Design of the Next Generation Smart Card Wallet System

by Sin Wai Ming

A thesis submitted in partial fulfillment of the requirements

for the Degree of Master of Philosophy

in Department of Computing

at The Hong Kong Polytechnic University

The Hong Kong Polytechnic University

January 2003

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written nor material which has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

Signature: _____

Name: SIN WAI MING

ABSTRACT

Abstract of thesis entitled 'Design of the Next Generation Smart Card Wallet System' submitted by Sin Wai Ming for the degree of Master of Philosophy at the Hong Kong Polytechnic University in January 2003.

The smart card is playing an important role in electronic commerce, particularly in the area of electronic payments. It is expected that the next generation of smart card will be more intelligent, more interactive and more interoperable (the 3i requirements). The aim of this project is to design a smart card wallet system for fulfilling these 3i requirements. The smart card wallet is multi-functional, such that it can be used to carry electronic coins, electronic identity cards and other electronic items. It can also interact with other devices on the Internet. An object-based framework is proposed to store various electronic items as objects, and some security protocols are presented to ensure that objects can be stored and managed securely, effectively and efficiently. Furthermore, an anonymous offline payment protocol has been developed by enhancing the CAFE payment system.

To take into account the memory-limited nature of smart cards, a Lightweight eXtensible Markup Language (LXML) is proposed for storing data in the smart card wallet. The basic idea is to employ special navigation tags instead of normal XML tags in the data files in order to reduce the size of a data file. By using these navigation tags, the data entries in a LXML document can be linked to their respective elements in the document object tree.

To utilize the memory of a smart card wallet more efficiently, certain objects can be stored in an Object Server (OS) outside of a smart card. With the aim of minimizing the mean object retrieval time, a dynamic memory management algorithm has been developed to determine which objects should be stored externally by taking into account the available memory and the size and retrieval frequency of each object.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my project supervisor, Dr. Henry C.B. Chan of the Department of Computing at the Hong Kong Polytechnic University, who gave me valuable advice on this project throughout the years. He spent a lot of time helping me throughout the project. His encouragements, ideas and guidance are the most important factors of the project success.

Special thanks must also go to my friend, Mr. Pang Yu Kei, Ivan for his discussion on the development of the prototype.

I would like to thank the funding support of the Innovation Technology Fund (Teaching Company Scheme) and Macroview Telecom Ltd.

I would also like to express my appreciation for the Tuition Scholarships for Research Postgraduate Studies at the Hong Kong Polytechnic University.

I have been very fortunate to have my girlfriend, Heidi, who shows consideration to me throughout the years. She has always provided me with support and encouragements.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1: Introduction

A card embedded with a microprocessor was first invented by 2 German engineers in 1967. It was not publicized until a French journalist called Roland Moreno, reported the Smart Card patent in 1974 [1]. With the advances of microprocessor technology, the development cost of the smart card has been greatly reduced. In 1984, a breakthrough was achieved when French Postal and Telecommunications services successfully carried out a field trial with telephone cards. Since then, smart cards are no longer restricted to the traditional bankcard applications.

In May 1996, several companies including Microsoft, Hewlett-Packet and Schlumberger formed a PC/SC workgroup, which aimed at integrating the smart card with personal computers (PCs). This workgroup mainly concentrates on producing a common smart card and PC interface standards for the smart card and PC software producers. Many of the interface standards and hierarchies have already been established. Some of these prototype products are now available in the market.

Netscape and Microsoft have also announced that the smart card will play an important role in their computer security and electronic commerce products. For example, Microsoft has indicated that Microsoft Windows 98 and Windows 2000 not only support smart card but also provide programming modules for developing smart card applications using C++, Visual J++ and Visual Basic.

Throughout the history of smart card development, various standards have been established for enhancing the interoperability problem. The very first standard is the ISO 7816 smart card standard published by the International Organization for Standardization (ISO) in 1987. Before that, card vendors and manufacturers developed their own proprietary cards and readers, which were not interoperable. With the ISO standard, smart cards can communicate using the same protocol. The physical appearance and dimensions of a card, the location of the contacts and the communication messages etc. are all standardized. This enables a smart card manufactured by one company to interact with devices of other companies.

Two other important standards related to smart cards are EMV (Europay, Mastercard and Visa) and GSM (Global Standard for Mobile Communications). The EMV standard is for debit and credit cards issued by financial institutions such as Visa, Mastercard and Europay. Issued in 1996, this standard covers the electromechanical characteristic, data elements and instruction commands together with the transactions involving bank microprocessor smart cards. The goal of the EMV specification is to enable payment systems to share a common Point of Sales (POS) Terminal, as they do for magnetic stripe cards. As it is expected that the magnetic stripe-based banking cards would soon be replaced by smart cards, this standard is developed to ensure that new smart-card based banking cards can be compatible with the existing bank transaction system. This also allows all banking smart cards to be compatible with one another. Moreover, terminal manufacturers can develop and modify their own sets of API in EMV standard for their terminals, so these terminals can be used in different payment systems. Credit, debit, electronic purse and loyalty functions can be processed on these EMV-compliant

terminals. With the flexibility provided by the EMV standard, banks are allowed to add their own features in the smart card payment system.

The GSM standard is one of the most important smart card and digital mobile telecommunication standards. GSM specification started in 1982 under CEPT (Conference Europeenne des Postes et Telecommunications) and was later continued by ETSI (European Telecommunications Standards Institute). Originally, this specification is designated for the mobile phone network. However, when the smart card is used in the mobile phone system as the Subscriber Identification Module (SIM), parts of the GSM specification becomes a smart card standard. This part of the GSM specification started in January 1988 by the Subscriber Identification Module Expert Group (SIMEG).

Within a GSM network, all GSM subscribers are issued with a SIM card which can be viewed as the subscriber's key to access the network. A SIM card has the same size as a credit card. Furthermore, a mini-sized version is also available. Because this card is used for handling various GSM network functions, a rather high-performance microcontroller (a 16-bit microprocessor) is used and the EEPROM memory is dedicated for storing the application data, including the network parameters and subscriber data.

The GSM specification is divided into two sections. The first section describes the general functional characteristics, while the second section deals with the interface description and logical structures of a SIM card. Details of this specification are given in [2].

Before the smart card could be widely adopted by the market, one or more standardized card development environment is needed. Currently, four major smart card standards have been developed in the smart card industry namely PC/SC, OpenCard Framework [3], JavaCard and MULTOS and they are compatible to the ISO smart card standard.

It is forecasted that in the next few years, the world-wide smart card sales will increase from two thousands million units to around four thousands million units (Figure 1).

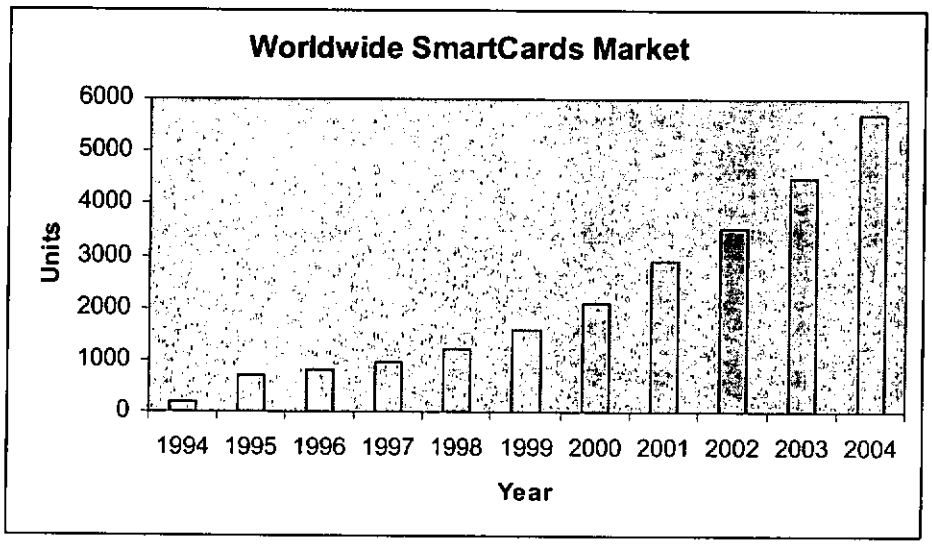


Figure 1: Worldwide Smart Cards Market Forecast

(Millions of Dollars and Millions of Units)

Source: Data from Frost & Sullivan

The World Wide Web is an ideal platform for supporting electronic commerce (e-commerce), particularly Business-to-Consumer (B2C) e-commerce, because it enables consumers to shop anywhere and anytime. In fact, e-commerce can also

complement physical commerce to provide better customer service. For example, a consumer can purchase a film ticket online before going to see the film at the physical cinema. Security and payment are currently two major requirements in B2C e-commerce. To carry out a transaction securely, we need to satisfy the CIA security requirements [5], where CIA stands for Confidentiality, Integrity and Authentication. The Authentication requirement is particularly important. Currently the X.509 digital certificate system is commonly used for authentication purposes or creating a public key infrastructure. Once the identities of a consumer and a merchant have been verified, they can establish various cryptographic keys for performing encryption to ensure data confidentiality (i.e., to satisfy the confidentiality requirement), and for generating digital signatures to ensure data integrity (i.e., to satisfy the integrity requirement).

In traditional commerce, we have the 4Cs payment methods, namely Cash, Credit card, Check and Credit/debit (i.e., funds transfer) [1]. While we also have the cyber versions of the 4Cs payment methods, the credit card is currently the most commonly, used payment method for B2C e-commerce. In particular, the Secure Electronic Transaction (SET) protocol [7] has recently been adopted to support secure credit card payment over the Internet. Besides the traditional payment methods, which are commonly referred to as macropayment methods, many micropayment methods (e.g., Millicent [4]) are also emerging to handle very small value transactions.

While the public key infrastructure and various payment methods are available to support B2C e-commerce, the current solutions have a major shortcoming.

Currently, a digital certificate is usually installed in a Web browser and hence is not portable (i.e., it is terminal based rather than personal based). For example, once you install a digital certificate in a computer, you need to use that computer for authentication purposes. Similarly, many payment solutions, such as SET, are also terminal-based rather than personal-based.

Smart cards provide an effective solution to support portability and hence overcome the above shortcoming. Our research focus is to use smart cards as wallets (i.e., smart card wallets). People can store digital certificates, electronic cash, etc. in a smart card wallet and carry it to conduct electronic transactions anywhere in a secure and efficient manner through the Web. Integrated with the smart card wallet, the Web becomes an even more effective tool to support B2C e-commerce. However, current smart cards have two major limitations. First, the memory of a smart card is very limited. Therefore, it is not possible to store a large amount of data inside a smart card, and hence its application is limited. For example, it is not feasible to store a large image file inside a smart card. Second, smart cards produced by different manufacturers are generally not inter-operable because each manufacturer often employs its own proprietary development tools and application programming interface. The aim of this project is to address these two issues. To extend the memory of a smart card, we propose that for some objects, an agent of the original object is stored inside a smart card whereas the actual object is stored somewhere on the Internet. Whenever the actual object is required, the agent retrieves the object over the Internet for the user. To develop an inter-operable smart card wallet, we propose an object-oriented framework, and the Java card [5] is used to realize this framework. A Java card is a smart card with a

Java virtual machine built inside the card for running Java applets. It not only facilitates programming, but also allows Java applets to run on other Java-compatible cards. This greatly facilitates the development of interoperable smart card applications.

In summary, the objectives of our project are:

- To design an object-oriented framework for the next generation smart card wallet such that data can be stored effectively and efficiently.
- To design effective protocols for the smart card wallet.
- To investigate effective methods to extend the memory of the smart card wallet.
- To build a prototype for showing the basic functions of the smart card wallet.

The organization of the rest of this thesis is as follows. Chapter 1 gives a general overview of smart card as well as its current development. In chapter 2, we introduce the basic operation of a smart card and its role in an electronic cash payment system. In Chapter 3, the architecture of the next generation smart card wallet is introduced. In particular, we consider the Hong Kong Identity (HKID) card, the credit card, the e-coin bag and the name card object as examples to illustrate how different types of objects can be stored inside a smart card wallet. An Object Server (OS) is also discussed such that objects can be placed in a remote secure server. In Chapter 4, a novel LXML, which is backward compatible with general XML document, is introduced. By employing navigation tags, LXML

enables data to be stored more efficiently inside a smart card. In Chapter 5, a dynamic memory management algorithm is proposed to extend the limited memory of a smart card. The algorithm determines which objects should be stored externally while minimizing the mean retrieval time. A backward induction algorithm is presented to solve the problem.

CHAPTER 2: Overview of Smart Card

2.1 Physical Structure of Smart Card

Basically, a smart card is a plastic card embedded with a microprocessor chip [1], [9]. The physical structure of a smart card is specified by the International Standards Organization (ISO) 7810, 7816/1 and 7816/2 [10]. In general, it is made up of three elements, namely a plastic card with a typical dimension of 85.60mm x 53.98mm x 0.80mm, a printed circuit which conforms to ISO standard 7816/3, and an Integrated Circuit Chip. The chip has a CPU for controlling the operation, a ROM for storing the operating system, an EEPROM for storing long-term data, and a RAM for storing short-term data. Figure 2 shows an overview of the physical structure of a smart card.

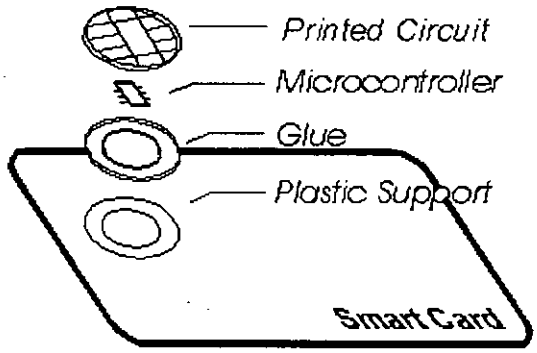


Figure 2: Physical structure of a smart card (Source: Philips DX smart card reference manual, 1995)

The printed circuit conforms to ISO standard 7816/3 and provides five connection points for power supply and data transfer. It is hermetically fixed in the recess

provided on the card and is burned onto the circuit chip, filled with a conductive material, and sealed with contacts protruding. The printed circuit protects the circuit chip from mechanical stress and static electricity. Communications with the chip are accomplished through contacts on the printed circuit.

The capability of a smart card is defined by its integrated circuit chip. Typically, an integrated circuit chip consists of a microprocessor, read only memory (ROM), nonstatic random access memory (RAM) and electrically erasable programmable read only memory (EEPROM) which will retain its data when the power is removed. The current circuit chip is made from silicon which is not flexible and particularly easy to break. Therefore, in order to avoid breakage when the card is bent, the chip is restricted to only a few millimetres in size.

Furthermore, the physical interface which allows data exchange between the integrated circuit chip and the card acceptor device (CAD) is limited to 9600 bits per second. The serial communication line is bi-directional and conforms to ISO standard 7816/3. All the data exchanges are under the control of the central processing unit in the integrated circuit chip. Card commands and input data are sent to the chip that responds with status words and output data upon the receipt of these commands and data. Information is sent in half duplex mode, which means that data is transmitted in one direction at a time. This restriction together with the low bit rate prevent massive data attack on the card. Nowadays, smart cards have numerous applications, such as electronic payment [11], [12], authentication [13] and health care [20]. In particular, smart cards are playing an important role in e-commerce [21], [22].

The size and thickness of a smart card are designed such that it can prevent the card from being spoiled physically. However, this also limits the memory and processing resources that may be placed on the card. Due to this limitation, a smart card needs to operate in conjunction with other external peripherals. For example, it requires a device to provide user input and output, time and date information, power and so on. As external devices are involved, the security of a smart card may be affected under some circumstances.

2.2 Logical File Structure

Generally, in terms of data storage, a smart card can be viewed as a disk drive where files are organized in hierarchical directories. Similar to the MS-DOS and UNIX environment, there is one master file (MF) that functions like the root directory. Under the root, we can store different files called elementary files (EFs). We can also create various subdirectories called dedicated files (DFs). Under each subdirectory, there can be various elementary files. The main difference between a smart card file system and a MS-DOS file system is that dedicated files may also contain data. Figure 3 shows the logical overview of a smart card file structure.

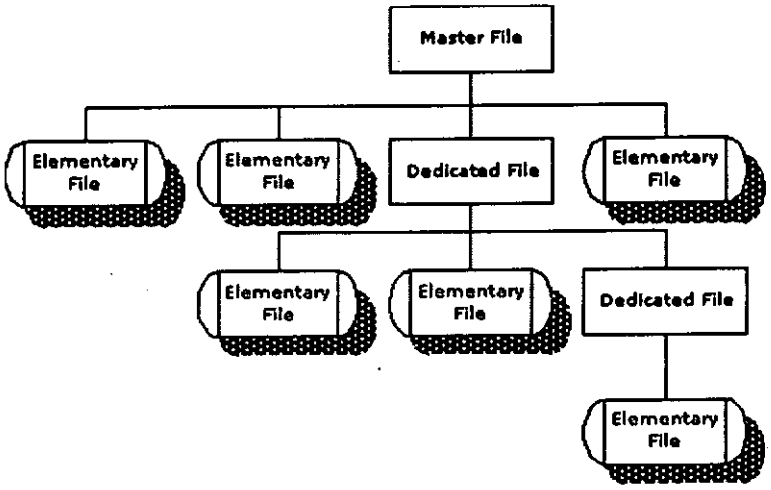


Figure 3: Logical file structure of smart card

In smart card terminology, the body part of a master file (MF) contains the headers of all of the dedicated files and elementary files. The dedicated file (DF) is a functional grouping of files including itself and all its child files. The elementary file (EF) simply consists of its header and its body that stores the data.

The ways data is managed depends on the operating system. Some of them may manipulate the data file simply by specifying the offset and length, while others may organize data in fixed or variable lengths of records such as the Global System for Mobile Communication (GSM) system. In any cases, the file must be selected before performing any operations. This is equivalent to opening a file.

The logical access and selection mechanisms are activated after the power is supplied to the card. Initially the master file is selected automatically. The selection operation allows movement around the logical tree. It can select an EF or a DF by

descending the tree, or select a MF or DF by ascending the tree. Selecting another EF from an EF involves horizontal movement within the same level.

After successful selection, the header of the file can be retrieved. It contains the information about the file such as its identification number, description, types, size. Particularly, it stores the attribute of the file which indicates the access conditions and current status. Access of the data in the file depends on whether the specified conditions can be fulfilled or not.

In short, the file structure of the smart card operating system is similar to other common operating systems such as MS-DOS and UNIX. However, in order to provide greater security control, the attribute of each file is enhanced by adding access conditions and file status fields in the file header. Moreover, a file lock function is also provided to prevent the file from being accessed by an unauthorized person. These security mechanisms and algorithms provide strong protection for smart cards.

2.3 Smart Card communication protocols

To communicate with a smart card, we need to write an application for sending commands to the smart card through a card reader. Command and response APDUs are transported in and out the smart card by using a pre-defined communication protocols as defined in the ISO7816/4 standard (T=0 (character protocol) or T=1 (block-chaining protocol)). However, currently there is no standardized protocol between the host and the reader. To permit the description of card commands

independent of host-reader and reader-card protocols, a standard has been established. This standard defines the command message sent from the client application, and the response message returned by the card to the client application as Application Protocol Data Units (APDUs). Each manufacturer provides a driver to transport APDUs with its proprietary host-reader protocol. By means of the driver, we are referring to the operating drivers that are responsible to make the smart card reader operable. For example on the windows platform, it is a set of DLLs and system files that define the operation of the smart card reader. Figure 4 shows how a client's application can communicate with a smart card by exchanging APDUs.

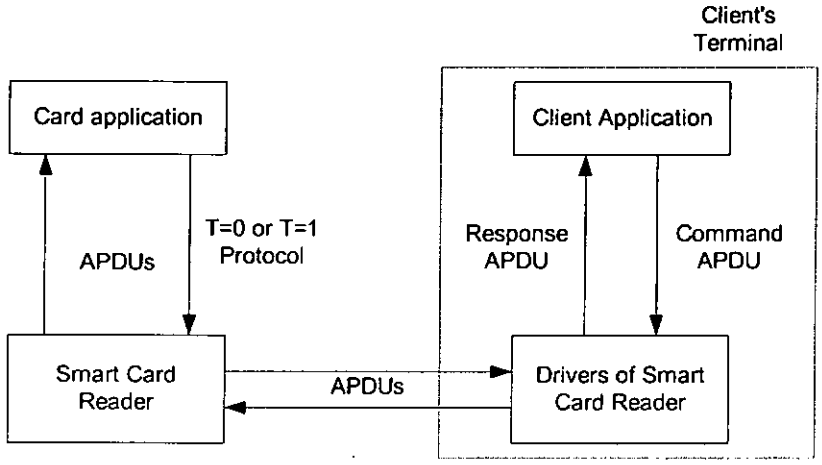


Figure 4: Communication with Application Protocol Data Units

While the meaning of the each APDU command is defined by the standard, it can also be redefined by a manufacturer, or a designer. Some command APDUs are standardized according to the capabilities provided by smart card applications. For example, ISO 7816-4 defines commands for file management, ISO 7816-7 defines commands for database management, and EMV defines commands for general

purpose payment management. Smart cards manufacturers may also define their own APDUs for their application's needs. In general, the following steps are involved when communicating with a smart card.

1. Based on the card command set, that is the list of command APDUs that smart cards can process, obtain the possible values for their parameters, and the format of data.
2. Build the command APDUs sent to smart cards regarding the application data retrieved from local variables, databases, user inputs, etc.
3. Use a driver to transport APDUs from the host to the smart card through a reader.
4. Decode the response APDUs regarding the format of data returned by the card and analyze the different possible values of the status bytes.

Taking the Java Card as an example, for instance, it does not specify how a card applet has to be loaded or initialized, or how the terminal may obtain information about a smart card. The Java Card specification is a standard for programmers, whose goal is to guarantee the interoperability of Java Card applications at the source code level. To support this level of integration, a Java Card applet is defined as a *javacard.framework.Applet* object with a set of methods to enable the applet to communicate through *javacard.framework.APDU* objects.

As shown in Figure 5, the format of a command APDU is made up of a header and a body. The header contains the CLA (class byte), INS (instruction byte), P1 and P2 (parameters), whereas the body contains the optional data, Lc (Length command)

and Le (Length expected). The response APDU has a simpler format. It contains the optional data and two status word bytes, namely SW1 and SW2.

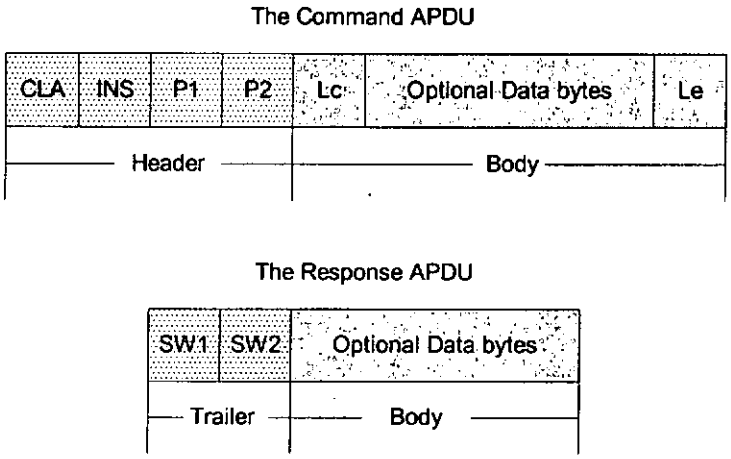


Figure 5 Format of the command APDU and response APDU

The argument used in an applet's 'install' and 'process' methods is an APDU object. This object is used to encapsulate the command and response APDUs exchanged by the card with both the T=0 and T=1 protocols. An APDU object is handled as a buffer in which the application can first read the command APDU received, and then can write the response to be sent back (See Figure 6). Finally, the 'End' state is reached by setting the response APDU's status bytes.

Method	Description
Void install(APDU apdu)	Method called by the JCRE during the final stage of applet installation. Allocates and initializes the applet. A command APDU can serve to provide initialization parameters to this method, and a response APDU can return information on the processing of this method.
Void deselect()	Called by the JCRE when the applet is deselected (i.e. when another applet is selected). Contains the code that must be executed at the end of each session with the applet.
Boolean select()	Called by the JCRE when the applet is selected. Contains the code that must be executed at the beginning of each new session with the applet.
Void process(APDU apdu)	Main method, which receives and processes all the APDUs for the applet, while it is selected.

Figure 6: Main methods of the class javacard.framework.Applet

Three cases may occur during the exchange of APDUs:

1. If the process or install method executes and completes successfully, the JCRE sends response APDU with the status code 0x9000 (normal completion). If the process or install method throws an ISOException, the JCRE returns the response APDU with the reason for that exception in the status bytes.
2. If the process or install method throws an IOException, the JCRE returns the response APDU with the reason for that exception in the status bytes.

3. If any other exception has been thrown, the behavior of the underlying operating system is undefined. In fact, the JCRE may decide to mute the card, block the applet, or perform any operations to safeguard the applet or the card itself.

2.4 Magnetic Card versus Smart Card

A magnetic card is composed of a layer of magnetic material for storing information. It is easy to carry and its main purpose is for basic authentication. The physical size of magnetic card is 8.5cm X 1.2cm, which is defined based on ISO 7811 and the data structure follows the ISO 7811/2. On the magnetic strip, it composes a of maximum of 3 strips and can store around 1K bits of data. Track 1 is developed by International Air Transportation Association (IATA) which contains adaptive 6-bit alphanumerical characters. Track 2 is used by American Bankers Association (ABA) that stores 4-bit numerical information containing the identification number and other control information. Track 3 is originated by the Thrift Industry that contains information intended to be updated in each transaction. Figure 7 shows the properties of the 3 tracks.

Track Record	Density bits/inch	Capacity
1	210	79 (7bits/char)
2	75	40(5bits/char)
3	210	107(5bits/char)

Figure 7: Properties of the 3 tracks in magnetic card

2.5 Java Card Technologies

A Java Card [28] is a special type of smart cards. It conforms to the ISO smart card standards and is thus compatible to the existing smart card applications. However, Java Card has its unique features as well. In particular, a Java Virtual Machine is implemented in its read-only memory (ROM) mask. The JVM controls the access to all the smart card resources, such as memory and I/O, and thus essentially serves as the smart card's operating system. The JVM executes a Java byte code subset on the smart card and provides the functions accessible by applications or devices.

As its name implies, the Java Card is based on the popular Java programming language. It provides an object-oriented development environment through a Java-based platform. A major advantage of the Java Card is that it is more Web-compatible. For example, a Java Card can be accessed through a Java Applet running on a Web browser. As shown in Figure 8, the JVM executes the Java byte codes (or Java Applets) stored inside the smart card and communicates with the external device(s) (e.g. smart card readers) by exchanging the application protocol data units as defined in the ISO 7816 standard. Therefore from the smart card reader's point of view, a Java Card is almost the same as other smart cards that follow the ISO 7816 standards [30].

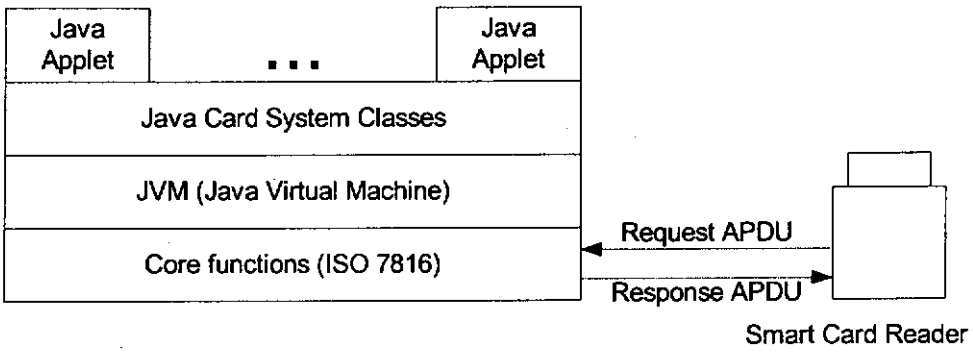


Figure 8: Schematic overview of a Java card (modified from the work of [30])

2.6 Current smart cards and their characteristics

- There are mainly 4 types of smart cards nowadays. They are memory card, MPU IC card, Crypto-processsor card and contactless card. Memory cards are of primitive type and they were originated from the 80s in France as a telephone card. They consist of mainly an EEPROM or PROM and perform very simple functions. Usually, it is used as a prepaid card and it does not have any processing power inside the card. MPU IC smart cards are composed of MCU (Micro Controller Unit). It is software driven and has higher flexibility and intelligence. However, it only provides basic security features. Crypto-processor IC cards are composed of crypto-processor and PROM. It is a powerful Microprocessor Unit and can recognize illegal signal and security features. Contactless smart cards are similar to contact smart card. However, the transmission between the card and the reader is carried out through a wireless channel. Basically, it uses inductive coupling or transformer coupling for transmission and reception.

The advantages of smart cards when comparing with traditional magnetic card such as credit cards are large storage capacity and high flexibility. The largest memory of a smart card is 32Kbytes with a processor or 64Kbytes without a processor, however, a magnetic card can only store data up to 1Kbits of memory. A smart card is also flexible and intelligent. It has lower power consumption and more effective packaging.

The card operating systems (COS) are the operating systems resided inside the smart card integrated circuit chip. The function of the COS is to provide an operating environment for the card to store and manipulate data. Currently, there are three types of COS, namely Multos, Java Card and Windows PC/SC.

Multos is developed by Master Card and it requires secured terminals and approved software code distribution. Therefore, before downloading a program to the card, one needs to present the software code to the relevant parties for approval. This is called configuration control. The reasons for this operation are to detect bugs and backdoor programs such as Trojan horses.

As discussed in the previous section, Java Card-based applications are more portable. Moreover, applications can be securely loaded to the card post-issuance—after it has been issued to the customer. This lets vendors to enhance Java Cards with new functions over time. For example, banking cards used for secure Internet access might be upgraded to support e-wallet such that e-cash, flier miles, and e-mail certificates can be stored inside the smart card.

WfSC (Windows for Smart Card) is only supported by Microsoft. It supports some popular Windows command but it is unstable and unreliable. It is an 8-bit, multi-application operating system for smart cards with 8K ROM. It is designed to be a low-cost, easy-to-program platform for running Visual Basic applications. Essentially, it extends the PC environment for smart card use. Windows for Smart Cards uses the same development tools--Microsoft Visual C++ and Visual Basic--that millions of independent software vendors (ISVs) and in-house corporate developers use. Additionally, because Windows for Smart Cards is part of the PC Smart Card (PC/SC) program that has already become part of Windows NT logon capabilities, smart cards based on Windows for Smart Cards will be able to be read by any certified NT card reader. Issuers will not have to determine if their card will be readable across different PC servers or networks, as long as the reader carries the PC/SC certification logo.

2.7 Electronic cash payment system

An electronic cash system is a system in which digital information, together with electronic devices, (e.g. personal computers, cash registry and smart cards) is used to replace the everyday use of cash money. In every electronic payment system, it includes the main protocols for withdrawal, payment and deposit. The use of these protocols involves three parties in general: a consumer (payer), a merchant (payee) and the bank. Consumer can download money to the cyber wallet in several currencies. A merchant can accept several currencies, convert it to local currency and upload the electronic cash to the bank account. Although the system sounds simple, we have to ensure the acceptability, anonymity, non-duplicability, non-

traceability, non-repudiation and guaranteed payment of electronic money. Acceptability means that the coin should be accepted by the bank and the system would not cause any false rejection. Anonymity is very important in the sense that the bank should not be able to know the identity of the owner of the electronic cash. Like our real-world situation, when you pay ten dollars to buy a newspaper, the merchant should not be able to know your identity based on the cash. Therefore, we should ensure anonymity such that the identity of the electronic cash owner would not be revealed on condition that the electronic cash is not double spent.

Basically, there are two types of payment model namely macro-payment and micro-payment. Macro-payment model are used by most e-commerce system nowadays and generally, it uses credit card or real-time bank transfer method in making payments. In a less secure model, credit card number, expiry date as well as the cardholder's identity are transmitted to the merchant by using the basic SSL protocol. In this model, the credit card number and the identity of the cardholder is revealed to the merchant. In a more secure model, Secure Electronic Transaction (SET) is used such that the credit card number is protected and can only be viewed by the bank. Obviously, the latter one is recommended because the former one cannot protect the credit card from being used again by the merchant for further procurements. This macro-payment model suits low-to-medium volume transactions of medium-to-high value e.g. books, food, toys, home appliances and so on. However, the use of macro-payment model always ties the consumer to the web site where he/she registered with where even if a few items are to be purchased, he/she needs to subscribe to every site they want to make procurement.

This does not guarantee the anonymity of the consumer and is not efficient for small-value payment.

An alternative model, the micro-payment model is used where it emphasizes the ability to make payment of small amounts. Usually, electronic coins are pre-downloaded to the consumer's e-wallet or smart card. Later on, the consumer can buy goods accordingly.

Payment protocols can be off-line or on-line. On-line payments refer to payments where the payee needs to contact the bank or the credit-card company during the payment process (e.g. for validation). For off-line payment protocol, a contact with a third party is not required during the payment process. However, the payee still needs to contact their respective acquirer on a regular basis to clear all received payments.

Electronic money can be stored in the electronic wallet using either the valued-based or token-based approach. Value-based is a direct method of representing electronic cash and clearly it is efficient and flexible. Any amount can be paid from the card as long as the amount does not exceed the value stored inside the wallet. Token-based electronic money acts like ordinary coins and each coin has a fixed amount. Any amount can be paid as long as all available coins can be summed up to the desired amount.

Later in the thesis, we will propose an anonymous off-line token-based payment protocol based on the CAFE project. It ensures that double spending can be

detected and the identity of the respective user will be revealed when double spending occurs. In the following, we give the background of the CAFE project.

2.8 The CAFE project

CAFE (Conditional Access for Europe) was a project funded under the European Community's ESPRIT program. It began its work in 1992 and has been lasted for three years. In this project, an advanced electronic payment system is developed based on the idea of untraceable electronic coins proposed by David Chaum. The goals of CAFE are to design a global prepaid and off-line payment system. It is off-line such that the payer and the payee are not required to contact any third parties including the bank during payment. This lowers the costs of establishing and setting up a communication channel between the payer and the payee.

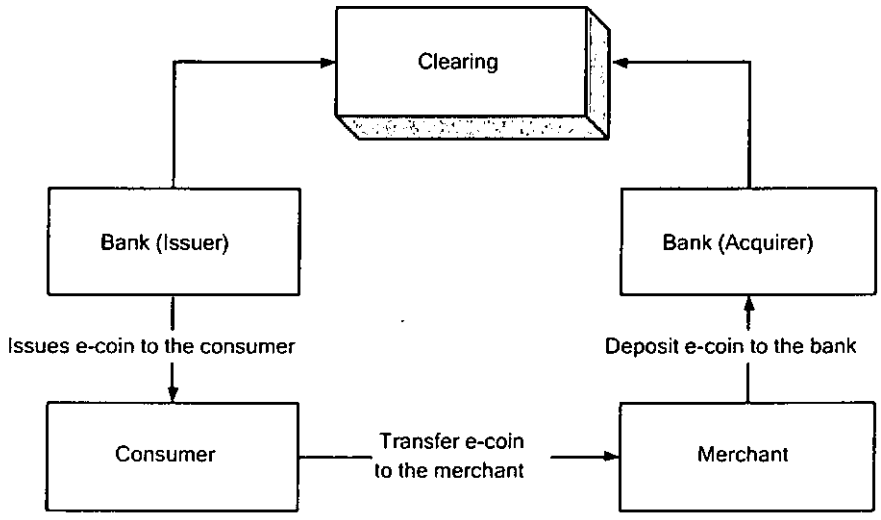


Figure 9: Architecture of CAFE

The architecture of CAFE is similar to most other payment systems in which three main parties are involved in the payment protocol. They are the consumer (payer), the merchant (payee) and the bank. Figure 9 shows the general architecture of CAFE. The devices used in CAFE are tamper resistant secure electronic devices. It includes the smart card and the wallet. The smart card, referred to as the α (alpha) system, stores all the information on the chip and performs the cryptographic computations. The wallet consists of two parts that work in conjunction with each other. One part is known as an observer that protects the bank's interest and the other part is known as the purse that protects the consumer's interest. Inside the wallet, an observer is built in the microprocessor. Unlike the smart card in the α system, the observer exclusively protects the interest of the bank. It ensures the correctness of all transactions performed by the user. A user will not be able to successfully complete a payment transaction without the cooperation of an observer.

CAFE employs two types of security mechanism to protect the payment protocol from being attacked. The first one is the use of secure tamper-resistant devices to store cryptographic keys. The second one, which is the most powerful feature of CAFE, is that it supports the detection of double spending of electronic coins. Due to its off-line feature, every electronic coin transferred from the payer to the payee cannot be checked real time with a central database of the bank. Instead, CAFE allows the detection of double spending at the time where the coins are deposited back into the bank. Normally, when no double spending occurs, the bank cannot know the identity of the owner due to the blind signature protocol [39]. However, if

double spending is detected, the identity of the coin owner is revealed and the bank can perform suitable actions. The algorithm works as follows.

Basically, a coin consists of two parts, the first part and the second part. The algorithm follows that any single part of the coin does not contain enough information to identify the identity of the payer. When the coin is spent the first time, the first part of that coin is read and when the coin is spent the second time, the second part of that coin is read. At the time the coin is double spent, the bank will obtain both parts of the coin and the identity of the payer is revealed. The idea works in such a way that the first part of the coin contains the identity of the payer (I) XOR with a one-time generated random number (R) to produce $I \oplus R$ with an encryption scheme to produce $C(I \oplus R)$ whereas $C(I \oplus R)$ denotes the encryption of $I \oplus R$ with an encryption scheme such as the RSA encryption algorithm using the bank's public key to produce the cipher $C(I \oplus R)$. The second part of the coin contains only the encrypted random number C(R). When the coin is spent the first time, either $C(I \oplus R)$ or C(R) is read and when the coin is spent the second time, the bank can obtain both $C(I \oplus R)$ and C(R). After decryption, the bank can obtain $(I \oplus R)$ and R such that I can be obtained because $I \oplus R \oplus R = I$. In other words, the identity of the coin owner (I) can be revealed.

2.9 XML and compression

XML is a markup language for documents containing structured information. It is the subset of the Standard Generalized Markup Language (SGML) that is designed to make it easy to interchange structured documents over the Internet.

By defining the role of each element using a formal model, known as the Document Type Definition (DTD), users of XML can check whether a XML document is valid or not. One special characteristic of XML is that it specifies neither semantics nor a tag set. In fact XML is a meta-language for describing markup languages. In other words, XML provides a facility to define tags and the structural relationships between them flexibly. Since there is no predefined tag set, there are no special semantics. The semantics of an XML document are either defined by the applications that process them or by the associated style sheets.

XML is rapidly becoming the industry standard for the exchange and processing of data between different software applications and platforms. While the XML format makes the interchange of data easier, XML substantially increases the file size by adding tags. Therefore, there is a need to conduct XML compression for some applications.

In general, data compression is concerned with the minimization of the amount of data needed to represent information, which is produced by a source and described by messages composed of symbols. Compression is often related to coding, since its objective is to represent source messages with codes. Source coding is related with the semantics of data, whereas entropy coding refers to its redundancy [13], [15], [16].

The most important difference among the compression techniques concerns with their reversibility. If the decoded data is identical to the original one, the

compression method is called lossless; otherwise, it is called lossy. Lossless compression schemes refer e.g. to the work by Human [17], the algorithm by Lempel and Ziv [18] and the more recent Arithmetic Coding. Lossy compression has higher compression ratios so as to preserve a representative subset of original data. Approaches to lossy compression include wavelet transformations, histograms and methods for the extraction of significant parts from a free text. XML uses markups to identify and describe data. The schema-related information is contained in documents themselves so the language is called self-describing.

Recently, a number of methods for conducting lossless XML compression has been developed. XMill [19] applies classical entropy-based compression techniques by executing ad-hoc compression rules driven by the XML structure of the document and the semantics of the data. The main ideas are to separate structure from data; grouping related data items into homogeneous classes; applying semantic compressors to these data classes and applying general-purpose compressions. XMill makes use of the above characteristics of data identifiably and markup structure compressibility. It can therefore achieve higher compression ratios that are significantly higher than those produced by general-purpose compressors, at almost the same speed.

Our focus is on the compression of XML documents stored inside a smart card wallet. The characteristic of this type of XML compression is that the memory inside a smart card is very limited. As a result, we need a compression technique that can compress an XML document as small as possible. The proposed Lightweight XML (LXML) is designed to cater for this requirement.

CHAPTER 3: Next Generation Smart Card Wallet

3.1 Architecture of the next generation smart card wallet

Emulating physical wallets, our focus is to use smart cards as wallets for storing different types of items. While current smart cards are "smart", future smart cards will be even "smarter". We think that future smart cards will be more "Intelligent", "Interactive" and "Interoperable". We call these the 3i requirements, and they are further explained as follows. 'Intelligent' refers to the nature of smart cards. Unlike traditional smart cards, next-generation smart cards are multi-functional in that different applications can be loaded into the same smart card [23]. In other words, the same smart card can be used to support different applications and this is done automatically without the intervention of the user. 'Interactive' means that a smart card can interact with not only the attached smart card reader but also with other devices on the Internet. 'Interoperable' means that a smart card can operate in multiple platforms and interact with smart cards of different manufacturers. In other words, the smart card applications should achieve the useful property of "written once, run in many cards". The OpenCard Consortium [3] has been formed to establish an open framework for developing smart card applications. In particular, the Java card provides an effective development platform [5].

3.2 Classification of objects

To satisfy the 3i requirements, we propose to use an object-oriented framework such that everything is stored inside a smart card as an object. The object-oriented

approach greatly facilitates design and implementation through reusability. In other words, we can design a generic model such that when new functions and objects are needed, we can extend the functions of the existing objects rather than re-designing them from scratch. This not only shortens programming time but also increases the extensibility of smart cards.

Our focus is on the design of a smart card wallet. Figure 10 shows that objects stored inside a smart card wallet can be classified into three categories, namely transferable objects, non-transferable objects and agents.

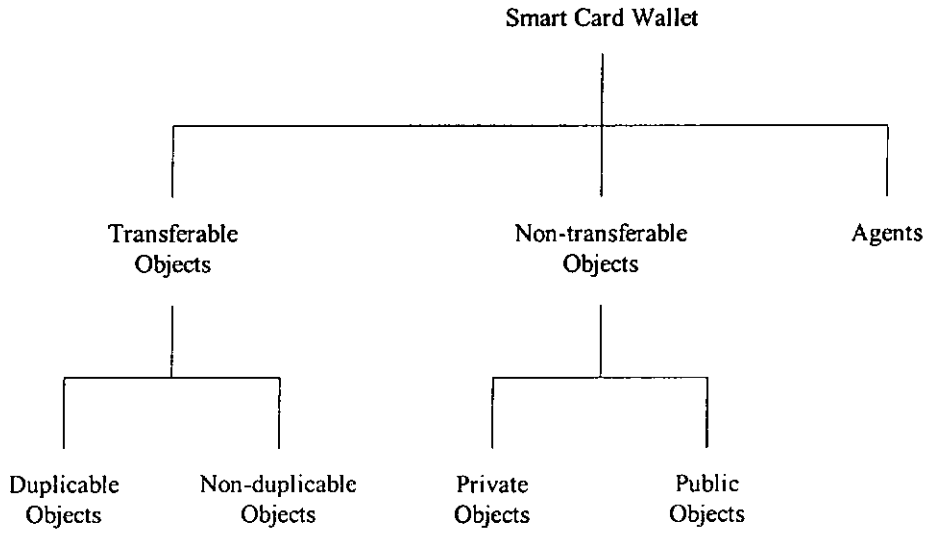


Figure 10: Classification of smart card wallet objects

Transferable objects are objects that can be transferred out of the smart card wallet. These objects can be further divided into duplicable objects and non-duplicable objects. Duplicable objects can be duplicated or copied. Examples of transferable and duplicable objects are name cards, purchase receipts and digital photographs.

When these objects are transferred from one smart card wallet to another smart card wallet, they are not deleted in the original wallet. Therefore these objects are not unique because duplicated objects may exist. For transferable and non-duplicable objects, they are totally removed from the original smart card wallet after being transferred to another smart card wallet. A typical example of transferable and non-duplicable objects is electronic cash stored in the form of a secure token. Unlike duplicable objects, non-duplicable objects are unique.

In contrast to transferable objects are non-transferable objects. As the name implies, these objects cannot be transferred to another smart card wallet. Non-transferable objects can be further divided into public objects and private objects. An example of public objects is the identity card of the cardholder. The content/information of public objects can be accessed by other smart card wallets through the predefined methods. However the object itself must remain inside the smart card wallet. An example of non-transferable private objects is the private key used for RSA encryption. These objects are typically used to perform various supporting functions inside a smart card wallet. Its access is strictly restricted to the owner. Figure 11 shows further examples of the four different types of objects, which can be stored inside a smart card wallet.

Object	Transferable duplicable	Transferable non-duplicable	Non-transferable public	Non-transferable private
Name card	√			
Receipt	√			
Digital Photograph	√			
Electronic coin		√		
Identity Card			√	
Medical Record			√	
Private Key				√
Personal Diary				√

Figure 11: Examples of different types of objects stored inside a smart card wallet

Apart from the transferable and non-transferable objects, the third type of object is called an agent. It can be thought of as a physical key stored inside a smart card wallet. The key is used to 'open' the door or to gain access to other things. This is similar to the situation in which a key is used to open a locked drawer. Once the draw is opened, we can take the things out from the drawer. Working in a similar manner, the agent acts as a key for accessing other objects stored outside the smart card wallet. For instance, due to memory constraints, it is impossible to store a large image file in a smart card wallet. However an agent can be stored in the smart card wallet for retrieving the image file over the Internet. This not only overcomes the memory constraint of a smart card wallet but also opens many new applications and services.

3.3 Managing objects stored inside the smart card wallet

Objects stored inside a smart card wallet are managed by an object manager (OM) as shown in Figure 12.

As shown in Figure 12, each object is divided into three main components. The first one is the object header that contains the identity number of the object *</ID>...</ID>*; the type of the object (i.e. whether the object is duplicable or transferable) *<TYP>...</TYP>*; the textual description of the object *<DES>...</DES>*; the identity of the party that issues the object *<IID>...</IID>* and finally the signature of the object *<SIG>...</SIG>*; which is a hash value signed with the issuer's RSA private key. The reason of adding the digital signature is to ensure that the object content cannot be altered and is really issued by the correct object issuer. The second part of the object is the object content. Data is expressed in an XML string contained within the *<CONT>...</CONT>* tag. It can be noticed that storing the XML string inside the smart card occupies a large amount of memory space and therefore, we have designed a novel lightweight XML (LXML) such that it can greatly reduce the overhead of XML. Details will be discussed in chapter 4. The third part of the object is the method. It is a set of interfaces that exposes different operations which can be performed on the object.

The OM itself is a non-transferable and private object inside a smart card wallet. It contains the ID of all the objects stored inside the smart card wallet.

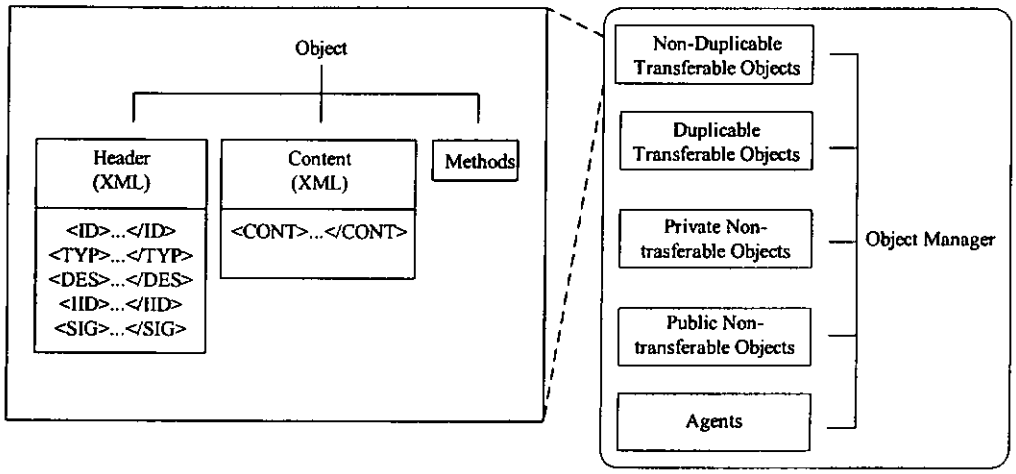


Figure 12: Detailed explanation of the object manager

The main function of OM is to handle the addition, removal, retrieval and modification of the objects stored inside the smart card. Basically each object has a header that contains the object ID, object type, object description and object signature. The object ID is a unique identifier for identifying the object. The object type defines the type: duplicable transferable object, non-duplicable transferable object, public non-transferable object, private non-transferable object or agent object. The object description gives a short description of the object. The signature provides the hash of the object content. It is to ensure that the object content has not been altered.

The UML model for a generic object is shown in Appendix B. A generic object contains the object header, the *log* for logging all the transactions conducted and various methods. The method *getValue()* is responsible to get the value of a particular variable inside the object. The *setValue()* method is to set the value of a particular variable. The *getDigitalSignature(), setDigitalSignature() and verifyDigialSignature()* methods are to get, set and verify the digital signature of

the object, respectively. The *writeLog()* function is to write the transaction details to the *log* variable.

With the construction of a generic object, new objects can be added to the smart card and extends the features from the generic object with the use of object-oriented concept. We are going to take four real-world objects as examples and discuss their respective uses in section 3.4.

The API for the object manager is shown in Appendix C. Basically, there are three classes. The class *Connect* is responsible to establish the connection between the terminal computer and the smart card. The class *DMI* stores the methods used by the smart card's object manager. The class *OMClient* is the main class to format an XML request and to send it to the smart card.

The object manager is responsible to determine whether a particular object is to be stored internally inside the smart card or externally in a remote object server. To do this, the object manager needs to calculate the minimum mean retrieving time among all the objects. To increase the efficiency of calculating the minimum mean retrieving time, a Dynamic Memory Decision Model $(DM)_2$ is proposed. The main idea is to formulate a Markov decision model and solve it using the backward induction algorithm. Details are discussed in Chapter five.

To demonstrate the functions of the object manager, a prototype system called JCOM (Java Card Object Manager) has been built. Details of the system are described in section 3.6.

3.4 Using the next generation smart card wallet

The next generation smart card wallet is designed in such a way that new objects can be added to the wallet easily and it is done by using an object-oriented concept in defining the objects. Apart from this characteristic, the next generation smart card wallet is secure through the use of cryptographic technologies.

While using the RSA encryption technologies, we need to have a digital certificate preinstalled in each smart card wallet. Each smart card has its digital certificate. Each party that is going to use the object is also required to install a digital certificate. The reason for applying a digital certificate is to prove the identity of a particular party and to define which operations can be performed on a particular object.

The digital certificate is obtained from a certificate authority, which manages the trust relationships between all parties in the smart card wallet system. We call the authority: Smart Card Certificate Authority (SCCA). Its function is to issue digital certificates based on the X.509 framework. The architecture of the SCCA is illustrated in

Figure 13.

Generally speaking, to use the smart card wallet, one needs to apply for a digital certificate from the SCCA and the name (CNAME) will be uniquely identified in the certificate. All smart cards, before distributing to the users, are preloaded with a digital certificate and the public key of the SCCA. Every time, when the smart card

wants to verify the identity of the object issuer, it can extract the public key of the CA and then validate the certificate accordingly. Details are shown in Figure 14.

After discussing the general architecture of the next generation smart card wallet, we consider four objects: The Hong Kong ID card, the credit card bag, the e-coin bag for micro payment and the name card in the subsequent discussions.

Basically, there are four kinds of operations that can be performed on a particular object. They are: adding a new object (addition), getting the object details (retrieval), modifying the object content (modification) and deleting an object (removal). While there are only a few operations, there exist many issues as discussed below.

For instance, the addition operation requires that the object issuer is authenticated. In other words, only the authorized party can add a new object to the smart card wallet. The reason for this is that we want to prevent the adding of a 'fake' object to the smart card wallet. If we allow anyone to add a new object to the wallet, we cannot prevent one from adding a fake object to the wallet. To achieve this goal, we have introduced a challenge and response protocol such that the object issuer's identity is verified before a new object can be added to the smart card wallet. Details of this protocol are illustrated in section 3.4.1.

Apart from the authentication problem mentioned above, we need to know whether an authenticated party is eligible to add a new object to the smart card wallet. It is different from the authentication problem. In the authentication problem mentioned

before, we only need to prove that the party is authenticated and is the real object issuer. However we do not know whether the party is eligible to add the object or not. In some cases, we only allow a party to 'retrieve' the object content but we do not allow that party to 'add' a new object to the smart card wallet. It is an authorization problem. The solution to this problem is to introduce the concept of 'authorization level'. The authorization level defines which object operations are allowed for a particular party. Illustrated in Figure 14, each party's authority level is defined in the corresponding digital certificate. The authority level is defined in the 'O' field of a standard X.509 digital certificate and the format is: OBJECT_NAME@add/get/delete/modify. OBJECT_NAME defines the name of the object where we are operating on. The '@' sign is used to delimit the object name and different operations that are allowed by that party. After the '@' sign, we have the 'add/get/delete/modify' specification. 'Add' defines that the party is eligible to perform the add object operation. 'Get' defines that the party can perform the get object operation, i.e., to retrieve the content of the respective object content. 'Delete' defines that the party is authorized to perform the delete object operation, i.e., to remove the object from the smart card wallet. Finally, 'Modify' defines that the party is authorized to perform the modify object operation, i.e., modifying the content of the object.

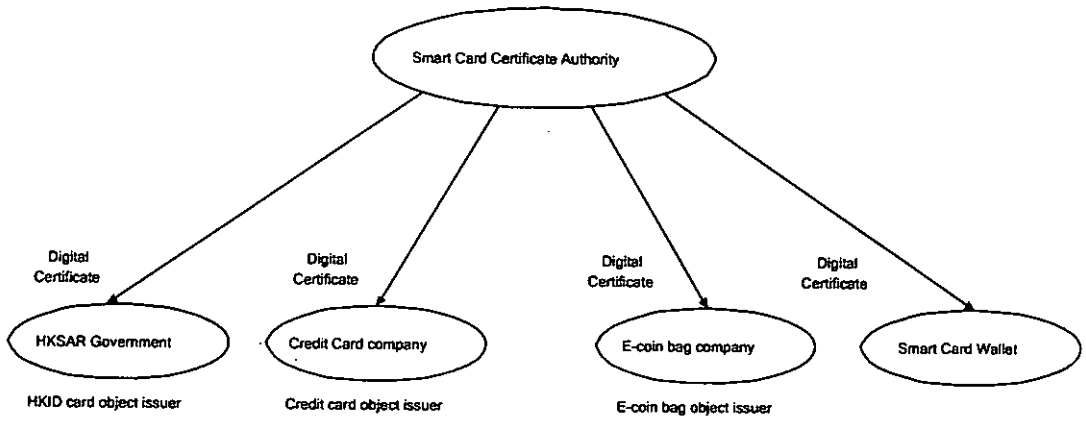


Figure 13: The architecture of SCCA

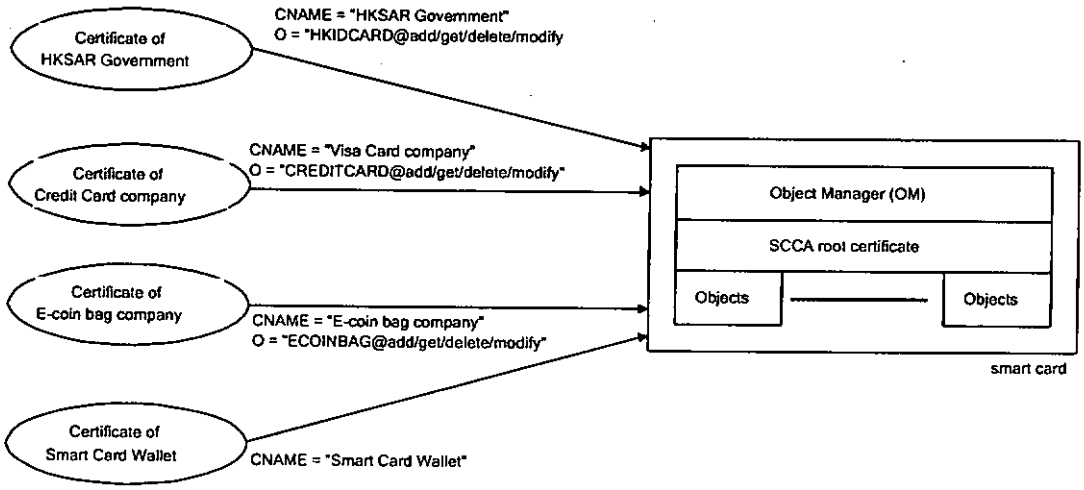


Figure 14: Unique CNAME and authority level of each object issuer

The UML model for the four kinds of objects are shown in Appendix A. Variables and methods are extended from the generic object. Furthermore, new variables and methods can be added for each object in future.

3.4.1 Authenticating the object issuer

As have been discussed in the previous section, when a new object is to be added to the smart card wallet, we need to ensure that the object issuer is authorized and have the right to perform the action. As a result, we have designed a simple challenge and response protocol such that we can ensure that the object issuer is authenticated and has the required authority level to add a new object to the smart card wallet. Details of the protocol are illustrated in Figure 15.

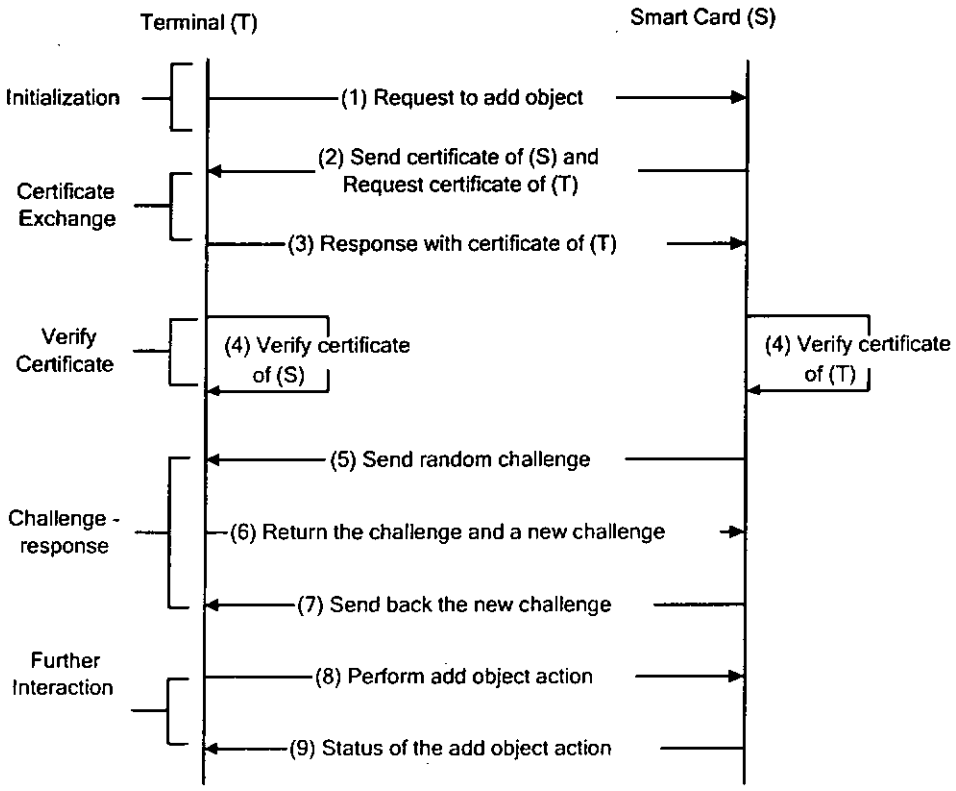


Figure 15: Secure communications between the terminal and the smart card

1. In step 1, the terminal initiates the add object action by sending a request to the smart card wallet.

2. The smart card, upon receiving the add object request, requests the terminal for its certificate and at the same time, sends the smart card's certificate to the terminal.
3. The terminal then sends its own certificate to the smart card wallet.
4. Both the terminal computer and the object manager (OM) inside the smart card verifies the certificate of both parties.
5. After verifying the certificate, the smart card sends a random challenge, a newly generated DES [32] session key, a time stamp and the identity of the terminal to the terminal. All the messages are encrypted with the public key of the terminal derived from the digital certificate of the terminal.
6. If the terminal possesses the respective private key, it decrypts the challenge and then generates a DES session key to be used in subsequent communications. The terminal then returns the challenge back to the smart card together with a newly generated random challenge. All messages are encrypted using the public key of the smart card derived from the smart card's certificate.
7. The smart card verifies the correctness of the random challenge and returns the newly generated challenge back to the terminal encrypted using the DES session key received.
8. The terminal verifies the new challenge and sends the new object encrypted with the DES session key to the smart card wallet.
9. Finally, the smart card wallet returns to the terminal the status of the add object command encrypted with the DES session key.

In our protocol, apart from just encrypting the messages in the 'Further Interaction' phase, we also add a challenge and response phase. The reason for this is to ensure that the public key used in RSA encryption is really the one owned by the real object issuer. Each object issuer carries a digital certificate with a unique CNAME and therefore, we can check the CNAME of the certificate to verify that it is the real object issuer. Apart from authenticating the object issuer, we need to ensure that the smart card itself is authenticated. The reason is that someone may produce a 'fake' smart card and attack the system. Therefore, by conducting the challenge and response protocol, both parties can be authenticated. The object issuer authenticates itself to the smart card by decrypting the random challenge generated by the smart card and the smart card authenticates the object issuer by decrypting the new challenge generated by the object issuer.

3.4.2 Logging mechanism of the smart card wallet

For some secure applications, each transaction conducted between the terminal and the smart card has to be logged for recording purposes. Basically, the login operation achieves the non-repudiation requirements. By logging the transactions securely, we can trace back the transactions.

As discussed before, the memory space inside a smart card is very limited. Therefore, we need an effective and efficient algorithm to store these transaction details. Obviously, storing each transaction command in plain text inside the smart card is not feasible as it occupies too much memory. It is still not feasible if we consider storing a little amount of data, for example the hash value of each transaction inside the smart card. Although it greatly reduces the memory requirement, we cannot store unlimited hash values inside a smart card. On the other hand, the terminal can store the transaction details because of its large memory.

The algorithm mainly involves the techniques of MD5 [34], bit-wise XOR operation and digital signature. Firstly, as shown in Figure 22, we need to perform initialization, certificate request, certificate verification and challenge/response. For the first message received by the smart card, the corresponding hash value is calculated. The hash value is computed by using the MD5 algorithm to produce a 24-character BASE64 encoded string and it is stored inside the smart card for further use. For the second message sent from the smart card, a hash value is also calculated and is XORed with the hash value produced by the first message. This

process goes on until the last message of the whole transaction is reached. After the last message of the transaction, we add the identity of the smart card and the timestamp to the hash value and eventually, we get the resultant 24-character hash value for the transactions. Figure 16 shows the iteration process.

If i not equal <Last message>

$H_i = H [H_{i-1} \oplus H [M_i]]$

Else

$H_i = H [H_{i-1} \oplus H [M_i]]$

$H_{i+1} = H [H_i \oplus H [$ Timestamp + <ID of smart card> + Transaction ID $]]$

where H_i = Resultant hash value of i^{th} message

$M_i = i^{th}$ message sent or received

$H [M_i]$ = Hash value of i^{th} message

Figure 16: Iteration process of creating the chained-hash

The resultant hash value is then stored inside the terminal. It is signed with both the private key of the smart card and the private key of the concerned party. Note that the same signature can only be produced by the same transactions (i.e. in the same order) conducted by the same parties. Therefore, when someone denies having performed a transaction, we can take out the transaction details in the plain text from the terminal and calculate the hash value for each message sent and received. Each calculated hash value is then XORed with the previous hash value. Finally, the resultant digital signature is re-generated. If the signature matches with the stored signature, we can conclude that the transaction has been performed. Figure 17 shows the table for storing all the logging information inside the database where

there are N transactions and $a,b,...,z$ refer to the number of messages for the transactions.

Transaction ID	Messages
1	$M_1, M_2, \ldots, M_a$
2	$M_1, M_2, \ldots, M_b$
...	...
N	$M_1, M_2, \ldots, M_z$

Figure 17: Log format inside the database

3.4.3 The HKID card object

In order to illustrate how to store different types of objects inside a smart card wallet, several objects are investigated. The first one is the HKID card object. It is used to uniquely identify a person by storing various data such as the name, sex, age, date of birth, identity card number, photo and etc. Apart from normal personal data, it can also store the template of the thumbprints by storing a hash value of the thumbprints. Also, other value-added applications such as the driving license and the library card can be optionally stored inside the object. The object itself is non-transferable. This means that it can never be transferred out from the smart card wallet. In other words, a HKID card object cannot be moved to another wallet. Therefore, when adding this kind of object to the smart card wallet, extra steps have to be taken to ensure that it 'sticks' with the smart card wallet.

One practical application in using the HKID card object is to use it in the authentication system in the Immigration Department. Whenever someone passes through the immigration, he/she needs to present the smart card wallet in which the HKID card object is stored. Detailed operations will be discussed later.

As shown in Appendix A, basically, a HKID card object contains the following variables: *holderName, dateOfBirth, cardNumber, dateOfIssue, sex* and *photo*. By using the *setValue()* and *getValue()* methods, the variables can be set and retrieved respectively.

3.4.3.1 Adding the HKID card object

The addition of new objects to the smart card wallet requires that the party adding the object is authorized. Only the correct issuer can add the respective object. For example, only the HKSAR government can add the HKID card object to the smart card wallet. Apart from authenticating the identity of the object issuer, different authority levels are assigned as discussed before. Basically, there are four types of actions that can be performed by the object issuer and they are the addition, retrieval, modification and removal of objects inside the smart card wallet. For instance, some object issuers are allowed to add new objects to the smart card wallet but are not permitted to delete objects from the smart card wallet. In this case, the authority level of the object issuer is specified inside the digital certificate accordingly as illustrated in Figure 18.

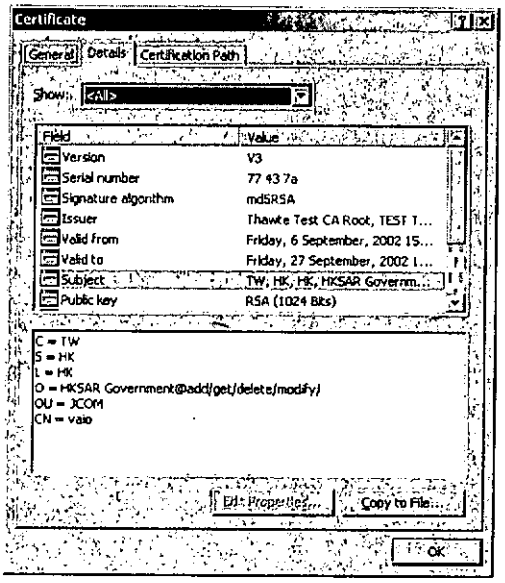


Figure 18: The definition of the authority level inside a digital certificate

Assume that a person getting our HKID card object wants to duplicate it. Even if the person can obtain the details of the object as well as the issuer's certificate (The Hong Kong SAR Government), the challenge in the challenge-response phrase cannot be decrypted because the respective private key cannot be obtained. In this case, the HKID card object is said to be non-duplicable. That person may register a new certificate from the SCCA, but the CNAME of the certificate cannot be the same with that of the Hong Kong SAR Government under the control of SCCA.

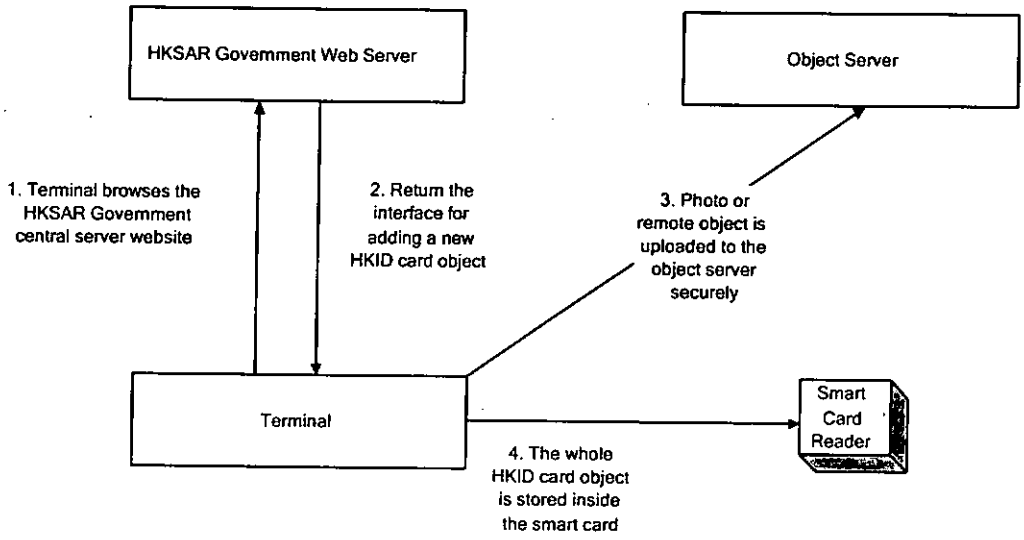


Figure 19: A general picture of adding a new HKID card object

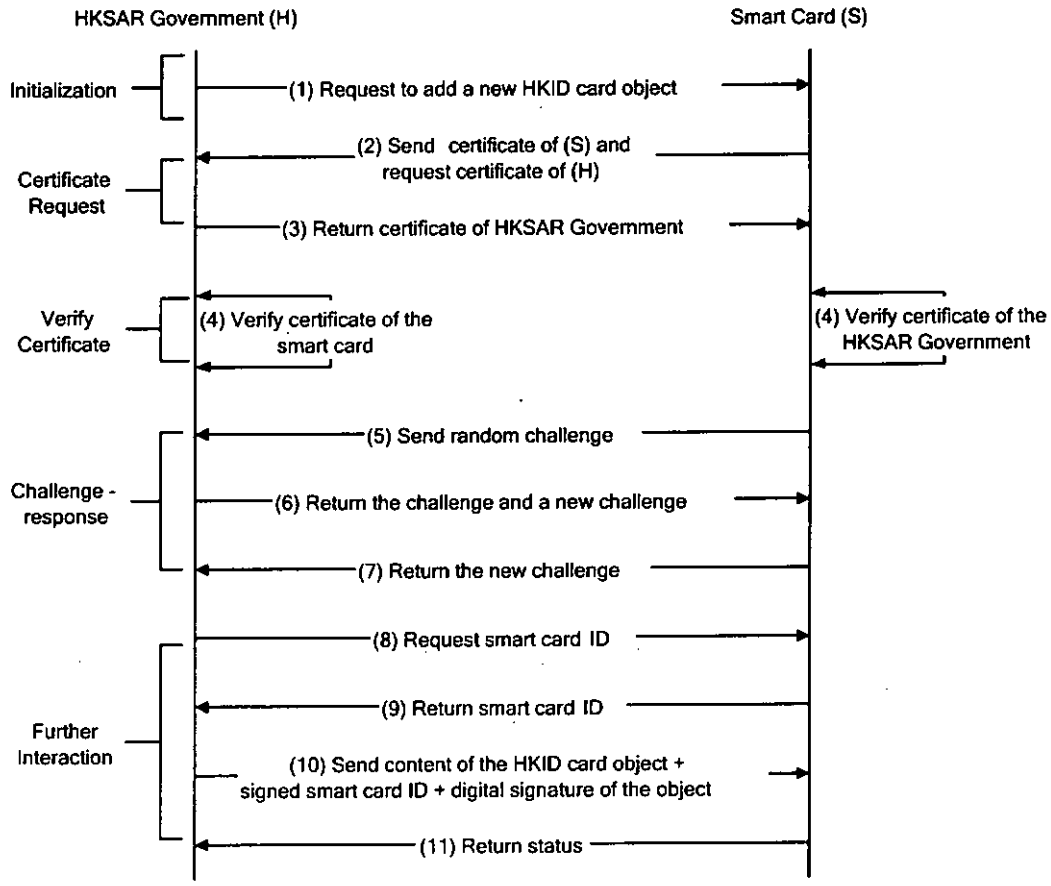


Figure 20: Protocol operation of adding a new HKID card object

Figure 19 shows the basic steps to add a new HKID card object. Firstly, the terminal browses the HKSAR Government central website to get the interface for adding a new HKID card object. In the interface, the content of the HKID card object is formulated and any remote object such as the photograph of the HKID card owner can be uploaded to the object server (e.g. through a SSL connection). Finally, the whole object is added to the smart card.

The issuer of the HKID card object, the Hong Kong SAR Government, needs to obtain a valid certificate from the SCCA before adding a new HKID card object to

the smart card. The CNAME of the HKID card company will be 'HKSAR Government'. The protocol operation of adding a new HKID card object is shown in Figure 20.

1. The Hong Kong Government issues a request telling the smart card that it wants to add a new HKID card object.
2. The smart card responds by requesting the digital certificate of the Hong Kong Government.
3. The Hong Kong Government sends its certificate to the smart card.
4. The smart card verifies the certificate by decrypting the certificate with the public key extracted from the certificate of the Smart Card Certificate Authority (SCCA). The object manager then checks to prove that the CNAME of the Hong Kong Government's certificate is 'The Hong Kong SAR Government'. Also, the object manager checks the extension field inside the digital certificate to obtain the respective authority level (i.e. which operation can be performed).
5. The smart card then generates a random number and a DES session key and sends them to the Hong Kong Government as a challenge.
6. The Hong Kong Government encrypts the challenge together with a newly generated challenge with the agreed DES session key and sends them to the smart card.
7. The smart card verifies the identity of the Hong Kong Government by decrypting the response with the DES session key. After that, the smart card sends a positive response to the terminal to notify that its identity is verified

and that the smart card is ready to communicate with the terminal using the DES session key.

8. At this stage, both the Hong Kong Government and the smart card have agreed with a common DES session key so that they can use the DES session key to encrypt the subsequent commands. Before actually adding the HKID card object, the HKSAR Government needs to request the ID of the smart card. The reason is that we need to ensure that the HKID card object can be linked to the smart card where it belongs. Details of this operation are shown below.
9. The smart card returns the smart card ID to the terminal.
10. After receiving the smart card ID, the HKSAR Government formats the whole HKID card object, the digital signature [33] on the object together with the signed smart card ID by the HKSAR Government's private key and sends them to the smart card. The digital signature on the object is to prove that the object is really issued by the HKSAR Government.
11. To add a new object, the Hong Kong Government formulates the content of the new HKID card object. To ensure that the HKID card object cannot be transferred, the ID of the smart card will be signed with the Hong Kong Government's private key and stored together with the object.

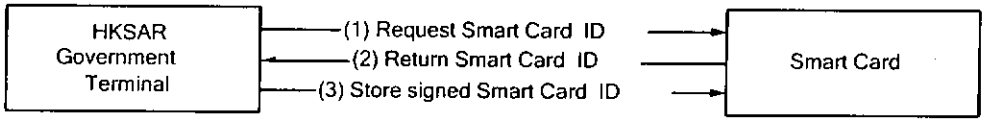


Figure 21: Storing the signed smart card's ID inside the HKID card object

1. After the initialization phase, we have the challenge and response phase. This allows both the terminal and the smart card to agree a DES session key for subsequent secured communications. To add a new HKID card object, the terminal sends a request to the smart card to retrieve the respective smart card ID. Notice that the smart card's ID is kept secret and therefore, other parties except the HKSAR Government cannot read it. Therefore, the method of getting the smart card's ID can only be invoked after verifying the identity of the HKSAR Government.
2. After the identity of the HKSAR Government is verified, the smart card's ID is retrieved and transferred to the terminal.
3. The HKSAR Government, upon receiving the smart card's ID will sign it with the HKSAR Government's private key. The signed smart card ID will be transferred to the smart card together with the HKID card object content. Therefore, when the object is retrieved later, the respective smart card ID is compared with the signed one to make sure that the object is really belonged to the respective smart card. Note also that all the messages exchanged in the operation as shown in Figure 21 is encrypted by the DES session key agreed by both parties in step 1.

The adding of a new object to the smart card wallet requires that the object issuer is authorized. In getting the object content, we have to ensure that the object is really issued by the correct and authorized object issuer. For instance, when getting the HKID card content, we have to make sure that the HKID card object is really issued by the HKSAR Government. To cater for this requirement, we make use of the digital signature method. The idea is to insert a digital signature of the object

content every time the content is added or modified. Therefore, in the last process of adding a new object to the smart card wallet, we calculate a hash value of the object content by using the MD5 algorithm and then digitally sign the hash value with the object issuer's private key as illustrated in step 10 of Figure 20. This digital signature is stored in the smart card wallet together with the object to prove that the object content is not altered and is really generated by the real object issuer.

When adding contents that cannot be stored inside a smart card due to their large sizes (e.g. the photograph of the HKID card owner), the reference link is stored instead. In order to ensure that only the correct person can view the photograph, the object will be stored in a secure remote object server. The operation will be discussed later.

3.4.3.2 Presenting the HKID card object

During the object retrieval process, the object content as well as the digital signature is retrieved. Anyone getting the object can validate the digital signature by decrypting it using the object issuer's public key extracted from the respective digital certificate. After verifying the digital signature, a hash value is calculated using all the commands sent and received by the smart card reader and the hash value is then signed using the private key of the smart card reader to produce the digital signature. The reason is to satisfy the non-repudiation requirement. Detailed operation of the logging algorithm has been discussed before. The digital signature is then sent to the terminal and is signed using the terminal's private key. The whole process is shown in Figure 22.

The verification of digital signature is required for retrieving non-duplicable objects. However, when retrieving duplicable objects, for example, the name card object, verification of the digital signature is not required.

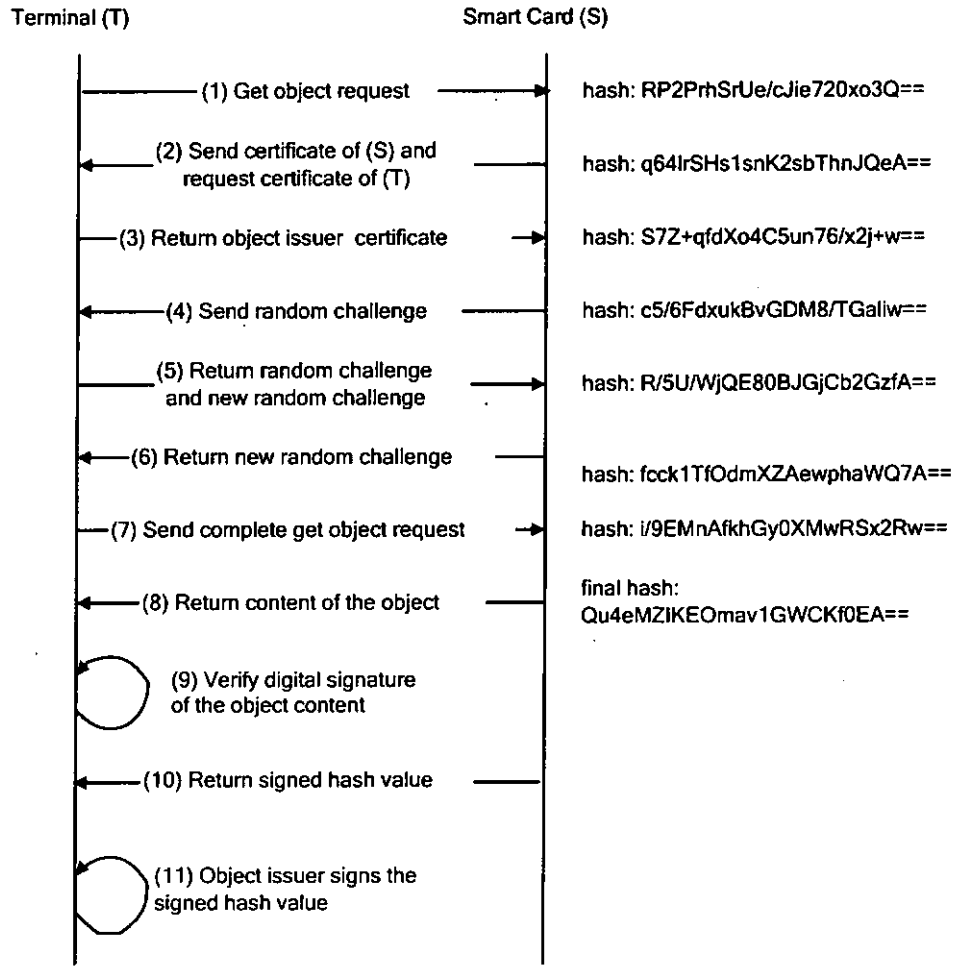


Figure 22: Protocol operation of getting object content and verifying digital signature

The operation of getting the object's content is as follows:

1. The terminal sends to the smart card the request to get the object's content.
2. The object manager inside the smart card sends a request to the terminal asking for the object issuer's digital certificate.
3. The terminal returns the object issuer's certificate to the smart card wallet.
4. The object manager verifies the certificate and sends a random challenge to the terminal.

5. The terminal returns the random challenge together with a newly generated random challenge to the smart card.
6. The smart card then returns the newly generated random challenge to the terminal.
7. The terminal verifies the new challenge and sends the get object command to the smart card wallet.
8. The smart card wallet then processes the get object command and returns the respective object content to the terminal.
9. The terminal retrieves the digital signature of the message and verifies it.
10. In steps 1 to 9, a hash value for each of the messages is calculated. The smart card then signs and sends the accumulative hash value to the terminal.
11. The terminal signs the hash value and stores it for record keeping purposes.

The security measures we have taken in using the HKID card object is shown in Table 1.

Security problems	Solutions
1. Anyone can add a new HKID card object.	• Only the party that has the correct digital certificate can add a new HKID card object. The identity of the object issuer is proved by the challenge and response protocol.
2. Information exchanged between the terminal and the smart card is not secure.	• By using the challenge and response protocol, a DES session key is

	established and agreed by both parties. Therefore, further communications can be encrypted with the DES session key.
3. When retrieving the HKID card object, we cannot make sure that the HKID card really belongs to the smart card.	• In adding the HKID card object, the smart card ID is signed by the HKSAR Government and is stored inside the smart card wallet. When retrieving the object, we compare the smart card ID and the signed smart card ID to ensure that the object belongs to the correct smart card.
4. All the messages exchanged between the terminal and the smart card cannot be traced.	• The logging algorithm, as discussed in section 3.4.2 is used to log the transactions between the terminal and the smart card

Table 1: Security measures taken in using the HKID card object

3.4.4 The credit card object

The credit card object contains all brands of credit cards, for example, Visa or Master. It functions like a physical credit card. It stores the cardholder's name, the credit card number, the expiry date and other important information. Obviously, when adding this kind of object to the smart card wallet, we have to ensure that only the authorized party can perform the operation. Therefore, the addition of the credit card object is similar to that for adding the HKID card object. When getting the credit card object, we have defined two different security levels. The basic one is for retrieving the cardholder's name, credit card number and the expiry date directly without any cryptographic operations. The secure one employs the idea of SET protocol for exchanging the cardholder's information. The reason for designing a secure protocol is that the current credit card payment system requires customers to enter their credit card numbers into the merchant's web site. This leads to a very serious problem because the sensitive information (e.g. credit card number) is exposed to the merchant. If the merchant is not truthful, it can 'steal' the credit card number and use it to conduct unauthorized transactions.

As shown in Appendix A, basically, a credit card object contains the following variables: *cardType, holderName, cardNumber, expiryDate* and *signature*. The variable *cardType* defines the brand (e.g. Visa or Master). By using the *setValue()* and *getValue()* methods, the variables can be set and retrieved respectively.

3.4.4.1 Presenting the credit card object

As discussed above, there are two modes in retrieving the credit card object, the insecure mode and the secure mode. In the insecure mode, we retrieve the content of the credit card object by sending a request to the smart card wallet to invoke the credit card object's methods to get the cardholder's name, credit card number and the expiry date.

Notice that the insecure mode resembles the traditional credit card operation. For example, when purchasing a book at an online bookstore, the buyer needs to submit an order form in which the cardholder's name, the credit card number and the expiry date are filled in. Therefore, anyone sitting in between may get all these sensitive information. Of course, the connection can be secured by using the SSL protocol to encrypt all the information sent to the merchant. But still, the credit card number is disclosed to the merchant.

The secure mode makes use of the SET protocol [7] in making credit card payment. The reason of using the SET protocol is that the merchant does not have access to the buyer's credit card number and it eliminates the need for a third party to monitor Internet credit card transactions. In addition, it involves the use of strong encryption and authentication scheme so as to ensure the privacy of buyer's information. The process in using SET in the smart card is illustrated in Figure 23 and the detailed steps are illustrated below.

1. Firstly, the buyer indicates to the merchant that he/she wants to pay by using a credit card.
2. Then, the merchant's system generates an invoice and sends it to the buyer. The buyer selects a type of credit card (VISA or MasterCard credit card) to be used.
3. Next, the buyer smart card initiates the payment process by sending a request to the merchant requesting for their RSA public key and the RSA public key of the payment gateway. The request indicates the type of credit card that the buyer will use.
4. The merchant generates a response to the request and replies back to the buyer's smart card. The response includes a unique transaction identifier generated by the merchant's system and the certificate of the merchant and the payment gateway.
5. The buyer's smart card then verifies the merchant's and payment's gateways using their digital certificate. The buyer generates two packets of information to send back to the merchant. They are the Order Information packet (OI), and the Purchase Instructions (PI) packet. OI contains the transaction identifier, brand of card being used, and the transaction date whereas PI includes the credit card number and expiration date, the purchase amount agreed to by the buyer and the description of the order. Each packet is encrypted separately. For instance, the PI is encrypted with the payment gateway's public key since the merchant should not have access to it. Next, the buyer sends the OI and PI to the merchant.
6. When the merchant receives the PI and OI, it checks the message to see if OI and PI have been altered. If not, the merchant starts the process of

requesting authorization from the merchant's acquiring bank. The merchant generates an authorization request for the credit card payment request. Included in this request is the transaction identifier that the merchant generated at the beginning of the payment process. The merchant sends to the payment gateway a message encrypted using the payment gateway's public key. This message includes the authorization request, the PI packet sent from the buyer and the merchant's digital certificate. The payment gateway then decrypts the message and all components such as the PI from the buyer. It then checks the integrity of various parts of the message.

7. The payment gateway then sends a request for payment authorization to the buyer's credit card issuer through conventional bankcard channels.
8. In response to the authorization request, the issuing bank returns an approval or denial code to the payment gateway.
9. The payment gateway then returns an authorization response message to the merchant. This message includes the issuing bank's response, an optional capture token to be used by the merchant. The payment gateway encrypts and sends the authorization response back to the merchant.
10. The merchant decrypts the authorization notice from the payment gateway. It examines the notice to find out if the request was approved or not. It then stores the authorization response and the capture token sent by the payment gateway for later use when capturing the sale. If the transaction is approved, the merchant then returns a purchase response to the buyer. This message informs the buyer that the payment has been accepted and that the respective product or service will be delivered. The buyer's terminal

processes the purchase response message and informs the buyer of the acceptance of the payment.

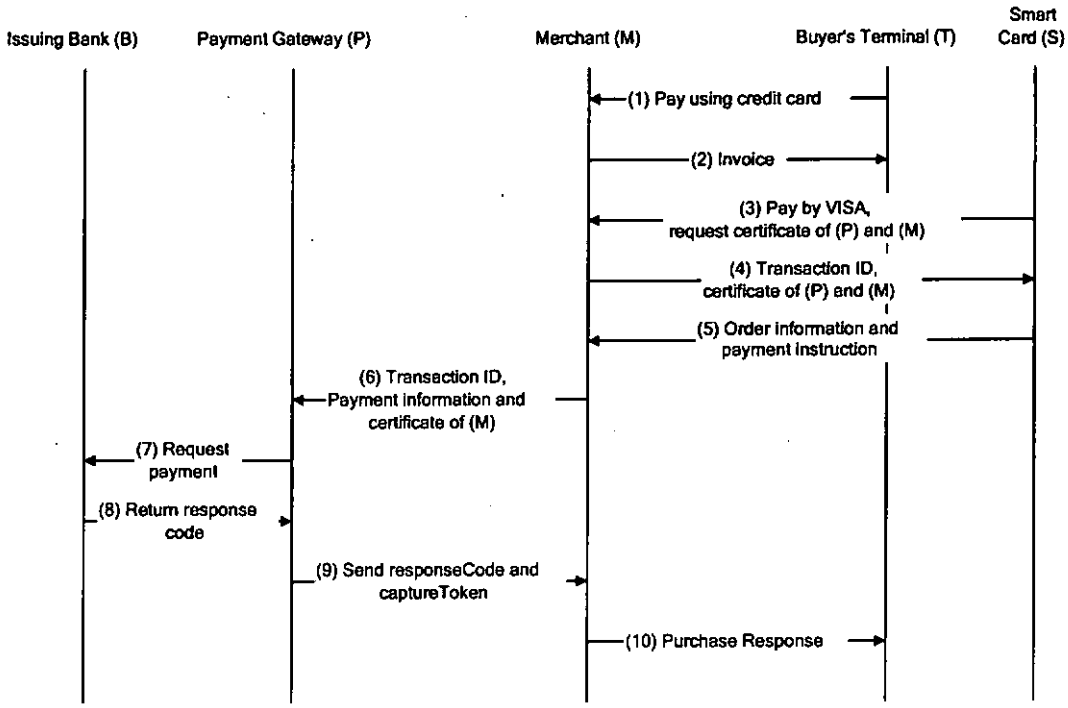


Figure 23: Using SET with smart card

The security measures we have taken in using the credit card object are shown in Table 2.

Security problems	Solutions
1. Anyone can add a new credit card object.	• Only the party that has the correct digital certificate can add a new object. The authority of the particular object issuer is defined in the certificate. The identity of the object issuer is proved by the challenge and response protocol.
2. Information exchanged between the terminal and the smart card is not secure.	• After the challenge and response protocol, a DES session key is established and agreed by both parties. Therefore, further communications are encrypted using this DES session key.
3. The credit card number, the expiry date and the card holder's name are sensitive information and should not be exposed.	• We provide two operating modes. The first one is to retrieve all the sensitive information directly. Similar to the one used by many e-commerce sites. The second mode uses the SET protocol such that all the sensitive information is protected.
4. All the messages exchanged between the terminal and the smart card cannot be traced.	• The logging algorithm, as discussed in section 3.4.2 is used to log all the transactions between the terminal and the smart card

Table 2: Security measures taken in using the credit card object

3.4.5 The e-coin Bag object

The e-coin bag object stores electronic cash for supporting micro-payments. As shown in appendix A, basically, a e-coin bag object contains the following variables: *moneyValue*. This value stores the amount of e-money inside the wallet. By using the *credit()* and *debit()* methods, the value can be incremented and deducted respectively.

3.4.5.1 Using the e-coin bag object

In a general procurement system, the consumer's bank needs to transfer money to the payment gateway. The payment gateway then formulates the electronic cash and sends it to the consumer's e-coin bag or electronic wallet for storage. When the consumer decides to buy something from a merchant, the electronic money is transferred from the consumer to the merchant. After the receiving the electronic money, the merchant delivers the product or service to the consumer. Later on, the merchant can redeem the electronic cash through the payment gateway. The payment gateway then credits the merchant's bank account accordingly.

Our aim of designing the e-coin bag object is to support anonymous off-line payment while being able to detect double spending. The credit card object, as discussed in the previous section is an on-line payment protocol using the SET algorithm whereas the e-coin bag object is off-line (i.e., it functions like our physical coin bags). Although the CAFE system can be used, the double spending handling capability of CAFE cannot detect double spending under some

circumstances. Hence, we propose an improved version of the CAFE protocol to address this deficiency.

3.4.5.1.1 Proposed anonymous off-line payment protocol

Project CAFE is well designed such that a cryptographic fallback mechanism is used to detect double spending. Recall from the previous section that the first part of the coin is read when the coin is spent the first time and that the second part is read when the coin is spent the second time. However, there exists a case where the user captures the first part of the coin C(I ⊕ R) and replays this message to the merchant. In this case, the identity of the user cannot be revealed because the second part of the coin is never sent out. Apart from this limitation, CAFE relies too much on tamper resistance devices and the role of observer built inside the microprocessor. This increases the costs of the whole payment system.

In view of this, we have design an improved version of CAFE to address the replay attack problem. In addition, our solution does not rely on tamper resistant hardware and therefore the costs can be lowered. The general flow of the proposed algorithm is shown in Figure 24.

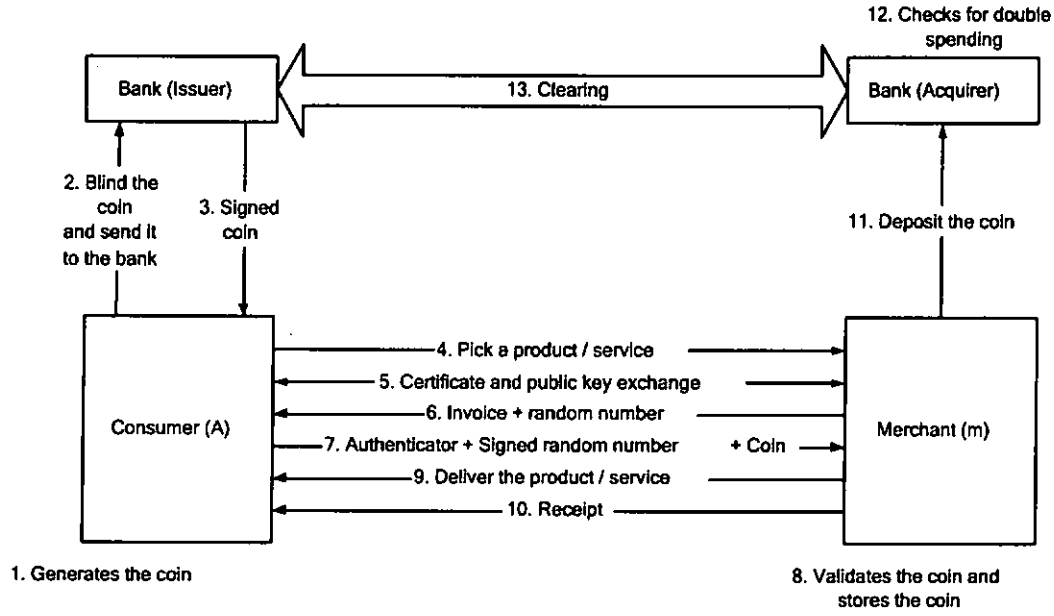


Figure 24: General flow of the proposed algorithm

The validity of the coin is proved based on the blind signature protocol [39] for issuing the coin (Figure 25). The coin contains the serial number of that coin as well as a digital signature for the respective serial number (signed by the bank). Therefore, any merchant can validate the coin by validating the corresponding digital signature. The value of the coin can be guaranteed by signing it with different private keys that represent the respective values.

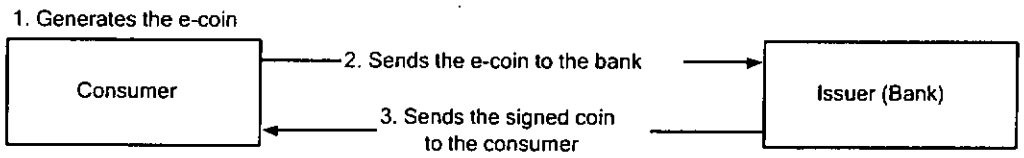


Figure 25: Blind signature algorithm

Our design follows the same algorithm downloading money to the smart card as specified in the CAFE protocol. During the payment process, the user, instead of sending C(I ⊕ R) to the merchant, sends out ($I_A \oplus R_1m \oplus Serial_{coinA}$) in which I_A is the identity of user A, R_1m is an unique random number generated by the merchant in which the prefix of R_1m is merchant specific and $Serial_{coinA}$ is the serial number of coin A. Next, ($I_A \oplus R_1m \oplus Serial_{coinA}$) is encrypted with the bank's public key to produce [$I_A \oplus R_1m \oplus Serial_{coinA}$] KeyPub_bank in which we called it the *authenticator*. KeyPub_bank indicates that we are using the public key of the bank to encrypt the message. Apart from sending the authenticator, the user also calculates the digital signature for the random number R_1m to produce [R_1m]Sig_user in which Sig_user indicates that the message is signed with the user's private key. To sum up, three components are sent from the user to the merchant as follows:

$$[I_A \oplus R_1m \oplus Serial_{coinA}] \text{ KeyPub\_bank} + [R_1m]\text{Sig\_user} + \text{coin}$$

Upon receiving the message sent by the user, the merchant can validate the coin by using the public key of the respective bank. After verifying the coin, the merchant then stores all the three components, [$I_A \oplus R_1m \oplus Serial_{coinA}$] KeyPub_bank , [$R_1m$]Sig_user and the coin for the later redemption operation. Later on, when the merchant wants to deposit the coin, both [$I_A \oplus R_1m \oplus Serial_{coinA}$] KeyPub_bank and the coin are sent to the bank. The bank, upon receiving the two components, verifies the coin using the bank's private key. If it is a valid coin, the bank will take out the serial number and match it with the database of all the spent coins. The coin is accepted only if the bank cannot find a match in the database. However, if a

match is found in the database, double spending of the coin is detected. The bank then issues a request to the merchant asking for the R_1m used in the transaction. Upon receiving R_1m, the bank can decrypt $[I_A \oplus R_1m \oplus Serial_{coinA}]$ KeyPub_bank using the bank's private key and get $[I_A \oplus R_1m \oplus Serial_{coinA}]$. Eventually, "$I_A$" can be obtained by XOR $[I_A \oplus R_1m \oplus Serial_{coinA}]$ with R_1m and $Serial_{coinA}$ again ($I_A \oplus R_1m \oplus Serial_{coinA} \oplus R_1m \oplus Serial_{coinA} = I_A$). In fact, there are several double spending scenarios: a untruthful user with truthful merchant; a truthful user with an untruthful merchant and a truthful user with two or more untruthful merchants as explained below.

3.4.5.1.1.1 Untruthful user with single truthful merchant

In the case where the user is untruthful, he/she can capture the payment message and replay it to the same merchant. As a result, the merchant will receive two payment messages from the same user with:

$[I_A \oplus R_1m \oplus Serial_{coinA}]$ KeyPub_bank + $[R_1m]$Sig_A + coinA

and

$[I_A \oplus R_1m \oplus Serial_{coinA}]$ KeyPub_bank + $[R_1m]$Sig_A + coinA

Obviously, it is not allowed because the serial number of the coin is unique and when the merchant receives the second coin with the same serial number, it can reject the payment message sent by the user. The merchant can then blacklists the user and sends his/her identity to the bank.

3.4.5.1.1.2 Untruthful user with two or more truthful merchants

In the case where the untruthful user replays the payment message to two different truthful merchants, the following occurs:

Merchant 1 sends to the bank:

$$[I_A \oplus R_1m \oplus Serial_{coinA}]\ KeyPub\_bank + coinA$$

and merchant 2 sends to the bank:

$$[I_A \oplus R_2m \oplus Serial_{coinA}]\ KeyPub\_bank + coinA$$

The bank can reveal the identity of the user by requesting both R_1m and R_2m from the merchants. Afterwards, the bank can blacklist the user.

3.4.5.1.1.3 Truthful user with single untruthful merchant

In the case where the merchant is untruthful, it can send to the bank the same coin with different authenticators.

Specifically, the merchant sends to the bank:

$$[I_A \oplus R_1m \oplus Serial_{coinA}]\ KeyPub\_bank + coinA$$

and

$$[I_B \oplus R_2m \oplus Serial_{coinA}]\ KeyPub\_bank + coinA$$

The bank, upon receiving the two coins, discovers that double spending has occurred and can immediately declare that the merchant is not truthful. As the

merchant can detect that the two coins have the same serial number, a truthful merchant should not send the previous message.

3.4.5.1.1.4 Truthful user with two or more untruthful merchants

The user sends to merchant 1:

$$[I_A \oplus R_1m \oplus Serial_{coinA}]\ KeyPub\_bank + [R_1m]Sig\_A + coinA$$

and to merchant 2:

$$[I_A \oplus R_2m \oplus Serial_{coinB}]\ KeyPub\_bank + [R_2m]Sig\_A + coinB$$

Now, merchant 1 gives the coin to merchant 2 and now, merchant 2 sends to the bank:

$$[I_A \oplus R_2m \oplus Serial_{coinB}]\ KeyPub\_bank + coinB$$

$$[I_A \oplus R_1m \oplus Serial_{coinA}]\ KeyPub\_bank + coinA$$

and merchant 1 sends to the bank:

$$[I_A \oplus R_1m \oplus Serial_{coinA}]\ KeyPub\_bank + coinA$$

Now, the bank detects that coin A is doubly spent and asks both merchants to provide the random number. Merchant 2 has two choices: the first one is to give the bank R_1m and the second choice is to give the bank a 'fake' random number. If merchant 2 gives a fake random number or a wrong authenticator to the bank, the bank cannot reveal the identity of the user A and cannot come up with a unique I_A from both merchants. Therefore, the bank can conclude that the merchant is untruthful. If merchant 2 gives a real random number to the bank, that is R_1m, then, the bank will notice that merchant 2 is using a wrong random number because the prefix of that random number belongs to merchant 1 and therefore, the bank can declare that merchant 1 is untruthful and blacklist the merchant. Notice that if a

merchant cheats the bank by using a correct authenticator from another consumer, the coin will not be valid because the authenticator has the serial number that ties with the coin.

3.4.5.1.1.5 Untruthful user with one or more untruthful merchants

If both parties are untruthful, then the bank can still detect double spending and either the merchant or the user is declared to be untruthful and the bank will blacklist both of them.

3.4.5.1.2 Applying the proposed protocol in using the e-coin bag object

The installation procedures for the e-coin bag object are similar to that for adding the HKID card object. To use the e-coin bag object, it involves the addition of e-coin to the e-coin bag (credit), the retrieval and transfer of e-coin (payment) and the deposit of the e-coin back to the bank. To use the e-coin bag object, we follow the CAFE [37] payment algorithm, which uses an anonymous off-line protocol for electronic payment. The payment protocol is anonymous such that the bank can never know the owner of a particular coin on the condition that the coin is not double spent. Due to the off-line characteristic, both the payer and the payee are not required to contact the bank or any third parties during the payment process.

To obtain the e-coin from the bank, the user first creates a coin with a unique serial number and sends the coin to the bank. The bank, upon receiving the coin, will digitally sign the signed coin and then return it to the user. Notice that in this stage,

the bank does not know the serial number of the coin and therefore, cannot link a user to the serial number of the coin. Upon receiving the coin, the user verifies the bank's signature on the coin by using the bank's public key and then stores the coin in the e-coin bag.

To initiate a payment, firstly, the merchant terminal displays the amount of money that is going to be deducted from the consumer's e-coin bag. The consumer checks the displayed amount and agrees on the transaction by pressing a button on the merchant's terminal. After that, the public keys of both the merchant and the consumer are exchanged accordingly. Then, the merchant sends to the user an invoice stating the amount going to be deducted from the e-coin bag as well as a signed R_1m. The message is encrypted using the merchant's public key to ensure privacy. After receiving this message, the user decrypts it using the merchant's public key and checks the invoice for the agreed amount as well as checking the digital signature on R_1m. If everything is correct, the user takes out the appropriate amount of e-coin from the e-coin bag and sends to the merchant the coin as well as the encrypted $[I_A \oplus R_1m \oplus Serial_{coinA}]$ with the bank's public key. The whole message is encrypted using the merchant's public key to ensure privacy. Next, when the merchant receives the message, the coin is checked for validity with the bank's public key. The merchant will also check in its database for the serial number of the coin. If the merchant finds a match in its database, the payment is refused and the user is blacklisted. Otherwise, the merchant can store the authenticator ($[I_A \oplus R_1m \oplus Serial_{coinA}]$ KeyPub_bank) and the coin for later redemption.

To deposit the e-coin, the merchant sends to the bank the coin and the authenticator. The bank then checks the validity of the coin by decrypting it with its own public key. After the coin is validated, the coin is checked for double spending. If no double spending is detected, the bank accepts the coin and credits the merchant's bank account accordingly.

The security measures we have taken in using the e-coin bag object is shown in Table 3.

Security problems	Solutions
1. Anyone can add a new e-coin bag object.	• Only the party that has the correct digital certificate can add a new object
2. Information exchanged between the terminal and the smart card is not secure.	• By using the challenge and response protocol, a DES session key is established and agreed in both parties. Further communications are encrypted using this DES session key.
3. Double spending cannot be detected	• We have proposed a new protocol that detects double spending
4. All the messages exchanged between the terminal and the smart card cannot be traced.	• The logging algorithm, as discussed in section 3.4.2 can be used to log all the transactions between the terminal and the smart card

Table 3: Security measures taken in using the e-coin bag object

3.4.6 The name card object

The name card object contains less confidential and sensitive information than the previous objects. Lie a pyhsical name card, it contains your company name, company logo, company address, your name, job title, qualifications, email address and your contact number. Note that as people can distribute your name card to other people, a name card object is an example of the duplicable and transferable object.

As shown in appendix A, basically, a name card object contains the following variables: *name, title, email, mobilePhone, companyName, companyAddress, companyPhone, companyFax* and *companyWebSite*. By using the *getValue()* and *setValue()* methods, the variables can be retrieved and set, respectively.

The design purpose of the name card object is to make it as flexible as possible. Therefore, apart from the standard information stored inside a normal name card, we can add other special and dynamic contents. For instance, we can store multimedia contents such as voice, video and animation. Furthermore, we can store the link to our resume or a link to a video introducing ourselves inside the name card object. When other people receive our name card object, the link can be retrieved and the respective information can be obtained using a web browser. For the reason that the name card object is flexible and it is used to store public information, we do not need to care about the security issues.

A name card object is a duplicable object and therefore the procedure for adding it to a smart card wallet is quite simple because it involves the lowest level of authentication. We allow other person to duplicate this type of object and therefore, it is not required to check the object issuer before interacting with the object. The smart card owner can add a new duplicable object by accessing the smart card wallet with his/her PIN. After logging in, the card owner can freely add a new duplicable object to the smart card wallet.

As have been mentioned before, the name card object carries the personal information of a particular person. For instance, we may exchange our name card with another person to show our identity. In a practical situation, it is nice to synchronize the name card object with the address book of the PIM (Personal Information Management) softwares inside our laptop or desktop computer. It can be easily achieved because all the information stored inside the name card object is presented in the XML format. As XML documents are self-described, we can export the contents of our name card object according to the format required by the PIM softwares.

Taking the Microsoft Outlook Express as an example, we can import our name card into the Outlook Express's address book through the use of Comma Separated Values (CSV) files. A simple CSV file that can be imported into Outlook Express is shown in Figure 26. With the knowledge of the name of different fields in the CSV file, it is easy and straighforward to export the XML document to the corresponding CSV file.

```
Name,Nickname,E-mail Address,Home City,Home Country/Region,Home
Phone,Company,Job Title


cswmsin,dereksin,cswmsin@comp.polyu.edu.hk,Hong
Kong,China,(852)99998888,Macroview Telecom Ltd.,Programmer
```

Figure 26: A CSV file

3.5 Object Server (OS)

An object server is a remote server for storing the objects that are too large to be stored inside the smart card. For example, when storing a HKID card object, the photograph may be too large to fit into the smart card wallet. Only the object reference (presented as a hyperlink) is added to the wallet and the real photograph is stored in the object server.

The object server also provides an interface for the smart card to access the remote objects. To prevent others from capturing the object contents between the terminal and the remote server, the connection needs to be secure. To cater for this requirement, the SSL protocol is used when connecting from the terminal to the remote server. When using SSL, the information transferred between the terminal and the object server will be encrypted. Therefore, no one can obtain the object content when capturing the data in between.

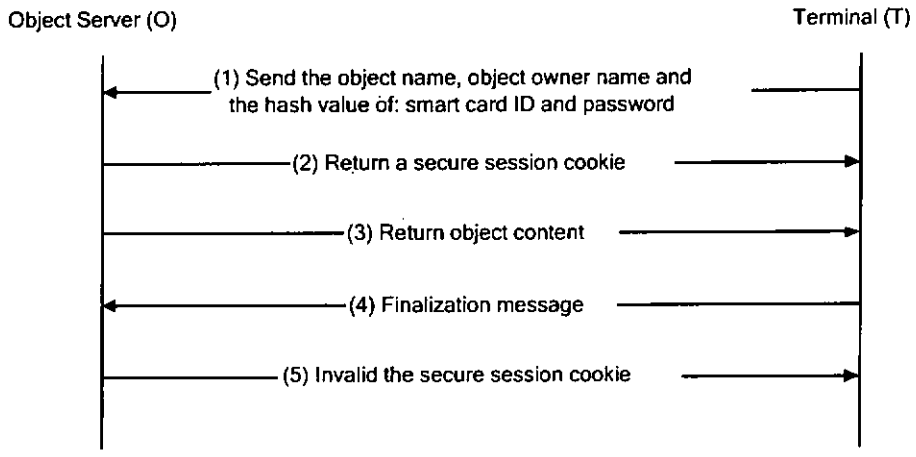


Figure 27: Mechanism of retrieving the remote object

Apart from SSL protocol to ensure privacy, we also need to ensure that only the correct smart card owner can view the correct objects, we have introduced a special authentication algorithm such that only the correct owner can view the respective objects. Every time, when the owner wants to read or access the remote object, the name of the owner and the object name have to be provided. In addition, a hash value of the owner's password and the smart card ID has to be provided for authentication purpose. Although other people may know the name of the object owner and the respective object name, they cannot get the object content because they do not know the respective password and the smart card ID. Figure 27 shows the mechanism when an authorized smart card owner wants to access the remote object. The detailed operations are explained below:

1. The terminal tells the object server which object is to be retrieved by sending the object's name and the object's owner name. In addition, a hash value of the password and smart card ID is sent to the object server for authentication. Notice that all the values sent to the object server is protected by the SSL protocol.
2. The object server checks the object for the correctness of its respective hash value. If the hash value sent by the terminal matches with the one stored in the object server, the object server sends a secure cookie to the terminal's browser.
3. Once the terminal receives the secure cookie, it can issue further request to retrieve the same object without further authentication. In the mean time, the object server can send the object content to the terminal.

4. After successfully retrieve the object content, the terminal sends to the object server a finalization message. The function of this message is to instruct the object server to make the secure cookie invalid.
5. The object server, after receiving the finalization message, makes the secure cookie stored in the terminal's browser invalid.

3.6 Java Card Object Manager (JCOM)

A prototype system called JCOM has been developed to illustrate the basic operation of the object manager (OM) for adding new objects, retrieving object contents, modifying the object contents and deleting the objects. Figure 28 shows the main screen for JCOM.

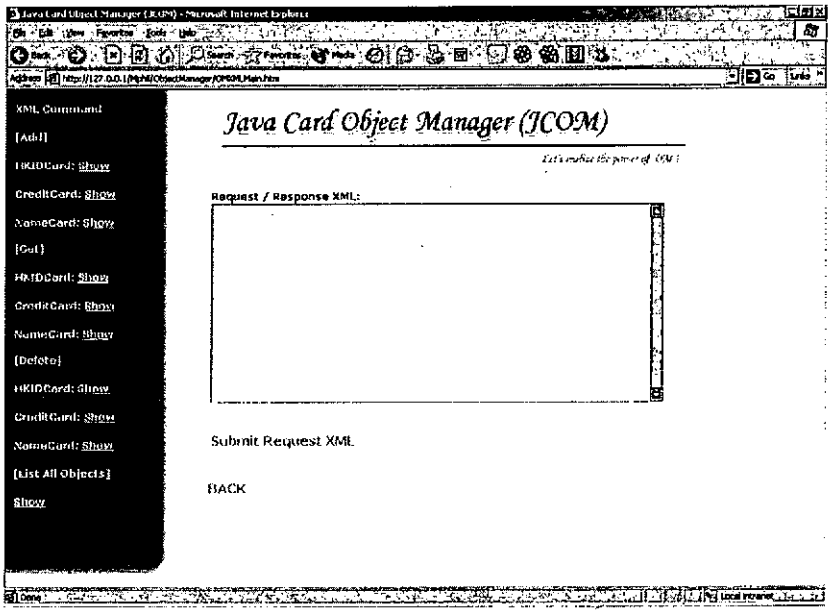


Figure 28: The main screen of JCOM

Basically, the browser and the OM communicate through the request and response commands expressed in XML. Users can type in the XML command for performing different operations as shown in Figure 29. For example, when adding a new object, we can call the *addObject()* method in the OM in order to add a new object to the smart card wallet. All methods supported by the OM are shown in Appendix B.

3.6.1 Using JCOM

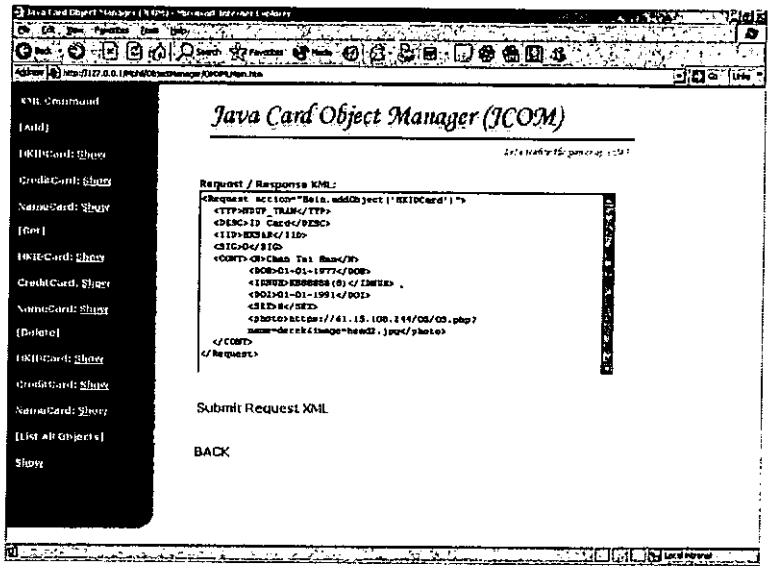


Figure 29: XML request for adding a new object

To add a new object, firstly, we provide the XML command. On the left-hand side of the JCOM, under [Add], we can see different kinds of objects. For example, if we want to add a new name card, we can click on the 'Show' hyperlink located next to the object name, the respective XML command is then shown on the right-hand side.

In the request/response XML text box, the information is displayed. To add the object, we simply click the 'Submit Request XML' hyperlink and then an alert will be popped up asking for confirmation. After confirmation, JCOM will communicate with the smart card and send the respective XML command to the OM inside the smart card. The OM then performs the operation and finally a response is returned as shown in Figure 30.

```
<Response>
        Object Added Successfully!!
</Response>
```

Figure 30: Return status after adding a new object successfully

The process of retrieving the object is similar. We first input the '*getObject()*' command and click on the '*Submit Request XML*' hyperlink. To delete an object, we can simply provide the '*deleteObject()*' command. To list all the available objects stored inside the smart card, we can issue the '*listAllObjects()*' command. In summary, all the commands are shown in Figure 31.

```
<Request action="OMClient.getObject('NameCard1')">
</Request>
```

```
<Request action="OMClient.deleteObject('NameCard1')">
</Request>
```

```
<Request action="OMClient.listAllObjects()">
</Request>
```

Figure 31: XML commands for adding/deleting/listing all objects inside the smart card wallet

As illustrated in Figure 31, to get a name card object of the name '*NameCard1*', we provide the XML request and send it to the smart card through the *getObject()* method. The *OMClient.getObject('NameCard1')* action is to invoke the *getObject()* method in the class *OMClient*. *OMClient* is a Java applet running in the JVM of the

terminal's browser. It is responsible for establishing a connection between the terminal computer and the smart card. By invoking this method, the class *OMClient* sends the respective request to the smart card through the methods as defined in the class *DMI*. The OM API is shown in Appendix C.

3.6.2 Using JCOM to visualize the HKID card object

To visualize the HKID card object, we have designed a simple interface. After clicking the 'Get HKID card object' hyperlink, the XML command for getting the object content will be issued to the smart card wallet. The OM will process the '*addObject()*' command and return the HKID card object to the terminal. Upon receiving the response from OM, the browser will parse the XML response and display the respective information as shown in Figure 33. As the photograph of a HKID card object is stored in a remote server. Securely, we use the SSL protocol for retrieving the photograph. Therefore, a security alert is displayed as shown in Figure 32 before the photograph is retrieved from the remote server.

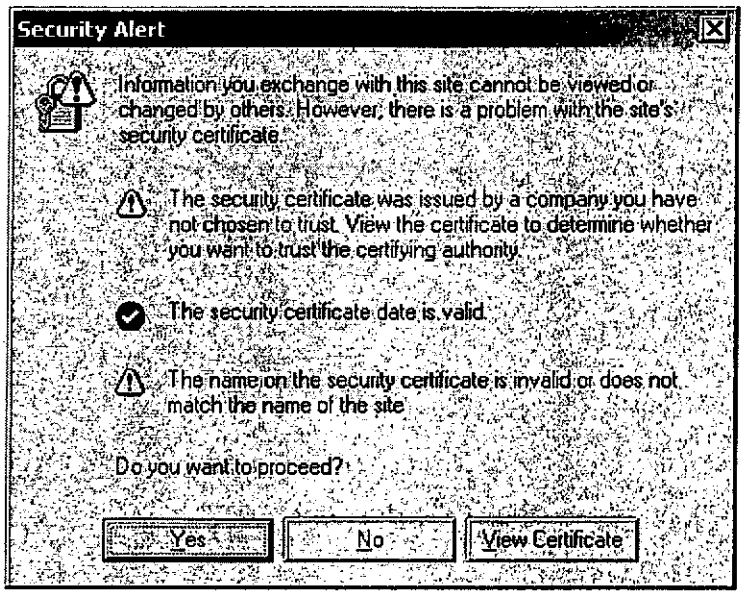


Figure 32: Secure channel for retrieving the photograph of a HKID card object

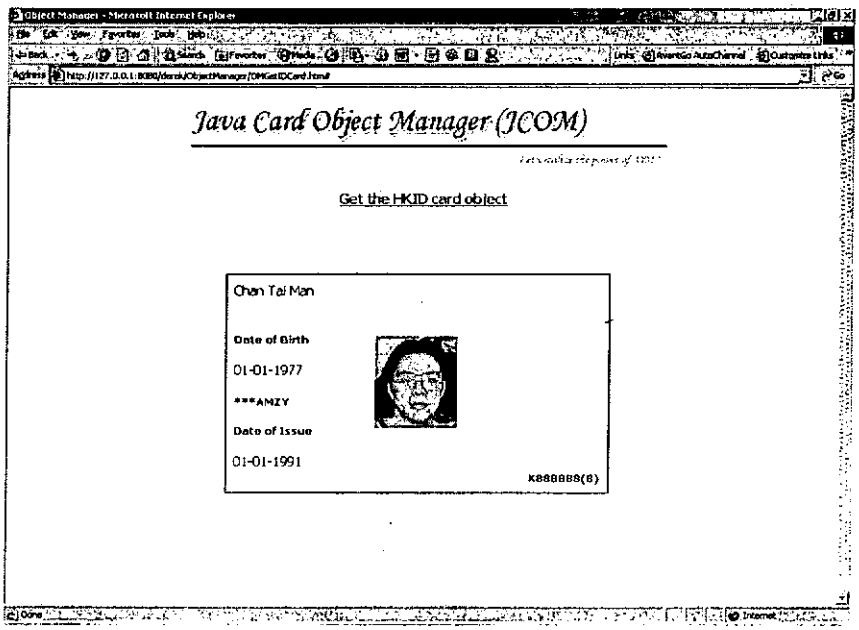


Figure 33: Visualizing a HKID card object retrieved using JCOM

CHAPTER 4: Lightweight XML (LXML)

As mentioned before, XML is well suited for storing data in the object-oriented smart card wallet because it allows users to define and modify the data structure easily by means of tags. However unlike general computers, which have abundant memory, smart cards are extremely memory-limited. Hence in our opinion, a new form of XML is required to cater for this requirement. In this section, we introduce the novel LXML method [29]. The major design objective is to reduce the overhead of XML while maintaining its capabilities and functions. In particular, LXML is backward compatible with XML such that it can be transformed back to the original XML document. The basic idea is to employ special navigation tags in a LXML document so that each data entry can be linked to the respective element or attribute in the Lightweight DTD (LDTD) file or the document object tree. By storing the navigation tags instead of the original XML tags, the data file size can be greatly reduced as shown later.

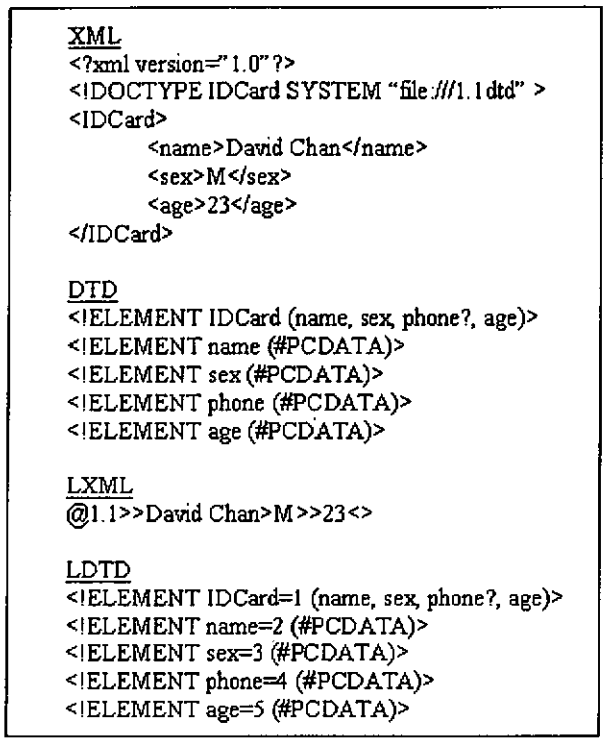


Figure 34: The original XML, DTD, LXML and LDTD files for the simple example

Let us first explain the idea of LXML by means of a simple example. Consider that we want to store a simple identity card in the smart card wallet. Its XML file and the corresponding DTD are as shown in Figure 34. Note that there is an optional element (i.e. "phone"). Figure 34 also shows the LXML document and the corresponding LDTD file. It can be seen that the LDTD file is formed by simply assigning numbers to the original DTD file. The numbers are for identification purpose (i.e. to identify the elements or attributes in the document object tree). In this simple example, the numbers are assigned sequentially when going through the elements one by one (or travelling down the document object tree). For example, the following definition:

<!ELEMENT IDCard=1 (name, sex, phone?, age)>

means that the element IDCard is assigned a number of 1. With the numbers assigned, the data entries can be linked to their respective elements. The same procedure can also be applied for assigning numbers to attributes. In the LXML document, special navigation tags are defined to tell the parser how to link each data entry to the corresponding element. Specifically, the ">" tag (or the forward tag) tells the parser to move to the next element/attribute. Imagine that there is a pointer whose movement is controlled by the navigation tags. Note that more tags will be defined in the next section. Initially the pointer is at the root element (i.e. IDCard) with a number of 1. After the first ">" tag is read, the pointer moves to 2 or the second element (i.e. Name). The data for this element (i.e. "David Chan") is read. Then the pointer moves to element 3 as specified by the next ">" tag and the data entry is read. After this element, there are two ">" tags. Hence the pointer moves to element 5 rather than element 4. This is because element 4 is optional and it has no data, so it is skipped. The process continues until the "<>" tag (or the end tag) is reached. This tag signals the end of the LXML document.

4.1 Specific syntax rules and a general LXML example

In the last section, we have presented a simple example to illustrate the basic operation of LXML. In this section, we present the syntax rules and a general example. By default, each element has one instance only. In a DTD file, it is also possible to specify how many times an element can repeat (i.e. multiple instances) [35]. LXML/LDTD needs to cater for this requirement. For example, consider the following elements:

<!ELEMENT Person (Name, HomeAddress, PhoneNo)>

In this case, we can number the elements Person, Name, HomeAddress and PhoneNo as say 1, 2, 3 and 4, respectively and use the forward tag (>) to identify all the elements without any problem. However if we allow the user to specify multiple addresses as defined in the DTD below:

<!ELEMENT Person (Name, HomeAddress*, PhoneNo)>,

the situation is different. In this case, we cannot simply use the forward tag to identify all the elements: Person, Name, HomeAddress and PhoneNo because there may be multiple addresses. To resolve this issue, a new navigation tag called the "separator" (|) is introduced. As shown in the later example, it is used to specify multiple instances. In summary, the full set of syntax rules for a LXML document is given as follows:

- @ indicates the start of LXML document followed by the DTD code to be used
- <n> indicates moving to the element or attribute with number 'n'
- <+n> indicates moving forward by n
- <-n> indicates moving backward by n
- > indicates moving forward to the next element or attribute (i.e. same as <+1>)
- | indicates repeating the current element because it has multiple instances
- < indicates moving backward to the previous element or attribute
- <> signals the end of the LXML document
- \ indicates that the next character is a real data character

Inspired by a XML document in [35] (chapter 9), the following shows a general XML example incorporating the above requirements/syntax rules. The LDTD file and the corresponding XML document are shown in Figure 35 and Figure 36, respectively.

By reading the LDTD file, the original XML document can be converted to the LXML document as shown in Figure 37.

The main points for forming the LXML file are explained as follows. Initially the navigation pointer is at the first element. After moving forward by two steps (i.e. >>), the first data element is reached (i.e. element 2).

```
<!ELEMENT IDCard=1(Cardholder)>
<!ELEMENT Cardholder (Name, Othername+, TelephoneNo*, Company*, HomeAddress?) >
<!ATTLIST Cardholder
        resident=2 (YES|NO) #REQUIRED
>
<!ELEMENT Name(Firstname , Familyname) >
<!ATTLIST Name
        title=3 (Ms.|Mr.) #IMPLIED
>
<!ELEMENT Firstname=4 (#PCDATA)>
<!ELEMENT Familyname=5 (#PCDATA)>

<!ELEMENT Othername=6 (#PCDATA)>
<!ELEMENT TelephoneNo=7 (#PCDATA)>
<!ELEMENT Company (CompanyName, CompanyAddress, CompanyEmail)>
<!ATTLIST Company
        type=8 CDATA #IMPLIED
>
<!ELEMENT CompanyName=9 (#PCDATA)>
<!ELEMENT CompanyAddress=10 (#PCDATA)>
<!ELEMENT CompanyEmail=11 (#PCDATA)>
<!ELEMENT CompanyBranch (BranchAddress, BranchTel, BranchFax)>
<!ELEMENT BranchAddress=12 (#PCDATA)>
<!ELEMENT BranchTel=13 (#PCDATA)>
<!ELEMENT BranchFax=14 (#PCDATA)>
<!ELEMENT HomeAddress=15 (#PCDATA)>
```

Figure 35: LDTD for the general example

```
<?xml version="1.0"?>
<!DOCTYPE IDCard SYSTEM "file:///1.dtd">
<IDCard>
        <Cardholder resident="YES">
                <Name title="Mr.">
                        <Firstname>Derek</Firstname>
                        <Familyname>Sin</Familyname>
                </Name>
                <Othername>|Derek|</Othername>
                <Company type="Software Development">
                        <CompanyName>Macroview Telecom</CompanyName>
                        <CompanyAddress>
                                Flat 2222, Block 22, Company Estate, HK
                        </CompanyAddress>
                        <CompanyEmail>abc@macroview.com</CompanyEmail>
                        <CompanyBranch>
                                <BranchAddress>
                                        Flat 3333, Block 33, Happy Center, HK
                                </BranchAddress>
                                <BranchTel>(852)8889999</BranchTel>
                                <BranchFax>(852)8888888</BranchFax>
                        </CompanyBranch>
                </Company>
                <TelephoneNo>(852)99999999</TelephoneNo>
                <TelephoneNo>(852)55555555</TelephoneNo>
                <HomeAddress>
                        Flat 1111, Block 11, 111 Estate, HK
                </HomeAddress>
        </Cardholder>
</IDCard>
```

Figure 36: The XML data file for the general LXML example

Alternatively, we can move to this element by applying <2>, which uses the same number of characters. Note that it is sometimes better to move to a specific number directly if less number of tags can be used. Furthermore we can use <+2> to achieve the same purpose but obviously it is not preferred since it requires more characters. At the element TelephoneNo, the separator | is used because it has more than one instance. Finally the <> tag indicates the end of the LXML document. Note also that for the element Othername, we need to specify that | represents a data character (i.e. not a control character) by using the escape tag \.

```
@1>>YES>Mr.>Derek>Sin>\|Derek\|>(852)99999999|(852)55555555>Software
Development>Macroview Telecom>Flat 2222, Block 22, Company Estate, HK>abc@macroview.com>
Flat 3333, Block 33, Happy Center, HK >(852)8889999>(852)8888888>Flat 1111, Block 11, 111 Estate,
HK<>
```

Figure 37: Resultant LXML document

We are going to compare the performance of four different approaches for storing data inside a smart card wallet, namely standard XML, compressed XML, agent and LXML. The focus is on evaluating the required file size and storage/retrieval time.

4.2 Comparisons on different approaches in storing objects

4.2.1 XML

In this method, the data XML file is stored directly in the smart card. As shown in Figure 38, after the XML file is filled in with data, the file is transferred to the smart card. The OM then stores the file in the corresponding object as shown in the previous section. To lower the data storage requirement, the DTD file is usually kept outside the smart card as specified by the URL in the XML file. To read the XML data file, the OM transfers the XML data file to the Web browser of the terminal accordingly as shown by the dotted line in the figure. The advantage of this method is that it is simple to use and is compatible with the well-established XML standard. However as XML is not designed for storing data in memory-limited storage devices, the solution is not optimized. In fact, storing the XML tags requires a lot of overheads.

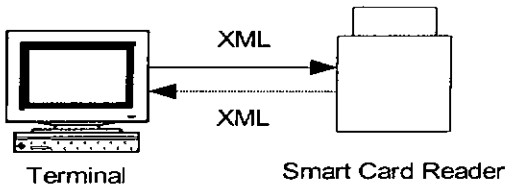


Figure 38: XML approach

4.2.2 Compressed XML

A straightforward way to lower the storage requirement is to store a compressed rather than plain XML data file inside the smart card. As shown in Figure 39, the XML data file is first compressed using the WinZip program before being stored inside the smart card. The retrieval process is shown by the dotted line. In this case, the compressed data file is read and decompressed to obtain the original XML data file for display by the browser. Again, the DTD file is kept outside the smart card.

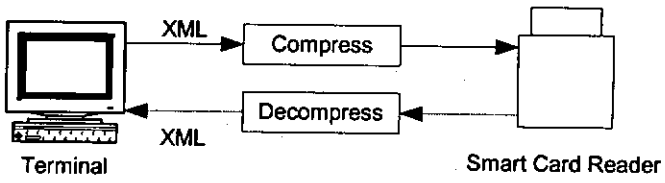


Figure 39: Compressed XML approach

4.2.3 Agent

In this method, the smart card only maintains an agent object. The original object content is kept in an object server on the Internet. Functioning like a proxy server, the agent obtains the object content from the object server whenever required. This is inspired by a similar approach presented in [36]. To ensure content integrity, a message digest is computed and stored in the agent object. As shown in Figure 40, the agent object with the message digest is kept in the smart card. Here we assume that special tags are defined for this purpose. When the object is retrieved, the agent object is first obtained from the smart card. From the agent object, the terminal can determine where (i.e. the URL) the object content is stored. Having obtained the object content, the terminal computes the message digest and then compares it with the message digest specified in the agent object. The terminal will

not display the object content unless the two message digests are the same. This operation verifies that the object content belongs to the specified object originally kept inside the smart card. Depending on the user requirement, the above method can be extended to ensure content confidentiality. In this case, the object content is encrypted by a secret key before being transferred to the object server. The secret key is then kept in the agent object. During the retrieval process, the terminal obtains the secret key from the agent object for decrypting the object content. Furthermore the connection between the terminal and the object server can be protected by the secure socket layer (SSL) protocol. The advantage of the agent method is that very large objects can be stored because the size of the agent is almost independent of that of the object content. However more processing overhead is needed. Note also that this method only works if the terminal is connected to the Internet. That means, unlike other methods in this section, it does not work with standalone terminals (i.e. it only works with networked terminals).

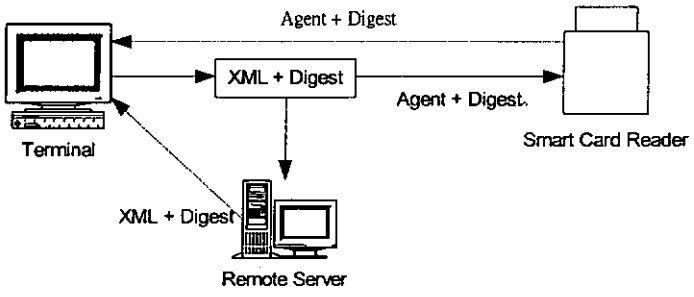


Figure 40: Agent approach

4.2.4 LXML

Finally, data can be stored in a smart card using the proposed LXML method. As shown in Figure 41, the original XML data file is first converted to the LXML data file at the terminal based on the corresponding LDTD file. It is assumed that the

LDTD file is available and stored in the terminal or elsewhere in the network. After conversion, the LXML data file is stored in the smart card. To retrieve the content, the LXML data file is obtained from the OM and then converted to the XML data file by using the corresponding LDTD file. Finally the XML file is displayed by the terminal accordingly. The major advantage of the LXML method is that it can reduce the overhead introduced by XML tags. However, more processing is needed. Figure 42 shows the data files (object content) stored inside the smart card by the four approaches.

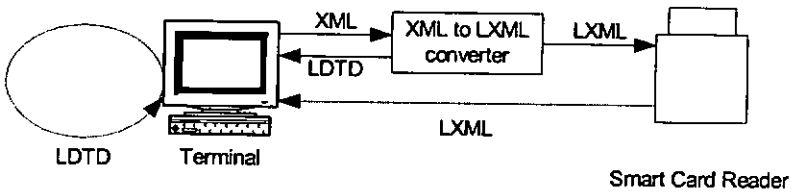


Figure 41: LXML approach

XML:
<NAME>Chan Tai Man</NAME><BIRTH>01-01-1977</BIRTH><IDNUM>K888888(8)</IDNUM><ISSUEDATE>01-01-1991</ISSUEDATE><ADDRESS>Flat 1111, Block 11, Happy Estate, Causeway Bay, Hong Kong</ADDRESS><SEX>M</SEX><TITLE>Senior Programmer</TITLE><COMPANYNAME>Macroview Telecom Ltd.</COMPANYNAME><COMPANYADDR>Flat 2222, Block 22, Happy Industrial Center, Hong Kong</COMPANYADDR>

LXML
@1>>Chan Tai Man>01-01-1977>K888888(8)>01-01-1991>Flat 1111, Block 11, Happy Estate, Causeway Bay, Hong Kong>M>Senior Programmer>Macroview Telecom Ltd.>Flat 2222, Block 22, Happy Industrial Center, Hong Kong<>

Compressed XML (binary data shown in Base64 encoding here)
H4sIAAAAAAAAAFVP72vCMBD9V+7jBm5Z90WF19AfGRZNJ6bC9vGowZW1iaRx0v9+qaWijyM53su7e8EiloKnP2SgpBokGWRXCpN8V674W/QSKIrO58hGBvOs2Eu+XlzxtHhGNjKYK7UXWVyKm2sZBfHGYpxIO6EU/2jlQxQwg6Sx1S8M3YpOpx5E58nrGaR07vSFekioD5o1R1iHA9k0A5X44hLZcGGZIxvBITa1dbBI9uiobbVDNgqYfsptXHxfPyapcvav1hcodeMr28LGH16R3b+ZDMOyMe17wJR26Me0uTmcO+9qeiDVxmv3EPV+yD8UBfrfaAEAAA==

Agent
<AGENT><TYPE>HKID Card</TYPE><object xlink:type='simple' href='http://www.myobject.com/user001/object01'>My HKID Card</object><digest>Ea2KZIKHXmvJJuoGGETREg==</digest></AGENT>

Figure 42: Data files stored inside the smart card wallet by the four different approaches

We have conducted an experiment to compare the four methods in terms of storing time, retrieving time and required data memory. As shown in Figure 43 the XML method requires the largest memory. Besides storing the data, we need to store the tags (i.e. the overhead). While the overhead is insignificant for general computers, it is very significant for smart cards because of their memory-limited nature. The LXML method is designed to address this overhead problem. As the data files are larger, it is found that the XML method also requires a slightly larger storing/retrieving time.

The compressed XML method requires the second largest memory. Note that in general this method is only effective if the data file contains much redundant data such as repeated characters etc. In fact, data compression may not work well for small data files because the compression process may generate overhead data as well. Furthermore data compression does not address the tag overhead issue as discussed above. It is found that although compression/decompression requires additional processing time, the time is rather insignificant compared to the processing time to retrieve and display the file. Hence the overall storing/retrieving time for the compressed XML method is slightly less than that for the XML method.

	Storing time (seconds)	Retrieving time (seconds)	Memory requirement (bytes)
XML	10.38	10.55	2702
Compressed XML	9.37	9.10	2114
Agent	7.12 (smart card) + 4.56 (remote server) = 11.68	8.59 (smart card) + 4.80 (remote server) = 13.39	971
LXML	6.50	7.08	1270

Figure 43: Comparison of the four data storage approaches

As expected, the agent method requires the least memory because the object content can be kept elsewhere on the Internet. For this method, the memory requirement is almost fixed (i.e. less dependent on the content size). However the agent method requires more time to store and retrieve the object content as shown in the table. This time depends largely on the network condition as the Internet only provides best-effort service. In our test, the round-trip network delay was about 0.6 seconds and the total time to store an object in the remote server was 4.56 seconds (including the transfer time and processing time). For simplicity, we have only linked to the object content directly without comparing the message digests so the result actually reflects the best-possible time.

In terms of memory requirement, the LXML method ranks second after the agent method. However it is more flexible than the agent method because it can be used for standalone terminals provided that the LDTD is available in the terminals. Among the four methods, the LXML method requires the least storing and retrieving time. Although more processing is needed, the data file is smaller, which results in better overall storing / retrieving time. The above experiment illustrates the attractiveness of the LXML method.

To demonstrate the functionality of LXML, we have developed an XML to LXML converter (by using Java) as shown in Figure 44. The function of the converter is to convert an XML document and the corresponding DTD document into the LXML and LDTD documents, respectively.

To convert an XML document into an LXML document, we type "*java tolxml file.xml file.dtd*" where *tolxml* is the Java program. The files *file.xml* and *file.dtd* are the XML file and DTD file that we are going to convert. After typing the command, the LXML document and the LDTD document are generated accordingly as shown in the figure.

The LXML document generated is stored into the smart card whereas the LDTD document generated is stored either inside the smart card or the terminal.

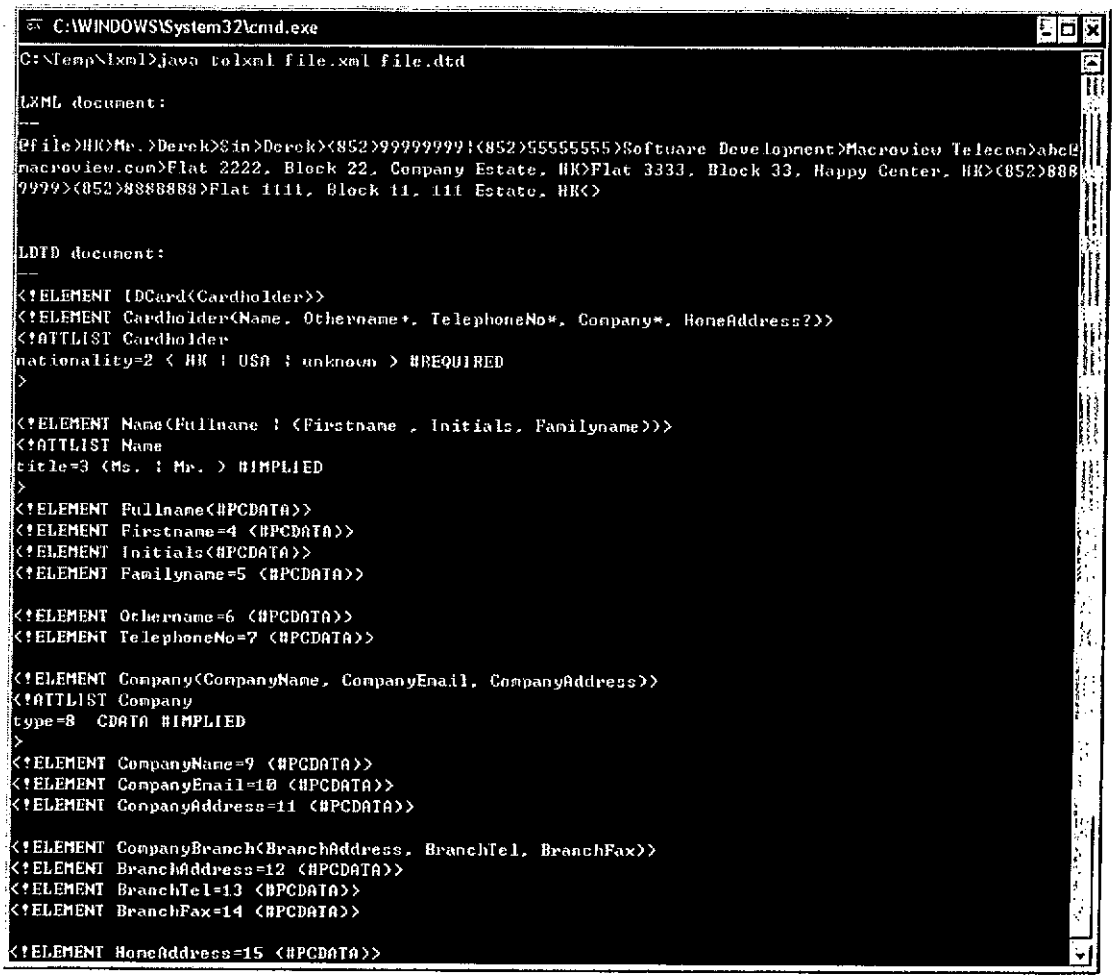


Figure 44: XML to LXML converter

CHAPTER 5: Dynamic Memory Decision Model $(DM)_2$

As the memory of a smart card is very limited, it is impossible to store a large object inside it. For instance, it is impossible to store a large photograph or a video file inside it. Inspired by the WebCard proposal [36], we have developed an Object Server (OS) to cater for this limitation. The purpose of the object server is to provide a secure place to hold all objects that are too large to fit into a smart card or not frequently accessed. While storing the object in a remote object server, we also need to store a linkage (agent) inside the smart card such that we know where to retrieve the remote object. Ideally, the most efficient way is to store all objects inside the smart card because the retrieval time can be greatly minimized. However, it is impossible due to the limited memory of a smart card. Therefore, to maximize memory usage, we need an algorithm to determine which objects should be stored internally. For example, assume that the total memory inside a smart card is 7 units. If we have three objects that occupy 3,4,5 units respectively, it is not possible to store all of them in the smart card. However if the access-frequencies are known, it is possible to determine which objects should be stored internally/externally. In the following section, we formulate a Dynamic Memory Decision Model $(DM)_2$ to tackle this problem.

5.1 Formulation of the $(DM)_2$ problem

Following the notations in [38] and [39], we formulate a Markov decision problem as follows. The aim is to determine whether an object should be stored internally or externally in order to minimize the average retrieval time.

Consider that we want to store N objects. The objects are added to the smart card/the object server step-by-step at stages 1,2,3,...,N. The stages form the decision point set, which is denoted as:

$$T = \{1, 2, 3, ..., t, ..., N\}$$

The object to be added to the smart card/object server at stage t is O_t where t=1,2,3,...,N. We denote the set of objects as:

$$O = \{O_1, O_2, ..., O_t, ... O_N\}$$

To store object O_t into the smart card, a memory space of m_t is required. We denote the set of memory space (in bytes) required by each object as:

$$H = \{m_1, m_2, m_3, ..., m_t, ..., m_N\}$$

The total memory available inside the smart card is M.

In our problem, the system state is defined as the available or residual memory at each stage. At stage t, the available memory is s_t. The set of system states is denoted as:

$$S = \{s_1, s_2, s_3, \ldots, s_t \ldots, s_N\}$$

Denote A as the set of actions that can be performed at each decision point. There are two possible actions, I or E as follows:

$A = \{I, E\}$ where I: Store the whole object (Store internally)

E: Store only the agent object (Store externally)

To store an agent object, a constant memory space of C bytes is required.

Assume that the rates of retrieving data from the smart card and from the object server are R_{SC} (bytes/second) and R_{OS} (bytes/second), respectively. Obviously we have $R_{OS} < R_{SC}$. Therefore, the time needed to retrieve 1 byte of data from the smart card is $1/R_{SC} = T_{SC}$. Likewise, the time needed to retrieve 1 byte of data from the object server is $1/R_{OS} = T_{OS}$.

Denote the average access frequency as the average number of times a particular object is accessed per unit time. The set of the average access frequency for the N objects is

$$F = \{f_1, f_2, f_3, \ldots, f_t, \ldots, f_N\}$$

The reward (cost) after taking an action A in state s at stage t is $r_t(s, A)$, which is the mean time needed to retrieve the object from the smart card. Based on the above information, we have

$$r_t(s, I) = f_t m_t T_{SC} \qquad \text{when taking action } I$$

and

$$r_t(s, E) = f_t(m_t T_{OS} + C T_{SC}) \qquad \text{when taking action } E$$

After taking an action A in state s at stage t, the available memory in the smart card will be reduced. Referring to the state definition, we would move to another state as determined by the following transfer functions (state transition functions):

$$w_t(s, I) = s - m_t \qquad \text{when taking action } I$$

subject to:

$$s - m_t \geq 0$$

and

$$w_t(s, E) = s - \mathrm{C} \qquad \text{when taking action } E$$

subject to:

$$s - C \geq 0$$

Denote the total expected reward (cost) in state s at stage t as $v^t(s)$, which can be found as follows:

$$v^t(s) = \min \{ r_t(s, I) + v^{t+1}(w_t(s,I)), r_t(s, E) + v^{t+1}(w_t(s,E))\}$$

and $t = 1, \ldots, N$ (1)

subject to

$$s_t \geq 0$$

and $t = 1, \ldots, N$

Hence if we know all the possible values of $v^{t+1}(s)$, $v^t(s)$ can be found. Therefore starting from the last stage N, we can work backward to find the optimal policy or the optimal action to be taken at each stage.

Our problem is similar to the shortest problem in [38] [39]. In our case, we need to find the minimum cost path from stage 1 to N where the cost is the mean time in retrieving the respective object.

There are two ways to solve the minimum cost path problem in the graph. For instance, this minimum cost path problem can be solved by tracing the path forward or by using the aforementioned backward induction algorithm.

To trace the path forward, we apply the following route selection algorithm:

1. Set $t = 1$ and for all possible states s, define $d_t(s) = a^*$. The decision to take at state s in time t to be a^*.
2. Set $u = w_t(s, a^*)$, i.e. the state to be occupied after taking action a^* at state s at time t.
3. For all decisions u, define $d_{t+1}(u) = a^*$ at state u in time t+1. Replace u by $u = w_{t+1}(u,a^*)$.
4. If $t + 1 = N$, stop; otherwise, set $t = t + 1$ and return to step 3.

This algorithm traces in a forward manner so as to determine the minimum value of Eq (1). As a result, the minimum cost path can be found.

The alternative method is to use the backward induction algorithm to solve the minimum cost path problem. The main idea of the backward induction algorithm is as follows:

1. At stage N, the best action for each state is selected. Clearly, $v^N(s)$ gives the optimal value for state s at stage N.
2. For each state at stage N - 1, an action is found to minimize the immediate reward plus the latest total reward. Obviously, $v^{N-1}(s)$ is the optimal reward from stage N - 1 onward starting in state s. Thus, this policy is optimal over these two stages since its value equals to the optimal value.
3. The algorithm is repeated at stages $N - 2$, $N - 3$, ... ,1 to determine the optimal policy, i.e., the best action $A^*_{s,t}$ at each stage.

The backward induction algorithm is more attractive to solve the minimum cost path problem because it requires less comparison. The mathematical operations are as shown below.

In the forward tracing algorithm, there are N stages (N objects to be added to the smart card wallet), 2 actions (either stored internally or externally) and M states at each stage. Hence there are $(2^M)^N$ polices. To solve the problem, there are $N \times (2^M)^N$ additions and $(2^M)^N$ comparisons to do.

If the backward induction algorithm is used, only $N \times M \times 2$ additions and $N \times 2$ comparisons are required under the same assumptions. Consequently it is far more efficient.

5.2 Simple examples illustrating the $(DM)_2$ model

Let us look at a simple example with three objects. It is equivalent to finding the minimum cost path of the following graph:

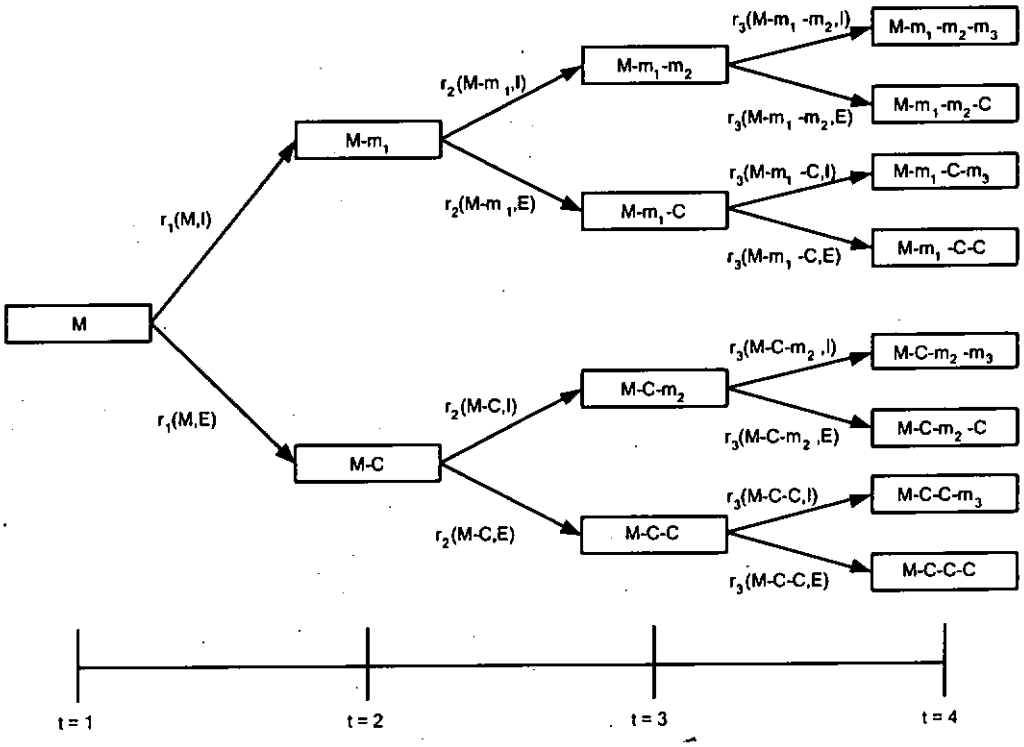


Figure 45: Retrieving different objects from decision point t=1 to t=4

The parameters are described as follows:

- Total memory available inside the smart card: 7 units
- Memory occupation of m_1: 5 units
- Memory occupation of m_2: 4 units
- Memory occupation of m_3: 3 units
- Time needed to retrieve 1 unit internally (I): 3 seconds

- Time needed to retrieve 1 unit externally (E): 4 seconds
- Time to store the agent object internally: 0 second
- Access frequency coefficient of m_1: 2
- Access frequency coefficient of m_2: 1
- Access frequency coefficient of m_3: 1

The graph becomes:

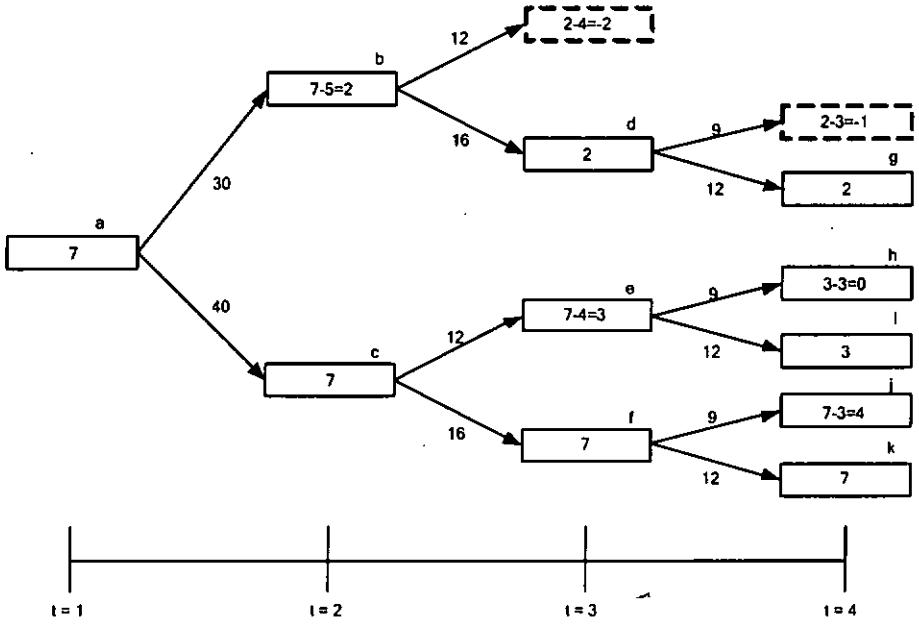


Figure 46: Retrieving objects of 5 units, 4 units and 3 units respectively

The boxes with dotted lines denote the impossible states. To solve this problem, we can calculate the mean retrieving time by considering all the possible paths and find out the best one among all the possible combinations.

The paths are:

- Path 1: a → b → d → g

- Path 2: a → c → e → h
- Path 3: a → c → e → I
- Path 4: a → c → f → j
- Path 5: a → c → f → k

The mean retrieving times are:

- Mean retrieving time of path 1 is: (30 + 16 + 12) / 4 = 14.50
- Mean retrieving time of path 2 is: (40 + 12 + 9) / 4 = 15.25
- Mean retrieving time of path 3 is: (40 + 12 + 12) / 4 = 16.00
- Mean retrieving time of path 4 is: (40 + 16 + 9) / 4 = 16.25
- Mean retrieving time of path 5 is: (40 + 16 + 12) / 4 = 17

Hence, path 1 has the minimum mean retrieval time, and therefore, we can conclude that a → b → d → g is the minimum cost path.

The method of finding the minimum mean retrieving time by comparing all the possible paths is efficient only when retrieving a few objects. If we need to find out the mean retrieving time for 10 objects, we need to calculate and compare 2^{10} = 1024 possible paths and it is clearly not efficient.

To solve this problem, we can use the backward induction algorithm such that we do not need to compare all the possible paths. The algorithm is as follows.

a. To start off, set t = 4; the total cost = 0

b. Since $t \neq 1$; continue. Set $t = 3$ and

$$\min \{v^3(d)\} = \min \{ 9, 12 \} = 9$$

$$\min \{v^3(e)\} = \min \{ 9, 12 \} = 9$$

$$\min \{v^3(f)\} = \min \{ 12 \} = 12$$

c. Since $t \neq 1$; continue. Set $t = 2$ and

$$\min \{v^2(b)\} = \min \{ 16 + 9, 12 + 9 \} = 21$$

$$\min \{v^2(c)\} = \min \{ 16 + 12 \} = 28$$

d. Since $t \neq 1$; continue. Set $t = 1$ and

$$\min \{v^1(a)\} = \min \{ 40 + 21, 30 + 28 \} = 58$$

e. Since $t = 1$; stop.

Hence, the same minimum cost path: a → b → d → g can be found.

Therefore, we can conclude that the mean retrieving time would be ((15+15+16+12)/4) = 14.5 seconds and the best arrangement is to store m_1 internally and m_2 and m_3 externally.

The mean retrieving time would be different when the access frequency coefficient is changed. If the access frequency coefficients for all the objects are the same, then, the best approach is to store as many objects as we can inside the smart card.

The reason for this is that the time in retrieving internal objects is smaller than that in retrieving external objects. However, when taking the access frequency into consideration, it is preferable to store objects with a higher retrieval frequency inside the smart card in order to reduce the object retrieval time. The object manager (OM) keeps track of the access frequency of all objects and transfers the objects periodically (e.g. to store the less frequently retrieved objects in the object server).

In order to evaluate the behavior of the dynamic memory system, a web-based CGI program has been developed based on the backward induction algorithm. Details are shown in the next section.

5.3 Analysis

To analyze the dynamic memory management algorithm, we have developed a simple web-based CGI program to simulate the object addition/retrieval process. With this program, we can simulate the addition and retrieval of objects by pressing the respective buttons. A screenshot is shown in Figure 47.

As can be seen in Figure 47, initially, we can define the number of objects initially stored inside the smart card. After that, we can change the memory occupation of each object by entering its respective object ID. If we want to simulate the object retrieval process, we can press the respective button under the 'Access Frequency' column. The program then updates the access frequency and run the previous dynamic memory management algorithm. If an object should be stored inside the smart card wallet, an 'O' symbol will be marked under the 'Internal' column and similarly, if the object should be stored in remote object server, an 'O' symbol will be marked under the 'External' column.

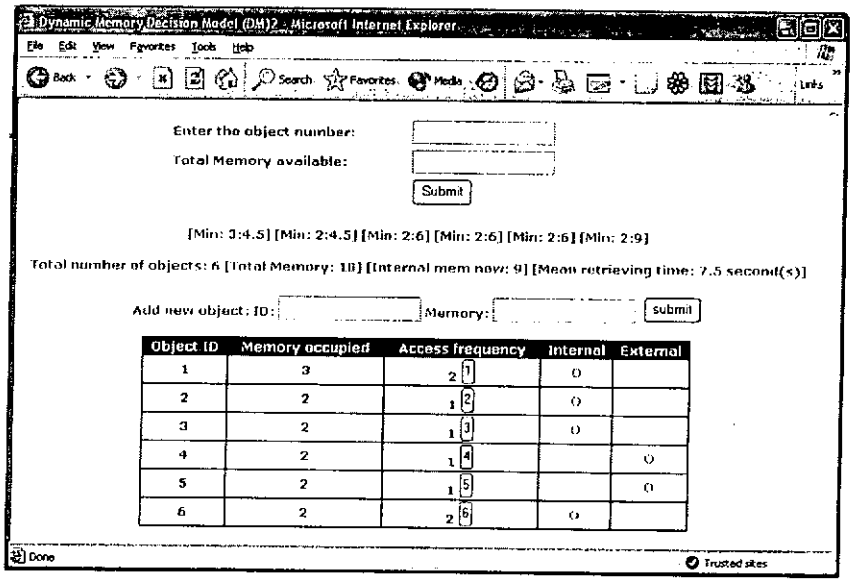


Figure 47: A web-based program solving the $(DM)_2$ problem

After the web-based CGI program has been developed, we have conducted several experiments to determine the 'cut off frequency' that refers to the access frequency required to bring the object that is stored externally back to the smart card. It is obvious that if the object should be stored internally when it is first added to the smart card wallet, the cut-off frequency will be 1.

In the first analysis, for simplicity, we assume that all the objects stored inside the smart card wallet are of fixed size (say 2 units) and that all the objects are 'just fit' into the internal memory (say five 2-unit objects with the overall memory space of 10 units). As can be seen in Figure 48, if a new object of the same size is added to the internal memory, the cut-off frequency will be the access frequency plus one. The reason is obvious as the memory occupation of all the objects are the same, the factor that determines whether the object should be stored internally or externally is the access frequency. Therefore, we need one more retrieval process to bring an externally stored object back to the internal memory. For example, if all the objects stored inside the smart card have an access frequency of 10. We need an access frequency of 11 to store a newly added object internally.

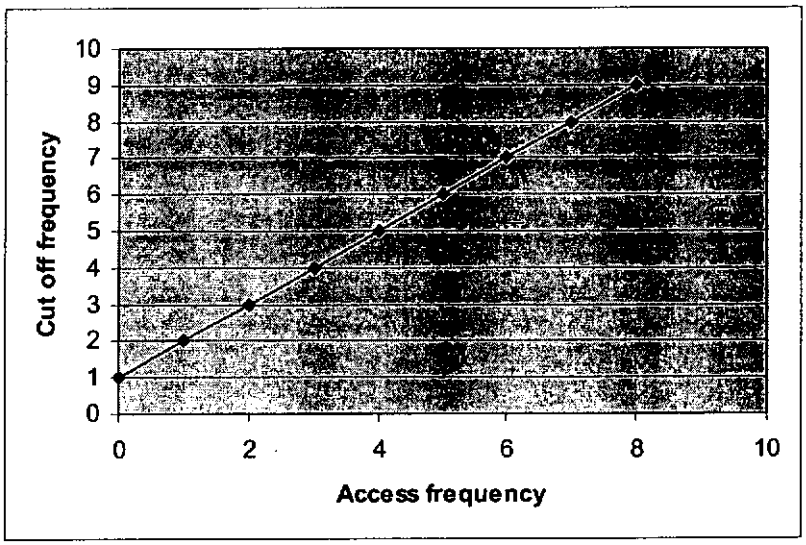


Figure 48: Relationship between access frequency and cut-off frequency

In the second analysis as shown in Figure 49, we investigate how the size of a newly added object affects the cut-off frequency.

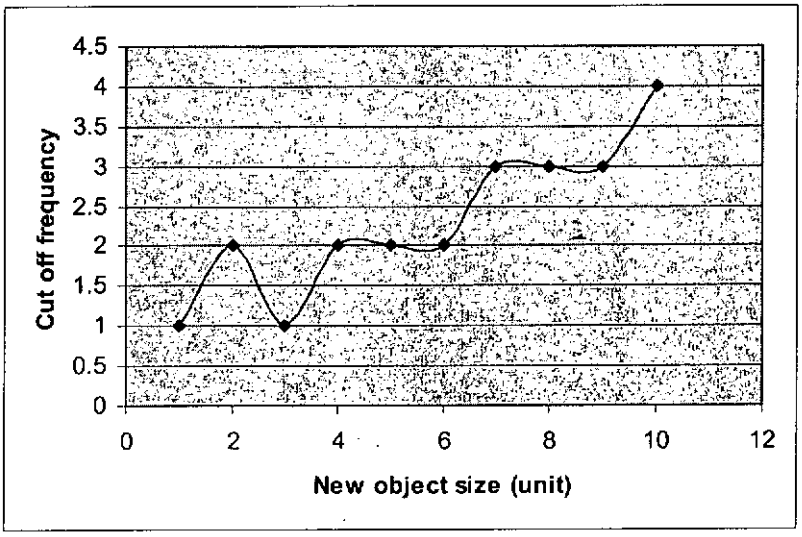


Figure 49: Relationship between the size of a newly added object and the cut-off frequency

In this analysis, we assume that several objects of the same size are already stored inside the smart card. It is clear that the cut-off frequency increases as the object size increases. With the increase in object size, we need to move more objects out

from the smart card in order to bring the new object into the smart card. For the ideal situation, it is preferred to store as many objects as possible inside the smart card. Therefore, the only factor that can affect the ideal situation is the access frequency.

In our third analysis as shown in Figure 50, we investigate how the mean retrieving time is affected by the access frequency for a particular object. Originally, we have one object occupying 9 units of memory inside the smart card. If we add a new object of 2 units into the smart card, it is obvious that the new object should be stored externally so as to minimize the mean retrieving time. Now, if we increase the access frequency of the new object from 1 to 38, Figure 50 shows that the mean retrieving time decreases slowly towards 6. The reason for the curve converging to the value '6' is that the retrieval time of the new object is 6 when storing it inside the smart card. Therefore, we can conclude that with the increase in the access frequency for a particular object, the mean retrieving time would tend to be the retrieval time of that object being stored internally, provided that there is enough memory space inside the smart card. If the memory space inside the smart card is not sufficient to accommodate the object, the mean retrieving time will tend to be the retrieving time of that particular object being stored externally.

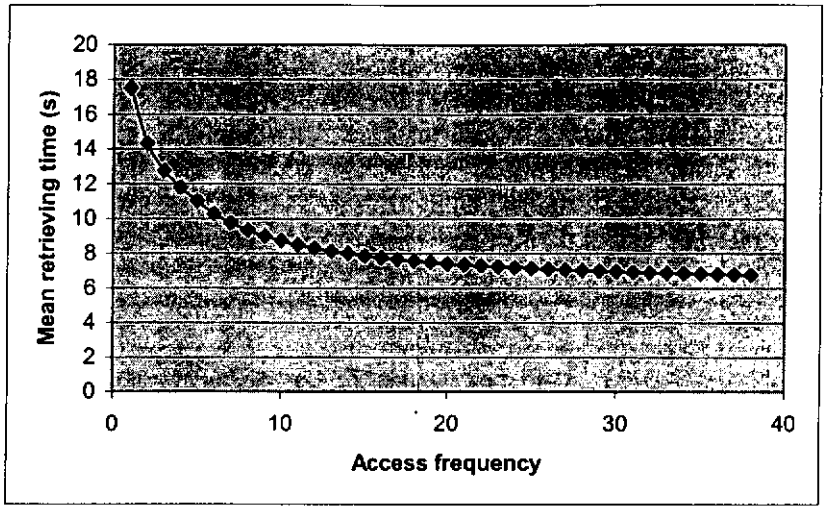


Figure 50: Relationship between the mean retrieving time and the access frequency

In our fourth analysis, we study how the available memory inside the smart card affects the mean retrieving time of various objects. We assume that there are 10 objects, each of 2 units, stored inside the smart card wallet. We increase the memory from 2 units to 28 units and investigate how the mean retrieving time changes.

As can be seen in Figure 51, the mean retrieving time decreases linearly and remains the same after the memory space reaches 20 units. The reason for the decrease in the mean retrieving time is that more space is available inside the smart card. As a result, more objects can be stored internally. After the memory reaches 20 units, all the objects can be stored internally inside the smart card and the mean retrieving time remains to be 6 with the increase in available memory.

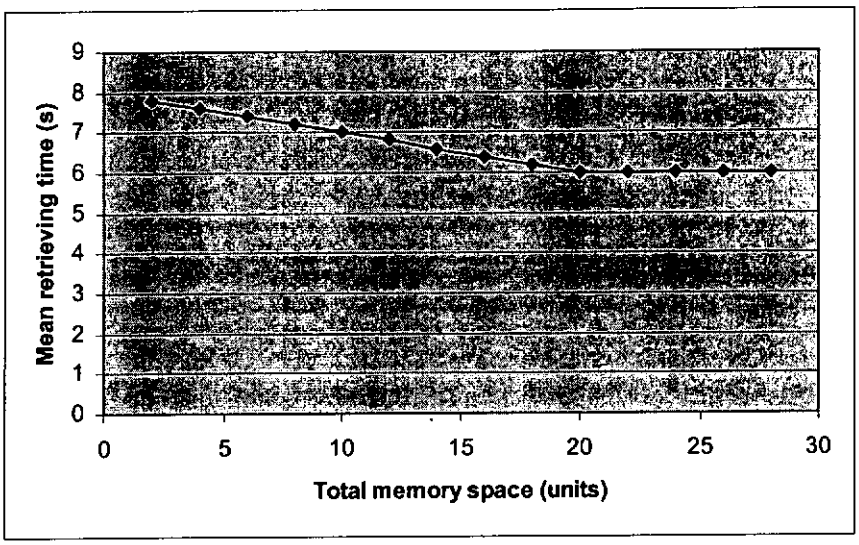


Figure 51: Relationship between the total memory space and the mean retrieving time

In the fifth analysis, we study how the total memory size and the number of objects affect the mean retrieving time. We assume that each object occupies 2 units of memory and we increase the available memory gradually.

As can be seen in Figure 52, when there are 20 units of memory space, the mean retrieving time remains unchanged as the number of objects increases. The reason is that there is enough memory to accommodate all the objects. Therefore, all objects can be stored inside the smart card and the mean retrieving time remains at the minimum value. When there are 10 objects and 16 units of memory space, some objects need to be stored externally because there is not enough memory to accommodate all the objects. The mean retrieving time increases gradually when more and more objects are stored externally.

We can conclude from the analysis that in order to obtain the minimum mean retrieving time, we need to have sufficient memory space. If the memory space is

insufficient, more and more objects are stored externally and the mean retrieving time increases respectively.

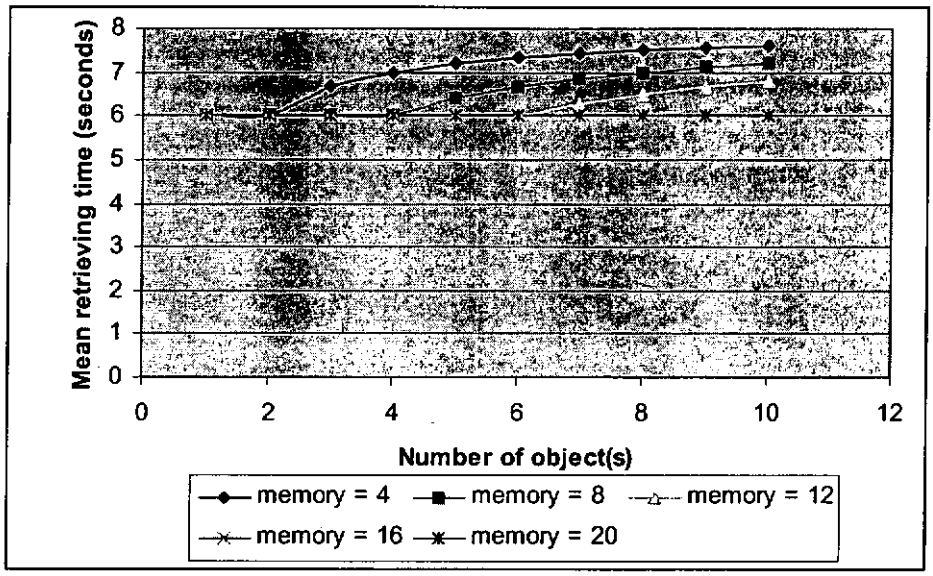


Figure 52: Relationship between the number of objects and the total memory available with the mean retrieving time.

In our final analysis as shown in Figure 53, we take several real-world objects and investigate how the cut-off frequency changes with respect to the access frequency. The objects and their sizes are as follows:

Object	Size (units)
RSA public key	1
Memo note	1
Photo	1.5
X-ray image	3

We assume that the smart card have a memory of 8 units. Previously, we have discussed how different kinds of objects can be stored inside the smart card wallet. They are the HKID card object, the credit card object, the e-coin bag object and the name card object. Now we assume that all these objects are already stored inside the smart card wallet and each object occupies a certain amount of memory space as follows.

Object	Size (units)
HKID card	1.5
Credit card	1.5
E-coin bag	1
Name card	1

Hence, after storing these objects, the available memory is 3 units.

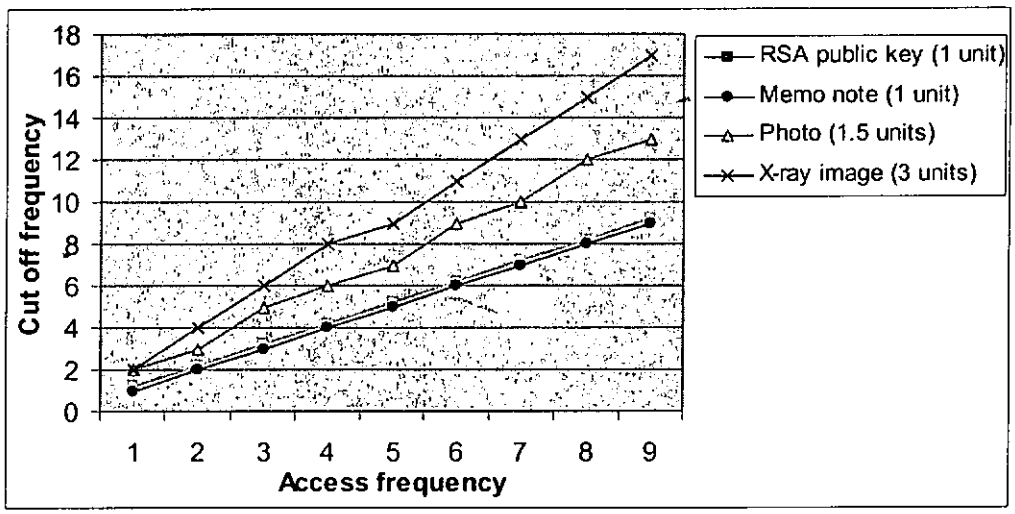


Figure 53: Relationship between access frequency and cut off frequency using several real-world objects

It can be seen from Figure 53 that when the access frequency increases, the cut-off frequency increases accordingly. For an object with a larger size, the cut-off frequency is higher. The reason has been discussed in the previous analysis. In summary, once it is stored externally, a larger object requires a higher access frequency to bring it back to the smart card.

5.4 Implementing the dynamic memory management algorithm in the smart card wallet

As have been mentioned before, the object manager manages all the objects stored inside the smart card wallet and also runs the dynamic memory management algorithm. Initially, when a new object is added into the smart card wallet, the OM sets the access frequency counter of the respective object to one. The counter is incremented by one each time the object is retrieved. Regularly, the object manager calculates the minimum mean retrieving time based on the dynamic memory algorithm and decides which objects should be stored internally and externally. If an object is less frequently accessed, that object is migrated from the internal memory to the object server (OS). Similarly, if an object is accessed more frequently, the object is migrated from the object server to the smart card wallet based on the dynamic memory algorithm.

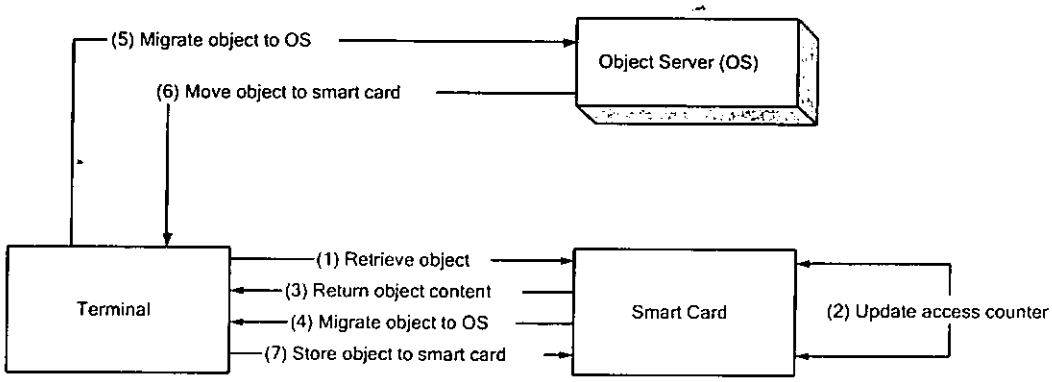


Figure 54: Implementing $(DM)_2$ in the smart card wallet

To migrate an object to the external object server, the detailed operations are shown as follows (see Figure 54).

1. Firstly, the terminal issues a command to the smart card wallet to retrieve a particular object.
2. The OM inside the smart card wallet updates the access frequency counter of that particular object and runs the dynamic memory algorithm. Suppose that an object is to be moved to the object server, the OM needs to determine if another object should be migrated from the object server back to the smart card wallet. If so, the OM issues a command to the object server to take the object back to the smart card wallet.
3. The smart card returns the content of the particular object to the terminal.
4. In the mean time, the smart card sends the object to the terminal and tells the terminal to migrate that object to the object server.
5. The terminal sends the object to the object server.
6. Some other objects may need to be migrated from the object server back to the smart card wallet to minimize the mean retrieving time. In this case, the object server also sends the respective object to the terminal.
7. Finally, the terminal forwards the received object to the smart card wallet.

With the above protocol, objects can be moved between the object server and the smart card wallet. It is obvious that if the retrieval of a particular object does not affect the other objects, step 4 and onwards can be omitted.

The channel between the terminal and the smart card is secured. All the messages exchanged are encrypted by the agreed DES session key generated in the challenge and response process as have been discussed in section 3.4.1. The communication channel between the terminal and the object server is protected by the SSL protocol and the secure session cookie as have been discussed in section 3.5.

CHAPTER 6: Conclusions

In conclusion, we have presented an overview of the object-oriented smart card wallet. Behaving like a physical wallet, the smart card wallet can be used to store various electronic items. Integrating with the Web system, the wallet can function as an effective tool for enabling e-commerce.

We have achieved the project objectives by designing an object-oriented smart card wallet. We have taken the HKID card for storing personal information, the credit card for online procurement using the SET protocol, the e-coin bag for storing electronic cash used for electronic payment and the name card object as examples and study their respective behaviors. In the e-coin bag object, we have proposed an anonymous offline protocol such that double spending can be detected.

To cater for the memory-limited nature of smart cards, we have proposed a novel LXML method to store data inside the smart card wallet efficiently. By storing the navigation tags instead of the XML tags, the data file size can be greatly reduced. Hence compared with XML, LXML is more "lightweight" because it imposes less overhead. To illustrate the advantages of using LXML, we have compared it with three different methods for storing data inside a smart card. Results indicate that the LXML method gives the best performance in terms of storing/retrieving time and memory requirement. For implementation, we have presented the basic syntax rules for forming a LXML document and a general example to explain its operation. Besides the smart card wallet, we expect that LXML can be used to store data in other memory-limited devices such as mobile phones to support mobile commerce.

Finally, in order to utilize the memory of a smart card efficiently, we have investigated a dynamic memory algorithm based on a Markov decision model. The algorithm determines whether an object should be stored internally or externally according to the access frequency and available memory inside the smart card.

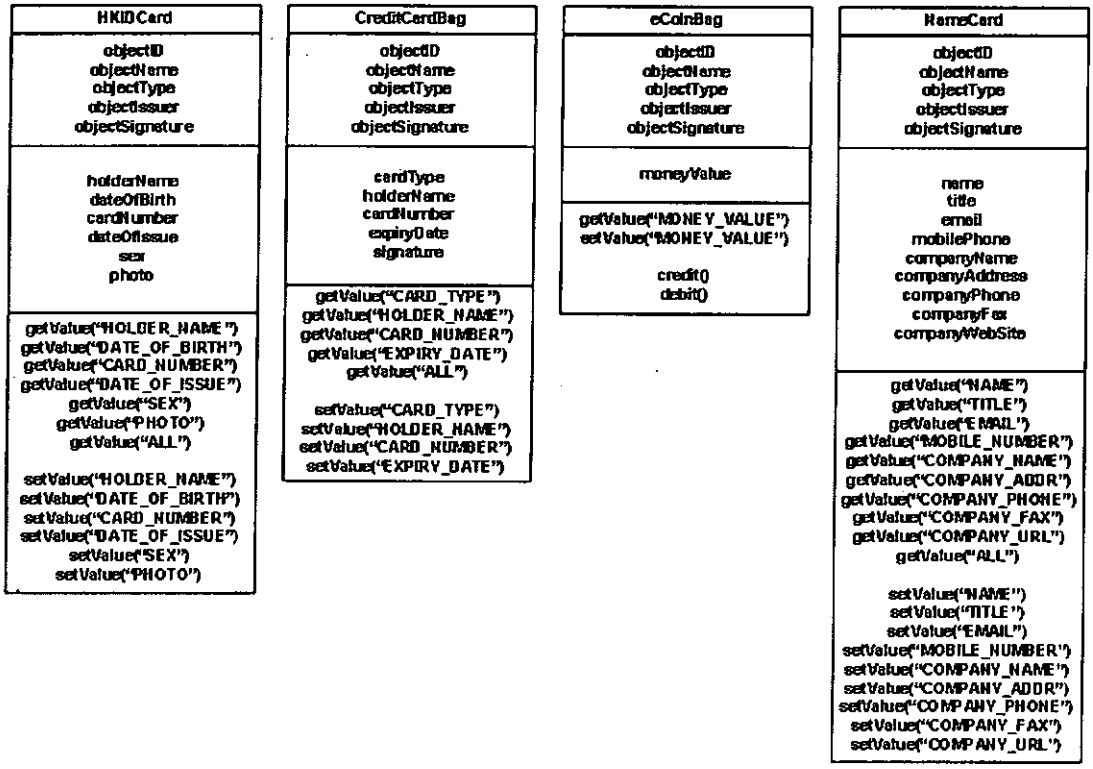
Some research results of this thesis have been published in the following conference proceedings:

Derek W. M. Sin and Henry C. B. Chan, "LXML: Lightweight XML for Storing Data in Smart Card Wallets", *Proceedings of International Conference on Internet Computing 2002*, pp. 103-110, 2002.

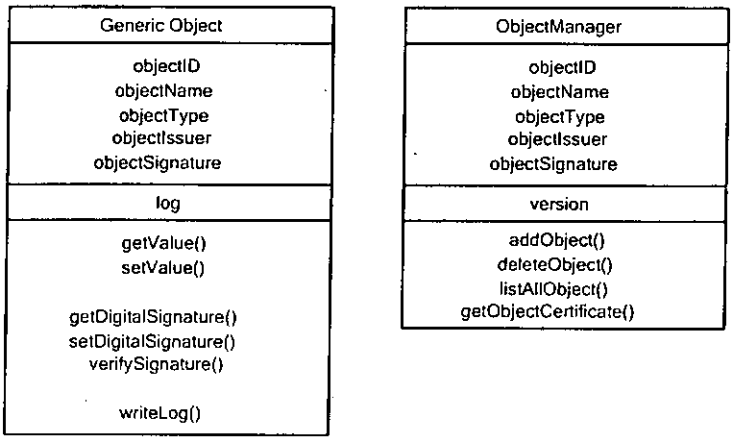
Derek W. M. Sin and Henry C. B. Chan, "JOBS: Javacard-based Online-ticket Booking System", *Poster Proceedings of the Tenth International World Wide Web Conference*, pp. 148-149, 2001.

APPENDIX

Appendix A: UML object model for four kinds of objects



Appendix B: UML object model for a generic object and the OM



Appendix C: Object Manager's API

Class BASE

java.lang.Object

|

+--**BASE**

---

public class **BASE**

extends java.lang.Object

Definition of some basic static variables

Constructor Summary
BASE() Definition of static variables
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

BASE

public **BASE**()

Definition of some basic static variables

Class Connect

java.lang.Object

|

+--**Connect**

---

public class **Connect**

extends java.lang.Object

Constructor Summary	
`Connect()`	

Method Summary	
`int`	`ConnectRC()` Try to make connection to the smart card.
`void`	`Dispose()` Disconnect the connection with the smart card
`com.gemplus.gcr.toolkit.gemxpresso.GxCard`	`Get Card()` Get the gemplux GxCard object

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Connect

public **Connect()**

Method Detail

ConnectRC

public int **ConnectRC()**

Try to make connection to the smart card.

Loop until the smart card is inserted into the smart card reader

---

Dispose

public void **Dispose**()

Disconnect the connection with the smart card

Get_Card

public com.gemplus.gcr.toolkit.gemxpresso.GxCard **Get_Card**()

Get the gemplux GxCard object

Class DMI

java.lang.Object

|

+--**DMI**

---

public class **DMI**

extends java.lang.Object

Constructor Summary	
`DMI()` Constructor	
Method Summary	
`byte[]`	`send XML(com.gemplus.gcr.toolkit.gemxpresso.GxCard myCard, int ID, byte[] CONT, int END_TAG, byte[] inputpin)` Sending a XML command to the smart card wallet to perform various operations
Methods inherited from class java.lang.Object	
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`	

Constructor Detail

DMI

public **DMI**()

Constructor

Method Detail

send_XML

public byte[] **send_XML**(com.gemplus.gcr.toolkit.gemxpresso.GxCard myCard,
int ID,
byte[] CONT,
int END_TAG,
byte[] inputpin)

Sending a XML command to the smart card wallet to perform various operations

Class OMClient

java.lang.Object

|

+--java.awt.Component

|

+--java.awt.Container

|

+--java.awt.Panel

|

+--java.applet.Applet

|

+--**OMClient**

All Implemented Interfaces:

javax.accessibility.Accessible, java.awt.image.ImageObserver, java.awt.MenuContainer, java.lang.Runnable, java.io.Serializable

---

public class **OMClient**

extends java.applet.Applet

implements java.lang.Runnable

See Also:

Serialized Form

Inner classes inherited from class java.applet.Applet
`java.applet.Applet.AccessibleApplet`

Constructor Summary
`OMClient()`

Method Summary	
`void`	`callJS(java.lang.String jsCommand)` Call the Javascript function
`void`	`init()` Initiation: runs when this class is loaded
`java.lang.String`	`performAction(java.lang.String ID,`

	`java.lang.String content,`
	`java.lang.String inputpin)`
	Format the XML string and call the DMI class to perform respective operations

`void`	**`run`**`()`
`void`	**`start`**`()`
`void`	**`stop`**`()` Call when the class is unloaded
`long`	**`timeNow`**`()` Get the time now
`boolean`	**`waitForCard`**`()` Wait for the insertion of smart card into the smart card reader

Constructor Detail

OMClient

public **OMClient**()

Method Detail

init

public void **init**()

Initiation: runs when this class is loaded

Overrides:

`init` in class `java.applet.Applet`

---

performAction

public java.lang.String **performAction**(java.lang.String ID,
java.lang.String content,
java.lang.String inputpin)

Format the XML string and call the DMI class to perform respective operations

---

waitForCard

public boolean **waitForCard**()

Wait for the insertion of smart card into the smart card reader

---

callJS

public void **callJS**(java.lang.String jsCommand)

Call the Javascript function

---

timeNow

public long **timeNow**()

Get the time now

---

start

public void **start**()

Overrides:

`start` in class `java.applet.Applet`

---

stop

public void **stop**()

Call when the class is unloaded

Overrides:

`stop` in class `java.applet.Applet`

---

run

public void **run**()

Specified by:

`run` in interface `java.lang.Runnable`

REFERENCES

[1] W. Rankl and W. Effing, *Smart Card Handbook*, John Wiley & Sons Ltd., Chichester, 1997.

[2] John Scourias, Overview of the Global System for Mobile Communications, http://ccnga.uwaterloo.ca/~jscouria/GSM/gsmreport.html

[3] OpenCard Consortium http://www.opencard.org

[4] Millicent http://www.millicent.com/works/details/papers/millicent-w3c4/millicent.html

[5] H.C.B. Chan, R.S.T. Lee, T.S. Dillon and E. Chang, *E-commerce: Fundamentals and Applications*, John Wiley and Sons (UK), 2001.

[6] JavaCard Forum http://www.javacard.forum.org

[7] Secure Electronic Transaction LLC http://www.setco.org/

[8] U. Hansmann, M. S. Nicklous, T. Schack and F. Seliger, *Smart card: application development using Java*, Springer-Verlag, Berlin, Heidelerg, 2000.

[9] S. B. Guthery and T. M. Jurgensen, *Smart card developer's kit*, Macmillian Technical Publishing, 1998.

[10] International Organization for Standardization (ISO). Draft International Standard ISO/IEC 7816: Integrated circuit(s) cards with contacts.

[11] Mondex Electronic Cash
http://www.mondex.com

[12] Visa Cash,
http://www.visa.com

[13] M. S. Hwang and L. H. Li, "A new remote user authentication scheme using smart cards", *IEEE Transactions on Consumer Electronics*, Vol. 46, No. 1, pp. 28-30, Feb 2000.

[14] T. Bell, I.H. Witten, J.G. Cleary, "Modeling for text Compression", *ACM Computing Surveys,* Vol. 21, No. 4, pp. 557-591, Dec 1989.

[15] D.A. Lelewer and D.S. Hirschberg, "Data compression", *ACM Computing Surveys,* Vol. 19, No. 3, pp. 261-266, Sept 1987.

[16] M.A. Roth and S.J. Van Horn, "Database Compression", *SIGMOD Record* Vol. 22, No. 3, pp. 31-39, 1993.

[17] D.E. Knuth, "Dynamic Huffman Coding", *Journal of Algorithms,* Vol. 6, No. 2, 1985.

[18] A. Lempel and J. Ziv, "A Universal Algorithm for Sequential Data Compression", *IEEE Transaction on Information Theory,* Vol. 23, No. 3, pp. 337-343, May 1977.

[19] H. Liefke and D. Suciu, "XMill: an efficient compressor for XML data", *in Proc. SIGMOD Conference*, 2000.

[20] A.T.S. Chan, J. Cao, H. C. B. Chan, G. Young, "A Web-enabled Framework for Smart Card Application in Health Services", *accepted for publication in Communications of the ACM.*

[21] S. Elliot and C. Loebbecke, "Smart-card based electronic commerce: characteristics and roles", *in Proc. 31st Annual Hawaii International Conference on System Sciences*, pp. 242-250, 1998.

[22] E. Turban and D. McElroy, "Using smart cards in electronic commerce", *in Proc. 31st Annual Hawaii International Conference on System Sciences*, pp.62-69, 1998.

[23] J. Elliott, "The one-card trick: multi-application smart card e-commerce prototype", *Computing and Control Engineering Journal*, pp. 121-128, June 1999.

[24] Tessella Technical Supplement – XML
http://www.tssp.co.uk/Literature/Supplements/XML.htm

[25] Elementary XML
http://msdn.microsoft.com/library/en-us/dnxml/html/elxml.asp

[26] Developing Smart Card-Based Application using Java Card
http://www.gemplus.com/smart/r_d/publications/art1.htm

[27] The Hong Kong Electronic Industries Association Ltd., *Study on the impact of the Development and Manufacturing Technology of Smart Card products to the Hong Kong electronic industry*, October 1997, Hong Kong SAR Government Industry Department.

[28] Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*, Addison-Wesley, Reading, MA, 2000.

[29] Derek W. M. Sin and Henry C. B. Chan, "LXML: Lightweight XML for Storing Data in Smart Card Wallets". *In Proceedings of International Conference on Internet Computing 2002*: pp. 103-110.

[30] D. Husemann, "Standards in the smart card world", *Computer Networks 36* pp. 473-487, 2001

[31] A. Niemi, J. Arkko and V. Torvinen, "HTTP Digest Authentication Using AKA", *RFC 3310*, September 2002.

[32] "American National Standard for Information Systems - Data Link Encryption". *American National Standards Institute.* ANSI X3.106, 1983.

[33] K. Davidson and Y. Kawatsura, "Digital Signatures for the v1.0 Internet Open Trading Protocol (IOTP)", *RFC 2802*, April 2000.

[34] R. Rivest, "The MD5 Message-Digest Algorithm", *RFC 1321*, April 1992.

[35] David Hunter, *Beginning XML*, Wrox Press Inc, 2nd edition, 2001.

[36] Extended Memory Card. http://www.gemplus.com/smart/r_d/publications/art6.htm

[37] Jean-Paul Boly, Antoon Bosselaers, Ronald Cramer, Rolf Michelsen, Stig Mjølsnes, Frank Muller, Torben Pedersen, Birgit Pfitzmann, Peter de Rooij, Berry Schoenmakers, Matthias Schunter, Luc Vallée and Michael Waidner. "Digital Payment Systems in the ESPRIT Project CAFE". *In Proceedings of Securicom '94. Paris*, June 1994.

[38] M.L. Puterman, *Markov Decision Process: Discrete Stochastic Dynamic Programming*, 1st edition, Wiley-Interscience, 1994.

[39] M.L. Puterman, "Dynamic programming", *Encyclopedia of Physical Science and Technology*, Vol. 4, Academic Press Inc., pp. 438-463, 1987.

[40] D. Chaum, "Blind signatures for untraceable payments, Advances in Cryptology". *In preceedings of Crypto '82*, Springer-Verlag, pp. 199-203, 1983.