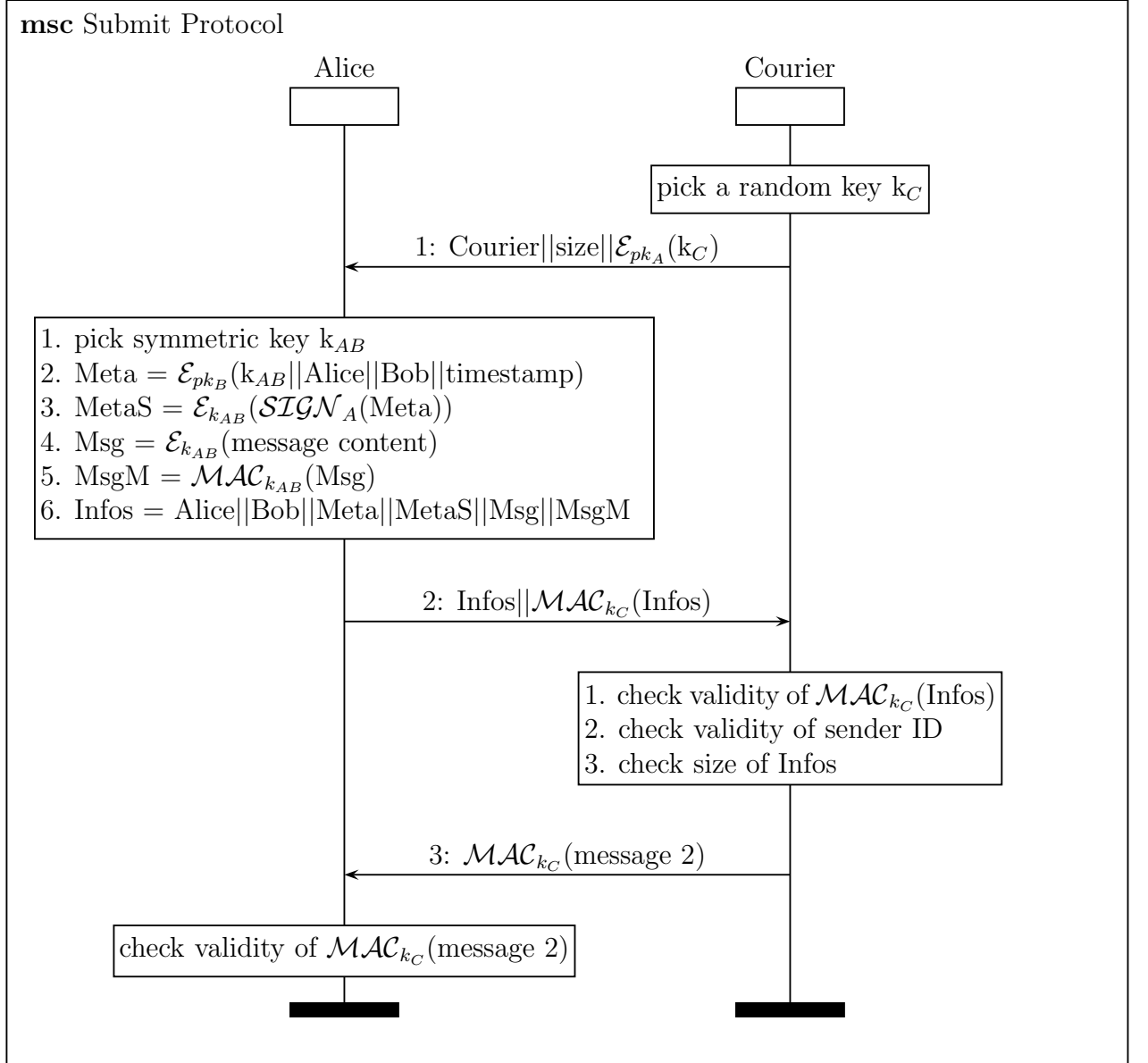
4.2.1 Preconditions

- Alice holds a unique pair of asymmetric key (pk_A, sk_A)
- Courier knows Alice’s public key pk_A

4.2.2 Postconditions

- Alice knows all the messages have been successfully sent to someone
- Alice doesn’t know the identity of the receiver
- Alice doesn’t know whether the message will be eventually deliver to Bob
- Courier knows the integrity of the message is preserved
- Courier knows the authenticity of origin of Alice’s messages

4.2.3 Message Sequencing Chart



4.2.4 Specification

Before the start of the protocol, Courier should randomly creates a symmetric key k_C which will be sent to Alice and used for deniable authentication later. Then Courier actively connects to Alice and sends first message to Alice. The first message should contain (1) The ID of Courier, (2) The maximum storage size of Courier, (3) Random key k_C encrypted by Alice's public key.

Once Alice receives the first message from Courier, it prepares for sending the

second message as reply. Firstly, Alice creates a random symmetric key k_{AB} which is supposed to be used as the session key with the message recipient Bob. Then Alice prepares the Meta block and its digital signature block MetaS. The Meta block contains meta information of the message content for Bob and it is encrypted under the public key of Bob. The plaintext of Meta block contains (1) Symmetric key k_{AB} , (2) The \mathcal{ID} of Alice, (3) The \mathcal{ID} of Bob, (4) A timestamp indicates the time Alice creates this message. To prove the authenticity of Meta block, Alice creates a digital signature of Meta block and encrypts the signature with k_{AB} which forms MetaS block.

After Alice has Meta and MetaS blocks, it creates a Msg block which is the encrypted message content for Bob under k_{AB} . Similarly, to ensure the integrity of the Msg block, a MAC of Msg block is created under k_{AB} , called MsgM block.

After Alice gets above 4 blocks, it creates a Infos block which is the whole information for Courier. Infos block is the concatenation of (1) The \mathcal{ID} of Alice, (2) The \mathcal{ID} of Bob, (3) Meta block, (4) MetaS block, (5) Msg block, (6) MsgM block. Then Alice appends a MAC of the Infos block to ensure its integrity. The MAC key is k_C , which can be revealed by decrypting $\mathcal{E}_{pk_A}(k_C)$ in the first received message. Finally Alice examines the size of Infos block. If the total size of Meta, MetaS, Msg, MsgM blocks exceeds the storage limitation of Courier, it either reduces the size of those blocks and prepares them again, or reports an error and aborts the protocol. If it doesn't, Alice sends the whole message 2 which contains Infos block and its MAC, to Courier.

Once Courier receives the message 2 from Alice, it should check the validity of message 2. It first verifies $\mathcal{MAC}_{k_C}(\text{Infos})$, if true then checks the \mathcal{ID} of Alice to see if it is indeed the entity it is going to connect. After that, it checks the total size of the Meta, MetaS, Msg and MsgM blocks, making sure it does not exceeds the storage limitation. If any of above checks violate, Courier should report an error and abort the protocol. If all checks success, Courier uses k_C to create a MAC of the whole message 2 received from Alice and sends it to Alice as message 3.

At the end of the protocol, Alice checks the validation of message 3. It verifies the received MAC using k_C . If it verifies true, it means the protocol success and all postconditions are held. Otherwise, the protocol fails. However, the verification result does not prove the fact whether Courier has received the correct message 2 or not. Alice only knows its message may or may not be successfully sent. So further actions can be taken by Alice such as waiting for next Courier to send the same message or report an error.

4.3 Transmit Protocol

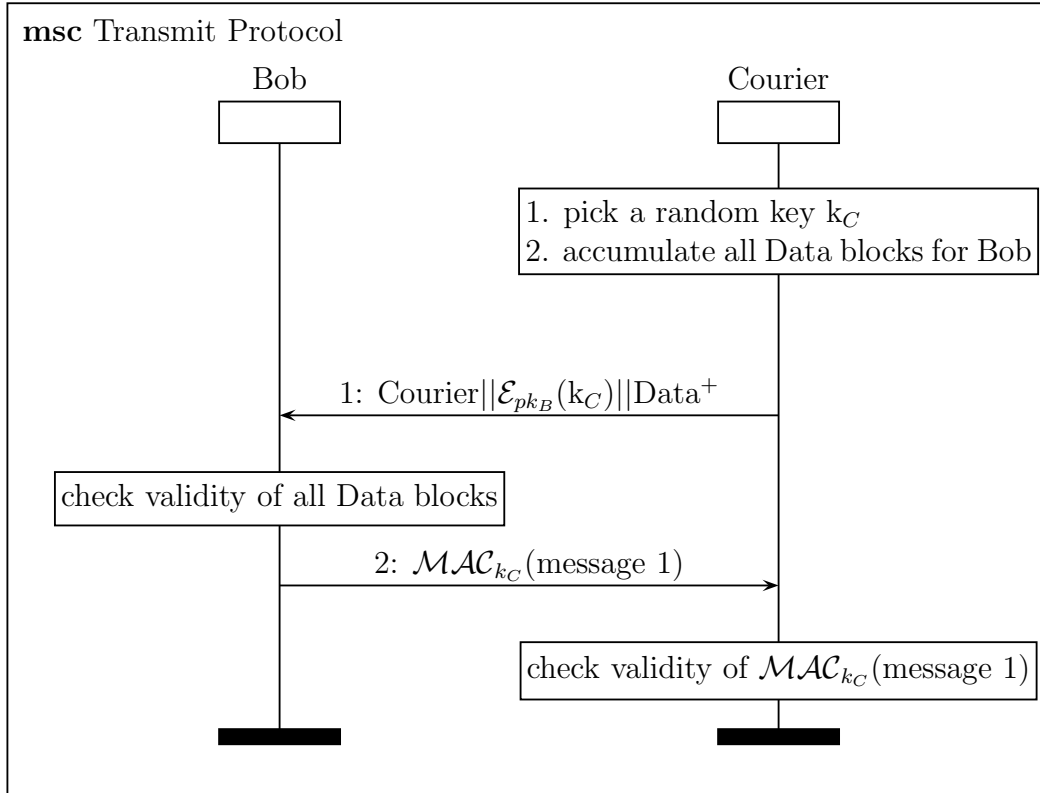
4.3.1 Preconditions

- Bob holds a unique pair of asymmetric key (pk_B, sk_B)
- Courier knows Bob's public key pk_B

4.3.2 Postconditions

- Bob accepts the message, knowing it is created by Alice
- Bob doesn't know the identity of the message sender
- Courier knows Bob has successfully received and accepted the message

4.3.3 Message Sequencing Chart



where:

$\text{Data} = \text{Meta} || \text{MetaS} || \text{Msg} || \text{MsgM}$

$\text{Meta} = \mathcal{E}_{pk_B}(k_{AB} || \text{Alice} || \text{Bob} || \text{timestamp})$

$\text{MetaS} = \mathcal{E}_{k_{AB}}(\text{SIGN}_A(\text{Meta}))$

$$\text{Msg} = \mathcal{E}_{k_{AB}}(\text{message content})$$

$$\text{MsgM} = \mathcal{MAC}_{k_{AB}}(\text{Msg})$$

4.3.4 Specification

Before start of the protocol, Courier first creates a random symmetric key k_C which will be sent to Bob and used for deniable authentication. Once Courier has k_C , it could prepare the data for Bob. By the time Courier starts the Transmit Protocol with Bob, it should have carried certain pieces of data for Bob which are collected earlier. As introduced in Submit Protocol, each piece of data from Alice contains (1) Meta block, (2) MetaS block, (3) Msg block, (4) MsgM block. We call such a piece of data a Data block. Courier now fetches all those data, and accumulate them together, forming a Data^+ block. Then Courier actively connects to Bob and sends the first message to Bob. The message 1 contains (1) The \mathcal{ID} of Courier, (2) Random key k_C encrypted by Bob's public key, (3) The accumulation of all Data blocks for Bob.

Upon receipt of the first message, Bob checks the validity of all Data blocks received. Basically, Bob will check each Data block one by one, and it will accept all Data blocks that pass all verifications while discard all Data blocks that fail one of the verifications. The detailed algorithm used for checking all Data blocks is displayed below in the algorithm "check all Data blocks".

Algorithm 1: Check all Data blocks

Data: The concatenation of all Data blocks

Result: Bob accepts valid Data blocks and discards invalid ones
initialization;

while *there is still Data block unchecked* **do**

 currentDataBlock = nextDataBlock;

 use sk_A to decrypt Meta block in currentDataBlock, reveal $\{k_{AB}, \text{senderID}, \text{receiverID}, \text{timestamp}\}$;

if *receiverID* \neq "Bob" **then**

 | discard currentDataBlock

else

if *timestamp expires* **then**

 | discard currentDataBlock

else

 use k_{AB} to decrypt MetaS in currentDataBlock, reveal $SIGN_A(\text{Meta})$;

 verify $SIGN_A(\text{Meta})$ using pk_A ;

if *verifies false* **then**

 | discard currentDataBlock

else

 verify MsgM in currentDataBlock using pk_A ;

if *verifies false* **then**

 | discard currentDataBlock

else

 | use pk_A to decrypt Msg and reveal the message content

After all Data blocks have been received and checked, Bob send back a MAC of whole message 1 received from Courier. The key used for creating the MAC is k_C which is revealed by decrypting $\mathcal{E}_{pk_B}(k_C)$ in message 1.

Finally Courier checks the validation of message 2. It verifies the received MAC using k_C . Same to the Submit Protocol, if the last MAC is verified true, it means the protocol success and all postconditions are held. Otherwise, the protocol fails. However, the verification result does not prove the fact whether Bob has received the correct message 2 or not. Courier only knows its message may or may not be successfully sent. So further actions can be taken by Courier such as restart the protocol again or report an error.

Chapter 5

Security Analysis

5.1 Protocol Properties

Following are the properties that claimed to be held by CDSP, they are briefly introduced in this section and then will be proved afterwards.

5.1.1 Submit protocol

- **\mathcal{A} is authenticated to \mathcal{CR}**

As it has been assumed that physical transportation for a Courier is extremely costly, Courier should not carry messages for any arbitrary entity. It is expected that Courier only carries messages for those entities that have been authorized, so \mathcal{A} should prove its identity when submitting the message.

- **The Integrity of message from \mathcal{A} to \mathcal{CR} is preserved**

It is expected that the message is not modified or forged after being submitted to Courier.

- **\mathcal{CR} is not able to access the message content**

As the message content from \mathcal{A} to \mathcal{B} is highly confidential, it should not be disclosed to any other entity other than \mathcal{B} . Thus making message content inaccessible to \mathcal{CR} prevents the content from being released when adversary examining the Courier's data.

- **\mathcal{A} is able to deny sending any message to \mathcal{CR}**

In certain scenarios which require high privacy protection - like exchanging military intelligence or voting, the action of sending out a message itself might become a sensitive information. It is expected that this protocol protects the

privacy of entity \mathcal{A} in such a way that it can deny ever sending message to Courier at any time.

5.1.2 Transmit protocol

- **\mathcal{B} is authenticated to \mathcal{CS}**

As physical transportation is costly for Courier, Courier should not transmit its data to some arbitrary entity. It is reasonable that Courier ensures the message is transmitted to the target entity \mathcal{B} . Thus \mathcal{B} should prove its identity when receiving data from Courier.

- **The Integrity of message from \mathcal{CS} to \mathcal{B} is preserved**

It is expected that the message is not modified or forged after being transmitted to \mathcal{B} .

- **\mathcal{B} is able to deny receiving any message from \mathcal{CS}**

Similar to the Submit Protocol, in some privacy-sensitive scenarios, it is plausible for \mathcal{B} to deny the fact of receiving data from Courier.

5.1.3 On the whole

- **\mathcal{A} is authenticated to \mathcal{B}**

As this CDSP is for sending message from \mathcal{A} to \mathcal{B} , it is required that the message sender proves its identity to the recipient. Otherwise malicious Courier would be able to forge arbitrary messages to \mathcal{B} that never exists.

- **Confidentiality of message from \mathcal{A} to \mathcal{B} is preserved**

As the message content from \mathcal{A} to \mathcal{B} is highly confidential, it should not be disclosed to any other entity other than \mathcal{B} .

- **Integrity of message from \mathcal{A} to \mathcal{B} is preserved**

It is expected that the message is not modified or forged during the transportation.

- **\mathcal{A} can not deny sending a message to \mathcal{B}**

This property is not deliberately designed in the protocol. As CDSP is an one-way protocol, the authentication of \mathcal{A} must be non-interactive. Thus a compromise has to be made between computational complexity and the full deniability of a non-interactive deniable authentication. Finally we choose to put the efficiency as first priority. So in the protocol, \mathcal{A} is not able to fully

deny sending a message to \mathcal{B} . However, as the next property states, \mathcal{A} is able to deny the message content which almost achieves the same goal. Besides, the property of non-repudiation could still be useful for \mathcal{B} as it proves the number of messages sent from \mathcal{A} .

- **\mathcal{A} is able to deny the message content for \mathcal{B}**

In some privacy-sensitive scenarios, it is reasonable for \mathcal{A} to be able to deny the message content it has sent. According to above property, \mathcal{A} is not allowed to fully deny sending a message to \mathcal{B} , however, capability of denying the message content has approximately the same effect, because \mathcal{A} can always argue that the message content is empty.

5.2 Property Proof

5.2.1 Primitives

As CDSP is built based on many primary cryptographic operations, its security also relies on them. Thus before proving the security properties of CDSP, some essential assumptions on the security of the primary cryptographic operations have to be made as the security primitives. Then it can be proved that all the properties listed above can be deduced to those primitives.

- Message Authentication Code \mathcal{MAC}

The MAC [23] function used in this protocol is assumed to be secure in the sense that it has all the properties that a cryptographic hash function possesses and additionally, it resists to existential forgery under chosen-plaintext attack. That is, even if attacker is able to access an oracle which possesses the secret key and generates MACs according to the attacker's input, it is computational infeasible for attacker to guess MACs of other messages (not used to query the oracle).

- Signature Function \mathcal{SIGN}

The signature function used in this protocol is assumed to be secure in the sense that it resists existential forgery under an adaptive chosen message attack. [19][24]

- Encryption Function \mathcal{E} and Decryption Function \mathcal{D}

To assure the security primitives of this protocol, all encryption/decryption schemes used are required to be secure in the sense that without the decryption

key, it is computational infeasible for attackers to reveal the plaintext of a cipher with non-negligible probability.

- Symmetric Key Generator \mathcal{G}

The symmetric key generator used in this protocol is assumed to be no less secure than a Cryptographically Secure Pseudorandom Number Generator. That is:

(a) It should satisfy the next-bit test. That is, given the first k bits of a random sequence, there is no polynomial-time algorithm that can predict the $(k+1)$ th bit with probability of success better than 50%. [27]

(b) It should withstand "state compromise extensions". In the event that part or all of its state has been revealed (or guessed correctly), it should be impossible to reconstruct the stream of random numbers prior to the revelation. Additionally, if there is an entropy input while running, it should be infeasible to use knowledge of the input's state to predict future conditions of the CSPRNG state.

Furthermore, the keys generated are required to fit the Symmetric Encryption Scheme described above.

5.2.2 Submit protocol

THEOREM 1: \mathcal{A} is authenticated to \mathcal{CR}

Proof :

Assuming \mathcal{G} is secure, then \mathcal{CR} is the only entity who knows the randomly generated key k_C . Providing \mathcal{E}_{pk_A} scheme is secure, because k_C is encrypted by pk_A before sent out \mathcal{A} will be the only entity who can reveal the encrypted k_C . Similarly, providing \mathcal{MAC} scheme is secure, \mathcal{A} is also the only one who can create MESSAGE 2 and its \mathcal{MAC}_{k_C} . Thus when (MESSAGE 2, \mathcal{MAC}_{k_C}) pair is received by \mathcal{CR} and is verified true, it can be sure this message is created by \mathcal{A} . So \mathcal{A} is authenticated to \mathcal{CR} .

THEOREM 2: The Integrity of message from \mathcal{A} to \mathcal{CR} is preserved

Proof :

Assuming \mathcal{MAC} scheme is secure, any modification of MESSAGE 2 will lead to unpredictable changes in the \mathcal{MAC}_{k_C} (MESSAGE 2) and cause its verification to be false. And according to THEOREM 1, no one else but \mathcal{A} can create the message and its valid MAC, message forgery is prevented. As consequence, the message integrity

is preserved.

LEMMA 1 The message content cannot be revealed by any entity but \mathcal{B}

Proof :

Assuming \mathcal{G} is secure, the randomly generated key k_{AB} is only held by \mathcal{A} . k_{AB} is encrypted by \mathcal{E}_{pk_B} and sent to \mathcal{CR} , so assuming the \mathcal{E} scheme is secure, only \mathcal{B} can decrypt the cipher and reveal k_{AB} . As message content is encrypted by $\mathcal{E}_{k_{AB}}$, the only entity can decrypt it is \mathcal{B} because only it knows k_{AB} (other than the message creator). Consequently, only \mathcal{B} can reveal the message content created by \mathcal{A} .

THEOREM 3: \mathcal{CR} is not able to access the message content

Proof :

According to LEMMA 1, only \mathcal{B} can reveal the message content, we can easily deduce that \mathcal{CR} is not able to reveal the message content.

THEOREM 4: \mathcal{A} is able to deny sending any message to \mathcal{CR}

Proof :

The whole message sent from \mathcal{A} to \mathcal{CR} contains three parts - (1) entity IDs, (2) encrypted message from \mathcal{A} and (3) \mathcal{MAC}_{k_C} , if \mathcal{CR} wants to prove the authenticity of the origin of the whole message, it must show that at least one of these three parts can be created only by \mathcal{A} . However, all these three parts are forgeable by \mathcal{CR} itself:

1. entity IDs are plaintexts, thus can be created by \mathcal{CR} .
2. as has been proven in LEMMA 1, no entity but \mathcal{B} can reveal the securely generated key k_{AB} , so \mathcal{CR} is not able to reveal $\mathcal{SIGN}_{\mathcal{A}}$ or message content, thus the encrypted message is just a block of random data for \mathcal{CR} , thus can be created by \mathcal{CR} .
3. \mathcal{MAC}_{k_C} can also be created by \mathcal{CR} because \mathcal{CR} has k_C and is able to forge the whole former message.

To sum up, \mathcal{CR} is able to create the whole message itself, it can not convince others the authenticity of the origin of the the message, so \mathcal{A} is able to deny sending the message to \mathcal{CR} .

5.2.3 Transmit Protocol

THEOREM 5: \mathcal{B} is authenticated to \mathcal{CS}

Proof :

Similar to proof of THEOREM 1, providing \mathcal{G} and \mathcal{E} scheme are secure, only \mathcal{B} knows the symmetric key generated by \mathcal{CS} . So, assuming \mathcal{MAC} scheme is secure, if $\mathcal{MAC}_{k_C}(\text{MESSAGE 1})$ can be successfully verified by \mathcal{CS} , it must be \mathcal{B} who create the MAC. Thus, \mathcal{B} is authenticated to \mathcal{CS} .

THEOREM 6: The Integrity of message from \mathcal{CS} to \mathcal{B} is preserved

Proof :

Similar to proof of THEOREM 2, any modification on MESSAGE 1 will leads to unpredictable changes in its MAC, and no one else but \mathcal{B} can create the verifiable MAC of arbitrary message. So if $\mathcal{MAC}_{k_C}(\text{MESSAGE 1})$ is verified true by \mathcal{CS} , it means it is originally created by \mathcal{B} and has not been modified. Thus the message integrity is preserved.

THEOREM 7: \mathcal{B} is able to deny receiving any message from \mathcal{CS}

Proof :

Similar to the proof of THEOREM 4, the message from \mathcal{B} is totally forgeable by \mathcal{CS} because it creates the whole MESSAGE 1 and holds k_C , it can create $\mathcal{MAC}_{k_C}(\text{MESSAGE 1})$ by it own. Therefore \mathcal{CS} cannot prove to others that the MAC is sent from \mathcal{B} , \mathcal{B} can deny receiving any message from \mathcal{CS} .

5.2.4 On The Whole

THEOREM 8: \mathcal{A} is authenticated to \mathcal{B}

Proof :

As \mathcal{B} receives two pieces of data - Meta and Msg, the authentication will be done for both of them separately.

Meta: Because \mathcal{B} can decrypt the encrypted Meta from \mathcal{A} , it can reveal the sender \mathcal{ID} and k_{AB} , then it can further decrypt $\mathcal{E}_{k_{AB}}(\mathcal{SIGN}_A(\text{Meta}))$ to get $\mathcal{SIGN}_A(\text{Meta})$. Assuming \mathcal{SIGN} scheme is secure, if \mathcal{B} verifies the signature true under \mathcal{A} 's public key, \mathcal{B} knows Meta can only be created by \mathcal{A} .

Msg: Assuming \mathcal{G} and \mathcal{E} scheme is secure, only \mathcal{B} can decrypt $\mathcal{E}_{k_{AB}}(k_{AB})$ and reveal k_{AB} in Meta. Thus only \mathcal{B} (other than \mathcal{A}) can create $\mathcal{MAC}_{k_{AB}}(\text{Msg})$. So, if $(\text{Msg}, \mathcal{MAC}_{k_{AB}})$ pair is verified true by \mathcal{B} , \mathcal{B} knows Msg is created by \mathcal{A} .

To sum up, \mathcal{A} is authenticated to \mathcal{B} as all message sent by \mathcal{A} is authenticated.

THEOREM 9: Confidentiality of the message from \mathcal{A} to \mathcal{B} is preserved

Proof :

As proven in THEOREM 3, the message content from \mathcal{A} to \mathcal{B} can not be revealed by any other entities but \mathcal{B} , even the Courier itself. So after physical transportation and transmitting message to \mathcal{B} , this property still hold. This means only the creator and recipient of the message can reveal its content, so the confidentiality is preserved.

THEOREM 10: Integrity of the message from \mathcal{A} to \mathcal{B} is preserved

Proof :

Similar to the proof of THEOREM 6, assuming \mathcal{G} and \mathcal{E} scheme is secure, \mathcal{B} is the only entity who can decrypt $\mathcal{E}_{k_{AB}}(k_{AB})$ and reveal k_{AB} . So when message is received by \mathcal{B} , only \mathcal{A} and \mathcal{B} hold k_{AB} and are able to create $\mathcal{MAC}_{k_{AB}}(\text{Msg})$. So any forgery will lead to MAC verification fail. Further more, if the Msg is modified, it will lead to unpredictable changes in $\mathcal{MAC}_{k_{AB}}(\text{Msg})$ and fail the MAC verification as well. So if $(\text{Msg}, \mathcal{MAC}_{k_{AB}}(\text{Msg}))$ pair is verified true, the Msg must be created by \mathcal{A} , and remain unchanged. Thus the integrity of message from \mathcal{A} to \mathcal{B} is preserved.

THEOREM 11: \mathcal{A} can not deny sending the message to \mathcal{B}

Proof :

Similar to the proof of THEOREM 8, if $\text{SIGN}_{\mathcal{A}}$ is verified true, it proves the authenticity of the origin of the Meta, so \mathcal{A} can not deny sending the message.

THEOREM 12: \mathcal{A} is able to deny the message content sent to \mathcal{B}

Proof :

The message content contains two parts - (1) encrypted message $\text{Msg} = \mathcal{E}_{k_{AB}}(\text{message})$ and (2) its MAC $\mathcal{MAC}_{k_{AB}}(\text{Msg})$. Because k_{AB} is contained in the Meta and Meta is encrypted under \mathcal{B} 's public key, \mathcal{B} is able to reveal k_{AB} . Then the whole message content part sent from \mathcal{A} is forgeable by \mathcal{B} :

1. Msg is encrypted under k_{AB} , \mathcal{B} can create any Msg it wants.
2. $\mathcal{MAC}_{k_{AB}}(\text{Msg})$ also can be created by \mathcal{B} because \mathcal{B} can create Msg and hold k_{AB} .

As \mathcal{B} can create the whole content part by its own, \mathcal{B} can not prove to others that the message is from \mathcal{A} . Thus \mathcal{A} can deny the message content for \mathcal{B} .

Chapter 6

Protocol Implementation

6.1 Application Overview

To run this protocol in a practical scenario, at least 3 different devices are required - one be Alice, one be Bob and one be Courier. This application implements all 3 entities within it, so once the device has this application, it can be any entity in the protocol. To achieve this, at the start of the application, user is asked to choose a particular role for the device to be. There are 4 options:

1. **DataCreator**, who creates message and wants to send it out.
2. **CourierReceiver**, who connects to DataCreator and receives the message from it.
3. **DataReceiver**, who is the recipient of DataCreator.
4. **CourierSender**, who possesses the message from DataCreator and should transmit it to DataReceiver.

Once the choice has been made, the graphical user interface will adjust to the selected role, and user will be requested to input some information related to the selected role.

Below is a snapshot of the GUI of the application, it shows all the selections and textfields that will be used when the application interacts with user.

To run Submit Protocol, Alice should choose to run as DataCreator and Courier should choose to run as CourierReceiver. Then Courier transports to Bob and chooses to run as CourierSender, meanwhile Bob should choose to run as DataReceiver to run Transmit Protocol with Courier. The detailed instruction of how to run those two sub-protocols is given in the appendix.

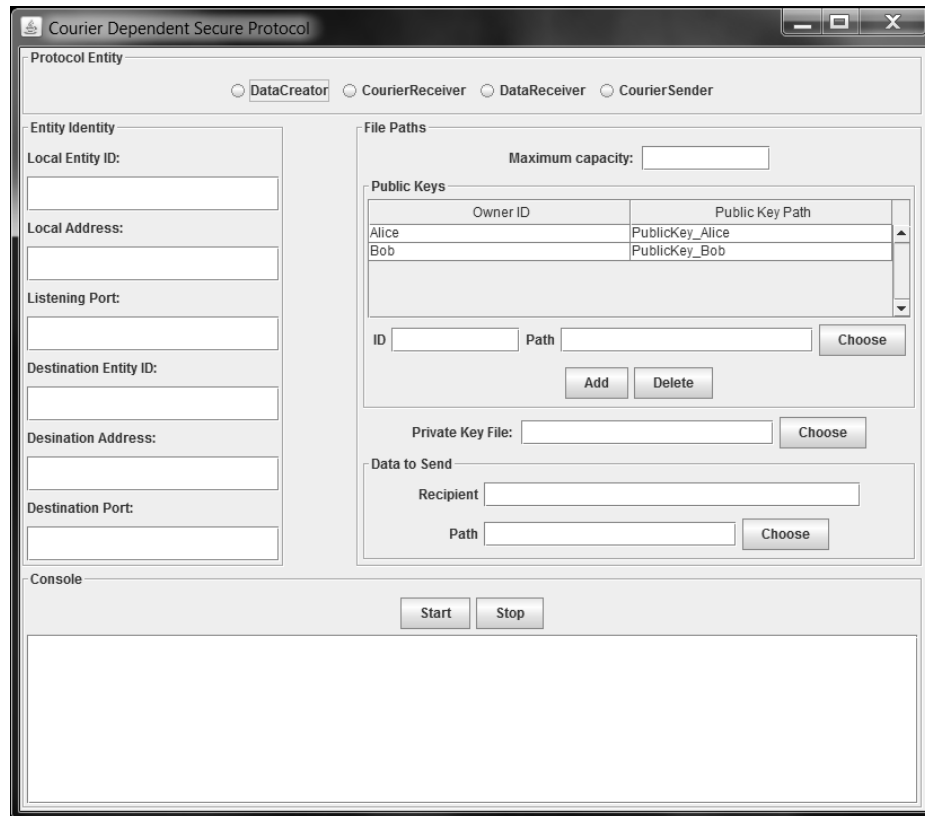


Figure 6.1: GUI Overview

To further illustrate how user can interact with the system, a use case diagram is given and every use cases appear in the diagram will be explained. For disambiguating, the “Alice”, “Bob” and “Courier” appeared in the use case diagram do not mean the protocol entity as introduced in the system model, but are the names of the actors who control the device. The name of the actor simply reflects which entity the actor wants to be in the protocol.

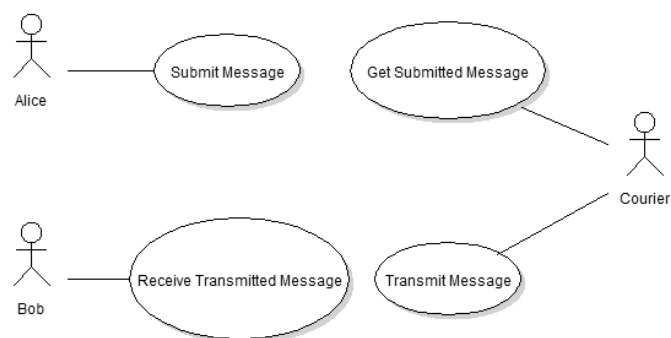


Figure 6.2: Use Case Diagram

Use Case 1: Submit Message

Title	Submit Message
Primary Actor	Alice
Goal in Context	Alice tries to submit the message to a Courier
Scope	System (Black Box)
Level	
Stakeholders	Alice and Courier
Success Guarantees	Message is received by Courier and Alice confirms the fact
Trigger	Alice starts the Submit Protocol
Main Success Scenario	<ol style="list-style-type: none">1. Alice starts the application.2. Alice selects “DataCreator” as the protocol entity in the application.3. Alice specifies all related information as instructed in the “Run Submit Protocol” section.4. Alice starts running the protocol.

Use Case 2: Get Submitted Message

Title	Get Submitted Message
Primary Actor	Courier
Goal in Context	Courier tries to obtain the message of Alice
Scope	System (Black Box)
Level	
Stakeholders	Courier and Alice
Success Guarantees	Courier gets a message from Alice
Trigger	Courier starts the Submit Protocol
Main Success Scenario	<ol style="list-style-type: none">1. Courier starts the application.2. Courier selects “CourierReceiver” as the protocol entity in the application.3. Courier specifies all related information as instructed in the “Run Submit Protocol” section.4. Courier starts running the protocol.

Use Case 3: Receive Transmitted Message

Title	Receive Transmitted Message
Primary Actor	Bob
Goal in Context	Bob tries to receive message of Alice carried by Courier
Scope	System (Black Box)
Level	
Stakeholders	Bob and Courier
Success Guarantees	Bob receives the message
Trigger	Bob starts the Transmit Protocol
Main Success Scenario	<ol style="list-style-type: none">1. Bob starts the application.2. Bob selects "DataReceiver" as the protocol entity in the application.3. Bob specifies all related information as instructed in the "Run Transmit Protocol" section.4. Bob starts running the protocol.

Use Case 4: Transmit Message

Title	Transmit Message
Primary Actor	Courier
Goal in Context	Courier tries to transmit Alice's message to Bob
Scope	System (Black Box)
Level	
Stakeholders	Courier and Bob
Success Guarantees	Message is received by Bob and Courier confirms the fact
Trigger	Courier starts the Transmit Protocol
Main Success Scenario	<ol style="list-style-type: none">1. Courier starts the application.2. Courier selects "CourierSender" as the protocol entity in the application.3. Courier specifies all related information as instructed in the "Run Transmit Protocol" section.4. Courier starts running the protocol.

6.2 Software Design

The implementation of this protocol mainly consists of two parts - a core library and an user interface. The core library defines the framework of the program and provides

all essential components for running the protocol. While the user interface takes user's input, configures components provided by the core library, runs the protocol and outputs running results if necessary. This design separates the implementation of user interfaces from the actual running of the protocol, the major benefit is that when this protocol is running on different devices or platforms, its user interfaces can be customized and easily plugged to the core library without modifying it.

6.2.1 Program Architecture

As introduced above, the whole program contains 4 roles - DataCreator, DataReceiver, CourierSender and CourierReceiver. Classified by their main structure, those 4 different roles fall into 2 categories - (1) Acceptor, who continuously listens to the port and waits for incoming connection. (2) Initiator, who actively connects to a Acceptor. Although these two categories sounds similar to Client-Server pattern, it is not quite the case because Acceptor does not actually provide any service. Based on their behaviour definition in protocol specification, DataCreator and DataReceiver belong to Acceptor, while CourierSender and CourierReceiver belong to Initiator. Following paragraphs will introduce the internal architecture for Acceptor and Initiator respectively.

Acceptor Acceptor contains 5 main parts:

- Dispatch Thread

The Dispatch Thread listens to the program port, waits for incoming connections. Once it receives a connection, it will create a new session, then handover the connection to the newly created session. After that, it will continue listening to the port, wait for next incoming connection.

- Sessions

Sessions are created by the Dispatch Thread, each session corresponds to a single connection and sessions are independent between each other. Once a session takeovers a connection, it is fully responsible for it. Inside a session there is a Delegate and a Sub-thread.

The **Delegate** defines all communication content (e.g. a Delegate of Alice defines the content of messages to send out, and it also defines what to do when receives a certain message), it behaves like a state machine which takes message as input, processes the message and output a new message for response. During

the processing of input messages, it may interact with the 3 global objects - User Interface, Cryptographic Kit and Data Manager. The detail of Delegate will be explained in Delegate Model.

The **Sub-thread** listens to the connection which is handover by Dispatch Thread, capture any message from the connection. It delivers the captured message to Delegate, and receives a new message from Delegate. If the output message from Delegate does not indicate an exception or termination, the Sub-thread sends the message through the connection, otherwise it reports an error or terminates.

- User Interface

The User Interface is shared between all sessions in Acceptor. It displays all relative information to show the progress of the protocol running and reports errors to the user.

- Cryptographic Kit

The Cryptographic Kit is shared between all sessions in Acceptor. It provides functionalities of all cryptographic operations that is needed in the protocol, such as encryption, decryption, key generation, MAC generation and verification, and digital signature generation and verification. When any session need to do cryptographic operations, it just call from Cryptographic Kit and doesn't care the internal implementation. For the consistency concern, it is required that any two communicating device should use same Cryptographic Kit.

- Data Manager

The Data Manager is shared between all sessions in Acceptor. It keeps record of all data files in the device's disk which is related to the protocol, such as files of public keys, secret keys, and the message Courier carries. Data Manager is configured at the start of the application, when any session need data in disk, it just request from Data Manager.

The relation between those 5 components is illustrated in the figure "Acceptor Architecture".

Initiator Similar to Acceptor, Initiator consists of 4 main parts: (1) Session, (2) User Interface, (3) Cryptographic Kit and (4) Data Manager. The major difference between Initiator and Acceptor is that Initiator does not have a Dispatch Thread, every Initiator only has one Session. The Session is created once the Initiator is

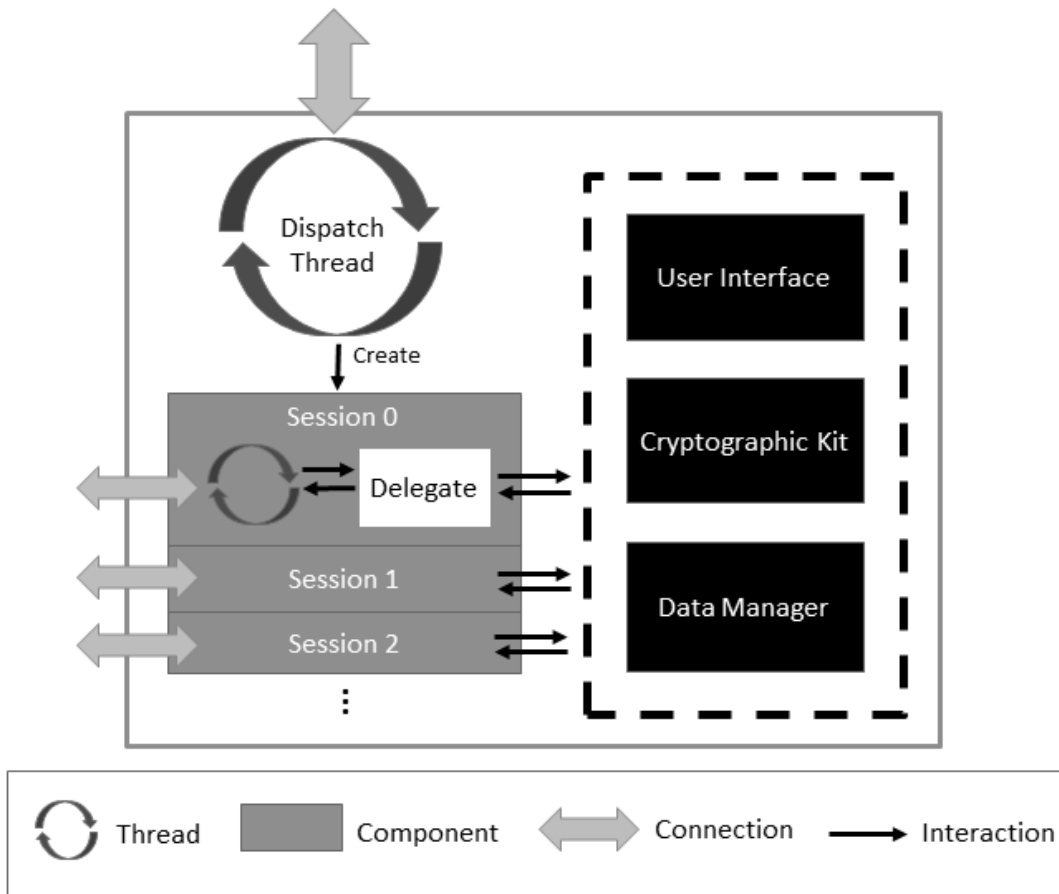


Figure 6.3: Acceptor Architecture

started, and it will ask its Delegate for a “initial message” - which will be the first message of the protocol, and will be sent to the specified Acceptor. The architecture design of Initiator is a simpler version of Acceptor’s, and it is illustrated in the figure “Initiator Architecture”.

6.2.2 Delegate Model

The object Delegate is the essence of the application, it directly refers to the specification of the protocol, defines how an entity processes a message - here “process a message” may involve doing cryptographic operations, checking message content validity and giving a response message.

Basically, every Delegate acts like a finite-state machine, who possesses an internal state, takes messages as inputs and outputs new messages as response. The internal state controls the behaviour of the Delegate - Delegates in different states will respond

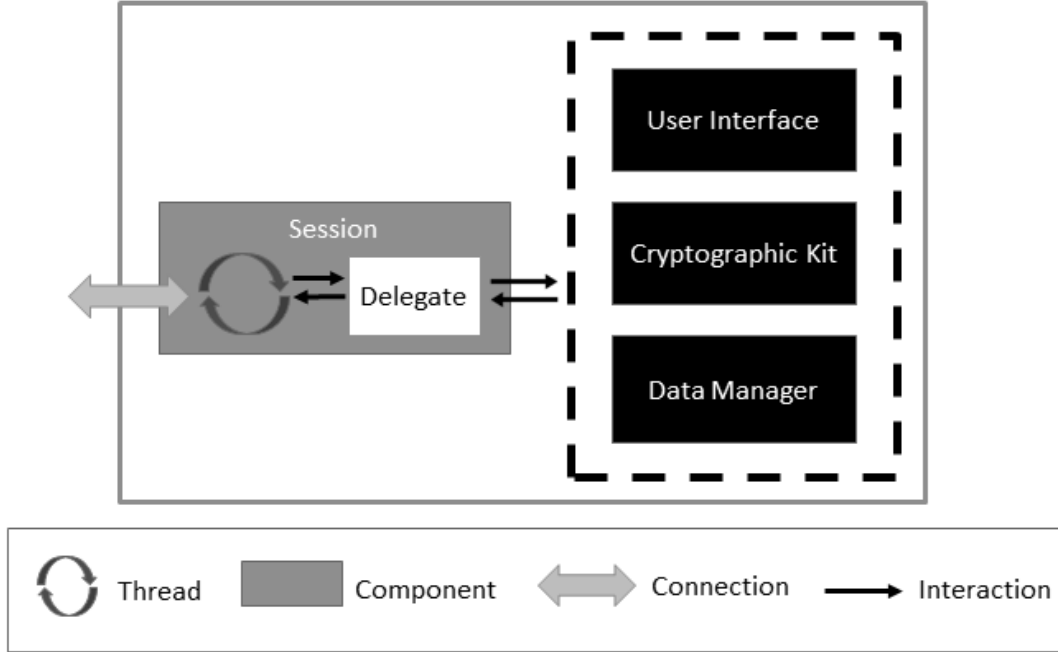


Figure 6.4: Initiator Architecture

differently to a same input. So when an input message comes, Delegate will process the message according to its current state, if the message is successfully processed, it will change its current state and wait for next message, until it reaches the final state.

Taking entity \mathcal{A} as example, the figure below shows a 4-internal-state machine which abstracts the behaviour of \mathcal{A} : initially \mathcal{A} is in Wait state, waiting for the first message. Once it receives the first message, it will process the message (according to the Submit Protocol, it will check the first message and submit its data to Courier). If \mathcal{A} successfully processed M_0 , it enters Submitted state, waiting for the second message. Then it will receive and process M_1 (according to the Submit Protocol, it will check the validity of the MAC). If it is successful again, it enters the final state Checked and stops. If any error occurs in processing the input messages in early states, \mathcal{A} directly enters final state Error and stops.

6.2.3 Message Structure

Based on the protocol specification, there are totally 5 different kinds of messages to exchange in a single success protocol run - 3 messages are needed to achieve Submit

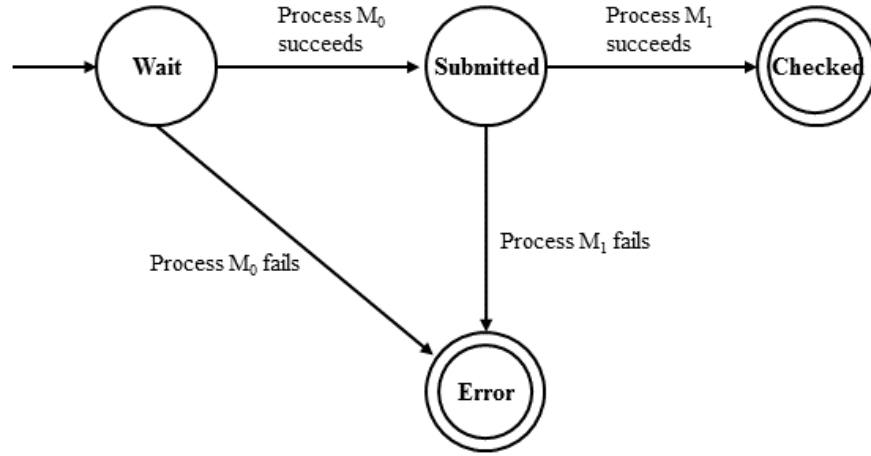


Figure 6.5: State Machine of \mathcal{A}

Protocol and 2 messages are needed to achieve Transmit Protocol. The design of message structure attempts to maximize the time efficiency of the protocol, thus no extra message will be exchanged and the lengths of messages will be minimized. All messages exchanged in this protocol are concatenations of 6 different kinds of primary blocks:

- **Integer Block**

Integer Block simply contains a single Integer, it occupies 4 bytes (based on the current JAVA's Implementation).

- **MAC Block**

MAC Block contains a cryptographic MAC, whose size is defined by the Cryptographic Kit the application uses.

- **Asymmetric Cipher Block**

Asymmetric Block contains an asymmetric cipher. Because encrypting a long plaintext using asymmetric encryption takes long time, to reduce the computational consumption, the length of plaintext for asymmetric encryption is restricted in this protocol, so that only one cipher block will be produced after the encryption. As the consequence, the size of Asymmetric Cipher Block is fixed, which is 128 bytes (based on a key of size 128 bits).

- **ID Block**

ID Block contains two parts. The front part is a single byte which indicates

the total length of the ID string and the following part is a string of characters representing the ID. The size of ID Block is variable, which is the length of ID string plus 1 bytes.

- **Meta Block**

The Meta Block in application design is different with “Meta block” in protocol specification. It contains a concatenation of 3 parts - (1) an integer indicates the size of the rest of this Meta Block, (2) an asymmetric cipher of the meta of the message, (3) a symmetric cipher of the signature of the meta. As explained in Asymmetric Cipher Block, the size of all asymmetric ciphers is fixed, so the size of the asymmetric cipher is (size of Meta Block - size asymmetric cipher).

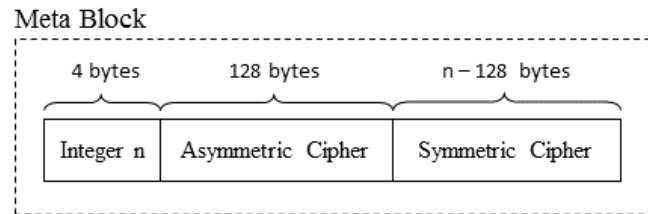


Figure 6.6: Structure of Meta Block

- **Message Block**

The Message Block is also different with “Msg block” in protocol specification. Similar to Meta Block, it is the concatenation of 3 parts - (1) an integer indicates the size of the rest of this Message Block, (2) a symmetric cipher of the actual message content, (3) a MAC of the part (2). The size of symmetric cipher can also be deduced as the size of MAC is fixed.

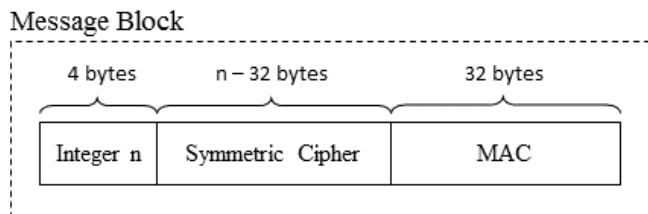


Figure 6.7: Structure of Message Block

To summarize, 3 of these 6 primary blocks are fixed-size blocks (Integer Block, MAC Block and Asymmetric Cipher Block), while other 3 of them are variable-size blocks (ID Block, Meta Block and Message Block). All the messages exchanged in the

protocol are constructed by these primary blocks through concatenation, and they will be illustrated below.

Submit Protocol

Message 1, from Courier to Alice:

ID Block	Integer Block	AC Block
----------	---------------	----------

Note that “AC Block” denotes Asymmetric Cipher Block.

Message 2, from Alice to Courier:

ID Block	ID Block	Meta Block	Message Block	MAC Block
----------	----------	------------	---------------	-----------

Message 3, from Courier to Alice:

MAC Block

Transmit Protocol

Message 1, from Courier to Bob:

ID Block	AC Block	Meta ₀	Message ₀	Meta ₁	Message ₁
----------	----------	-------------------	----------------------	-------------------	----------------------	-------

Multiple Meta Blocks and Message Blocks can be appended here.

Message 2, from Bob to Courier:

MAC Block

Error Message Flag

There are two kinds of messages exchanged in the protocol, all above messages are one of them, called Normal Message, the others called Error Message. Those two kinds of messages are distinguished by an Error Message Flag, which is the first byte of the message. Thus there is an extra step before above Normal Messages to be sent - they should be wrapped by an extra byte 0 at the front of them. While an Error Message contains two parts - (1) a byte indicates the size of the rest of the message, (2) a string of error information. As consequence, all messages start with a byte 0 will be treated as Error Message. The figure below further illustrates the difference between Normal Message and Error Message.

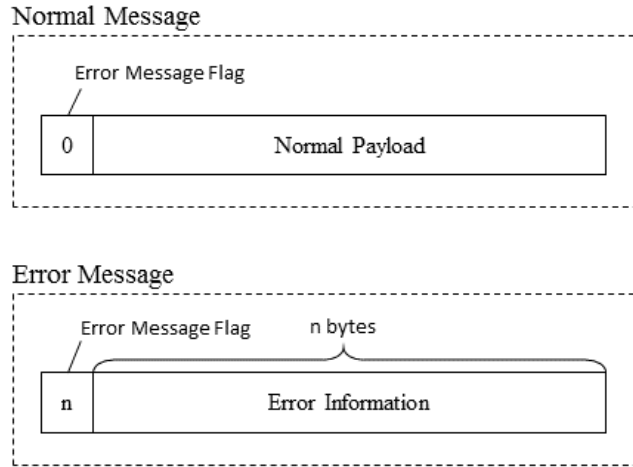


Figure 6.8: Normal Message and Error Message

6.3 Implementation Details

6.3.1 UML Class Diagram

A UML Class Diagram has been plotted to show all the main classes in the program and the relations between them.

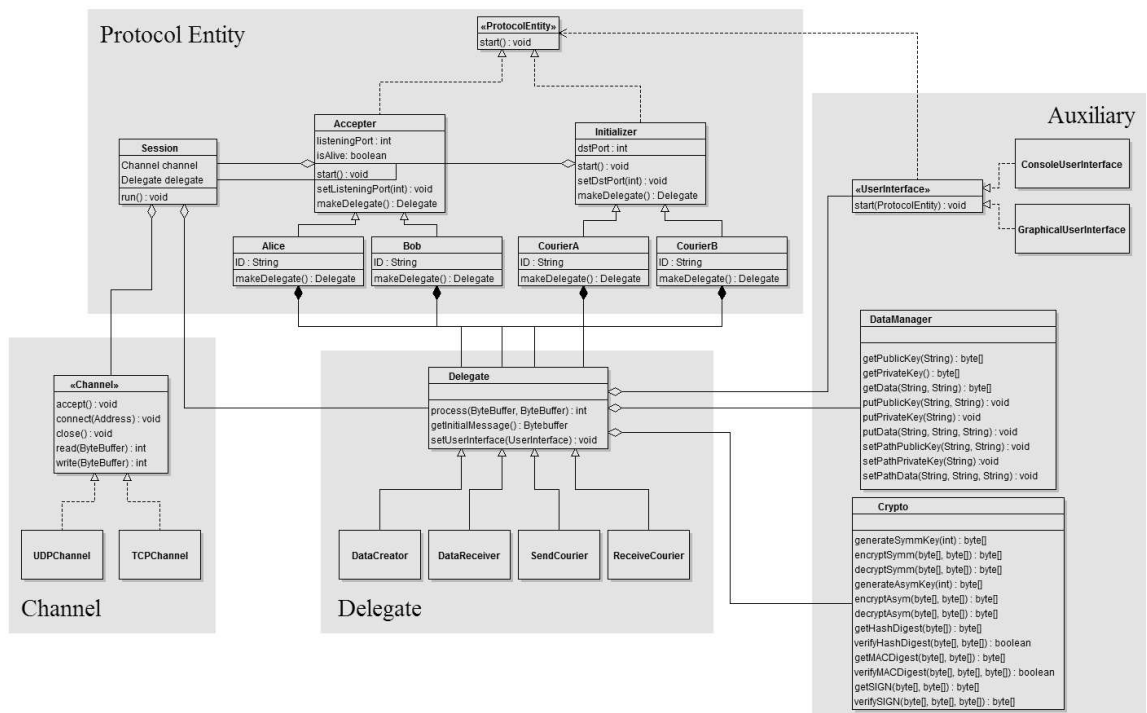


Figure 6.9: UML Diagram

According to the diagram, the whole program can be separated into 4 main parts:

- **Protocol Entity**

This part defines the framework of this application. Any protocol entity can be first specified into an Acceptor or an Initiator, then based on the role it plays, it can be further classified into a particular role like Alice, Bob, CourierA (Courier who connects to Alice) or CourierB (Courier who connects to Bob). As shown in the diagram, every level of classification corresponds to a group of classes and lower level classes inherit from the higher level classes. The main job of Protocol Entity classes is creating, initializing and managing its Session objects. While a Session class - which is possessed by every protocol entity, deals with the message transmission issues for the entity, such as requesting next messages from Delegate, sending messages to the Channel, etc.

- **Channel**

The Channel is used in Session class, it is an Interface denotes a place where messages can be sent and received. Ideally, when two channels are connected, devices can easily push message in and receive message from their channels without awareness of any detail about the connection. However, due to different networks and protocols suites may be used when running CDSProtocol, various implementations of Channel are necessary. As shown in the diagram, possible implementations of Channel Interface can be TCPChannel and UDPChannel, corresponds to connecting under TCP or UDP. Except above two, even more different channels could be developed like BluetoothChannel, CableChannel, etc. In this prototype program, only UDPChannel is currently implemented and the detail of it will be explained in the later “Connection Establishment” section.

- **Delegate**

As has been demonstrated in the “Delegate Model”, Delegate class defines the “rule” of an entity. It takes a message as input, processes the message and outputs a responding message if there is one. However, Delegate is just an abstract notion, its children classes define the real “rules” of each entity respectively. As shown in the diagram, DataCreator, DataReceiver, CourierSender, CourierReceiver are 4 different implementations of Delegate class, they define the “rules” for the 4 entities correspondingly - Alice, Bob, CourierA and CourierB.

- **Auxiliary**

This part contains 3 components that can be totally independent with the design of CDSProtocol. The `UserInterface` only deals with inputs from user and displays output to user. Reflect to the software design, its implementation can be various to fit different devices and platforms. The `DataManager` takes charge of managing the disk files that is related to the running of the protocol. And the `Crypto` represents “Cryptographic Kit” in the software design, its job is to provide all cryptographic operations that is needed in the protocol.

Extensibility and Reusability The design of the program structure attempts to maximize the extensibility and reusability of the program in order to make the program easy for modifying and extending after release. It should be noticed that all components of the program are loosely coupled which makes them easy for reuse, and frequently used inheritance and interfaces makes the program extensible. Developers can modify or create their own entities, delegates, channels or even cryptographic kits without affecting other parts of the code.

6.3.2 Connection Establishment

The `Channel` is used to establish connections between entities in program. Each `Channel` object is associated with a local address and listening port before connected. There are 2 steps for two entities to get connected using `Channel`. Firstly, `Initiator` initiates the connection by calling the `connect()` method of its `Channel` object, specifying the destination IP address. Then the `Acceptor` accepts the connection by calling the `accept()` method of its `Channel` object. The `accept()` method will wait until there is a connection to accept and what it really does is creating a new `Channel` object which is connected to the incoming connection, and returns that newly created `Channel` object. Once the connection is accepted, both entities can use `Channel`’s `read()` and `write()` method for receiving and sending messages. If any of the entity chooses to terminate the connection, it calls the `close()` method of its `Channel`, then both `Channels` are closed.

UDPChannel The above connecting mechanism is effective in this protocol, as the `Channel` object who accepts connections will not be connected to any `Channel`, thus it can accept several connections without changing its listening port - imaging if channel gets connected once it accepts a connection, then when a new connection comes, it will not be able to accept it. However, to implement this mechanism for

UDP requires extra effort because UDP itself is connectionless and every packet sent or received is independent between each other. The real process for establishing connections using UDPChannel is: When the Initiator calls the connect() method of its Channel object, the channel records the destination address and port number but sends nothing. So when Acceptor calls accept() nothing will happen until Initiator sends the first message, at which point accept() will create a new Channel object with a different listening port and all future messages will be sent through this Channel object. So, when Acceptor sends back the second message, the port number of the channel sending the message is different with what Initiator specified in the first message. So when Initiator receives a reply message from Acceptor, it changes its channel's destination port number to the coming channel's port number, so that its future messages will be sent to the Acceptor's newly created channel. Finally the connection is built between Initiator's channel and Acceptor's newly created channel.

6.3.3 Message Processing

When a message is received from the channels, it will be examined by the session first, where the Error Message Flag will be checked. If the Error Message Flag (the first byte of the received message) is greater than 0, it indicates the message is an Error Message and the number of the Error Message Flag represents is the length of the error information. Then session will extract the error information and displays it on the user interface. If the Error Message Flag is 0, it means the message is a Normal Message. Then session will extract the payload and feed it to the delegate. Delegate will further process the payload and returns an integer indicating the size of the responding message. If the integer is 0 it means the message is successfully processed and no further messages to be sent. If the integer is negative, it means an exception occurs during the processing.

When session gets a message from delegate, it should wrap it into a Normal Message by adding a 0 byte in the front of the message. Then session sends the wrapped message to the channel and waits for receiving next message from the channel.

6.3.4 Cryptographic operations

The cryptographic functions provided by the Crypto class in this program are mainly built from two Java packages "java.security" and "javax.crypto", thus the security of this application highly relies on the implementation inside those two Java packages. Below will list and explain all the cryptographic operations used in this protocol. As

there is no sign indicating there is any flaw inside the Java security packages, we will not dig too deep into the security cryptographic implementation.

- **Symmetric Encryption / Decryption / KeyGeneration**

The symmetric encryption / decryption scheme used is AES [25]. The mode of operation used is CBC [18]. The padding scheme is PKCS#5 [22]. The size of key used is 128 bits and it should not exceed 936 bits.

- **Asymmetric Encryption / Decryption / KeyGeneration**

The asymmetric encryption / decryption scheme used is RSA. The mode of operation used is ECB, however, for efficiency concern, the size cipher block of asymmetric encryption will be restricted to 1, so mode of operation will not actually be used in the program. The padding scheme is PKCS#1 [21]. The size of key used is only 1024 bits, as this is only a prototype application and key size can easily be increased in future.

- **MAC Creation / Verification**

The MAC algorithm used is Hmac-SHA-256 [16]. The size of key used is same to symmetric encryption / decryption, which is 128 bits.

- **Digital Signature Creation / Verification**

The signature algorithm used is RSA. The cryptographic hash function used is SHA256 [16]. The key size is same to asymmetric encryption / decryption, which is 1024 bits.

6.3.5 File Management

It is assumed that adversary is not able to hack into its target entity's system, that is, the adversary can not access to the memory or disk of a honest entity's device. Based on this assumption, all the data files are stored explicitly in device's disk.

Keys As a single device is able to act as several different entities, it is possible that one device possesses several secret keys and presumably a long list of others' public keys. The user should take care of his/her own key files, and before the protocol starts, he/she specifies the locations of all keys. Then during running, the application will read those files when necessary.

Alice's Message Content Similar to the keys, Alice's message content should be stored in the disk of Alice's device. Before the start of the protocol, user will be requested to specify the location of that file. Then during Submit Protocol, the file will be read when necessary.

Courier's Payload Courier's payload files are not specified by user, it is managed by the application. For every Courier entity, it has a working directory, by default, Courier's working directory is in the application's directory, named after its ID. Once Courier gets data from Alice in Submit Protocol, it will save the data into its working directory, the file name will be the destination entity's ID. For example, if Courier0 receives data from Alice to Bob, the data will be saved in the file `“./Courier0/Bob”`. As Courier may receives data from multiple datacreators, all further data will be appended to the file named after the receiver's ID, if no such file, program will create it. So, after the Message Acquisition phase, Courier may have several data files, and each contains data from multiple datacreators. Finally in Transmit Protocol, Courier transmits the corresponding data file to its destination entity base on the file name.

6.3.6 Error Handling

In this application, errors are classified into 4 main classes - user input error, I/O error, protocol error and timeout. Below will explain the details of these 4 errors and how they are handled in the program.

User Input Error User input errors occur when user input meaningless content into the application's textfield, such as inputting non-numeric string as port number, etc. This kind of errors will be caught before the protocol starts running, after user click “Start” button. Once user input error is detected, its detail will be sent to user interface where the error detail will be shown to the user.

I/O Error I/O error denotes errors happening when reading/write files or sending/receiving through network channels, such as files not found when reading files, or port number is in use when creating a network socket, etc. These errors are caught during the running of the protocol and will cause termination of the protocol run. Similar to user input error, once I/O error is detected, its detail will be sent to user interface and then be shown to the user.

Protocol Error Protocol error is produced by the protocol itself, when there is some checking violation happens such as verifying a MAC to false, or the message size exceeds the capacity of Courier's storage, etc. All the checking processes have been fully described in the protocol specification, and the program strictly follows those processes. Because all checking operations happen in delegates while delegates processing the received messages, the protocol error is detected by delegates. Once a protocol error is detected, firstly, the error detail will be sent to user interface and be shown to the user. Meanwhile, the delegate will return a negative integer indicating error happens during processing. Then session will send a Error Message to the other entity reporting the error. However, for the security concern, Error Message will not include any detail about the error, currently it just carries a string "Protocol Error".

Timeout Technically timeout is not an error but a mechanism of protecting an entity from waiting responses for too long time. And it is essential in this protocol because when an I/O error occurs during the protocol, it immediately terminates the program in this device, without reporting anything to the other device which still waits for reply. Timeout allows the other device to abort the protocol if no message is received after waiting for a certain amount of time. Timeout is detected in sessions. After sending out a message session will first check whether it is the end of the protocol, if it is, session will close the channel and terminate. If it is not, session will monitor the channel with a timer. Upon the time runs out, it will voluntarily close the channel and terminate.

Chapter 7

Test and Evaluation

7.1 Latency Test

7.1.1 Submit Protocol

7.1.2 Transmit Protocol

7.2 Scalability Test

7.2.1 Submit Protocol

7.2.2 Transmit Protocol

Chapter 8

Conclusion and Future Work

8.1 Conclusion

The recently introduced Delay-Tolerant Network architecture assembles networks with different latencies, bandwidth limitations or node longevities together and makes the interoperation between them achievable. Although DTN's high level security issues have been analysed and implemented in many publications, there is no a proper protocol specifically focusing on its courier-dependent custody transferring. Two of the related security protocols - Bundle Security Protocol and DTN Anonymity and Secure Architecture, have been examined and discussed, consequently, it has been shown that there are still some improvements to be made.

Then a new security protocol specifically designed for courier-dependent communication is created, named Courier-Dependent Security Protocol (CDSP). It has following main security achievements - authentication of originator and recipient, authenticity of origin of the message, confidentiality of the message content and deniability for originator and recipient, and the time efficiency of the protocol is raised to a high priority. To show the protocol, firstly the whole system that the protocol will be working in is fully described. Then is the protocol specification, specifies the detailed procedure of CDSP step by step. The last part of the protocol illustration summaries all security properties of CDSP, showing its capability and then proves every one of those properties.

After CDSP has been thoroughly described, a concrete implementation of this protocol is developed in JAVA. The program mainly consists of a core library which provides all essential functions to run CDSP and a runnable application uses the core library. The core library is designed with emphasized extensibility and reusability, thus it can be easily modified and extended after the prototype released. The program

architecture is illustrated by understandable figures and its security-related designs are fully documented afterwards.

Finally the application is tested for performance evaluation. Two main aspects - transferring latency and scalability have been taken into account, and some interesting conclusions of the protocol behaviour have been made.

8.2 Discussion and Future Work

The most obvious defect of this protocol system is that the success delivery of messages is never guaranteed by courier. Due to the efficiency and simplicity concerns, couriers are never required to authenticate to message creator, which means any entity can claim itself as courier and get the data. It leads to the problem that message creator will never know whether her message has been submitted to the real courier, neither will her know whether her message will eventually be delivered to the recipient. This kind of uncertainty can be problematic in some scenarios where the success delivery guarantee is strongly demanded - like transmitting military intelligence. One potential solution can reduce the extend of its uncertainty by forcing couriers to authenticate to message creator before they run Submit Protocol. It ensures only authorized couriers can successfully get data from message creator, and assume only well behaved couriers can get authorized, the probability of success delivery will definitely be increased. Furthermore, if any message is lost by a courier, later it will be very easy to trace back to the troublemaker. Nevertheless, the drawback of it is also non-negligible. Firstly it increases the overhead for key distribution and management as not only end users possess unique key pairs but also every authorized couriers. Secondly it burdens the running of CDSP as mutual authentication is needed when both Submit and Transmit Protocols. As a consequence, in the future implementation, it would be reasonable to provide both two sets of protocols in the application and leave it for users to choose which one to run based on the specific circumstances.

Another imperfection in the protocol design is that: after running the protocol, message originator can not fully deny sending a message to the recipient. As stated in the protocol property specification, this compromise is made to minimize the computational complexity while running the protocol as non-interactive deniable authentication is computational consuming. So, in the future implementation, it is also expected that user can choose which deniable authentication method to use based on the specific circumstances.

Besides, the protocol specification does not cover the corresponding schemes for key distribution, key management and key revocation, instead, they are assumed to be taken care of before running the protocol. Although making such minimum assumptions about the requirements makes the protocol as generic as possible, however, it may reduce the consistency of the whole system. Thus we hope that the best fitted key manipulation schemes can be created for CDSP protocol in the future revised designs.

Also it will be plausible if application can be developed in portable devices, so that tests and evaluations can be done on the portable devices as well. It obviously is more close to the real scenario where only portable devices will be used as couriers in the protocol, and with the extensible and reusable core library developed, it will not be a too time-consuming task.

Finally the efficiency of the protocol is always eager to be improved as it is one of the main requirement of the protocol design. It could be accomplished by exploring some more newly invented efficient cryptographic operations and substitute the less efficient ones, or by further revising the code of the implementation.

Appendix A

Application Instruction

To start a protocol using this application involves 3 basic steps - (1) user chooses a specific protocol role. (2) user inputs the information related to the selected protocol role. (3) user starts the protocol. Below is a snapshot of current GUI of the application, it basically consists of 4 main panels:

- **Protocol Entity**

It is on the very top. It provides 4 roles (as mentioned above) to be chosen for the user. Because every device runs this protocol must be one of those four roles, it is essential that user specifies a role in this panel before start running the protocol.

- **Entity Identity**

It locates on the left. Here user specifies the information about the host device and the destination device, such like IDs, and IP addresses.

- **File Paths**

It locates on the right. Here user manages all the disk files that are related to the running of the protocol. Presumably, many important information like public keys, private keys and messages are all files in the device's disk, user has to specify those files' paths before running the protocol.

- **Console**

It is at the bottom. Here provides buttons for user to start and stop the protocol, and it displays information about the running protocol in a information board.

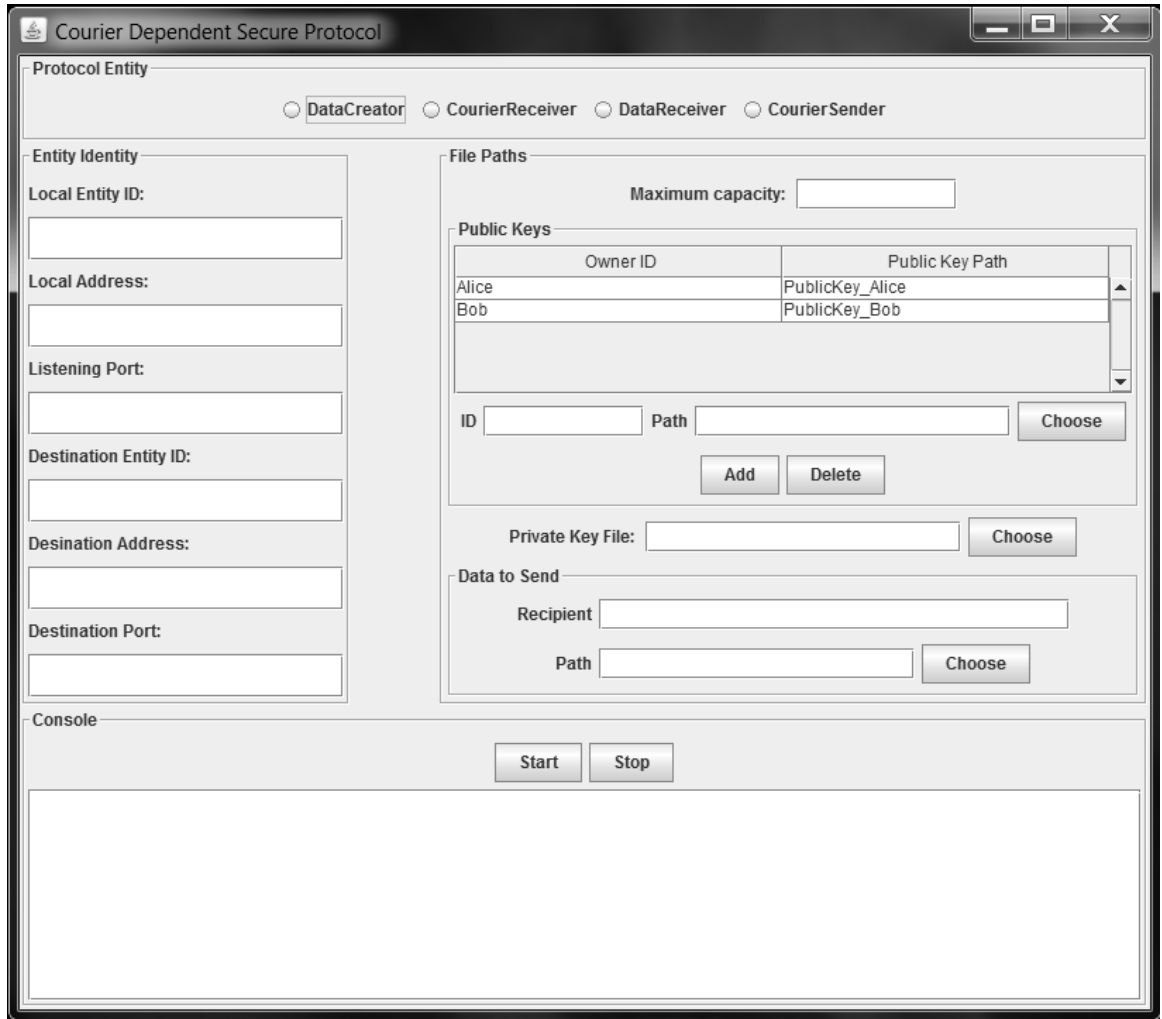


Figure A.1: GUI Overview

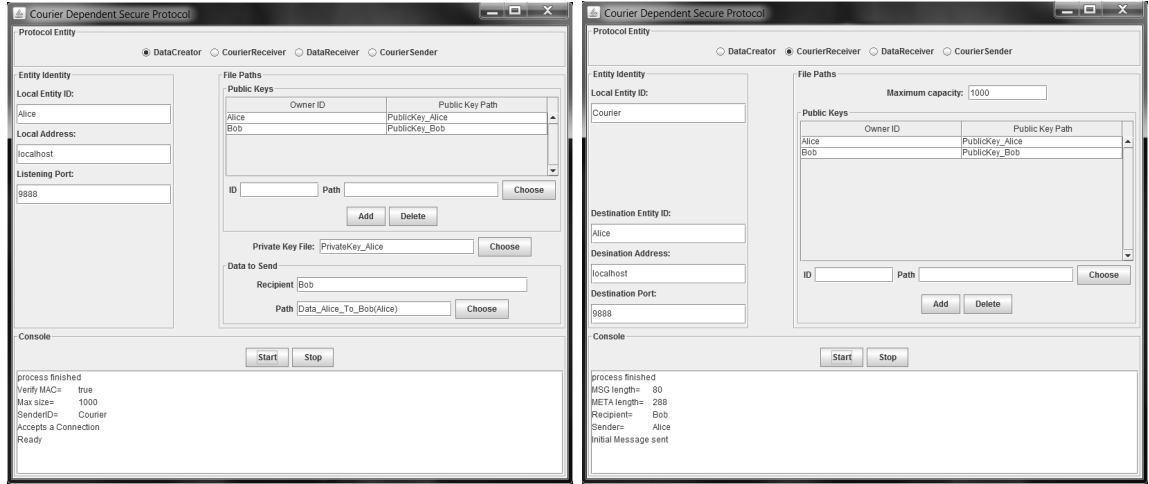
A.1 Run Submit Protocol

It requires at least 2 devices to run this protocol, we call them D_0 and D_1 . Assume D_0 is entity Alice, who possesses a message and wants to send it to Bob. Assume D_1 is entity Courier, who is able to carry the message and transmit it to Bob. To run the protocol successfully, it should be done by the following procedure:

1. D_0 starts the application.
2. D_0 user selects “DataCreator” in the Protocol Entity panel.
3. D_0 user specifies the entity ID, local address and listening port (9888 by default) by filling the corresponding textfields in Entity Identity panel.

4. D_0 user adds destination entity's public key into its public key list.
There are two ways to add an new entry to the public key repository:
 - (a) Adding through application: input the ID of the public key holder and the path of public key file by filling the textfields in the Public Key panel. Then click Add button. The (ID, path) pair will then appear in the public key list. This effect is temporary, the pair will disappear in the next launch.
 - (b) Adding through configuration file: by default, when application is launched, it will import all (ID, path) pairs from a file named PublicKeys in the current directory. User can appending a new line to PublicKeys file to add a new pair. The format is ID;Path, e.g. "Alice;/publickeys/alice.pk". This effect will take place since next launch of the application.
5. D_0 user specifies its private key by filling the textfield with the private key file.
6. D_0 user specifies recipient entity's ID by filling the textfield in the Data to Send panel.
7. D_0 user specifies the file that contains the message needs to be sent by filling the file path in the corresponding textfield.
8. D_0 user click Start button in the Console panel. The running progress of the protocol will be displayed in the information board in the Console panel.
9. D_1 starts the application.
10. D_1 user selects "CourierReceiver" in the Protocol Entity panel.
11. D_1 user specifies its own entity ID by filling the textfield in Entity Identity panel.
12. D_1 user specifies the ID of the entity he wants to contact (here D_0 's ID), together with its address and port number by filling corresponding textfields in the rest of Entity Identity panel.
13. D_1 user add the public key of the entity he wants to contact into the public key list. The way of doing that has been introduced in step 4.
14. D_1 user click the Start button in the Console panel. The running progress of the protocol will be displayed in the information board in the Console panel.

Below figures show snapshot of two applications successfully running Submit Protocol. After application consoles on both devices pops “process finish” successfully, the message of D_0 has been submitted to the D_1 , and is stored in the file named by the message recipient’s ID. If D_1 gathers messages from multiple devices, all messages to the same recipient will be accumulated in the same file.



(a) DataCreator

(b) CourierReceiver

Figure A.2: Running Submit Protocol

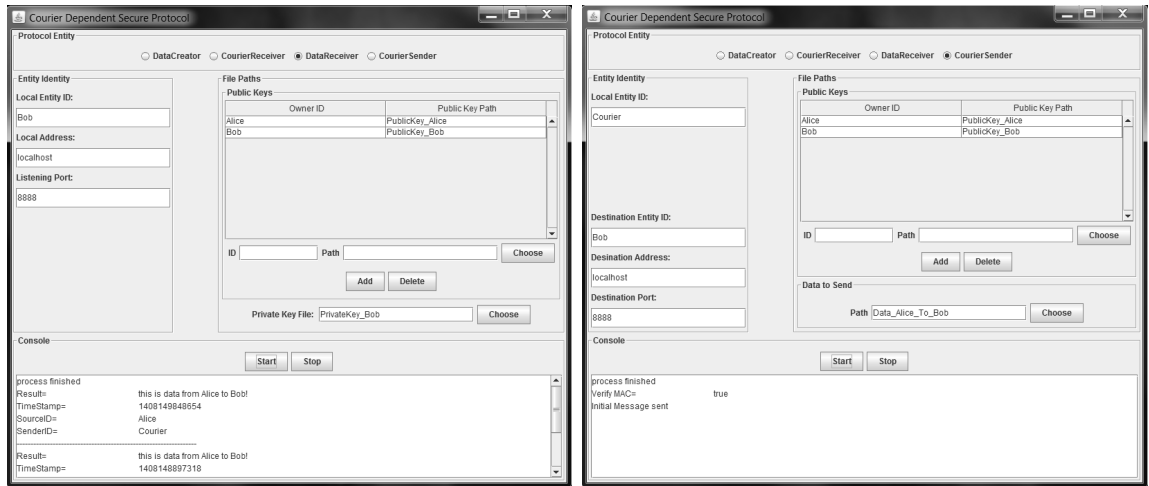
A.2 Run Transmit Protocol

It requires at least 2 devices to run this protocol, we call them D_1 and D_2 . Assume D_1 is entity Courier, who possesses an encrypted message from Alice and wants to send it to Bob. Assume D_2 is entity Bob, who waits to receive message from Alice. The running procedure of Transmit Protocol is:

1. D_2 starts the application.
2. D_2 user selects “DataReceiver” in the Protocol Entity panel.
3. D_2 user specifies the entity ID, local address and listening port (8888 by default) by filling the corresponding textfields in Entity Identity panel.
4. D_2 user adds destination entity’s public key into its public key list. The way of doing that has been introduced in the step 4 of running Submit Protocol.
5. D_2 user specifies its private key by filling the textfield with the private key file.

6. D_2 user click Start button in the Console panel. The running progress of the protocol will be displayed in the information board in the Console panel.
7. D_1 starts the application.
8. D_1 user selects “CourierSender” in the Protocol Entity panel.
9. D_1 user specifies its own entity ID by filling the textfield in Entity Identity panel.
10. D_1 user specifies the ID of the entity he wants to contact (here D_2 ’s ID), together with its address and port number by filling corresponding textfields in the rest of Entity Identity panel.
11. D_1 user add the public key of the entity he wants to contact into the public key list. The way of doing that has been introduced in step 4.
12. D_1 user specifies the file that contains the data needs to be sent, by filling the file path in the corresponding textfield in Data to Send panel.
13. D_1 user click the Start button in the Console panel. The running progress of the protocol will be displayed in the information board in the Console panel.

Below figures show snapshot of two applications successfully running Transmit Protocol. After application consoles on both devices pops “process finish” successfully, the data carried by D_1 has been transmitted to D_2 . If the authenticity of the data is successfully verified, D_2 will print the message content out in the information board. If the data contains messages from multiple senders, they will be verified and printed one by one so that user knows which messages have been discarded.



(a) DataReceiver

(b) CourierSender

Figure A.3: Running Transmit Protocol

References

- [1] Anonymity and security in delay tolerant networks.
- [2] Bundle protocol specification.
- [3] Bundle security protocol specification.
- [4] Concurrent zero-knowledge: Reducing the need for timing constraints.
- [5] Controlling the mobility of multiple data transport ferries in a delay-tolerant network.
- [6] A delay-tolerant network architecture for challenged internets.
- [7] Delay-tolerant networking: An approach to interplanetary internet.
- [8] Delay-tolerant networking architecture.
- [9] An improved deniable authentication protocol.
- [10] Lowcost communication for rural internet kiosks using mechanical backhaul.
- [11] Offtherecord communication, or, why not to use pgp.
- [12] Practical security for disconnected nodes.
- [13] Private authentication.
- [14] A secure and efficient deniable authentication protocol.
- [15] A simple protocol for deniable authentication based on elgamal cryptography.
- [16] D. Eastlake 3rd and T. Hansen. RFC 6234: Us secure hash algorithms (sha and sha-based hmac and hkdf), May 2011. Status: INFORMATIONAL.
- [17] D. Dolev and A. C. Yao. On the security of public key protocols. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:350–357, 1981.

- [18] W.F. Ehrtam, C.H.W. Meyer, J.L. Smith, and W.L. Tuchman. Message verification and transmission error detection by block chaining, February 14 1978. US Patent 4,074,066.
- [19] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, April 1988.
- [20] Sushant Jain, Kevin Fall, and Rabin Patra. Routing in a delay tolerant network. *SIGCOMM Comput. Commun. Rev.*, 34(4):145–158, August 2004.
- [21] B. Kaliski. RFC 2313: PKCS #1: RSA encryption version 1.5, March 1998. Obsoleted by RFC2437 [?]. Status: INFORMATIONAL.
- [22] B. Kaliski and RSA Laboratories. RFC 2989: PKCS5: Password-based cryptography specification version 2.0, September 2000. Status: INFORMATIONAL.
- [23] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC: Keyed-hashing for message authentication, February 1997. Status: INFORMATIONAL.
- [24] Wenbo Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall Professional Technical Reference, 2003.
- [25] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.
- [26] K. Ohgishi R. Sakai and M. Kasahara. Cryptosystems based on pairing. *SCIS 2000: Symposium on Cryptography and Information Security*, 2000.
- [27] Andrew C. Yao. Theory and application of trapdoor functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 80–91, Washington, DC, USA, 1982. IEEE Computer Society.