# A Secure Protocol for Courier-Dependent Communication


Chen Sun

Trinity College

Supervised by Dr. Kasper B. Rasmussen

University of Oxford


A thesis submitted for the degree of

*Master of Science*

Trinity 2014

# Acknowledgements

I would like to offer my sincerest appreciation to my supervisor Dr. Kasper B. Rasmussen, for his continuous support on my work. His vast knowledge guided me out of many difficulties and his serious and patient working attitude set a great example of being a responsible researcher.

Besides, I would also like to thank Dr. Andrew Ker whose enthusiasm for his work ignites my strong interests on computer security area.

Last but not least, I would thank to my parents, for their endless support remains as strong as always, especially during the difficult times of writing this thesis.

# Abstract

On the contrary of modern Internet - which provides reliable and continuous connection between end-to-end users, the so called “challenged networks” denotes a group of networks which are characterized by high latency, restricted bandwidth limitation, short node longevity or bad path stability. In those challenged networks, typical TCP/IP protocol will not be functioning any more, thus other ways have to be used to establish connections within these networks. One of the common methods to achieve communication within challenged networks appears to be using portable devices as couriers to deliver the message for end users. Although this method has been widely used, it is spotted that this intuitive method leaks the information about communication content thus extra work need to be done to ensure its secure communication.

To solve the problem, this thesis aims to create a new protocol to allow secure and efficient communication established in the courier-dependent networks. In this these, first it formalizes such courier-dependent communications by creating a specific communication scenario which takes its security properties into account. Then it proposes a secure protocol for the specific scenario. The protocol provides mainly 4 security guarantees during the communication - authentication, authenticity of origin, confidentiality and deniability. After that, an application of the protocol is developed and the evaluation of the application proves its efficiency and scalability meets the general requirement as an applicable implementation.

# Contents

# List of Figures

# 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 from all over the world. Its major achievement is 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 Alice and Bob 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 extremely emphasized:

Assuming there are some off-line entities $\text{Alice}_0$, $\text{Alice}_1$, ... $\text{Alice}_m$, 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 - $\text{Alice}_0$, $\text{Alice}_1$, ... $\text{Alice}_m$ 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 a security guard. Thus, to make a successful delivery, a Courier first approaches $\text{Alice}_0$, $\text{Alice}_1$, ... $\text{Alice}_m$ 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 any third party (including security guard and the Courier itself).

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 many courier-dependent communication scenarios are similar to CDSP.

## 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 scenario has been briefly introduced and the basic requirements have been clarified. Here more detail of CDSP's achievements will be listed. More concretely, the invented CDSP takes following properties into consideration:

- Authentication
  Because the cost of such communication will be high - the transportation of courier is costly, it requires the Courier only delivers message for authenticated entity. Namely, if Courier wants to deliver message for $\text{Alice}_0$, $\text{Alice}_0$ 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 to Bob.

- 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 identity 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
  In some secret delivery missions - such like exchanging military intelligent, it would be plausible if Alice is able to deny the fact of 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 a portable device 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.

Above are aims of the protocol that will be created in this project. And at the end of the project, following objectives are achieved:

1. A fully specified Courier-Dependent Security Protocol will be created and it meets the requirements mentioned in above list.

2. A Java library will be developed providing essential functions for implementing the protocol.

3. An application will be built to actually run this protocol.

4. A test of the application will 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 initial settings, assumptions and rules. It is designated to convey a detailed picture of the scenario. Chapter 4 will propose a well defined Courier-Dependent Security Protocol, and its specification will be fully illustrated with the help of message sequencing charts. Afterwards, in Chapter 5, the security properties of CDSP will be highlighted and all of them will be proved in this chapter. 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

## 2.1 Background

This project is not merged out of the void and the courier-dependent communication scenario is also not invented groundlessly. Some of the notions in this project have been introduced and discussed antecedently. To convey an entire overview of why this project is introduced and how it is designed, some details of the background information are given in this chapter.

### 2.1.1 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 [9]. 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 [9], 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 [4], 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 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.

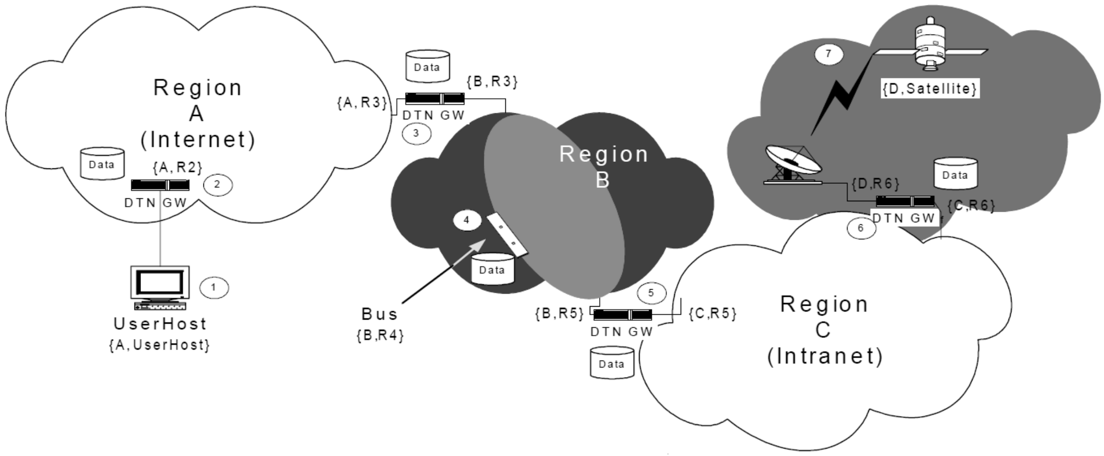



Figure 2.1: Overview of a Delay-Tolerant Network, cited from [9]

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 transfer, 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 some of DTNs [11][27]. The scenario introduced in this project is a concrete instance of such custody transfer and the network our CDSP works in is also referred to an instance of challenged networks.

Although the security issues of DTN have been discussed and evaluated in the high level design of the architecture [5][19], such like essential authentication and message integrity preservation, etc., they are specially designed to maintain efficient routing and prevent abuse of DTN, many other 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 predefined in the architecture, this project will dive into the detail of courier-dependent message delivery and create a practical and efficient security protocol for the specific scenario.

### 2.1.2 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 authentication methods do not take privacy as one of its goals, thus they require explicitly showing entity's unique identifier 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 as the identifier is exposed to any eavesdropper. 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 [2].

The concept of deniable authentication is created even earlier, by Dwork and Sahai [7]. Different with private authentication, 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 creating the message at all. It is extremely powerful in terms of privacy protection - 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 [3] and non-interactive [22][24][25]. Interactive deniable authentication involves interaction between entities, it 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 forward 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.

## 2.2 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.

### 2.2.1 Bundle Security Protocol [23]

Soon after the release of DTN architecture specification [5], a protocol called Bundle Protocol [19] 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 is 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 strictly 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. Its main contribution is that it defines 4 new types of bundle payload block - BundleAuthenticationBlock, PayloadIntegrityBlock, PayloadConfidentiality-Block 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.

Bundle Security Protocol has defined the security issues of DTNs 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

```
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 [19]

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 transfer is assumed to be accomplished securely and efficiently so that it only cares about the higher level. Unfortunately, it is not always the case. Every custody transfer 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 transfer and create a protocol for it. Although the message format of Bundle Security Protocol is highly referential, it still not quite fit the scenario we discussed before. The major problem is: as a general protocol it has a very heavy overhead to maintain the consistency of different blocks and extensions, which seems redundant in our scenario. So comparing to the Bundle Security Protocol, the newly created protocol will use less-sized messages and be more problem specific.

### 2.2.2 DTN Anonymity and Secure Architecture [14]

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 its long-period disconnection [20][21]. Although specific situations could be various in rural areas, Seth et al.'s study comes up with an unified approach: 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.

The DTN Anonymity and Secure Architecture is very close to the scenario of this project - buses can be regarded as couriers who carry the message for end users, and the protocol used in this architecture takes care of the authentication processes between end users and the buses, and it also 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 either is interactive or demands extra assumptions - such as in Sakai-Ohgishi-Kasahara (SOK) Key Agreement Scheme [18], it needs a trusted third party "private key generator", which increases the system complexity, thus one way authentication might be a better choice. (3) Its anonymity mechanism highly relies on the 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

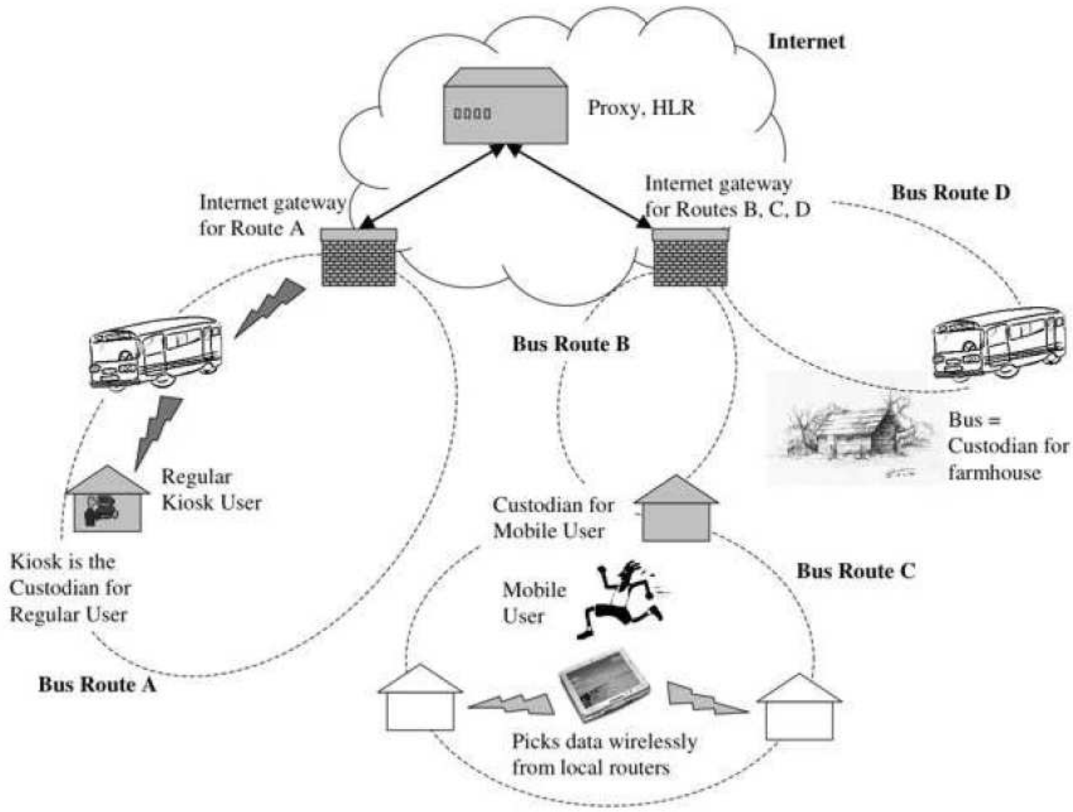



Figure 2.3: Rural Area DTN [21]

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.

In summary, the DTN Anonymity and Secure Architecture sets up a scenario similar to our project's courier-dependent scenario, and some methods it uses to achieve security protection is inspiring. However above 4 points show that in order to fully fit the scenario of this project, some improvements still need to be done.

# Chapter 3

# System Description

## 3.1 System Model

### 3.1.1 Overview

Based on the scenario demonstrated in Introduction chapter, this system describes the networks that have no reliable or stable networks existed between each end-to-end entities. Entities under such conditions are regarded 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. We define 3 abstract entities in the system - Alice, Bob and Courier: we abstract all entities which originate messages as Alice, all entities who receive messages as Bob and entities who deliver messages as Courier. We assume whenever an Alice has messages to send, eventually there will be a Courier approaches her and carries messages for her. Thus, if Alice wants to send a message to Bob, the Courier firstly gets the message from Alice and stores it, then physically transports to Bob and sends the message to Bob. During the transportation, Courier has to cross a border where Courier's data will be examined by the security guard.

The goal of the system 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 will never be trusted, which means the real content of the message will never 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) is able to deny sending the message content to message recipient as well.

### 3.1.2 Initial Setting

According to the description above, totally 3 different types of entities are defined within the system - Alice, Bob and Courier. Following specification lists the notations and jobs of those 3 entities together with the information they hold:

- Alice $\mathcal{A}$ denotes a set of devices who create messages and waits it to be delivered, so it is also called "message creator". It possesses an unique ID $\mathcal{ID}_A \in \{0,1\}^*$, a secret key $sk_A$ as part of its asymmetric key pair, and the message $\mathcal{M}$ to be sent.

- 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 ID $\mathcal{ID}_B \in \{0,1\}^*$, and a secret key $sk_B$ as part of its asymmetric key pair.

- 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. A single Courier will play two different roles in the system - when it receives message from Alice, it is denoted as $\mathcal{CR}$ (Courier Receiver), after that, when it delivers message to Bob, it is denoted as $\mathcal{CS}$ (Courier Sender). Initially $\mathcal{CR}$ only possesses an unique $\mathcal{ID}_C \in \{0,1\}^*$. After it gets messages from Alice and before it delivers message to Bob, it plays as $\mathcal{CS}$ and possesses some more information: the ID of message recipient $\mathcal{ID}_B$ 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 system, thus $\mathcal{A}$ and $\mathcal{B}$ are assumed to hold their asymmetric key pairs before the communication. 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 the communication.

**Devices v.s. Entities** A device $x$ denotes a physical object within the system. Differently, an entity denotes a particular role in the system. It should be noted that any device can communicate with multiple other devices simultaneously, thus a single device can be any both $\mathcal{A}$ and $\mathcal{B}$ at the same time. The role it plays in different communications is defined by what information $x$ holds and what its behaviour is (detail will be described in Trusted Entity Behaviour). On the contrary, an entity in the system can be associated with only one device.

### 3.1.3 Communication Model

Within the system, there will be several message creators $\mathcal{A}$ and they possibly will send messages to different message recipients $\mathcal{B}$ simultaneously, thus a single courier is able to collect messages from multiple message creators at a time. Because the storage capability of single courier is limited, to keep the fairness between each message creator, each message creator can submit maximumly one message to a courier at a time. Typically, a single communication requires courier collects all data from message creators and transports them to all message recipients. However, because each communication between several message creators and a single message recipient is totally independent with each other, the whole communication can be regarded as several independent M-to-1 communications. For example, achieving communication of $a_0 \in \mathcal{A}$ sends message to $b_0 \in \mathcal{B}$ and $a_1 \in \mathcal{A}$ sends message to $b_1 \in \mathcal{B}$ using a single courier can be regarded as achieving two independent communications which are (1) $a_0 \in \mathcal{A}$ sends message to $b_0 \in \mathcal{B}$, and (2) $a_1 \in \mathcal{A}$ sends message to $b_1 \in \mathcal{B}$ using two couriers, from the view of the system, they are totally the same. Thus this independent M-to-1 communications - M message creators each send one message to a single message recipient, can be regarded as the primary model for constructing any complex communications within the system. Because of this, the whole communication model of the system can be reduced to such M-to-1 communication model, and it will be described in following part.

An abstract M-to-1 communication involves following entities: a certain number of off-line devices $a_0, a_1, ...a_M \in \mathcal{A}$ send messages to a single off-line device $b \in \mathcal{B}$ independently, using a device $c \in \mathcal{C}$ as media. As physical transportation is slow and costly, the total number of physical transportation is expected to reduce to minimum. The optimized solution appeared to be separating the communication into two main phases - Message Acquisition phase and Message Delivery phase. In Message Acquisition phase, the courier $c$ will physically transport to every $a \in \mathcal{A}$ one by one, connect to it and get the message that is for $b$. After $c$ collects all needed messages, it enters Message Delivery phase, where $c$ transports to $b$ and transmit all acquired messages to it.

According to explanation above, the whole task can be divided into M + 1 individual sub-communications. In Message Acquisition phase, the courier establishes a sub-communication with each $a$ to get message from it, and in Message Delivery phase, the courier establishes sub-communication with $b$ to transmit messages to it. All these sub-communications are assumed to be established by typical network communication methods (e.g. cable, Wifi, Bluetooth, etc.) and apply their corresponding communication protocols (e.g. TCP/IP, UDP, Bluetooth protocol, etc.). The following figure shows how the communication is organized.

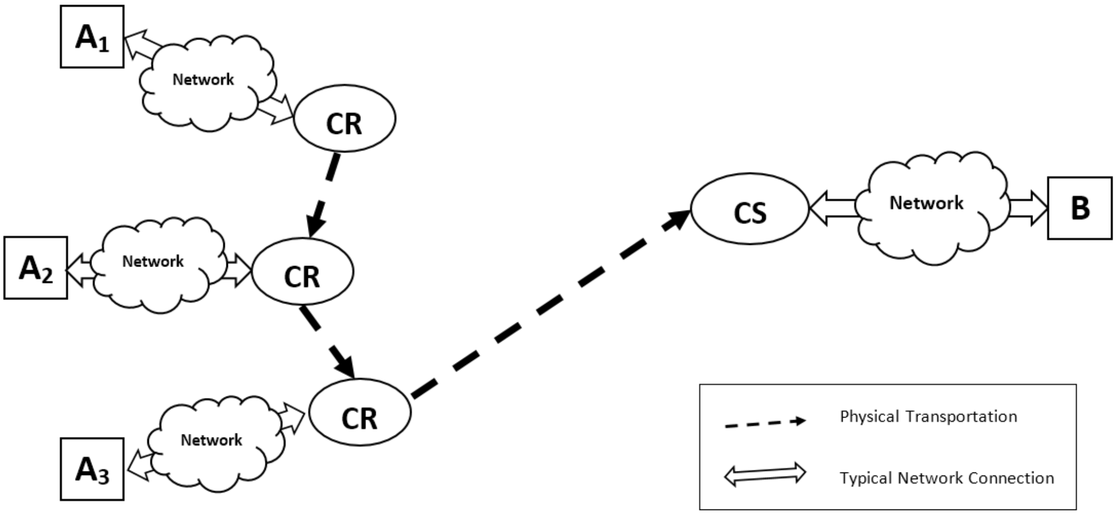



Figure 3.1: Communication Model

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 does 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 use any kind of typical network communication methods and do not have to be same, as long as both entities agree.

### 3.1.4 Honest Entity Behaviour

To achieve the communication, all three entities $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ have different behaviour patterns, the actions they can take to interact with the rest of the system are listed respectively in the following description:

**Alice $\mathcal{A}$**

- If Alice wants to send a message, it takes a timestamp when creating the message. Then it continuously listening to its network, waiting for incoming Couriers.

- Once a Courier connects to Alice, Alice immediately submits its message the Courier and explicitly specifies a message recipient.

- When Alice is submitting a message, she can stop the submitting process at any time.
- After Alice submits a message to a Courier, she can deny sending the message to the Courier at any time.
- If the message is not successfully sent, Alice waits for next Courier to send this message.
- Alice is allowed to submit any message arbitrary times, to as many Couriers as she wants. Alice herself does not care who carries her messages, nor how many Couriers carry her message.

**Bob $\mathcal{B}$**

- Bob must be continuously listening to its network, waiting for incoming Couriers.
- Once a Courier connects to Bob, Bob downloads all messages from the Courier. Then Bob checks the messages received. If any message is invalid, Bob simply discards it.
- Bob will discard all duplicated messages. Duplicated message are defined as messages whose creator IDs and timestamps are exactly the same. Because timestamp is created at the time that the message is created, sharing same IDs and timestamps reflects those messages are created by the same entity at the exact same time.
- When Bob is downloading a message, he can stop the downloading process at any time.
- After Bob receives a message from a Courier, he can deny receiving the message from the Courier at any time.

**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 controlled by an intelligent agent (such like human) who always knows where other entities located.

- Courier chooses some of the potential message creators to deliver messages for them. The specific choosing process is not defined in the system model. It can be regarded as (1) Courier routinely check all devices in the system, see if they have messages to send, or (2) The intelligent agent who controlled the Courier "magically" knows which device has message to send.

- Courier transports to every chosen message creators one by one and get their messages if they have.

- After collect messages from all chosen message creators, it crosses the border and transports to every message recipients that are specified by those message creators.

- Once Courier transports to a message recipient, it transmits all corresponding messages (which is designated to be delivered to this recipient) to it.

- If any transmission failed, Courier tries to retransmit, until it succeeds or Courier decides to abandon the transmission.

- Once a message has been successfully transmitted to its recipient, it is erased from the storage of Courier and will not make any effects in future.

## 3.2 Adversary Model

### 3.2.1 Adversary Capability

The adversary model in this system is mostly derived from Dolev-Yao model which implies "adversary carries the message" in the network [6]. Moreover, the adversary can also do something particular in this system. Specifically, an adversary $\mathcal{Z}$ has following capabilities:

- It supervises the whole network system, which means it knows when, where and how entities are communicating, and it knows what information is held by each entity in the system.

- It can access/rewrite any message passing through the network.

- It is a legitimate user of the system.

- It can access to all the Courier's data at any time.

- It will always have opportunity to be contacted by any other honest entities.

### 3.2.2 Adversary Goal

The goal of the adversary $\mathcal{Z}$ is various, it depends on what role it plays in a certain communication.

- If $\mathcal{Z}$ is **not involved** in a particular communication, its goal is to:
    1. reveal the message content exchanged in the communication
    2. modify the message sent by message creator and make it accepted by the message recipient
    3. impersonate the message creator or message recipient
    4. prove the identities of message creator and message recipient to any third parties.
- If $\mathcal{Z}$ acts as **courier** in a communication, its goal is to:
    1. reveal the message content exchanged in the communication
    2. modify the message sent by message creator and make it accepted by the message recipient
    3. prove the identities of the message creator and message recipient to third parties
    4. forge a message and make it accepted by any message recipient
- If $\mathcal{Z}$ acts as **message recipient** in a communication, its goal is to prove the authenticity of the message content to any third parties.

It should be noted that this network system is vulnerable to DoS attacks - same as any other existing network systems. It means $\mathcal{Z}$ can always prevent a message from being successfully delivered. However, it is not listed in the adversary goal, thus it will not be covered in the system security analysis.

# Chapter 4

# Protocol Specification

The protocol introduced for the system is called Courier-Dependent Security Protocol, it defines the interactions between Alice (message creator), Bob(message recipient) and Courier in the system described above when they engage communication. A successful run of CDSP 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, it defines how message is submitted from Alice to Courier, while the Transmit Protocol runs between Bob and Courier, it defines how message is transmitted from Courier to Bob. 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.

**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 $\mathcal{SIGN}_E$ :** It denotes an abstract digital signature function, and the subscript E indicates the entity who creates this signature using a secret key $\text{sk}_A$. And it can only be verified under the corresponding public key $\text{pk}_B$.

**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 "$\text{ID}^+$" equals a sequence of IDs concatenated together, which can be "$\text{ID}_0$" or "$\text{ID}_0||\text{ID}_1$" or "$\text{ID}_0||\text{ID}_1||$..." where it contains at least one ID and the content of IDs can be different.

## 4.2 Submit Protocol

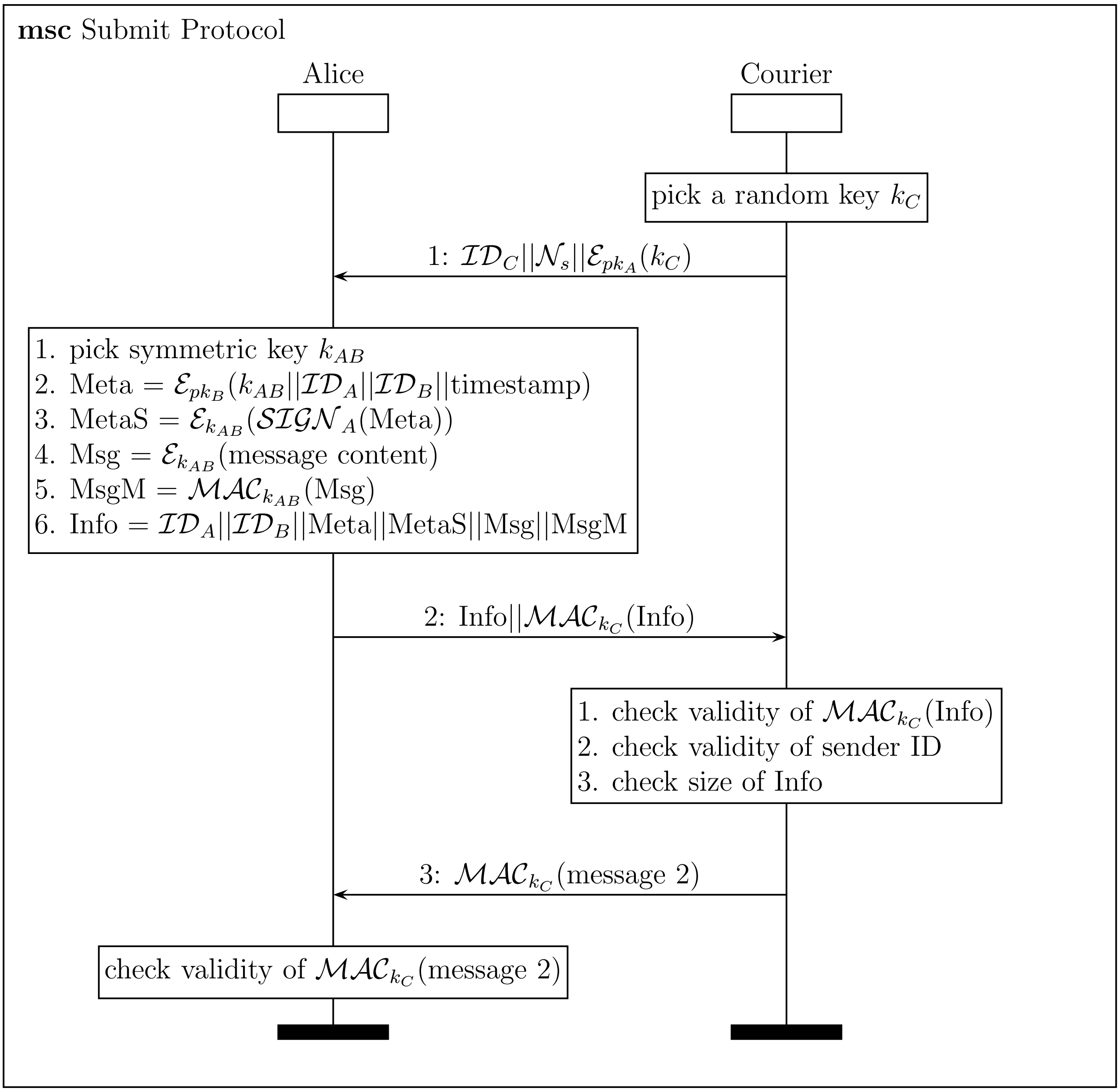
### 4.2.1 Preconditions

- 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 uses a symmetric key generator to create a random symmetric key $k_C$ which will be sent to Alice and used for deniable authentication later. The key $k_C$ must fit the symmetric encryption / decryption scheme which will be used in later of the protocol. Then Courier actively connects to Alice and sends first message to Alice. The first message should contain (1) The ID of Courier $\mathcal{ID}_C$, (2) The maximum storage size of Courier $\mathcal{N}_s$, (3) Symmetric key $k_C$

encrypted by Alice's public key, which is $\mathcal{E}_{pk_A}(k_C)$.

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 so that it is assumed only Bob can access the content of Meta block. The content of Meta block contains (1) Symmetric session key $k_{AB}$, (2) The ID of message creator (Alice) $\mathcal{ID}_A$, (3) The ID of message recipient (Bob) $\mathcal{ID}_B$, (4) A timestamp indicates the exact time when Alice creates this message. To prove the authenticity of the Meta block, Alice creates a digital signature of Meta block using her secret key $\mathrm{sk}_A$. However sending raw signature of Meta block release the authenticity of its origin to the Courier, in which case Courier will be able to prove the fact that the Meta block comes from Alice. Thus, to hide the authenticity of the origin of the Meta block, the signature is contained in MetaS block where it is encrypted with $k_{AB}$ so that only Bob can reveal it. The reason of encrypting signature under symmetric session key rather than Bob's public key is for efficiency concern.

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

After Alice gets above 4 blocks, it creates a Info block which reflects "all information" for Courier. Info block is the concatenation of (1) The ID of message creator (Alice) $\mathcal{ID}_A$, (2) The ID of message recipient (Bob) $\mathcal{ID}_B$, (3) Meta block, (4) MetaS block, (5) Msg block, (6) MsgM block. Then Alice appends a MAC of the Info 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, and it proves the identity of Alice to the Courier. Finally Alice examines the size of Info block. If the total size of Meta, MetaS, Msg, MsgM blocks exceeds the storage limitation of Courier $\mathcal{N}_s$, 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 Info block and its MAC, to Courier.

Upon receipt of the message 2 from Alice, Courier checks the validity of message 2. It first verifies $\mathcal{MAC}_{k_C}(\text{Info})$, if it fails it means either the Info block has been modified or the message sender used a wrong key, both leads to Courier abort the protocol. If it verifies true then Courier checks the ID of message creator to see if it is indeed the entity it is going to communicate (here whether it is $\mathcal{ID}_A$). After that, it checks the total size of the Meta, MetaS, Msg and MsgM blocks, making sure it does not exceeds the its 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 a confirmation message.

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, and they are not restricted by this protocol.

## 4.3 Transmit Protocol

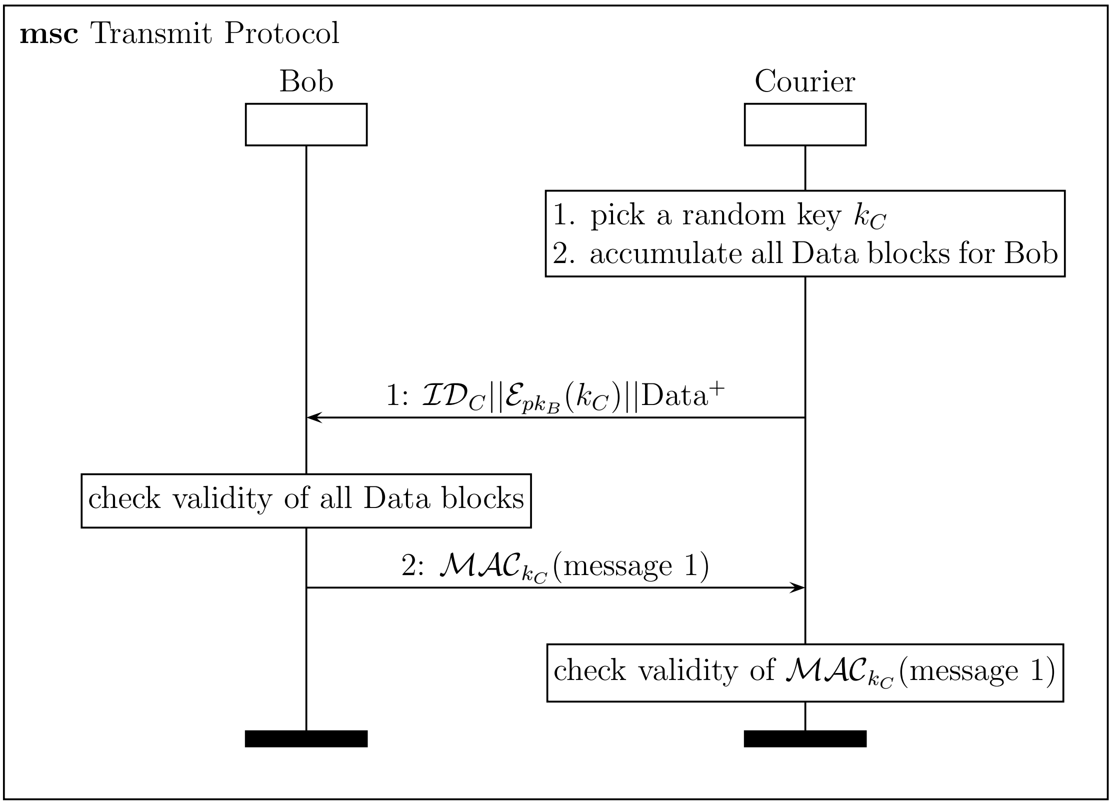
### 4.3.1 Preconditions

- 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:

Data = Meta||MetaS||Msg||MsgM

Meta = $\mathcal{E}_{pk_B}(k_{AB} || \mathcal{ID}_A || \mathcal{ID}_B ||$timestamp$)$

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

Msg = $\mathcal{E}_{k_{AB}}$(message content)
MsgM = $\mathcal{MAC}_{k_{AB}}$(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 prepares the data for Bob. Assuming by the time Courier starts the Transmit Protocol with Bob, it already has certain pieces of data for Bob which are collected earlier from various message creators. We define a Data block as essential data that is submitted by a message creator and need to be transmitted to a message recipient. Each Data block contains (1) Meta block, (2) MetaS block, (3) Msg block, (4) MsgM 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. Specifically, the message 1 contains (1) The ID of Courier $\mathcal{ID}_C$, (2) Symmetric key $k_C$ encrypted by Bob's public key, which is $\mathcal{E}_{pk_B}(k_C)$, (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 chop the Data$^+$ block up and check each Data block one by one. It will accept all Data blocks that pass all verifications while discard all Data blocks that fail any of the verifications. The detailed algorithm used for checking all Data blocks is displayed below in the algorithm "check all Data blocks".

According to the checking algorithm, there are 4 different checks for each Data block: First Bob checks the recipient ID contained in its Meta block, to see if this message is for Bob himself. Then Bob checks the timestamp contained in the message Meta block to ensure the message is not expired. The detail implementation of such timestamp checking is not defined in this protocol, it can be implemented in various ways by different applications. After checking the timestamp, Bob checks the digital signature of the whole Meta block of the Data block, by verifying it using the public key of the message creator - whose ID has been given in the Meta block. If the signature verifies true, Bob knows for sure that this message is originated from the message creator it indicates (here Alice). Finally, Bob checks the MAC of Msg block by verifying the MsgM block under the message creator's public key. If the result is true, Bob is convinced that the message content is created by the message creator (Alice).

**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 Data block unchecked* **do**
    currentDataBlock = nextDataBlock;
    use $sk_A$ to decrypt Meta block in currentDataBlock, reveal $\{k_{AB}$, creatorID, recipientID, timestamp$\}$;
    **if** *recipientID != "Bob"* **then**
        discard currentDataBlock
    **else**
        **if** *timestamp expires* **then**
            discard currentDataBlock
        **else**
            use $k_{AB}$ to decrypt MetaS in currentDataBlock, reveal $\mathcal{SIGN}_A$(Meta);
            verify $\mathcal{SIGN}_A$(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;
                    accepts the message content

After all Data blocks have been received and checked, Bob send back a MAC of whole message 1 received from Courier as message 2. The key used for creating the MAC is $k_C$ which is revealed by decrypting $\mathcal{E}_{pk_B}(k_C)$ in message 1. The message 2 is the confirmation of Bob receiving the 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, and they are not restricted by this protocol.

# Chapter 5

# Security Analysis

As stated above, CDSP is designed to ensure the secure communication in a courier-dependent scenario. In this chapter, all the security properties that CDSP claims to hold will be listed and proved. In the former part of this section, some primitives for the proving is defined beforehand, then in latter part, all security properties are described and presented as theorems which are provable.

## 5.1 Primitives

As CDSP uses many primary cryptographic operations - such like encrypting/decrypting, MACing and signing, its security properties also rely on them. Therefore, 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 later can be deduced to those primitives.

- Message Authentication Code $\mathcal{MAC}$
  The MAC [15] 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. [10][16]

- 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%. [26]
  (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 Security Proof

In this section all security properties of CDSP are presented as theorems to be proved. For each property, first a brief description will be given to show why this property is essential in the protocol, then authenticity of it will be proved. Before each proof, some primitives are assumed to be true (listed above), and after each proof, it will be clear that once the primitives are true, then the security property is held be CDSP.

### 5.2.1 Submit protocol

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

*Description:* As it has been assumed that physical transportation for a Courier is extremely costly, Courier should not carry messages for any arbitrary entity. Thus it is required that, when Courier chooses to carry a message for certain entity, the entity must prove its identity to the Courier before submitting the message.

*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 $\mathcal{CR}$. $\square$

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

*Description:* It is expected that the message is not modified and it is not forged after being submitted to Courier.

*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. $\square$

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

*Description:* It is an auxiliary theorem that helps to prove the THEOREM 3.

*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 $\mathcal{A}$. $\square$

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

*Description:* 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.

*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 $\mathcal{CR}$

*Description:* In certain scenarios which require high privacy protection - like exchanging military intelligence or voting system, 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 the fact that it ever sends a message to Courier after the message is submitted.

*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 $\mathcal{CR}$. □

### 5.2.2 Transmit Protocol

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

*Description:* 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 before receiving data from Courier.

*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 $\mathcal{MAC}_{k_C}$(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

*Description:* It is expected that the message is not modified and it is not forged after being transmitted to $\mathcal{B}$.

*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 $\mathcal{MAC}_{k_C}$(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 $\mathcal{CS}$

*Description:* 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.

*Proof.* Similar to the proof of THEOREM 4 in Submit Protocol, 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 $\mathcal{MAC}_{k_C}$(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 $\mathcal{CS}$. □

### 5.2.3 On The Whole

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

*Description:* As 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.

*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 (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

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

*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 if preserved. □

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

*Description:* It is expected that the message is not modified and it is not forged during the transportation.

*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 (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 $\mathcal{B}$

*Description:* 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 $\mathcal{A}$.

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

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

*Description:* 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.

*Proof.* The message content contains two parts - (1) encrypted message Msg = $\mathcal{E}_{k_{AB}}$(message) and (2) its MAC $\mathcal{MAC}_{k_{AB}}$(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 $k_{AB}$, so $\mathcal{B}$ can create any Msg it wants.

2. $\mathcal{MAC}_{k_{AB}}$(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 $\mathcal{B}$. $\square$

# Chapter 6

# Protocol Implementation

To prove the practicality of CDSP and also to evaluate its performance in real world, an implementation of CDSP is developed in Java. (The program is attached with this thesis) In this chapter, the implementation of CDSP will be fully described. As it will be absolutely redundant to dive into every implementation detail of the program, this chapter will only emphasize on the major decisions. And it will be organized as follows: First an overview of the application will be given, roughly demonstrates what the application achieves and how it can be used. Then a high level design will be illustrated, followed by the detail of its implementations.

## 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. **CourierReciever**, 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.

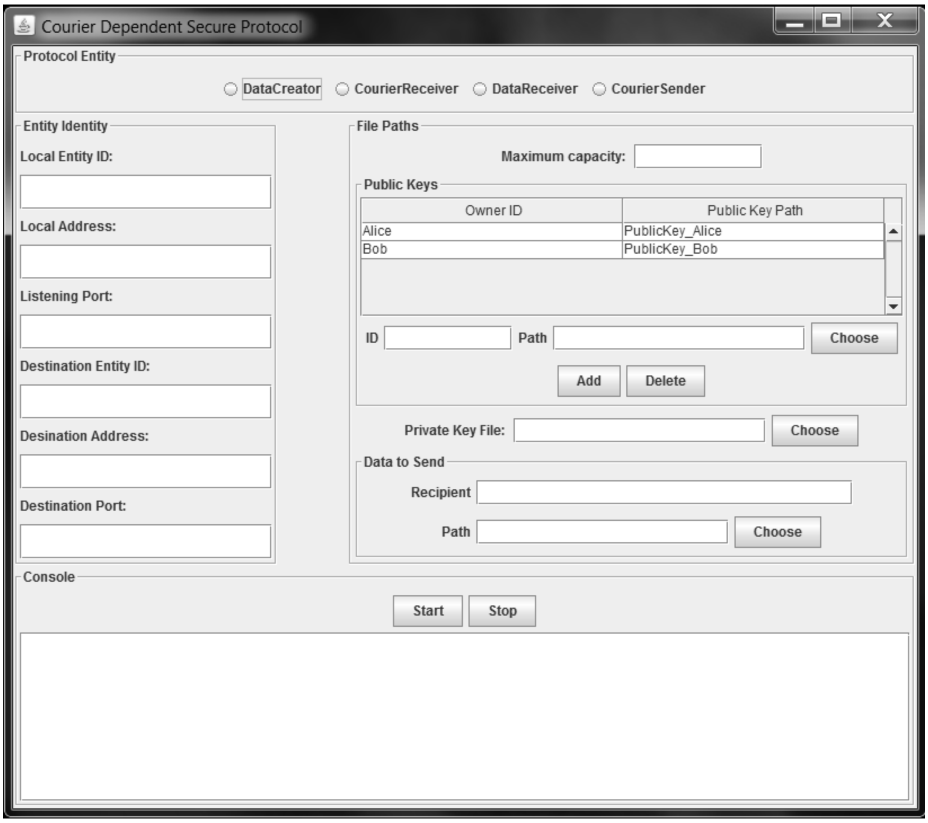



Figure 6.1: GUI Overview

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 start Transmit Protocol with Courier. The detailed instruction of how to run those two sub-protocols is given in the appendix.

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.

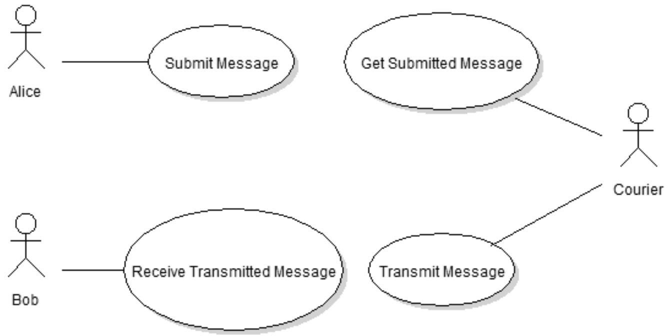



Figure 6.2: Use Case Diagram

## 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 | User Goal |
| 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 | 1. Alice starts the application.<br>2. Alice selects "DataCreator" as the protocol entity in the application.<br>3. Alice specifies all related information as instructed in the "Run Submit Protocol" section in Appendix A: User Manual.<br>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 | User Goal |
| Stakeholders | Courier and Alice |
| Success Guarantees | Courier gets a message from Alice |
| Trigger | Courier starts the Submit Protocol |
| Main Success Scenario | 1. Courier starts the application.<br>2. Courier selects "CourierReceiver" as the protocol entity in the application.<br>3. Courier specifies all related information as instructed in the "Run Submit Protocol" section in Appendix A: User Manual.<br>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 | User Goal |
| Stakeholders | Bob and Courier |
| Success Guarantees | Bob receives the message |
| Trigger | Bob starts the Transmit Protocol |
| Main Success Scenario | 1. Bob starts the application.<br>2. Bob selects "DataReceiver" as the protocol entity in the application.<br>3. Bob specifies all related information as instructed in the "Run Transmit Protocol" section in Appendix A: User Manual.<br>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 | User Goal |
| 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 | 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 in Appendix A: User Manual. 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) Accepter, who continuously listens to the port and waits for incoming connection. (2) Initiator, who actively connects to a Accepter. Although these two categories sounds similar to Client-Server pattern, it is not quite the case because Accepter does not actually provide any service. Based on their behaviour definition in protocol specification, DataCreator and DataReceiver belong to Accepter, while CourierSender and CourierReceiver belong to Initiator. Following paragraphs will introduce the internal architecture for Accepter and Initiator respectively.

**Accepter** Accepter 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 Accepter. 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 Accepter. 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 Accepter. 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 6.3 "Accepter Architecture".

**Initiator** Similar to Accepter, Initiator consists of 4 main parts: (1) Session, (2) User Interface, (3) Cryptographic Kit and (4) Data Manager. The major difference between Initiator and Accepter is that Initiator does not have a Dispatch Thread, every Initiator only has one Session. The Session is created once the Initiator is 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 Accepter. The architecture design of Initiator is a simpler version of Accepter's, and it is illustrated in the figure 6.4 "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 differently to a same input. So when an input message comes, Delegate will process

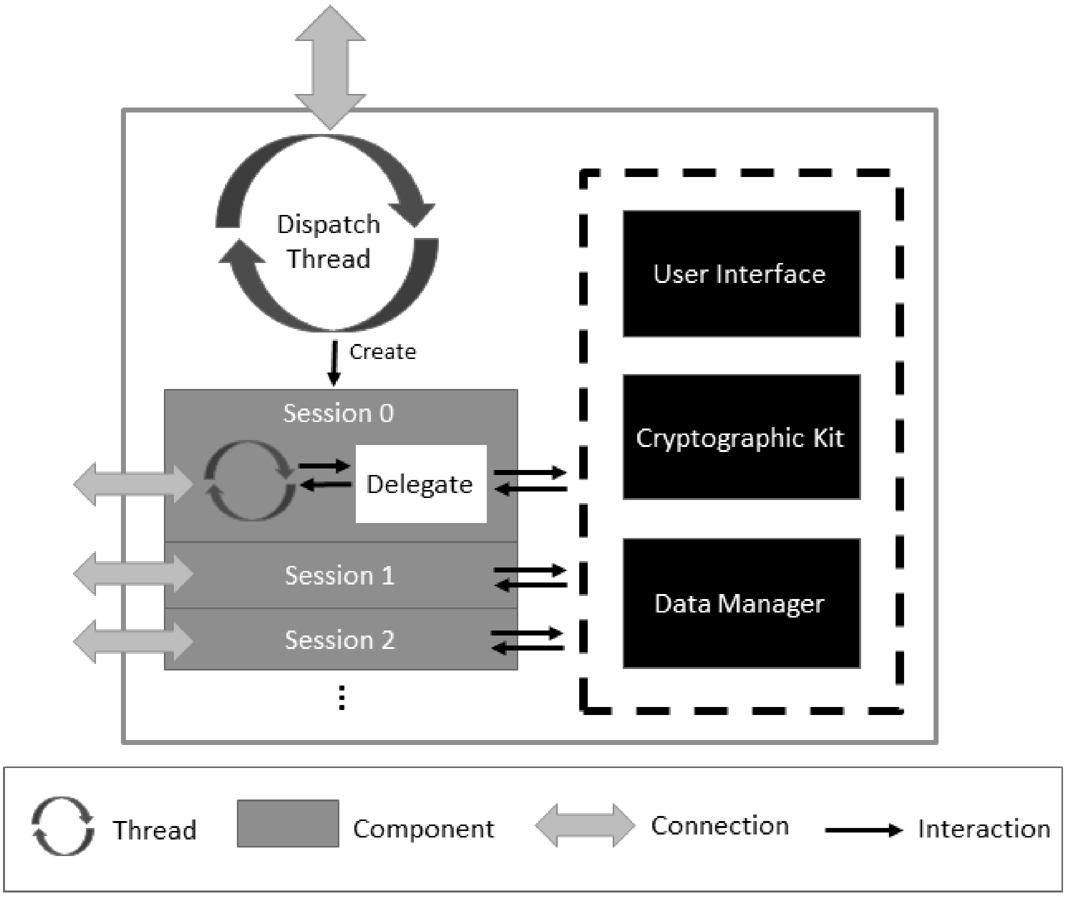



Figure 6.3: Accepter Architecture

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. The delegates for Bob and Courier will follow the same pattern and they will not be fully described here.

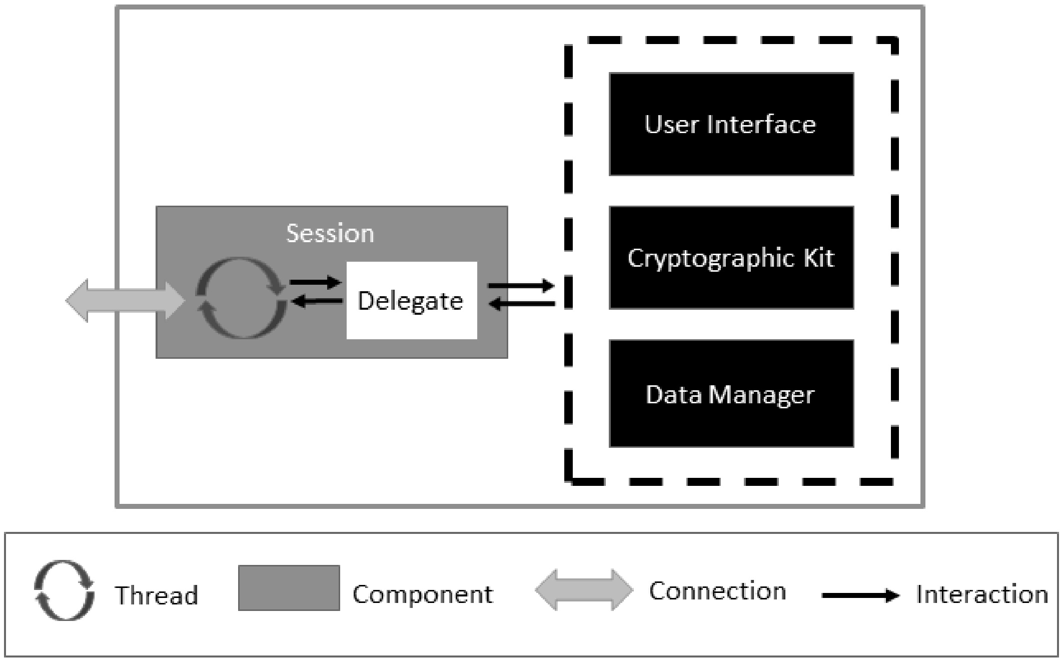



Figure 6.4: Initiator Architecture

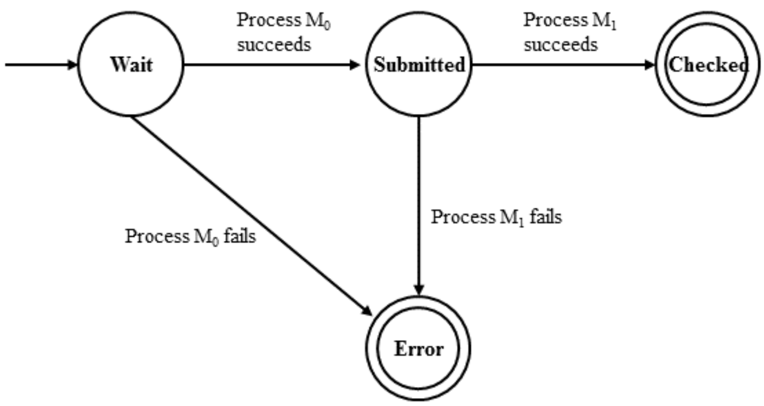



Figure 6.5: State Machine of $\mathcal{A}$

### 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 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).

- **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

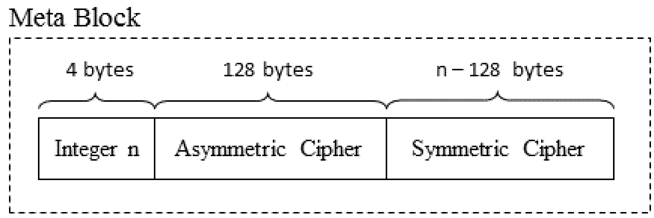



Figure 6.6: Structure of Meta Block

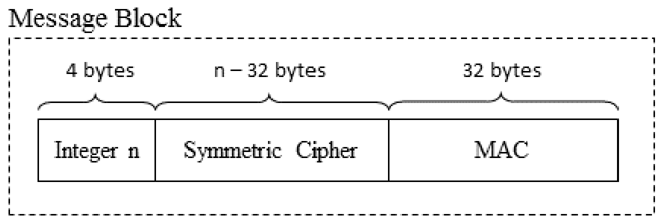



Figure 6.7: Structure of Message Block

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.

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 | $\text{Meta}_0$ | $\text{Message}_0$ | $\text{Meta}_1$ | $\text{Message}_1$ | ...... |

*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.

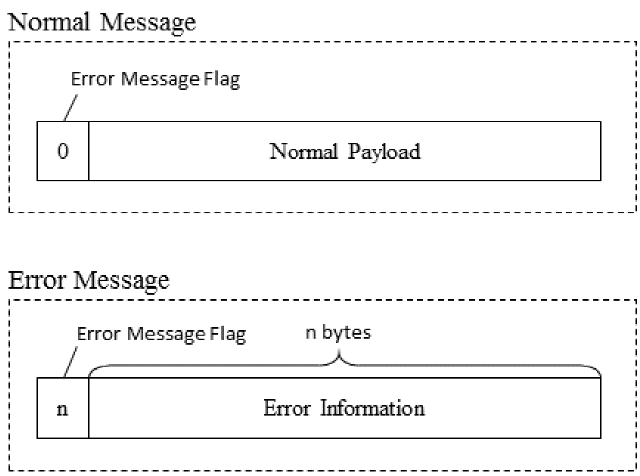



Figure 6.8: Normal Message and Error Message

## 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.

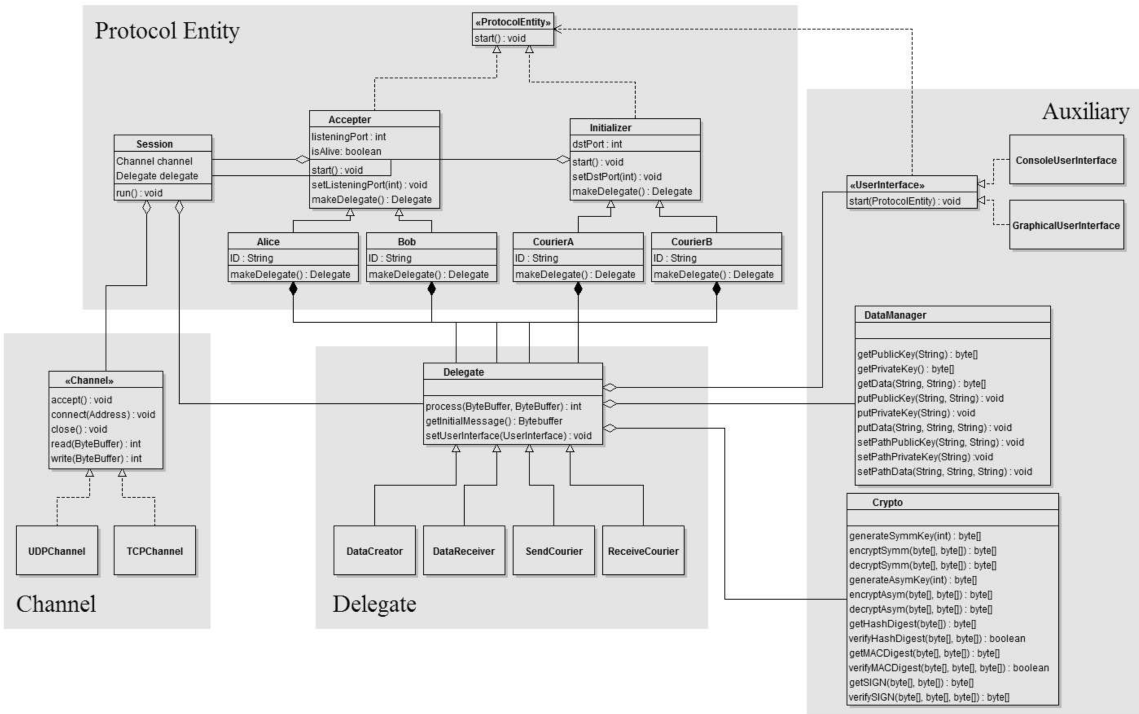



Figure 6.9: UML Diagram

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 Accepter 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 CDSP, 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 Accepter 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 actually connected to any Channel - instead it creates a new one to handle the connection, 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 Accepter 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 Accepter 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 fron Accepter, 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 Accepter's newly created channel. Finally the connection is built between Initiator's channel and Accepter'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 operations used in the implementation is carefully chosen so that theoretically, those operations meet the requirements of security primitives described in Section 5.1.

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 [17]. The mode of operation used is CBC [8]. The padding scheme is PKCS#5 [13]. 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. Th padding scheme is PKCS#1 [12]. 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 [1]. 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 in SHA256 [1]. They 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 inputing 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

In this chapter, several performance tests will be done to the program developed for running CDSP. As there is no any other existing program developed for CDSP, the test results will not be compared to any others. The main purpose of these performance tests is to show the efficiency of the current implementation of CDSP so that it can be taken as a point of reference for future development.

The rest of the chapter will first describe the system environment that is used for the testing. Then tests results will be given for both Submit Protocol and Transmit Protocol separately. For each sub-protocol, two main kinds of test are done - latency test and scalability test. Finally a summary will conclude the testing result for both Submit and Transmit Protocol.

## 7.1 Test Environment

Following is a snapshot of the system environment used for the testing:

| | |
|---|---|
| Processor | Intel® Core™ i7-3610QM CPU @ 2.30GHz × 4 |
| Installed Memory | 8 GB |
| OS Platform | Ubuntu 12.04 LTS |
| OS Type | 64-bit |
| Network Interface | Loopback |
| JRE Version | 1.7.0_07 - b11 |

To eliminate the influence of various UI implementations, a UI called "NullUserInterface" is used in the testing. In "NullUserInterface", all parameters required from user input are all hard-coded thus it can be run automatically by only using commands. On the other hand, "NullUserInterface" does not produce any output for user, it only outputs the testing results to the terminal so no extra computational resource will be used in dealing with outputs during the test.

## 7.2 Submit Protocol

### 7.2.1 Latency Test

The latency here refers to the total time cost for successfully running Submit Protocol once. It is defined as the time cost from the point that Courier initiates a connection to the point that Courier sends out the message 3 (a MAC as receipt confirmation). There are two reasons for defining the end point as Courier sending message 3, but not Alice receives message 3 : (1) It is very difficult to coordinate different times of two entities, especially in millisecond scale. (2) After Courier sends out message 3, it will leave immediately without caring whether the message will be received or not, and no further action of Courier depends on the arrival of message 3. Thus it is reasonable to define the sending of message 3 as the end of the Submit Protocol.

The test programs contains running of two entities:

1. An Alice continuously listens to the port and is ready to start the Submit Protocol.

2. A Courier initiates the Submit Protocol, and once it sends out a message 3, it immediately creates another Courier and initiates the protocol with Alice again. This process repeats for 1000 times.

The system records the time cost from the beginning of the program to Courier running the 100th, 200th, 300th,... 1000th times of the protocol respectively. The message content Alice sends is a 18 bytes string.

The test is done for totally 10 times and the final result is the average number of each testing result. Figure 7.1 shows the final result. Through the chart we can see the latency grows linearly with the number of communications - which is expected. And figure 7.2 shows the average latency for every protocol run. The chart indicates that when the number of consecutive communication grows large, the average latency for each communication becomes stabilized around 5.5 milliseconds. In practice, there might be hundreds of entities within the system, a single message creator may send thousands of messages in a short period of time, and above result just proved that a single message creator is able to sequentially complete 1000 of message sending tasks around 5.5 seconds. This outcome is satisfying.

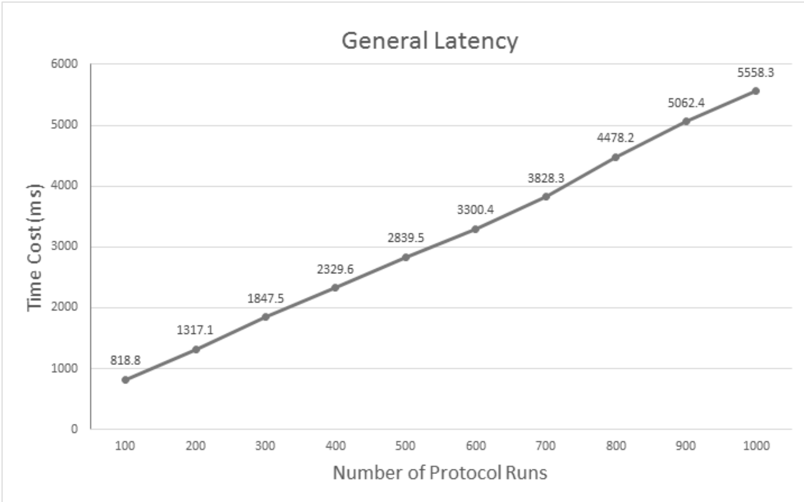



Figure 7.1: General Latency Test of Submit Protocol

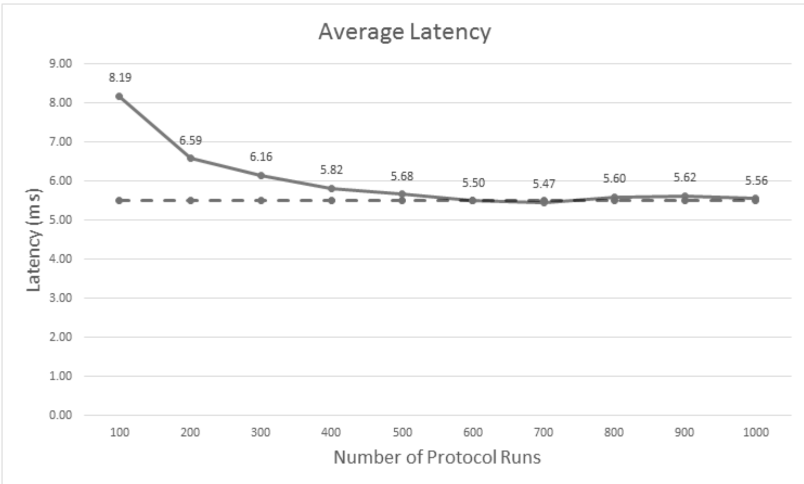



Figure 7.2: Average Latency of Submit Protocol

### 7.2.2 Scalability Test

To thoroughly show the scalability of Submit Protocol, the performance of the protocol will be measured under two conditions - the growth of message size and the growth of communicating entities. Their methodologies and analysis results will be given separately below.

#### 7.2.2.1 Scale of Message Size

In this test, it is expected to reveal how Submit Protocol will behave when the size of message to be sent increases.

The programs used are same with above latency tests - a normal message creator and a courier who repeatedly initiates Submit Protocol with the message creator.

The system records the latency of 1000th successful run of Submit Protocol with different sizes of messages. The message sizes are various from 1 byte to 10 KB. To make the test more convincing, we collect 5 data samples for every message size, and the result is plotted in a scatter chart figure 7.3. The chart shows the total latencies for submitting 1000 times 1 Byte, 10 Bytes, 100 Bytes, 1 KB, 2 KB, 3KB, 4KB, 5KB and 10 KB messages respectively. It is shown that compare to the dramatical increasing extent of message size, the rise of latency is quite subtle - from around 5500 milliseconds to around 6200 milliseconds while message size is 10000 fold! Furthermore, following the trend line in the chart, it can also be observed that the grow trend of latency is linear which is ideal for the protocol.

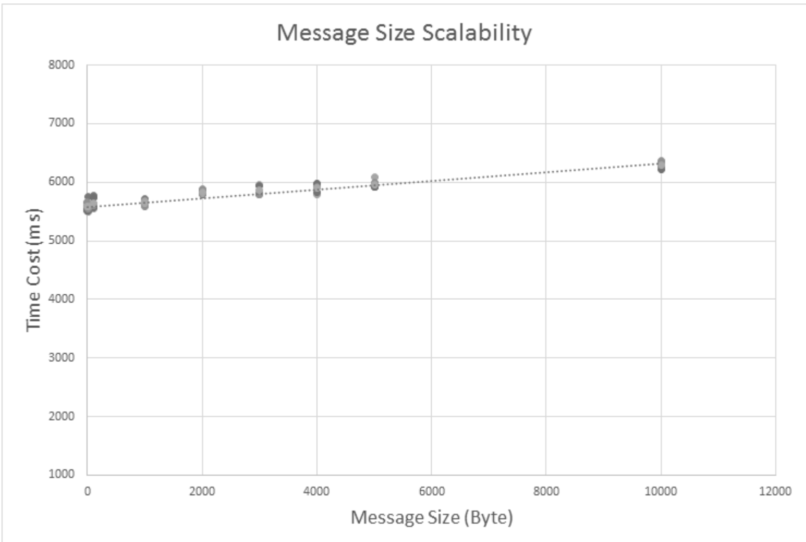



Figure 7.3: Latency with Message Size for Submit Protocol

Using the same data sample, we can also calculate the relationship between the throughput and message size. We define the notion "effective throughput" as the size of actual message content being sent per second. Namely, it does not count those extra bits which are the overhead for delivering the message. It represents the efficiency of the protocol in terms of delivering messages contents. Figure 7.4 illustrates how the effective throughput of Submit Protocol changed when the message size increased. As the chart shows, the correlation between effective throughput and message size is linear, which is a good news because it means that the larger the messages are the more effective the protocol is. And when the message size increased to 10 KB, the effective throughput reaches nearly 1.6 MB/s. Of course, the effective throughput will not keep increasing infinitely, it will eventually hit the ceiling and restricted by the network throughput and capability of devices' available memory, however as 10 KB size is already considered very large for a single message, the efficiency at this level is good enough.

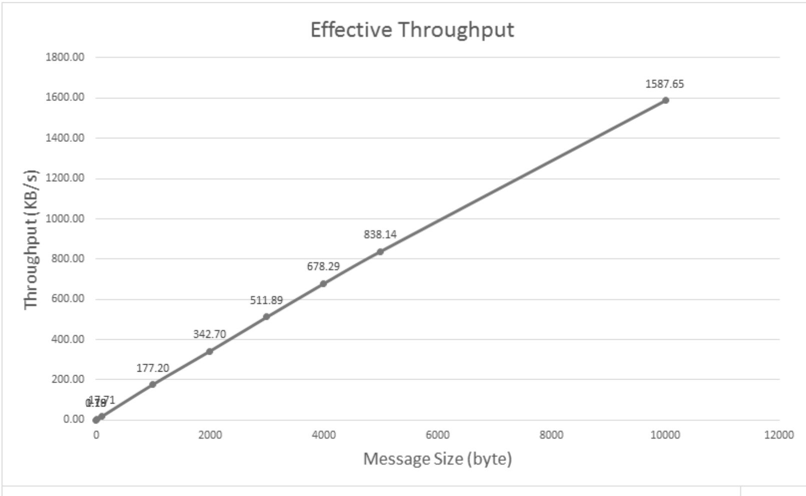



Figure 7.4: Effective Throughput with Message Size for Submit Protocol

#### 7.2.2.2 Scale of Concurrent Communication

In practice, it is highly probable that a single message creator will submit messages to several couriers simultaneously, this test reflects this scenario and aims to show the behaviour of Submit Protocol under the situation where concurrent communication is involved.

The testing program contains 3 entities:

1. An Alice continuously listens to the port and is ready to start the Submit Protocol.

2. A noisy Courier initiates the Submit Protocol with Alice, and once it sends out a message 3, it immediately creates another noisy Courier and initiates with Alice again. This process repeats for 100 times. It records the time cost for running 100 times of the protocol and yield the result.

3. Several silent Couriers initiate the Submit Protocol with Alice along with the noisy Courier. Same with noisy Courier, every silent Courier repeatedly initiates with Alice for 100 times, however, they do not record the time cost.

Above settings is designed to extract the latency of a single communication when several communications are running simultaneously. During the test, the message size is 18 bytes and the number of active couriers are 1, 2, 3, 4, 5, 10, 20 respectively. Because such concurrent performance highly relies on the implementation of platform's scheduling mechanism, some extreme result may appear during the testing. Thus, to make the result more persuasive, every test is done for 7 times and the highest and lowest results are eliminated. The rest of results are plotted in scatter chart figure 7.5. The trend line of the result appears to be polynomial in the order of 2. The reason that it is not linear is mainly because as the number of communication increases, the computational overhead for scheduling and switching processes also increases, so more concurrent communications it has, the more extra computation resource will be occupied.

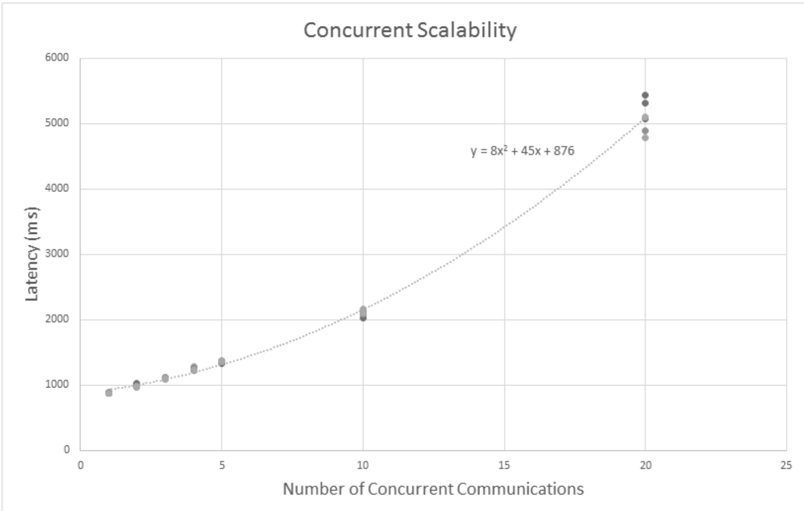



Figure 7.5: Latency with Number of Concurrent Communications for Submit Protocol

Despite the polynomial growing trend, the protocol is still proved to be efficient under relatively small scale (less than 20 communications). The result shows that even when the message creator is communicating with 20 different couriers at the same time, the latency for running 100 times of protocol is still around 5 seconds (50 milliseconds for a single protocol run), considering the average latency for a single protocol run is 5.5 milliseconds (tested above), this result is fairly tolerable.

## 7.3 Transmit Protocol

### 7.3.1 Latency Test

The definition of latency for Transmit Protocol is slightly different from Submit Protocol's. Referring to the protocol specification, Transmit Protocol requires only 2 messages, one forwarded from Courier to Bob, one replied from Bob to Courier. Here the latency is defined as the total time cost from the point that Courier initiates a connection, to the point that Courier receives the message 2 and finishes checking its validity.

Follow the same program settings of Submit Protocol, the latency tests for Transmit Protocol records the time cost for running 100, 200, 300, ..., 1000 times of the protocol. The message size is still 18 bytes.

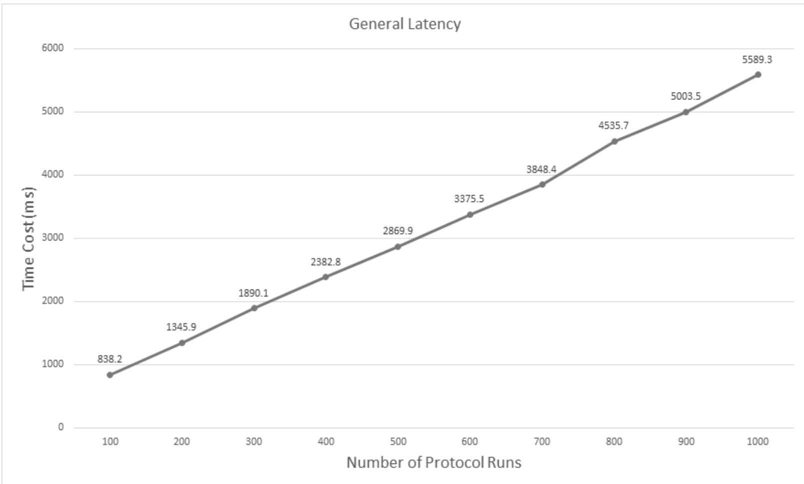



Figure 7.6: General Latency Test of Transmit Protocol

Figure 7.6 shows the testing result. As expected, the outcome is quite similar to Submit Protocol, which appears to be linearly increasing. And even the average latency for each run of Transmit Protocol is same to Submit Protocol, which is given in figure 7.7. It can be observed that after doing Transmit Protocol many times, finally the average latency stabilized around 5.5 milliseconds. It proves the protocol is as efficient as Submit Protocol.

### 7.3.2 Scalability Test

In the scalability test of Transmit Protocol, besides testing under different scale of message size and number of concurrent communications, a new parameter is to be

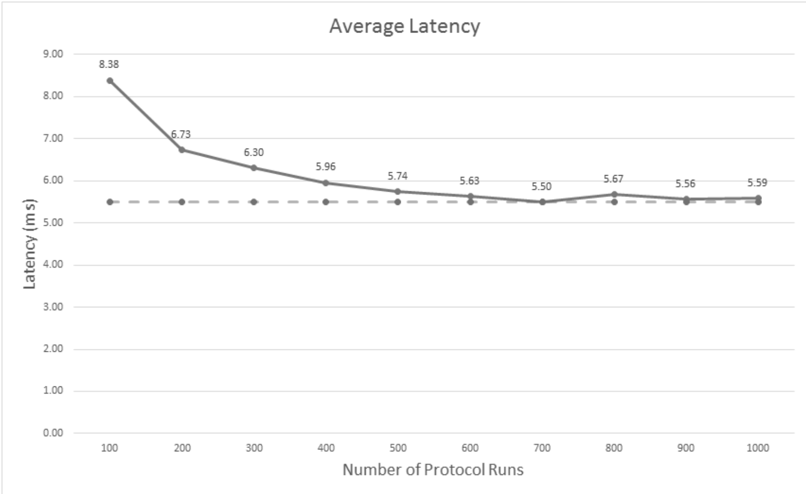



Figure 7.7: Average Latency of Transmit Protocol

tested - number of messages. Their methodologies and analysis results will be given separately below.

#### 7.3.2.1 Scale of Message Size

The methods and program settings used in this test is exactly the same with Submit Protocol, which is, a courier sends a message to Bob for 1000 times with different message size. The message sizes are 1 Byte, 10 Bytes, 100 Bytes, 1 KB, 2 KB, 3 KB, 4 KB, 5 KB and 10 KB respectively. The results are plotted in figure 7.8, which indicates that the latency increase is fairly little comparing to the increase of message size. And it can be observed that the performance of Transmit Protocol when transmitting large messages is slightly better than Submit Protocol, it only takes 6 seconds to transmit a 10 KB message 1000 times, while it takes 6.2 seconds to submit it.

The relationship between message size and effective throughput of Transmit Protocol is illustrated in figure 7.9. Similar to the Submit Protocol, the effective throughput has linear positive correlation with the message size. It means the larger the size of single message is, the more efficient of the protocol will be. And by observing figure 7.4 and figure 7.9, we can conclude that overall, the efficiency of Transmit Protocol is slightly better than Submit Protocol because its latency is always less than Submit Protocol when they are tested under the same condition.

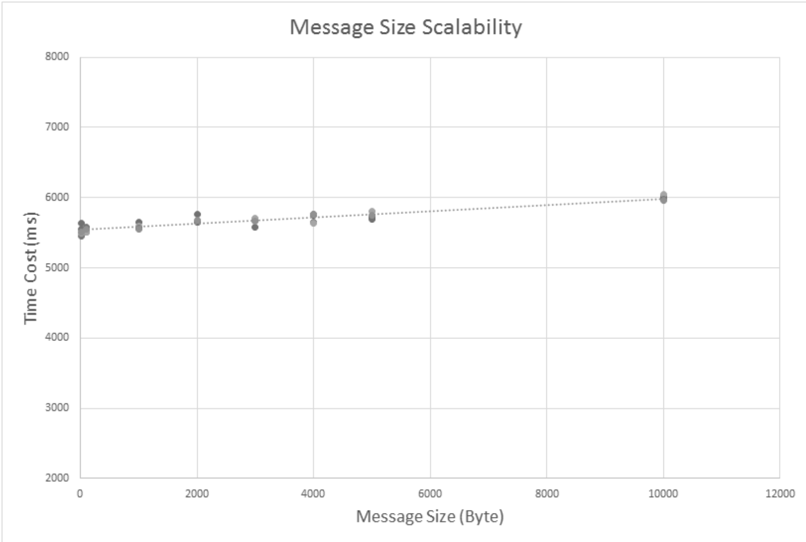



Figure 7.8: Latency with Message Size for Transmit Protocol

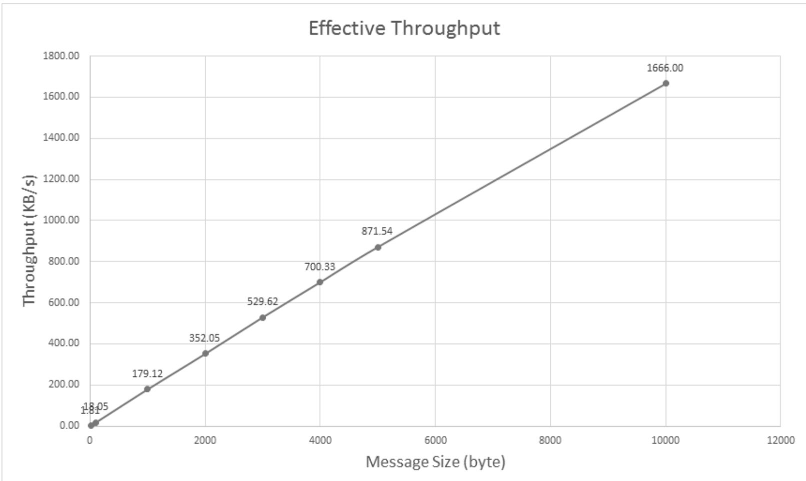



Figure 7.9: Effective Throughput with Message Size for Transmit Protocol

#### 7.3.2.2 Scale of Concurrent Communication

Same concurrency test is done to Transmit Protocol to reflect the practical scenario when a single message recipient (Bob) may receive messages from multiple couriers. The program settings are same to Submit Protocol: certain number of silent couriers keep communicating with Bob silently, while one noisy courier records the number of communication and yields the time cost.

The testing result is shown in the figure 7.10. The trend line of data samples is quite similar to the one of Submit Protocol, they both indicate a polynomial correlation. However, the growth rate of it in Transmit Protocol is much less than in Submit Protocol. We can find that when communicating with 20 couriers, Transmit Protocol takes only 3.2 seconds to complete 100 times of protocol, while Submit Protocol takes about 5 seconds. It means courier will be more efficient to transmit messages than to receive messages.

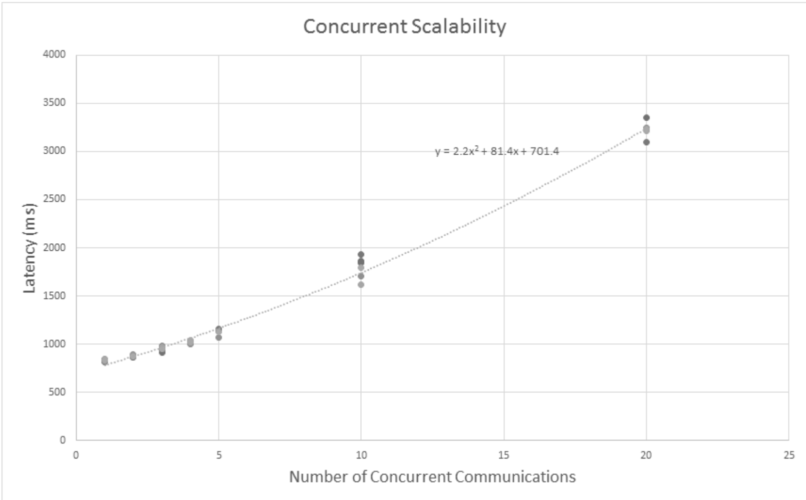



Figure 7.10: Latency with Number of Concurrent Communications for Transmit Protocol

#### 7.3.2.3 Scale of Message Number

This test simulates a common situation that a single Courier carries messages from several message creators and transmit it to the message recipient once for all. It is the most basic communication model in the system, so we are curious about how it performs in the actual implementation. Theoretically, the number of messages raises does not only lead to the increasing of message size, more extra checking processes are required as well. The test aims to find out how severe its influence will be.

This test measures the latency of total 100 Transmit Protocol runs with different number of messages transmitted. The number of messages are 1, 2, 3, 4, 5, 10, 20, and each message has size of 1 KB. Totally 5 tests are done and the average number of them is taken as the final outcome.

Figure 7.11 shows the linear correlation of between number of messages and latency. It indicates that to sequentially complete 100 Transmit Protocols for a courier who carries 20 messages, it takes about 4 seconds in total, which means 40 milliseconds for every completion of the protocol. It is a great cost increase comparing to 5.5 ms, which is the average latency per protocol run.

Furthermore, the figure 7.12 compares the efficiency difference between transmitting a large message as whole and chopping it into several pieces of smaller messages.

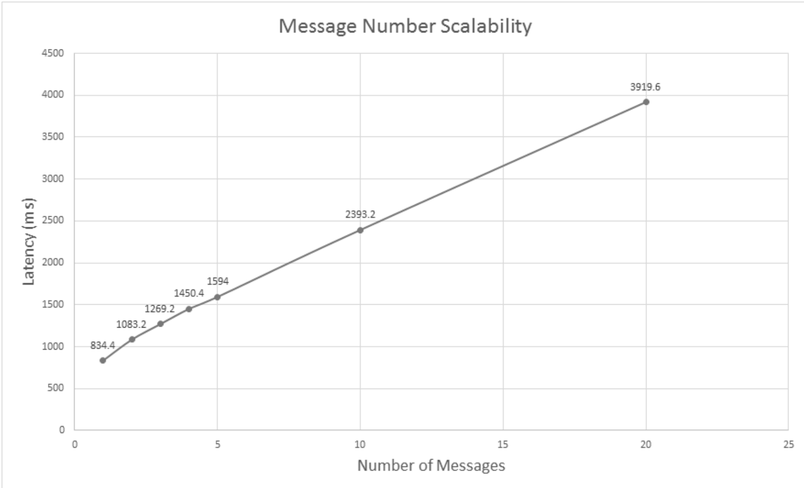



Figure 7.11: Latency with Number of Messages for Transmit Protocol

The comparison takes several large size messages, and (1) transmits it as a whole, (2) transmits it as several 1 KB size smaller messages, and show their effective throughput respectively. The result conveyed by the figure is absolutely clear: no matter what the total message size is, transmitting it as a large message has nearly twice the throughput than chopping it up.

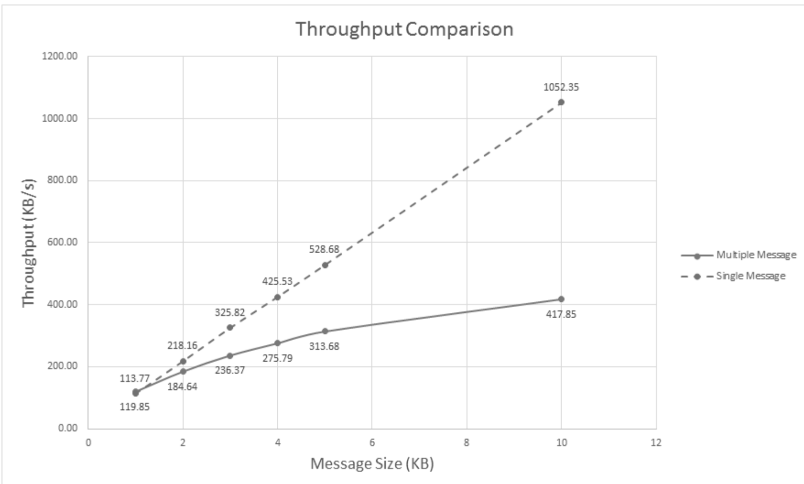



Figure 7.12: A Large Message or Several Small Messages?

Conclusions drawn from both figure 7.11 and figure 7.12 indicate that transmitting large number of messages in a single Transmit Protocol is very time consuming. Thus when Alice sending a large message to Bob, it is recommended to send it as a whole for the efficiency concern.

## 7.4 Summary

In this chapter, mainly two kinds of tests are applied to both Submit Protocol and Transmit Protocol respectively - latency test and scalability test. Latency test evaluates the time cost for completing a protocol under a relatively low workload, while scalability test reveals the behaviour of both protocols when the system scale dramatically grows. The testing results prove that both Submit and Transmit Protocol are very efficient, every single run of protocol in low-workload situation takes only 5.5 milliseconds for both of the protocols. And in terms of scalability, both sub-protocols behave surprisingly the same. Firstly they are both not sensitive to the increasing of message sizes, and their latencies have polynomial correlation with the number of concurrent communications. Besides, through the analysis of relation between latency and number of message sizes, it has been emphasized that sending a large message as whole is far better than sending it as several pieces of smaller messages.

# Chapter 8

# Conclusion and Future Work

## 8.1 Conclusion

The recently introduced Delay-Tolerant Network architecture assembles networks with different latencies, bandwith 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 security protocol specifically focused on its courier-dependent communication network. 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 tailored 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 running 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, the capability of the application is shown and some interesting conclusions about the protocol behaviour have been drawn after the evaluation.

## 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 targeted 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 authenticated 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 always need to be improved as it is one of the main requirement of the protocol design. Efficiency improvement 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

# User Manual

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.

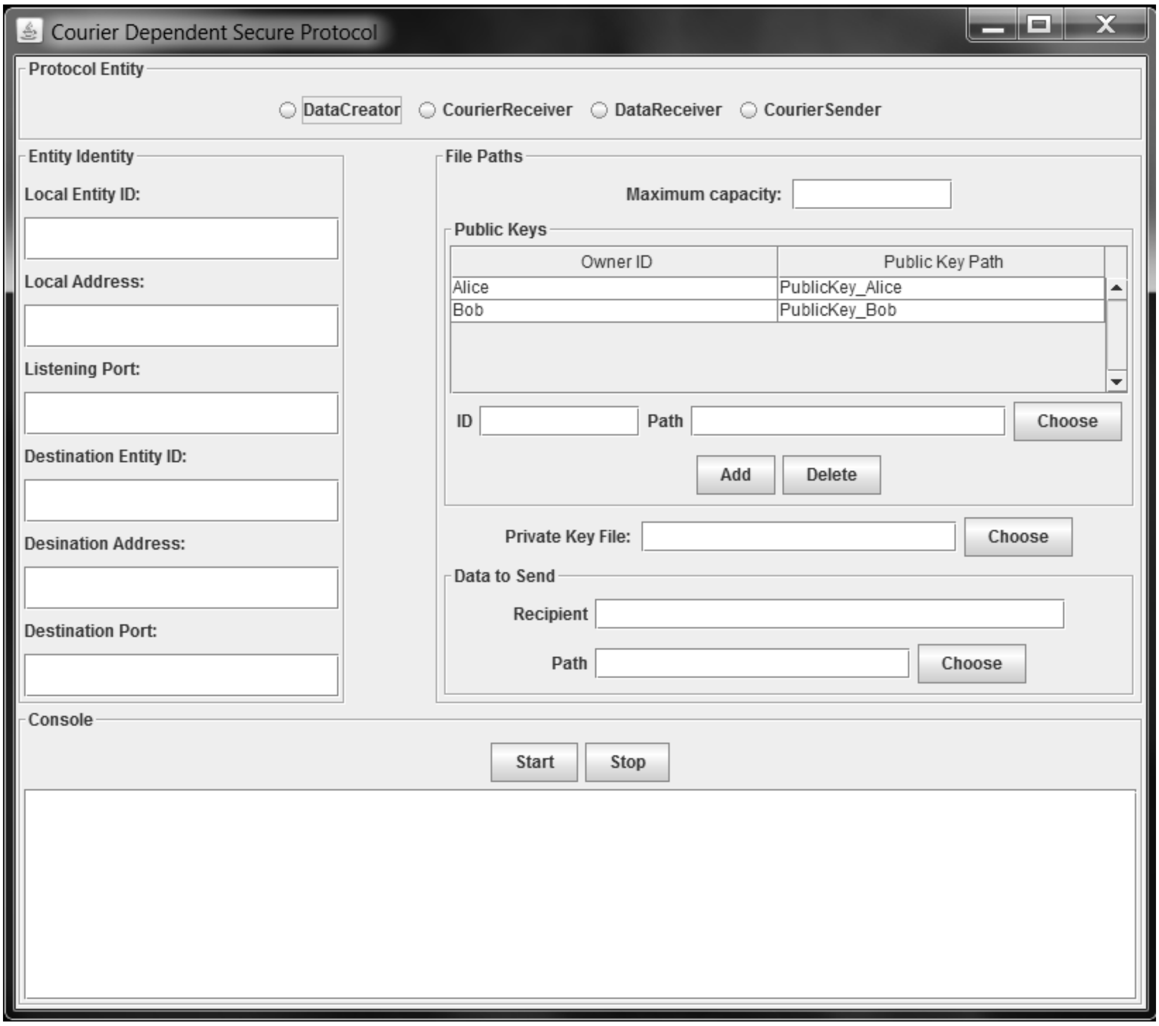



Figure A.1: GUI Overview

## 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:

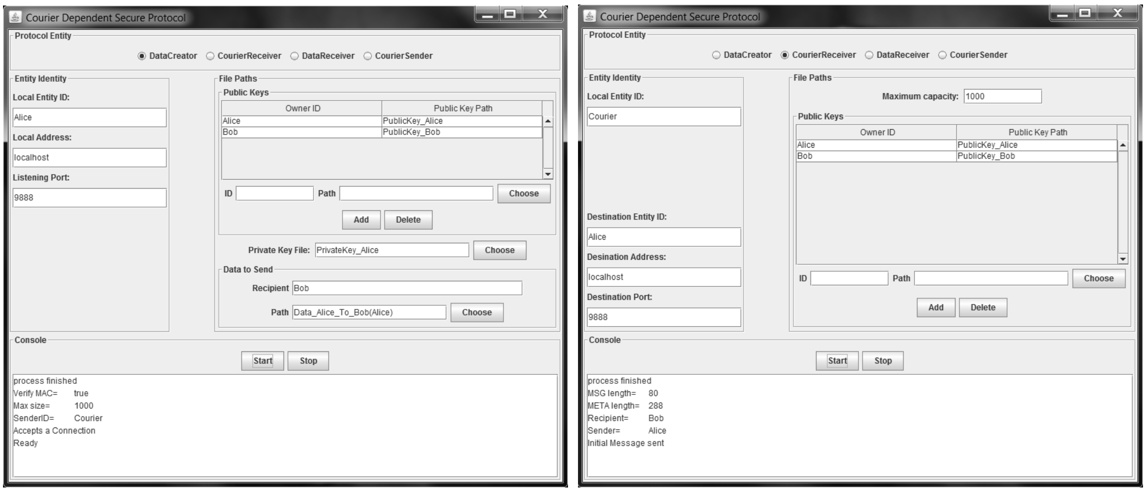
1. $D_0$ starts the application.

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 Entitiy 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 Entitiy Identity panel.

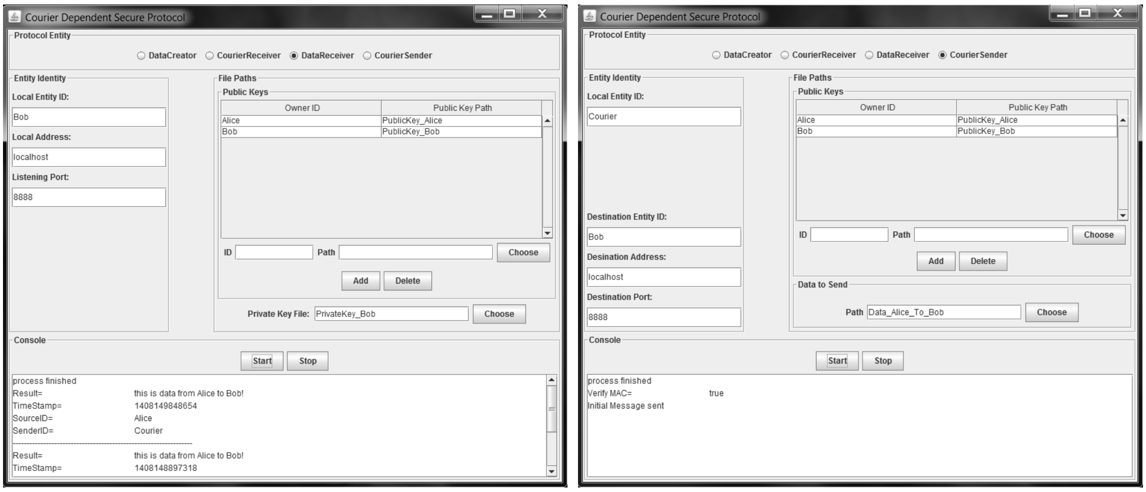
10. $D_1$ user specifies the ID of the entity he wants to contact (here $D_2$'s ID), together with its address and port number by filling corresponding textfields in the rest of Entity Identity panel.

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] D. Eastlake 3rd and T. Hansen. RFC 6234: Us secure hash algorithms (sha and sha-based hmac and hkdf), May 2011. Status: INFORMATIONAL.

[2] Martín Abadi and Cédric Fournet. Private authentication. *Theor. Comput. Sci.*, 322(3):427–476, September 2004.

[3] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[4] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, and H. Weiss. Delay-tolerant networking: An approach to interplanetary internet. *Comm. Mag.*, 41(6):128–136, June 2003.

[5] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss. RFC4838: Delay-tolerant networking architecture. April 2007. Status: INFORMATIONAL.

[6] 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.

[7] C. Dwork and A. Sahai. Concurrent zero-knowledge: Reducing the need for timing constraints. In *In Crypto98, Springer LNCS 1462*, pages 442–457, 1998.

[8] W.F. Ehrsam, 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.

[9] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 27–34, New York, NY, USA, 2003. ACM.

[10] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, April 1988.

[11] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. *SIGCOMM Comput. Commun. Rev.*, 34(4):145–158, August 2004.

[12] B. Kaliski. RFC 2313: PKCS #1: RSA encryption version 1.5, March 1998.

[13] B. Kaliski and RSA Laboratories. RFC 2989: PKCS5: Password-based cryptography specification version 2.0, September 2000. Status: INFORMATIONAL.

[14] A. Kate, G. Zaverucha, and U. Hengartner. Anonymity and security in delay tolerant networks. Technical report, In SecureComm, 2007.

[15] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC: Keyed-hashing for message authentication, February 1997. Status: INFORMATIONAL.

[16] W. Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall Professional Technical Reference, 2003.

[17] F. P. Miller, A. F. Vandome, and J. McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.

[18] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems based on pairing. *SCIS 2000: Symposium on Cryptography and Information Security*, 2000.

[19] K. Scott and S. Burleigh. RFC5050: Bundle protocol specification, November 2007. Status: INFORMATIONAL.

[20] A. Seth and S. Keshav. Practical security for disconnected nodes. In *Proceedings of the First International Conference on Secure Network Protocols*, NPSEC'05, pages 31–36, Washington, DC, USA, 2005. IEEE Computer Society.

[21] A. Seth, D. Kroeker, M. Zaharia, S. Guo, and S. Keshav. Low-cost communication for rural internet kiosks using mechanical backhaul. In *In MobiCom '06: Proceedings of the 12th annual international conference on Mobile computing and networking*, pages 334–345. ACM Press, 2006.

[22] J. Shao, Z. Cao, and R. Lu. An improved deniable authentication protocol. *Netw.*, 48(4):179–181, December 2006.

[23] S. Symington, S. Farrell, H. Weiss, and P. Lovell. RFC6257: Bundle security protocol specification, May 2011. Status: INFORMATIONAL.

[24] Y. Wang, J. Li, and L. Tie. A simple protocol for deniable authentication based on elgamal cryptography. *Netw.*, 45(4):193–194, July 2005.

[25] X. Xin and D. Chen. A secure and efficient deniable authentication protocol. pages 172 – 175, 2000.

[26] A. 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.

[27] W. Zhao, M. Ammar, and E. Zegura. Controlling the mobility of multiple data transport ferries in a delay-tolerant network. In *in IEEE INFOCOM*, 2005.