Clearinghouse: A Payment Framework for Distributed Object Systems

by

Ellis Y. Chi

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 7, 1997

Author .....
Department of Electrical Engineering and Computer Science
February 7, 1997

Certified by ....
Hal Abelson
Professor
Thesis Supervisor

Accepted by ....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Clearinghouse:

A Payment Framework for Distributed Object Systems

by

Ellis Y. Chi

Submitted to the Department of Electrical Engineering and Computer Science
on February 7, 1997, in partial fulfillment of the
requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the design and the details of the preliminary implementation of Clearinghouse: a "per-call" payment framework for an interoperable, distributed object environment.

Clearinghouse has two notable features. First, it has a *paymethod abstraction layer* which simultaneously supports multiple payment methods; in other words, Clearinghouse can be used for any payment method (e.g. PayWord [20] or credit card) as long as its implementation conforms to the interface of the abstraction layer. The paymethod abstraction layer allows individual payment methods to evolve without requiring changes to be made to the existing payment framework or to objects being paid using the payment methods. Second, this payment framework provides a user an "electronic wallet" residing in the local Clearinghouse process. This feature allows the user the option to select and configure one or more supported payment methods before initiating a network call to purchase a remote object invocation.

Clearinghouse is an extension module of ORBlite [6], a "software bus" which enables interoperability among distributed object systems possibly implemented in different programming models and on operating systems. Adding the payment framework to ORBlite provides an electronic commerce capability to the distributed objects.

Thesis Supervisor: Hal Abelson
Title: Professor

Acknowledgments

I would like to thank the whole ORBlite team for their support and care during the last six months, especially Evan Kishenbaum and Keith Moore, who offered valuable technical advice and personal support. They spent so much time and effort to ensure that I could proceed with the project. Their critical evluations have always helped me succeed on different occasions and in aspects during the last six months.

My thesis advisor, Professor Hal Abelson has displayed much patience and has offered me concrete guildlines for the thesis.

In addition, I would like to thank Marvin Keshner and Richard Baer for being my great on-site coordinators over the past three years. Marvin, in particular, has given me the confidence and courage to take on this challenging assignment. Yee-Hsiang Chang was my first supervisor at HP Labs, and since then he has offered me the advice and help beyond the normal reponsibilities of a supervisor. Vivian Shen has always been very helpful to me.

I would also like to thank Chris Yu for being my great cubicle mate during the first three months of my thesis assignment. Also I would like to thank Barabara Alles and Justin Liu for kindly reviewing the preliminary version of section 1.3 and chapter 14, Evan Kirshenbaum and Keith Moore for proofreading the abstract and chap 1 of my final draft, and Angie Hsu for reviewing part of this acknowledgement.

Last but not least, I would like to thank my parents, Vincent and Helen Chi, for having gone through so many hardships to ensure my proper growth and development for the last 22 years.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

This thesis describes the design and the details of the preliminary implementation of Clearinghouse: a "per-call" payment framework for an interoperable, distributed object environment.

Clearinghouse has two notable features. First, it has a *paymethod abstraction layer* which simultaneously supports multiple payment methods; in other words, Clearinghouse can be used for any payment method (e.g. PayWord [20] or credit card) as long as its implementation conforms to the interface of the abstraction layer. The paymethod abstraction layer allows individual payment methods to evolve without requiring changes to be made to the existing payment framework or to objects being paid using the payment methods. Second, this payment framework provides a user an "electronic wallet" residing in the local Clearinghouse process. This feature allows the user the option to select and configure one or more supported payment methods before initiating a network call to purchase a remote object invocation.

Clearinghouse is an extension module of ORBlite [6], a "software bus" which enables interoperability among distributed object systems possibly implemented in different programming models and on operating systems. Adding the payment framework to ORBlite provides an electronic commerce capability to the distributed objects.

1.2 Design Issues

The major design issues for developing a payment framework within ORBlite are:

- conforming with the ORBlite architecture, and
  - Since Clearinghouse is designed as a "per-call" payment framework, the payment should be appended to a request for an object invocation, rather than requiring the establishment of a separate channel to handle the payment. Therefore, any information used for triggering Clearinghouse should be appended as a transparent attribute for an object invocation.
- evolving independently of the ORBlite framework and ORBlite user applications.
  - A system which is built with this design issue in mind will be easier to maintain than a system built without it. ORBlite allows user applications to evolve independently of the communication infrastructure. As a service provided to ORBlite users, Clearinghouse should follow the same rule.

The major design issues for building a paymethod abstraction layer in a payment framework are:

- providing a complete set of interfaces to allow necessary interactions between a payment method and other modules in the payment framework, and
  - The paymethod abstraction layer should allow each payment method built underneath the layer to provide all the basic functionalities that the payment method can cover for a payment cycle.
- imposing minimum restrictions on the mechanism, data structures, and the set of configurable parameters,

- Each payment method has its own way to carry out a payment and its own representation of payment data. The paymethod abstraction layer should not impose any restrictions on them except for their ability to carry out the basic functionalities as mentioned in the previous item.

The major design issues for providing an ORBlite user an "electronic wallet" with the capability of selecting and configuring a supported payment method before making purchases are:

- interacting with payment methods through the paymethod abstraction layer, and

  - The paymethod abstraction layer separates the behavior of the basic functionalities from their implementation. Therefore, an "electronic wallet" taking advantage of the abstraction layer can treat all payment methods in the same way and does not need to modify its implementation when a user adds a new payment method to the wallet.

- providing a channel to configure and select supported payment methods and set default parameters for each transaction.

  - Recalling the second bullet of the design issues for building a paymethod abstraction layer, each payment method will have its own set of configurable parameters. Clearinghouse wallets should allow users to configure these parameters.

1.3 A Simple Example

Here is an example to illustrate the application of Clearinghouse.

`NetStock is a company that provides on-line stock quotes through the Internet.`

NetStock has a set of electronic agents which send out instant stock quotes. To make these electronic agents available to the public, a NetStock software developer ''publishes'' them using the ORBlite framework. Since the company is ''selling'' on-line accesses to stock quotes, the developer also needs to register each agent, its access price, and the NetStock's account information with a special repository.

A customer wants to get a stock quote. At a front panel (e.g a web browser), he issues a ''get-stock-quote'' command and an authorization to pay an ''electronic token'' for a poll. The front panel sends the ''get-stock-quote'' command to a NetStock server through the ORBlite framework. The ORBlite framework handles the payment and forwards the command to a stock-quote electronic agent at the NetStock server. Once the payment is cleared, the customer will get the requested stock quote. The customer may continue polling the stock quotes of the same company.

1.4 Organization

This thesis is divided into eighteen chapters. Chapter two gives a brief history for the growth of distributed objects in an open computing environment, introduces ORBlite (the communication backbone used for the payment framework), and states the motivation behind the thesis project. Chapter three compares some of the Internet commercial applications with Clearinghouse. Chapter four restates the thesis of the project. Chapter five gives an overview of Clearinghouse and a trust model on which it is based. From Chapter to six to Chapter thirteen, the thesis describes the design decisions and the implementation details of the data types of Clearinghouse. Chapter fourteen revisits the example in section 1.3, with a detailed descriptions for the internal operations of Clearinghouse. Chapter fifteen discusses the possible threats

to Clearinghouse. Chapter sixteen shows the result of the thesis project. Chapter seventeen proposes possible future work, and chapter eighteen concludes the thesis.

Chapter 2

Background

This chapter consists of two parts. The first part gives a brief description of CORBA and the ORBlite architecture. The intent is to provide enough information to address the rationale of the design of the thesis project. The second part defines the goal and the scope of the thesis based on the direction of ORBlite.

2.1 Building an Open Object Environment

2.1.1 Problems with Developing Large Computer Systems

There are two major problems with building large computer systems:

1. Systems are becoming complex: As computer systems become more popular, they are expected to perform larger and more complicated tasks. Building these systems entirely from scratch becomes infeasible. [7]

    - Observation: Various parts of a proposed system are often found at existing systems built by different vendors in different computing environments which include hardware platforms (e.g. PC-compatibles and Unix workstations) and operating systems (e.g. Windows NT and HP-UX) [7].

2. The evolution of the systems is unpredictable: The development of a system is usually at the very early stage of the system life cycle, and often it is too early

to predict how a system is going to evolve [6].

- Observation: A system becomes more evolvable if it is broken into individual pieces which only states the *behavior* rather than the actual implementation.

2.1.2 OMG

OMG (the Object Management Group) was founded in May 1989 by Christopher Stone and eight companies [13].[1] It aims at providing a solution to the problems listed in section 2.1.1 based on the observations.

It specifies standards which enable an open communication infrastructure for distributed systems built based on the standards. The open computing environment allows systems to interoperate across different hardware platforms and operating systems [13].

In addition, OMG specifies the management of different systems in the open environment as objects. This allows new systems to be built based on the existing objects (or systems) which may be implemented in another computing environment [13].

Today, OMG has over 700 members; they include software vendors, software developers, and end users. The consortium continues to promote the use of object technology in an open computing environment for distributed applications [13]. Implementations of OMG specifications have been found in over 50 operating systems across the world [13].

2.1.3 CORBA

CORBA is an acronym for Common Object Request Broker Architecture. It is a standard defined by OMG to enable interoperability across heterogeneous computing environment. CORBA defines a common language called IDL to describe an object

[1]The eight companies were: 3Com Corporation, American Airlines, Canon, Inc., Data General, Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems and Unisys Corporation

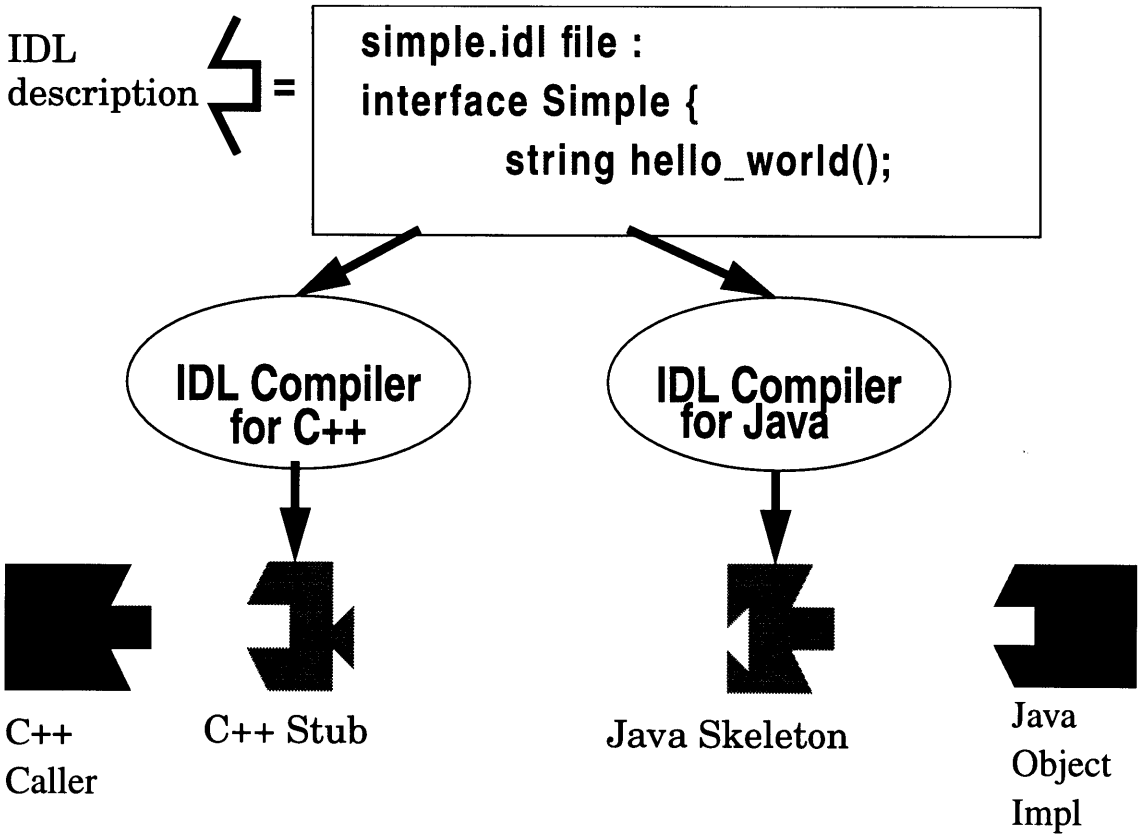

Figure 2-1: Stub and skel generated by an IDL compiler.

interface (rather than implementation). Objects written to IDL can interoperate with one another regardless of their operating systems and hardware platforms [17].

The implementation of the interoperability is accomplished by using IDL compiler. An IDL compiler takes in an IDL file defining the interface to an object and generates the code in a conventional programming language, such as C++ or Smalltalk, which allows objects to communicate across an open infrastructure. The code consists of two parts, a client-side stub and a server-side skeleton, which are both illustrated in figure 2-1.[2]

To make an invocation to an object, the caller calls an operation of the object presented in the stub. From the caller's viewpoint, all it sees is the stub code, whose

[2]Figure 2-1, figure 2-2, and figure 2-3 are adopted from [7]

Figure 2-2: Stub and skel with an ORB.

interface is presented in the same language as that of the caller code (in the above figure, C++). Once the stub is called, it will forward the call (through some communication channel) to the skeleton. Similar to the stub, the skeleton presents the invocation through a well-defined interface.

2.1.4 ORB

An Object Request Broker (ORB) provides the communication infrastructure between objects. The communication mechanism is transparent to the objects. The interaction among an ORB, a stub, and a skeleton is shown in figure 2-2.

2.2 ORBlite

ORBlite is an implementation of the CORBA2.0 standard. Therefore, it has an IDL compiler and an ORB. In addition, the focus of the ORBlite architecture is to allow piece-wise evolvability within ORBlite [6], and this is done by providing abstraction layers. The two layers that will be discussed are the language mapping abstraction layer and the protocol abstraction layer. The ORBlite architecture is shown in figure 2-3.

The IDL compiler of ORBlite offers a few language mappings (e.g. C++, Java,...) to application developers. To avoid tying the ORBlite core and the rest of ORBlite to a specific language mapping, the language mapping abstraction layer is provided between the IDL-compiler-generated code and the ORBlite core (see figure 2-3). The protocol abstraction layer allows the ORBlite core to select from a set of communica-

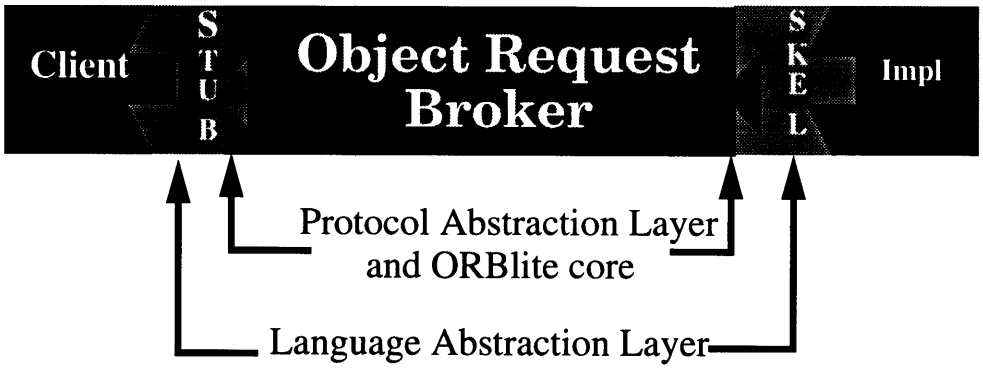

Figure 2-3: The ORBlite architecture.

tion protocols for the object without requiring the object implementation to conform with any convention imposed by the communication protocols. In other words, the actual communication protocol is transparent to application developers. Each module in ORBlite has a well defined interface which allows each of them to evolve independently. ORBlite is a prototype developed at Hewlett-Packard Laboratories, Palo Alto, California, USA. A detailed description of ORBlite is published in [6]. Figure 2-4 gives the overview of making an ORBlite remote invocation. [3]

2.3 Choosing Payment Framework as a Common Facility

The goal of ORBlite conforms with that of the OMG: to provide an open object infrastructure. As the ORBlite infrastructure provides great interoperability be-

[3]This figure is adopted from [6].

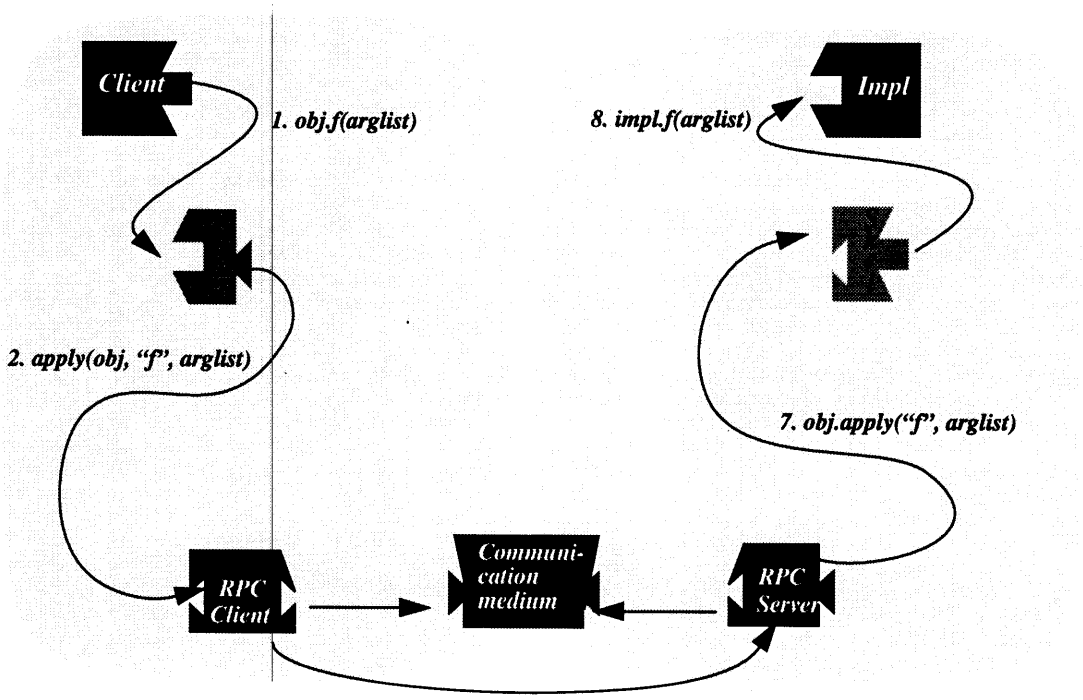

Figure 2-4: Making a remote invocation using ORBlite.

tween objects, it is envisioned that there will be commercial services implemented as CORBA-compliant objects. To support this, I chose to implement a payment framework for ORBlite. I believe that the common features enabling electronic commerce should be encapsulated in the middleware layer. In fact, this direction has been practiced in the World Wide Web where applications [12, 16, 5] are developed to provide common functionalities for financial transactions to take place.

Chapter 3

Comparison with Other Payment Frameworks

In this chapter, the payment models of other payment frameworks will be compared and discussed with Clearinghouse payment model.

3.1 Aggregated vs Per-Call Payments

Most current payment frameworks have an aggregated payment model, e.g. using a "shopping cart" to collect a list of "object descriptions" and then pay for them in another call to vendors [12, 16, 21, 5]. This model separates payments from object invocations. It is good for selling items which require to establish a secure session (e.g. SSL) to complete the payment. However, this model is not suitable for users who have a specific item in mind to purchase, or the items costs relatively cheap. The reason is that payment under an aggregated model requires additional network calls, e.g. sending a "check-out" signal to initiate the purchasing process.

An aggregated model requires a user to pay either before or after the receipt of objects. In a "pay-before-call" model, vendors are protected by the model since they can ensure the receipt of money before delivering the requested items to customers. However, the customers are not guaranteed to get the purchased items. In a "pay-after-call" model, vendors deliver goods to customers before payments take place.

This model is suitable for offering a preview for object contents, such as an abstract of a document [5]. This model offers more protection to customers; it ensures a success of delivery before paying for the requested items. In a "pay-per-call" model, customers have to include payment with the request of purchased objects, and vendors have to deliver the purchased object right away. Neither the vendor or customer is protected.

Clearinghouse has chosen the "pay-per-call" model because it focuses on providing the micropayment capability to objects, and it is more important not to add in extra network calls for enabling the commerce capability to the objects. Though the framework can support a macropayment method, such as credit card, it does not offer the protection found in the aggregated payment model.

3.2 Supporting Single vs Multiple Payment Methods

Some current payment frameworks have been designed based on a single payment method, e.g. credit card [12, 16]. Incorporating a single payment method into the frameworks may be easier to design; however, these frameworks cannot handle the evolvability of the single payment methods without making changes to the framework. In addition, these payment frameworks do not allow vendors to provide their own payment methods through the frameworks.

Other payment frameworks allow vendors and even customers to choose their preferred payment methods[21]. However, these payment frameworks provide the flexibility by processing payments off-line.

Clearinghouse aims at providing the evolvability of payment methods within the framework without affecting other modules of the framework or any existing objects in legacy applications. In addition, it supports multiple on-line payment methods simultaneously within the framework. Therefore, vendors and even customers can select any supported payment methods to make a payment.

Chapter 4

Thesis Project

The thesis project is the design and a first-stage software development of Clearinghouse: a payment framework for distributed systems using the ORBlite communication infrastructure. Clearinghouse provides "pay-per-call" capability to object invocations. This is different from most of the current payment paradigms in which payments are separate from purchased objects (or object descriptions). Clearinghouse has chosen a "pay-per-call" model to provide a suitable environment for deploying micropayment, since a "per-call" payment does not require additional network calls. Clearinghouse provides a *paymethod abstraction layer* to support different payment methods simultaneously. The paymethod abstraction layer allows individual payment methods to evolve without requiring changes made to the payment framework or objects being paid by the payment methods. Thus, Clearinghouse users do not need to suffer from implementing an entire payment framework for a new payment method. By using the existing of ORBlite framework, Clearinghouse users can make "per-call" payments using the interface of ORBlite, with an additional attribute which will be transparent to the ORBlite implementation. By using electronic wallets which locally reside in Clearinghouse users' processes, the users may take advantage of the paymethod abstraction layer to select their own sets of payment methods (either micro- or macropayment schemes) and configure them before issuing a "per-call" payment. In other words, Clearinghouse allows the users to add or delete their favorite payment methods in their wallets, unlike existing payment frameworks, which

give every user the same wallet. In addition to offering a payment service, Clearinghouse allows a user to determine whom to trust, and it lets an object provider (either a vendor or a broker) determine the pricing model for the object.

A design restriction of developing a payment framework within ORBlite is conformance with the ORBlite architecture; the payment framework should not change the existing implementation of the ORBlite core or any of the ORBlite abstraction layers. The design issues of building a paymethod abstraction layer are to form a unified transaction model for each payment method and build an interface based on this model. In order for a Clearinghouse wallet to configure a "per-call" payment, the wallet should provide ways to configure the payment methods.

The implementation goal of this thesis project is to prototype this framework. The major part of the payment framework is the design of the paymethod abstraction layer. A payment method will be shown to demonstrate the use of the paymethod abstraction layer with a user wallet. The implemented payment method only handles payments at this time. As later sections show, the redemption process can be modeled much in the same way as the payment process and therefore is considered as the next step for the framework development.

The implementation of Clearinghouse is formed by a set of data types which forms the abstraction layer, the wallet, a payment-capable communication protocol, and a payment method derived from the paymethod abstraction layer. The communication protocol is an enhanced implementation of IIOP (Internet Inter-Orb Protocol), a RPC OMG-standard protocol.[1] The enhanced IIOP is responsible for the network communication between the client and the server of the payment framework. The project implements a payment method called PayWord (a micropayment protocol proposed in [20]) to demonstrate the use of the payment framework.

[1]ORBlite includes an implementation of IIOP

Chapter 5

Overview of Clearinghouse

This chapter provides the overview of the payment framework. It shows how a customer issues a payment and sends it to a vendor, and it describes the trust model of Clearinghouse.

5.1 Overview of the Payment Framework

This section gives a general idea on how Clearinghouse operates by describing a purchase made by an ORBlite user. The user will get the change for the payment from a vendor and verifies its validity. Figure 5-1 graphically shows how an ORBlite user issues a payment and gets a change back from a vendor.

5.1.1 Client: Issuing an ORBlite Payment

To make a remote object invocation, the ORBlite user has an ORBlite-compliant front panel which interacts with the stub generated by an IDL compiler. To issue a "per-call" payment, the end-user application includes an _orblite_slip as an additional attribute to the remote invocation. The attribute is transparent to the rest of the ORBlite framework. When the request for a remote invocation reaches a payment-enabled communication protocol (such as PP, which will be discussed in section 11), PP_Client extracts the _orblite_slip from the ORBlite client's request. The _orblite_slip will be forwarded to the ORBlite user's wallet. The wallet then

sends the request for issuing a payment of a specific payment method through the paymethod abstraction layer. The specific payment method generates a payment and returns it to the wallet. The wallet also obtains payment-specific information for change through the paymethod abstraction layer, and the job dispatching is done in the same way as getting the payment. The wallet puts the results into a request_slip which will be forwarded to PP_Client.

5.1.2 Server: Verifying the Payment and Issuing an Change

The server of the payment-enabled communication protocol (PP_Server) receives the request_slip. PP_Server forwards the request_slip to the price module. The price module queries the price for the object invocation. Then it calls the vendor's wallet to verify the contents in the request_slip. The vendor's wallet unpacks the request_slip and dispatches the payment to the specific payment method through the paymethod abstraction layer, provided that the payment method is supported by the abstraction layer. Once the payment is verified, the vendor's wallet asks the specified payment method to generate a change through the abstraction layer. After that, the vendor's wallet creates an ack_slip, containing a receipt (change for the transaction) and the status of the transaction, and sends it back to the customer through PP_Server.

5.1.3 Client: Verifying the Change

Similar to the mechanism of issuing the request_slip, PP_Client forwards the ack_slip to the client's wallet. The client's wallet unpacks the ack_slip and forwards the receipt to a specific payment method through the paymethod abstraction layer. The payment method verifies the contents of the receipt. The result will be forwarded to PP_Server.

PP_Server puts the result of the verification of change and the price quoted by the server into the original _orblite_slip. PP_Server then sends the _orblite_slip back to the customer through the ORBlite.

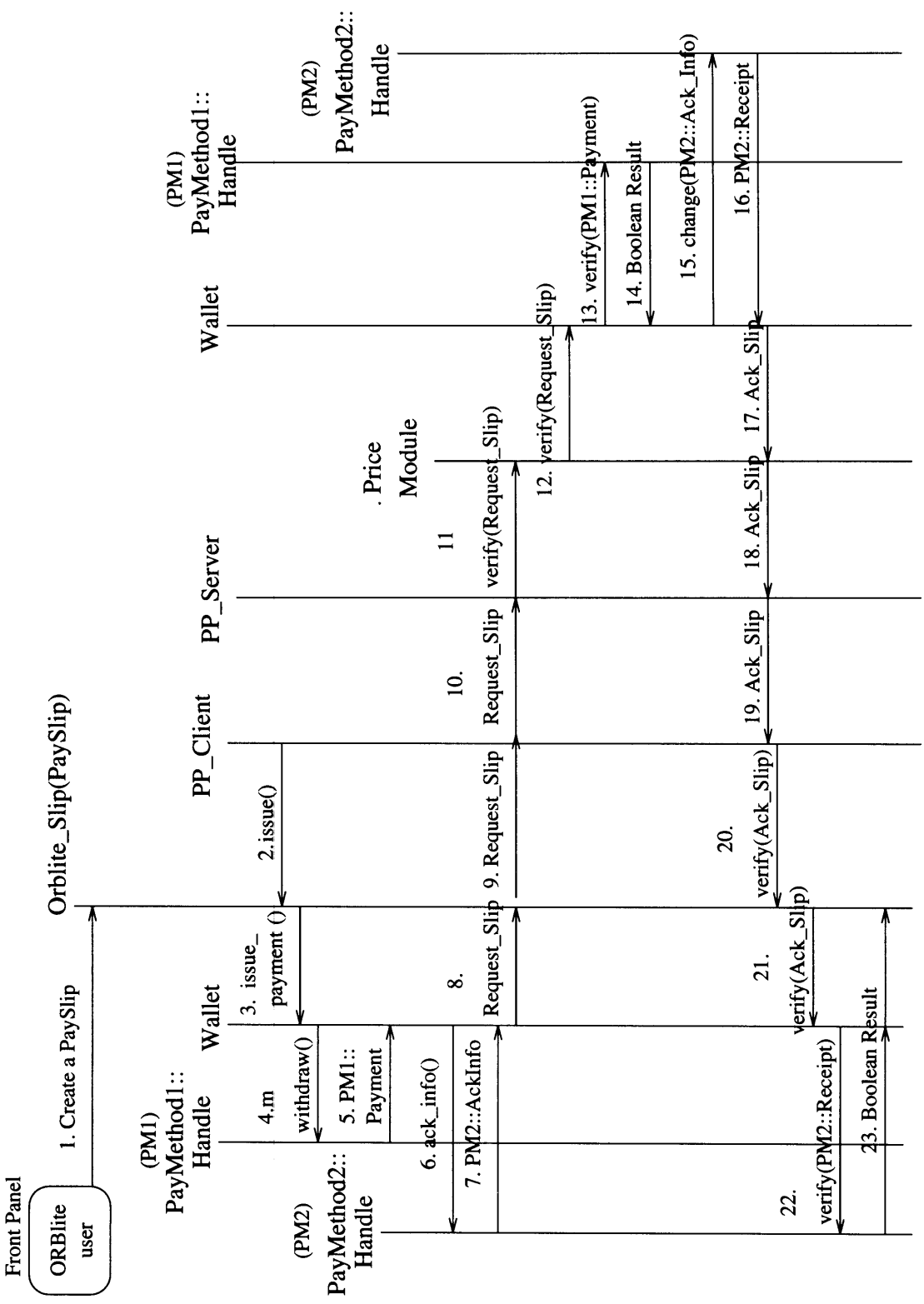

Figure 5-1: The steps of issuing a payment and getting change in Clearinghouse

5.2 Architectural Overview of Clearinghouse

Figure 5-2 shows the architectural overview of Clearinghouse. A detailed description of each module will be described from Chapter 6 to Chapter 13. PayMethod is the implementation of the paymethod abstraction layer. It manages payment methods currently supported in the framework and provides a separation between the implementation of individual payment methods and their behavior. PayWord is a mirco-payment protocol, and it is implemented according to the interfaces of the paymethod abstraction layer. It relies on Date and PGPGlue to generate digital signatures for payword commitments, and it uses MDGlue to generate payword hash values. User wallets are formed by a group of classes which make use of PayMethod to get a transparent and uniform way to access and configure individual payment methods. It is also responsible for assembling instances of Request_Slip and Ack_Slip. _Orblite_Slip is used to trigger Wallet to issue and verify request_slips or ack_slips. Its derived class, PaySlip (or RedeemSlip) is used as a transparent attribute in an ORBlite invocation to indicate an ORBlite user's wish to issue a payment for the object call. An ORBlite user, with a proper PGP keyID and a pair of public and private keys, configures the payslip.[1] Request_Slip and Ack_Slip contains a payment method data in an byte-stream form. They get the byte-stream representation through the interfaces of the paymethod abstraction layer. PP is a commerce-enabled, modified IIOP. It locates the _orblite_slip within the attribute list of an object invocation and triggers it to issue a request_slip. PP provides a communication channel for sending out the content in request_slips and ack_slips. PriceModule provides the price of the object invocation and is responsible for requesting Wallet to verify request_slips and issue ack_slips. Date, PGPGlue, and MDGlue (belong to a group of classes called Pay_Util) rely on external modules (system time, PGP [3], and MD5 [19] respectively) to provide common utilities for PayWord (and possibly to other parts of Clearinghouse in the future). AuthInfo and Certificate also belong to Pay_Util; they are the data

[1]This is because currently the authentication and authorization data in Clearinghouse is PGP-oriented. In the future, users may be able to use any type of security mechanisms supported by Clearinghouse.

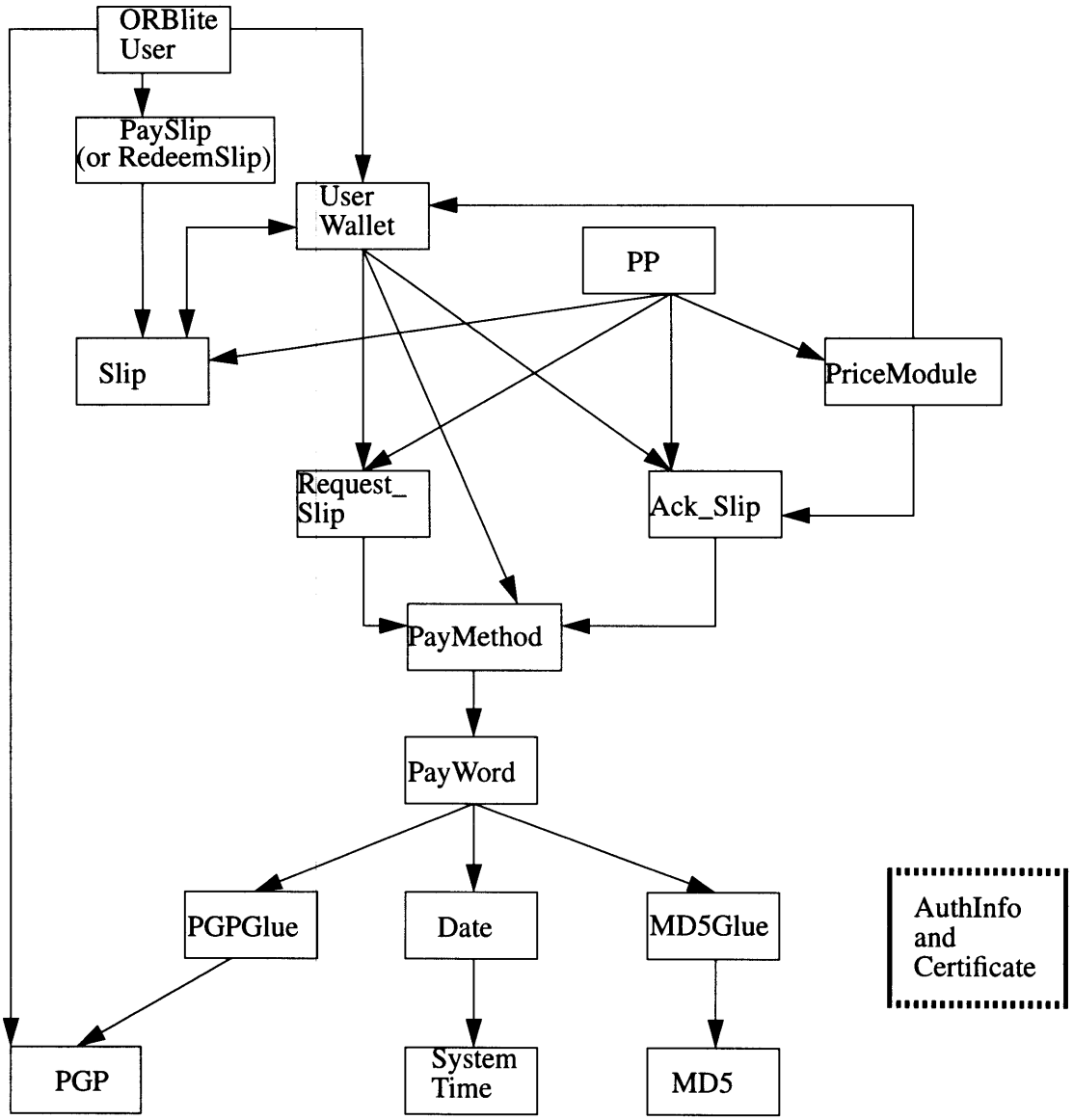

Figure 5-2: Architectural Overview of Clearinghouse. AuthInfo and Certificate are depended by most modules.

structures for user authentication and authorization within Clearinghouse.

5.3 Trust Model

A trust model is a set of rules and assumptions which form the skeleton of a security system. The security assessment of a system should evaluate the system against its trust model.

The trust model of Clearinghouse assumes that data are intact between the entry point of the ORBlite framework and the point right before the data are sent on to a network. In other words, only the network is not trusted. Under this assumption, data can be stored in cleartext format within the ORBlite process, and they only need to be encrypted if they are exposed to the communication channel. The communication protocol therefore does not need to be a secure medium; however, it should be reliable. The integrity and authenticity of the data which are transmitted through the communication protocol are enforced by the security measures of a payment method.

The action of an ORBlite user (a human or a computer process) cannot be controlled by Clearinghouse. They have to be trusted for providing the correct information. However, their exposure to the implementation of the payment framework and that of ORBlite should be minimal. They are only allowed to see the function declaration for classes AuthInfo, Certificate, C/V/BWallet, EWallet, PaySlip, and RedeemSlip.

Chapter 6

PayMethod Abstraction Layer

6.1 Design Issues

The purpose of the paymethod abstraction layer is to separate the implementation details of each specific payment method from the use of the payment method. By maintaining a stable API, the abstraction layer allows each payment method to evolve independently of the payment framework. To provide such a flexibility, the paymethod abstraction layer needs an interface that will cover any payment methods.

6.1.1 Payment-Cycle Modeling

The interface is designed based on the modeling of payment cycles. In a payment cycle, there are four entities: a customer, a vendor, an issuing broker and an acquiring broker, and there are four types of transaction processes being modeled: payment, redemption, change, and reimbursement.

A payment is a transfer of funds from one entity to another. The payer is viewed as a customer, while payee a vendor. In some payment methods, such as E-cash [2], money can only be used for payment once. For this reason, a payment in Clearinghouse is modeled as a transfer of "unspent" money from a customer to a vendor.

A redemption can be viewed as a transfer of "spent" funds from one entity to another. The entity that issues a redemption is modeled as a vendor, while the one

accepts the redemption is a broker. A redemption process is very similar to a payment process, except that the states of the funds are different.

Change is a response to an excessive payment, and it can be modeled in two ways. It can be another payment in response to a payment having been sent in the reversed direction. It can also be modeled as a transfer of "unspent" money from one entity to another. However, the state of the money will remain "unspent," which means that the recipient of the money can later on use the money for a payment.

Reimbursement is a response to the clearance of a redemption. It should provide a response to a redemption in the same way as a change to a payment. In other words, a reimbursement can be modeled as a change for a redemption.

Billing is not modeled in Clearinghouse. The reason is that billing is often a separate process and is handled in a separate channel.

6.1.2 Credit-Based and Debit-Based PayMethod

A payment method may be credit-based or debit-based. In a credit-based payment method, money is generated by a customer in the payment cycle. The money does not require a direct certification by a broker, and the customer will be billed after the payment takes place. In a debit-based payment method, a customer has to purchase the specific form of money before using the payment method [1].

The interactions among the entities in a credit-based payment method are quite different from that in a debit-based payment method.[1] The differences are shown in figure 6-1. In a debit-based payment method, the customer issues a payment to purchase money from an acquiring broker.[2] In a credit-based payment method, this step is modeled as a billing process. For both types of payment methods, there is always a payment between the customer and the vendor. A debit-based vendor does

[1] In this case, a debit-based payment system means that the payment method is debit-based from the viewpoints of all parties. The term "credit-based" also carries the same implication. This statement has to be made clear since there are payment methods which are in mixed mode (e.g. In MicroMint, customers have to purchase tokens from brokers in advance, while vendors have to redeem tokens from broker.) [1, 20] . It will be very confusing to ORBlite users if they try to reason the model with a mixed-mode payment methods

[2] For example, in Millicent [4], the customer has to purchase vendor's scrip using broker's scrip.

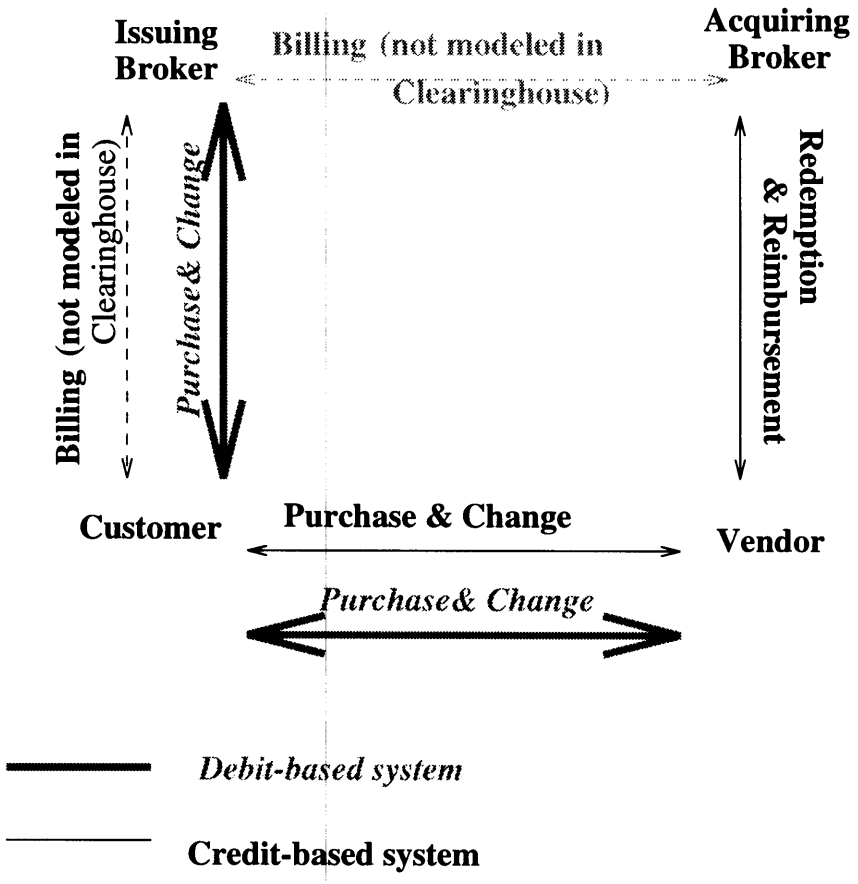

Figure 6-1: Payment-cycle model for credit- and debit-based payment methods in Clearinghouse.

not need to redeem a payment from the acquiring broker since a customer has paid for it in advance. A credit-based vendor, however, has to issue a redemption to the acquiring broker. The interactions between the issuing and the acquiring broker are always modeled as billing processes.

6.1.3 PayMethod Entities

Based on the modeling discussed in the previous two chapters, a payment method should provide the following three entities:

- Customer : A customer holds "unspent" payments.

- Vendor : A vendor should collect payments and turn them from an "unspent" state to a "spent" state.
- Broker : A broker collects "spent" payments.

The modeling of issuing brokers and acquiring brokers has been merged; this is because the relationship between the two is not modeled in Clearinghouse, and merging the two entities will not affect the operations of the rest of the entities.

6.1.4 PayMethod Data

A payment method should also have data structures for a payment, redemption, and receipt. A receipt is a merged data structure for change and reimbursement. As discussed earlier in section 6.1.1, they are modeled as the same data in Clearinghouse.

6.1.5 PayMethod Functions

To generate a payment, redemption, and a receipt, a payment method requires a set of data generating functions: withdraw, redeem, change, and reimburse. The data also require a set of verifying functions and functions which "adjust or undo" the effect of data generation and verification.

However, depending on whether a payment method is a credit-based, debit-based, or even a mixed-mode payment method, the above functions may belong to different entities for different payment methods. For example, in Millicent [4] (a debit-based payment method), a change has its own data structure and is issued by a Millicent broker or vendor. While in PayWord [20] (a credit-based payment method), a change can be modeled as a payment and thus can be issued by a PayWord customer. Therefore, the abstraction layer should not declare interfaces for the PayMethod functions in PayMethod entities.

6.1.6 Configuration

Each payment method may have its own set of parameters. For example, in PayWord [20], the denomination of a chain and the validity of a commitment should be configurable by Clearinghouse users. This set of data should not be pre-defined in Wallet or other data types within Clearinghouse, since this will violate the paymethod abstraction layer. Derived paymethods should be allowed to provide its own configuration parameters to Clearinghouse users.

6.2 Implementation Details: PayMethod

PayMethod is the implementation of the *paymethod abstraction layer*. It is an abstract base class, and it defines the interfaces on which a payment method (represented as a derived class) is based. Besides forming a skeleton for the design of derived payment methods, this paymethod abstraction layer maintains all payment methods currently supported by the payment framework. PayMethod consists of nine nested classes which form a basic structure for derived payment methods. These nested classes are: Info, Handle, Customer, Vendor, Broker, Payment, Redemption, Receipt, and AckInfo.

6.2.1 PayMethod::Handle

PayMethod::Handle is the core of the paymethod abstraction layer. It is the entry point for Wallet to reach other nested classes in PayMethod. PayMethod::Handle has three functionalities. It is responsible for declaring the interface for derived payment methods, creating (and deleting) derived paymethod data, and managing payment methods currently supported by the abstraction layer.

Job Dispatching Interface

The following is a list of job dispatching functions of PayMethod::Handle:

```
virtual PayMethod::Payment *withdraw(const AuthInfo &,
                                     const Certificate &recipient,
                                     const Orblite::ULong amt) = 0;
virtual PayMethod::Redemption *redeem(const AuthInfo &,
                                      const Certificate
                                      &recipient) = 0;
virtual PayMethod::Receipt *change(const AuthInfo &,
                                   const PayMethod::AckInfo &,
                                   const Orblite::ULong amt) = 0;
virtual PayMethod::Receipt *reimburse(const AuthInfo &,
                                      const PayMethod::AckInfo &,
                                      const Orblite::ULong amt) = 0;
virtual PayMethod::AckInfo *ack_info(const AuthInfo &) = 0;

virtual Orblite::Long verify(const AuthInfo &,
                             PayMethod::Payment &) = 0;
virtual Orblite::Long verify(const AuthInfo &,
                             PayMethod::Redemption &) = 0;
virtual Orblite::Long verify(const AuthInfo &,
                             PayMethod::Receipt &) = 0;
virtual Orblite::Boolean readjust(const AuthInfo &,
                                  PayMethod::Payment &) = 0;
virtual Orblite::Boolean readjust(const AuthInfo &,
                                  PayMethod::Redemption &) = 0;
virtual Orblite::Boolean revert(const AuthInfo &,
                                PayMethod::Receipt &) = 0;
virtual Orblite::Boolean revert(const AuthInfo &,
                                PayMethod::Payment &) = 0;
virtual Orblite::Boolean revert(const AuthInfo &,
                                PayMethod::Redemption &) = 0;
```

As discussed in section 6.1.5, the generation, verification, and reversion of a payment, redemption, and receipt may be defined in different entities (nested classes) in different derived payment methods. PayMethod::Handle has declared a set of interfaces (C++ pure virtual functions) so that derived instances of PayMethod::Handle can redirect incoming requests to the entities implementing the actual functions. Therefore, derived payment methods have the flexibility of implementing the generating, verifying, and reverting functions in different entities. Derived paymethod handles are expected to have references to the their entities.[3]

[3]The referencing should have been done in PayMethod with an additional set of functions handling

Creating Paymethod Data

Class PayMethod::Handle also provides functions to create empty instances of derived paymethod data. They are used for demarshalling data from a stream back to derived paymethod data (discussed further in section 10.1).

The following is a list of functions for creating paymethod data in PayMethod::Info:

```
public:
    static Payment *create_payment(const Orblite::Identifier &tag);
    static Redemption *
         create_redemption(const Orblite::Identifier &tag);
    static Receipt *create_receipt(const Orblite::Identifier &tag);
    static AckInfo *create_ackinfo(const Orblite::Identifier &tag);
private:
    virtual Payment *payment() const = 0;
    virtual Redemption *redemption() const = 0;
    virtual Receipt *receipt() const = 0;
    virtual AckInfo *ackinfo() const = 0;
```

Managing Currently Supported Payment Methods

PayMethod::Handle maintains a list of instances of derived PayMethod::Handle which have been linked into the payment framework. These functions for managing the currently supported payment methods are not to be inherited. PayMethod::Handle declares the interface for creating and deleting any payment methods. Wallet locates the desired payment method handle and asks it to create an instance of its own type. The deletion is handled by the paymethod abstraction layer, and it deletes any handles in a user wallet (even the ones that are no longer supported by the abstraction layer) as long as the action is authenticated. Each derived PayMethod::Handle has to provide implementation for creating and deleting a handle.

When each derived payment method is linked to the payment framework, the payment method will automatically register itself to the instance list of PayMethod::Handle. The implementation of this registration technique is the same as that of the transport abstraction layer in ORBlite [9].

---

down casting of the references to the derived type.

6.2.2 PayMethod::Info

PayMethod::Info is referenced by PayMethod::Handle. It is the only nested class whose derived instances will be exposed to an ORBlite user. It provides a means to an ORBlite user to configure default parameters of each derived payment method. Providing a configuration channel for each payment method removes the need to pre-define a set of parameters for different payment methods in user wallets.

6.2.3 PayMethod::Customer/Vendor/Broker

The actual job requests from wallet, such as withdraw and change, are implemented by the derived classes of PayMethod::Customer, PayMethod::Vendor, and PayMethod::Broker. PayMethod::Customer/Vendor/Broker provide very light interfaces. This provides the flexibility for the derived payment method to decide which entity should actually handle a dispatched job. The reason for defining the three entities is to provide a logical separation among the job requests from a wallet. Functions in a derived class of PayMethod::Customer should only handle "unspent payments." A derived PayMethod::Vendor is responsible for turning "unspent payments" into "spent payments." A derived PayMethod::Broker verifies "spent payments."

6.2.4 PayMethod Data

PayMethod Data is a general term for the nested classes whose derived instances will be sent through the communication protocol. These nested classes are: PayMethod:: Payment, PayMethod::Redemption, PayMethod::Receipt, and PayMethod::AckInfo.

As the names suggested, PayMethod::Payment and PayMethod::Redemption are the abstract base classes for a payment and a redemption respectively. PayMethod:: Receipt contains a change for a payment or a reimbursement for a redemption. PayMethod::
AckInfo provides the specification to Clearinghouse servers to generate change or reimbursement in a specified payment method.

Derived PayMethod data have to be sent through a communication protocol. An

IDL compiler generates the marshaling and demarshalling functions automatically and is a quick way to create a transmittable data representation for PayMethod data. The use of IDL-generated types will be seen often in other classes, such as PayWord, where data are going to be sent through a communication protocol.

The following shows the class declaration of PayMethod::Payment. The other three classes have the same function declarations.

```
class Payment
{
public:
  virtual ~Payment();
  virtual Orblite::Boolean
          marshal(_Orblite_Transport_OutStream &) = 0;
  virtual Orblite::Boolean
          demarshal(_Orblite_Transport_InStream &) = 0;

  virtual const Orblite::Identifier &tag() const = 0;
  static const Orblite::Identifier &type();
};
```

As a result of being transmittable data types, the four abstract base classes declare two pure virtual functions: marshal() and demarshal(). In addition, tag() is declared as a pure virtual function to obtain the name of the derived payment method. As explained later in section 10.1.2, it is important to accompany a payment data with this information, since data being demarshaled from the wire are series of bytes which do not provide any information to turn the data back to their original forms. Function type() is defined for the same reason. Since all derived classes of each payment data are of the same type, either be Payment, Redemption, Receipt, or AckInfo, this function needs not be virtual at all.

Figure 6-2 gives a structure of the PayMethod abstraction layer. The dotted lines mean that the dependencies between PayMethod::Handle and PayMethod entities are to be defined by PayMethod's derived class. The heights of the boxes reflect the number of functions defined for the classes.

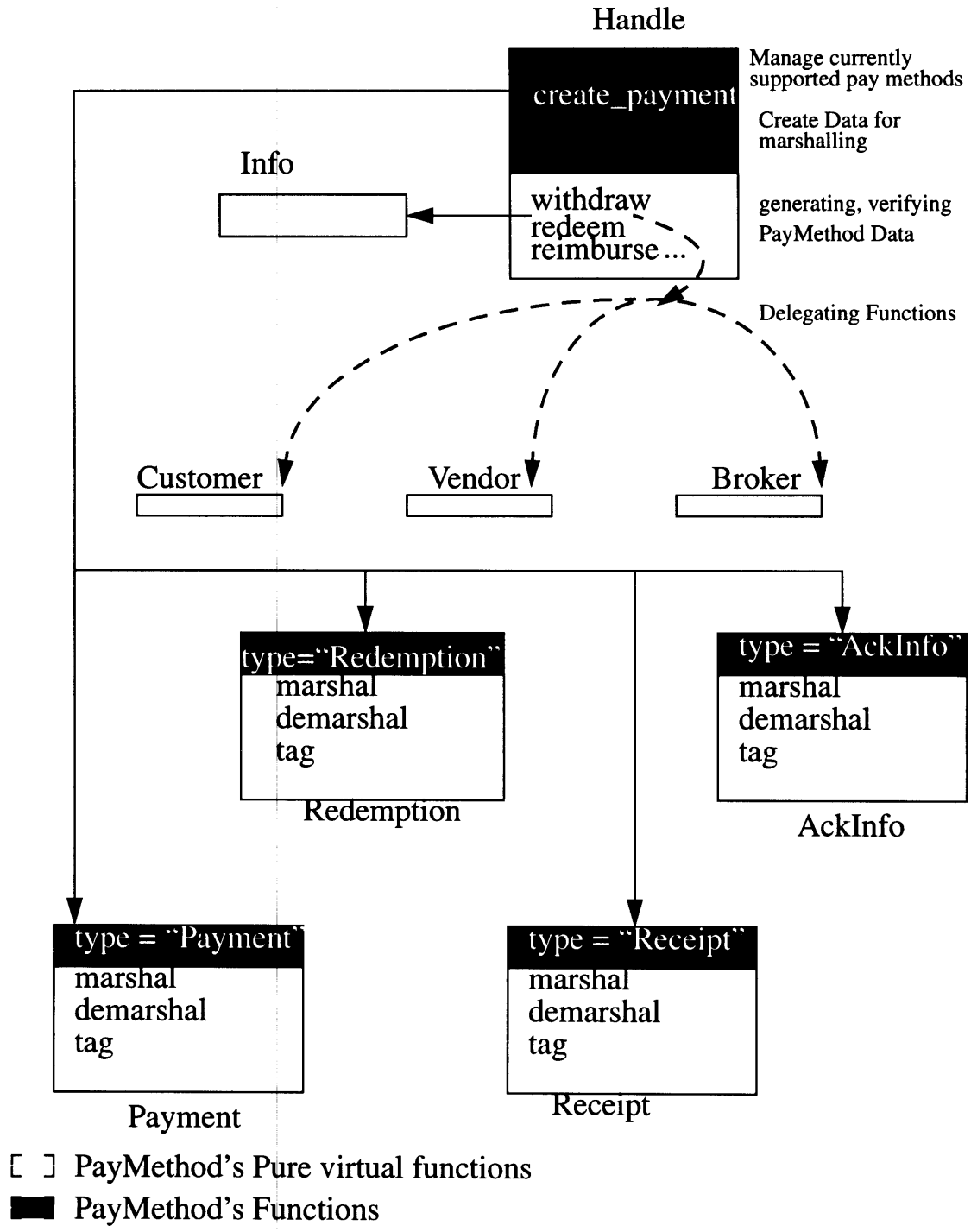

Figure 6-2: The module dependency diagram of PayMethod, the implementation of the paymethod abstraction layer.

Chapter 7

PayWord

7.1 Design Issues

PayWord is chosen as the payment method to demonstrate the use of the paymethod abstraction layer. The selection is based on two reasons. Firstly, an invocation of an object may cost infinitesimal amount. Therefore, the transaction should be handled by a payment method that provides a decent level of security and yet be computation- or storage-wise economic for small payments. PayWord amortizes the use of strong encryption algorithms (e.g. public key cryptography with long key lengths) to bring down the overhead cost of transactions.

Secondly, PayWord is a credit-based micropayment scheme. Money in PayWord is generated by customers. This has the advantage over a debit-based payment method in the framework development, since a debit-based payment method requires another payment method to acquire tokens prior to a payment.

7.1.1 The PayWord Protocol

The following is a summary of the PayWord protocol extracted from [1]. A detailed description of PayWord can be found in [20].

Introduction

The PayWord protocol is proposed together with MicroMint. PayWord is a credit-based protocol to a customer, vendor, and broker. It is based on a chain of hash values, called paywords [20]. Each payword represents a particular denomination or unit of value.

Paywords, Commitment, and User Certificate

Paywords are generated by a customer. They can be generated in advance or at the time of a purchase. To make a payword chain, a customer picks a random number as the nth payword, w_n, and it is the "seed" for generating the rest of the paywords in the chain according to the following rule:

$$w_{i-1} = h(w_i); \qquad where\ i = 1, ..., n \tag{7.1}$$

where h is a cryptographically strong function, such as MD5 [19], which has to be one-way and collision-resistant. The last value computed, w_0, is not part of the payword chain; it is the "root" of the chain and is embedded in a user-vendor-specific commitment.

A commitment authenticates w_0, and w_0 verifies the payword chain ($\{w_1, \ldots, w_n\}$). The chain is committed to a particular user-vendor relationship once its w0 has been bound to a commitment (M). M is a signed message by a customer (shown in EQ 7.2). It consists of w_0, a vendor identity V, the customer's certificate C_U, an expiration date D, and other information (I_M) necessary for the commitment.

$$M = \{w_0, V, C_U, D, I_M\}_{SK_U} \tag{7.2}$$

Before any transaction takes place, the customer must get a user certificate from a broker. The certificate (C_U) authenticates the customer's public key, which is used to sign a commitment during a purchase. C_U is a signed message consisting of the broker's identity, the customer's public key, the expiration date, and other related information.

$$C_U = \{Broker\_id, customer\_public\_key, expiration\_date, other\_info\}_{SK_U} \quad (7.3)$$

The three types of data work together to provide a secure purchase. A payword is sent unencrypted, but it is authenticated by a user-vendor-specific commitment. The user's public key used for signing the commitment is in turn authenticated by the user's certificate.

Making Purchases

First, a customer sends a vendor a commitment (M, defined in EQ 7.2). The vendor decrypts M with the customer's public key and verifies V and D. The customer signature is proven by C_U. Thus, a third party cannot forge a commitment by signing it with an invalid key, nor can he pretend to be the customer without knowing the customer's private key. If M is verified, the vendor stores it until it expires.

When the customer wants to make a purchase with one payword, he sends a pair, $P = (w_i, i)$, , where $1 \leq i \leq n$. The vendor will see if $h(w_i)$ equals the payword previously sent. If verified, the vendor stores this last received payword pair (P_{last}), and the payment is considered valid. Figure 7-1 gives a summary of the Payword protocol (including redemption, which will be discussed in next subsection).

If a purchased item costs more than the value of one payword, the customer can pay more by skipping paywords. Assuming that the next unspent payword is $w_i + 1$, each payword is worth one cent, and the item costs 5 cents, the customer can skip the first four unspent paywords and send $(w_i + 5, i + 5)$. The vendor verifies this pair by hashing the payword 5 times.

Redemption

A vendor only needs M and the last payword pair received (P_{last}) for redemption. A broker verifies M and makes sure the last payword can be hashed into w_i after last times. If every thing looks right, the broker debits the customer's account and credits

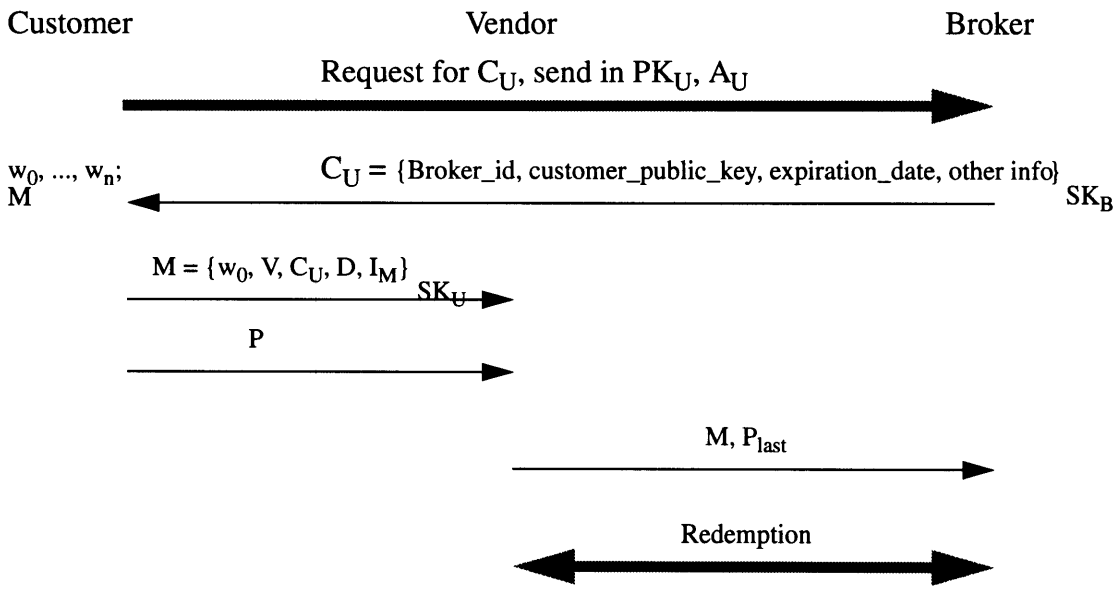

Figure 7-1: Summary of the Payword protocol. Thick arrows represent interactions using non-Payword protocols.

the vendor's.

Forgery Prevention

Spent paywords are the one-way hash values of the unspent paywords of the same chain; therefore, knowing the spent paywords should not expose the value of the unspent ones. A payword chain is authenticated by a commitment, and it is signed by the customer. The identity of the customer is ensured by the user certificate (C_U), and it is signed by the broker.

Double Spending Detection

A payment in the Payword protocol consists of a commitment and its corresponding set of paywords. Therefore, double spending in Payword means replaying the same paywords with the same commitment. The last spent payword and the root allow a vendor or a broker to keep track of all the spent paywords of a commitment. Replaying valid commitments can be found by searching for duplicates in a database. A vendor should store all the valid commitments received to prevent a customer from replaying a valid commitment. The broker should do the same to prevent the vendor from double depositing.

7.2 Implementation Details—PayWord

Class PayWord is a derived class of PayMethod. Besides the nine classes which are derived from the abstract base class, PayWord has implemented four additional classes: PayWord::Hash, PayWord_Chain, and PayWord_Commitment.[1] These four additional classes specific for the PayWord protocol and are used internally among the PayWord classes, and they will not be exposed to the paymethod abstraction layer. In the following subsections, the added four classes will be introduced first. Then PayWord_Payment, PayWord_RedemptionList, PayWord_AckInfo, and Pay-

[1]It seems that there is an inconsistent naming convention for PayWord. PayWord::Hash is defined as a nested class in an IDL file, while the rest are C++ implementation done by hand.

Word_Receipt will be discussed. After that, PayWord_Customer, PayWord_Vendor, and PayWord_Broker will show how the dispatched functions are categorized among the three according to the modeling of a credit-based payment method. Finally, the configuration capability of PayWord_Info will be discussed.

7.3 Implementation Details

7.3.1 PayWord::Hash

Design Issues

In the proposed PayWord protocol [20], the value of a payword hash represents a payment and is sent to the vendor. PayWord::Hash holds the hash value and is represented as a sequence of unsigned characters.

Implementation Details

PayWord::Hash contains a hash value of a payword chain, and its value has to be transmittable; therefore, it is represented as an IDL-generated type.

Currently, PayWord::Hash interacts with the rest of PayWord classes directly; however, a wrapper class could be defined to unify the naming convention of the PayWord classes.

7.3.2 PayWord_Chain

Design Issues

PayWord_Chain stores a series of payword hashes which are generated by the *root* (also a payword hash) of the chain. It is represented as a singly linked list of entries each of which holds a payword hash. It is used by all PayWord classes except for PayWord_Info and PayWord_Handle. PayWord_Chain is responsible for creating a series of paywords which are generated by a MD5Glue's function, verifying an incoming payword, and generate a payment.

PayWord_Chain also contains denomination and other data members required to keep track of the current status of the chain. Locks and mutexes are used to ensure that atomic insertion and deletion of entries are performed. Locks and mutexes are available from the standard ORBlite library.

Implementation Details

The following shows a subset of the class declaration of PayWord_Chain.

```
class PayWord_Chain
{
public:
  PayWord_Chain(Orblite::ULong amt, Orblite::ULong den = 1);
  PayWord_Chain(PayWord::Hash &root, Orblite::ULong den);
  ~PayWord_Chain();

  Orblite::Long verify(PayWord::Hash &, Orblite::ULong i);

  struct PayWord_Pair {
    PayWord::Hash pwh;
    Orblite::ULong pos;
  };
  PayWord_Pair payword(Orblite::ULong amt);

  PayWord::Hash root() const;
  Orblite::ULong length() const;
  Orblite::ULong amt_left() const;
  Orblite::ULong denomination() const;
};
```

PayWord_Chain has two constructors. The first one is used for chain generation when a payment is issued. It is stored in payword_customer. It takes in the total amount for the chain and the denomination as parameters; a chain created by this constructor will get a seed and automatically generate a series of payword hashes (according to EQ 7.1). The random number of the root is obtained from PGP::seed, and the series of hash values is generated by MD5Glue::create_md.

The second constructor is used for storing verified payword payments. This constructor takes in as parameters the root and the denomination from a received com-

mitment. The chain grows as subsequent payments are verified. This kind of chains is stored in payword_vendor and payword_broker.

To verify a payment, a payword chain will be given a received hash value and its corresponding position in the chain. The chain will hash the payword values down to the most recently received payword with the number of times determined by the index given to the chain. If the payword is valid, the result of the verification is the amount of the payment, which is determined by the number of hashes and the denomination of the chain. Forged paywords will fail the verification process described above. Replayed paywords can be detected by comparing the current length of the chain with the incoming index.

PayWord_Chain::payment returns a pair containing a payword hash and its corresponding position along a chain. The pair represents certain amount of money. PayWord_Chain::payment first takes in the amount to payment as a parameter. Then it determines the number of payword hashes required to be skipped based on the denomination of the chain and the requested amount, and returns a pair which contains a payword hash and its corresponding position along the chain. Once a payword hash is issued as payment, it will be deleted since it will not be used. Should a chain be restored, it can be regenerated from the unspent portion of the chain.

7.3.3 PayWord_Commitment

Design Issues

As mentioned in section 7.1.1, payword commitment is used for authenticating a payword chain. PayWord_Commitment holds a cleartext content of a commitment and its encrypted form. They are both IDL-generated data members. The cleartext representation allows other PayWord classes to obtain information easily, while the encrypted (and signed) part is the actual data sent on the communication protocol.

PayWord::CommitmentClear::Info

PayWord::CommitmentClear::Info contains the cleartext content of the commitment. It is an IDL-generated data structure. Representing the cleartext content of a commitment by an IDL-type eases the process of generating a signature for it. PayWord_Commitment has the root of a chain, the vendor's identity, an expiration date. However, unlike the commitment in EQ 7.2, the customer's certificate is not part of the the cleartext data. Currently, the signature generation algorithm uses an external module called PGP [3] through a class called PGPGlue (to be discussed in section 13.3, and it would treat the embedded user certificate as an ordinary input data. The actual inclusion of the user certificate has to be handled by the external module (in this case, PGPGlue::sign).

There are four other pieces of information in the cleartext: the customer's identity, the certifier's identity, the length and the denomination of the chain. This customer's identity is a keyID listed in PGP's default private key ring (secring.pgp) [3]. The customer identity is given to PGPGlue to locate the private key of the customer in PGP's default private key ring. PGPGlue then uses the private key to generate the signature for the commitment. The certifier's identity is a KeyID of the entity certifying the customer's public key. Currently, it is used for gathering a list of redemption for a broker.

The commitment has included the length and the denomination of the payword chain even though they are not required data in the original specification of PayWord [20]. They are used for specifying the maximum amount that the incoming chain is committed to. These two fields are configurable in a payword_info, and customers can use them to tailor the credit limit for each pair of customer-vendor relationship. This credit limit is not certified by the user certificate and is customer-vendor-specific.

PayWord::CommitmentEncrypt

PayWord::CommitmentEncrypt holds the signed and encrypted content of the commitment. It is represented as a sequence of characters. The sequence is obtained by first marshaling the cleartext data onto a character-based stream. A character string

is then extracted and passed to PGP::sign. PGP::sign generates an authenticated (signed with the customer's private key) and private (encrypted with the vendor's public key) character string for the input. This character string will be used to initialize PayWord::CommitmentEncrypt.

The encrypted part of the commitment is an IDL-generated data type for it can be used as a transmittable type easily. Its internal representation is a sequence of characters. A sequence of characters is chosen over a string because the string representation assumes inputs to be null-terminated. However, the series of encrypted characters is not so, and storing it in string will cause data truncation.

Implementation Details

The following shows part of the class declaration of PayWord_Commitment:

```
class PayWord_Commitment
{
public:
  PayWord_Commitment();
  PayWord_Commitment(PayWord::CommitmentEncrypt &);
  PayWord_Commitment(PayWord_Chain &chn,
  const Orblite::Identifier &vendor,
  const AuthInfo &,
  const Date &,
Orblite::String *other_info);
  PayWord_Commitment(const PayWord_Commitment &);

  Orblite::Boolean verify(const AuthInfo &);

  PayWord::Hash root() const;
  Orblite::ULong chain_len() const;
  Orblite::ULong denom() const;
  Orblite::Identifier vendor() const;
  Orblite::Identifier user_id() const;
  Orblite::Identifier certifier() const;
  Date dt() const;
  Orblite::String misc() const;

  Orblite::Boolean marshal(_Orblite_Transport_OutStream &);
  Orblite::Boolean demarshal(_Orblite_Transport_InStream &);
```

```
  Orblite::Boolean operator==(const PayWord_Commitment &) const;
};
```

The implementation of the cleartext representation of PayWord_Commitment is generated by an IDL compiler. Using an IDL-generated data structure allows the contents of the data to be marshaled into a character string which will be signed by PGP::sign.

The root of a payword chain is of type PayWord::Hash. It is obtained from the chain by calling root(). The vendor's identity is extracted from the vendor's certificate in _orblite_slip. The expiration date is generated using Date::create_expiration_time.

PayWord_Commitment::verify checks the encrypted part of the commitment using PGP::verifysig and convert the decrypted data back to the cleartext portion of commitment. It also verifies the ranges of denomination and the length of the chain, and ensures the existence of a root and a fresh timestamp in the commitment.

7.3.4 PayWord Data—PayWord_Payment

Design Issues

PayWord_Payment is a derived class of PayMethod::Payment. It contains a payword hash, its corresponding position of the payword of the chain, vendor's identity, customer's identity, an instance of PayWord_Commitment, and an IDL-generated data type.

Recall section 7.1.1, a payword payment contains only a payword hash and the index of the payword hash of the corresponding payword chain. PayWord_Payment has added the vendor's identity and a customer's identity. The vendor's identity lets the recipient check if the payment has been addressed to right entity. The customer's identity is used to locate the commitment which is previously sent to the vendor's site.

Besides the customer's and vendor's identities, a commitment can be included in a payment. A commitment is always included in the first payment generated from a new payword chain (see Figure 7-1). A secondary reason is that the customer may

re-send the commitment upon request regardless of the state of the chain.[2]

The IDL-generated data type holds the transmittable forms of the rest of the data members in payword payments. This IDL-generated type only includes the encrypted portion the commitment as its member, so that the cleartext portion will not be sent to the network.

Implementation Details

PayWord_Payment has to define the marshaling and demarshalling interfaces declared in PayMethod::Payment. The two functions delegates the actual work to the IDL-generated type.

The following shows the part of the class declaration of PayWord_Payment:

```
class PayWord_Payment: public PayMethod::Payment
{
public:
  PayWord_Payment(const PayWord_Commitment &,
                  const Orblite::ULong ,
                  const PayWord::Hash &);
  PayWord_Payment(const Orblite::Identifier &,
                  const Orblite::Identifier &,
                  const Orblite::ULong ,
                  const PayWord::Hash &);
  PayWord_Payment(PayWord_Payment &);
  PayWord_Payment();
  ~PayWord_Payment();

  const Orblite::Identifier &tag() const;

  PayWord_Commitment *cmnt() const;
  Orblite::ULong chain_pos() const;
  PayWord::Hash pwh() const;
  Orblite::Identifier vendor() const;
  Orblite::Identifier customer() const;

  Orblite::Boolean marshal(_Orblite_Transport_OutStream &);
```

[2]However, re-establishing the relationship between a customer and a vendor seems to only benefit the vendor, and it is not apparent that the customer is willing to do so.

```
  Orblite::Boolean demarshal(_Orblite_Transport_InStream &);
};
```

7.3.5 PayWord Data—PayWord_RedemptionList

As a credit-based system, PayWord has redemption. PayWord_RedemptionList represents the redemption of PayWord and is a derived class for PayMethod::Redemption. Its derived class, PayWord_RedemptionList, is an IDL-generated type, holding a sequence of the transmittable form of PayWord_Payment.

7.3.6 PayWord Data—PayWord_Receipt

PayWord_Receipt contains the change for a payment; it is derived from PayMethod:: Receipt. As mentioned in section 6.1.1, change can be modeled as a payment from a vendor to a customer. In PayWord_Receipt, this model is used. Thus, the internal representation of PayWord_Receipt is a PayWord_Payment.

7.3.7 PayWord Data—PayWord_AckInfo

PayWord_AckInfo derives from PayMethod::AckInfo. It allows the sender of a request_slip (containing a payment or a redemption) to provide the PayWord-specific information which is required to generate a change or a reimbursement. Currently, PayWord_AckInfo only contains the sender's identity, and is used as a parameter for encrypting a payword commitment.

7.3.8 PayWord_Handle

PayWord_Handle is a derived class of PayMethod::Handle; it provides the implementation for dispatching jobs to the entities of PayWord, namely PayWord_Customer, PayWord_Vendor, and PayWord_Broker.

As described earlier in section 6.1.3, PayMethod::Customer is responsible for giving out "unspent payments." As its derived class, PayWord_Customer is responsible for generating payword payments, and handling bounced payments as it may need to

change the state of the commitment. Moreover, it should generate change and reimbursement since both are modeled as payword payments in a credit-based payment method.

PayMethod::Vendor is responsible for turning "unspent payments" into "spent payments." PayWord_Vendor is the derived class of PayMethod::Vendor. It is responsible for verifying collected payments from PayWord_Customer and sending them to PayWord_Broker for redemption. Thus, it should be responsible for verifying a payment and a receipt, issuing a redemption, reverting the verified payments.

PayMethod::Broker is only responsible for collecting "spent payments." PayWord_Broker is its derived class and is responsible for verifying PayWord_Redemption List.

7.3.9 PayWord_Customer

PayWord assumes a long-term relationship between a customer and a vendor to amortize the use of public cryptography [1]; therefore, a payword customer has to maintain the state for each long-term relationship. A list is used to keep track of the current state for each customer-vendor-specific relationship. Each entry of the list consists of a commitment, the chain authenticated by the commitment, and the vendor's identity. Any modifications to the list have to be atomic actions, and this is ensured by locking the mutex of the list. Currently, the list imposes a policy that a customer can only establish one commitment with one vendor.

Issuing a negative payment is not allowed in PayWord since it indicates that the sender wants to charge the recipient who is expected to issue a redemption for the payment. It is doubted that the payee will issue such a redemption.

Implementation Details

The following shows part of the class declaration of PayWord_Customer. These functions are called by PayWord_Handle.

```
class PayWord_Customer: public PayMethod::Customer
```

```
{
  PayWord_Payment *withdraw(const AuthInfo &,
                            const Orblite::Identifier& vendor,
                            const Orblite::ULong amt);
  Orblite::Boolean readjust(const AuthInfo &, PayWord_Payment &);
  PayWord_Receipt *change_or_reimburse(const AuthInfo &,
                                       const Orblite::Identifier &,
                                       const Orblite::ULong amt);
  Orblite::Boolean revert(const AuthInfo &, PayWord_Receipt &);
};
```

PayWord_Customer::withdraw takes in as parameters a customer's authinfo, the amount of payment requested, and the recipient of the payment (the vendor). The function checks the sign of the amount. It also makes sure that the requested amount does not exceed the maximum amount of the credit limit imposed by payword info. PayWord_Customer::withdraw then searches for an entry for the vendor. If the search fails, an entry is created and filled with a newly generated payword chain and commitment based on the configuration provided in payword info. After that, PayWord_Chain::payword is called to retrieve a payword hash and its corresponding index. The result is put into a newly constructed payword payment together with the commitment.

If the customer and the vendor have established a relationship, and the remaining chain is sufficient for issuing the payment, no entries is created, and only PayWord_Chain::payword is called. The result is put into a newly constructed payword payment without the commitment. In the current implementation, if the chain has insufficient funds, the current entry will be deleted and a new one constructed. The original entry must be deleted to conform with the policy that only allows one entry in the list to associate with a vendor.

PayWord_Customer::readjust takes in the bounced payment as the parameter and adjusts the state of the "vendor list." In the current implementation, the entry is deleted. This implementation is applicable if the bounced payment is caused by the unavailability of the vendor or PayWord, or the payment was simply invalid. This implementation is not suitable if the bounced payment is caused by, say, the failure of the object invocation.

PayWord_Customer::change_or_reimburse simply redirects the call to PayWord_Customer::withdraw and then uses the result to create a payword receipt.

PayWord_Customer::revert voids a payword receipt (containing a change or reimbursement). Since a change and reimbursement is modeled as a payment, this function redirects the invocation to PayWord_Customer::readjust.

7.3.10 PayWord_Vendor

PayWord performs off-line redemption process; as a result, received payments have to be logged until the payword vendor issues a redemption. A "customer list" is used to store all the payments. This list has exactly the same structure and concurrency control mechanism as that of PayWord_Customer.

Implementation Details

The following shows part of the class declaration of PayWord_Vendor. These functions are called by PayWord_Handle.

```
class PayWord_Vendor: public PayMethod::Vendor
{
  Orblite::Long verify(const AuthInfo &, PayWord_Payment &);
  Orblite::Long verify(const AuthInfo &, PayWord_Receipt &);
  PayWord_RedemptionList *redeem(const AuthInfo &,
                                 Orblite::Identifier &broker);
  Orblite::Boolean revert(const AuthInfo &, PayWord_Payment &);
  Orblite::Boolean readjust(const AuthInfo &, PayWord_Redemption &);
};
```

PayWord_Vendor has two overloaded PayWord_Vendor::verify functions. One is used for verifying a received payment, while the other one is used for verifying a receipt. The first overloaded function verifies the commitment in the payment; the actual verification is handled by PayWord_Commitment. Once verified, it tries to locate the customer's commitment in existing entries of the "customer list." If the search fails, a new entry containing an empty chain is constructed. Then PayWord_Chain::verify is called to verify the payword hash in the payment and re-

turns the amount that the customer has given in the payment. The second PayWord_Vendor::verify function extracts the payment within the receipt and redirects the invocation to the first PayWord_Vendor::verify function.

PayWord_Vendor::redeem takes in a broker's identity as the parameter. The function then iterates the entire "customer list," takes out the payments certified by the broker, puts them into an instance of PayWord_RedemptionList, and returns it to PayWord_Handle. PayWord_Vendor::redeem belongs to the redemption process and thus has not yet been tested in the first-stage implementation.

PayWord_Vendor::revert is called if the verified payment has to be voided. It rolls back the chain of the entry. Currently, revert has not been tested yet.

PayWord_Vendor::readjust handles a bounced redemption request for a broker. Currently, this function belongs to the redemption process and is not implemented for in the first stage. It is predicted that this function cannot do more than storing this unanticipated event in a transaction log which will be reported to a judging authority.

7.3.11 PayWord_Broker

It should define PayWord_Broker::verify. It works similarly to PayWord_Vendor::verify, except that it has to verify a series of payments instead of one payment. The following shows the function signature of PayWord_Broker::verify:

```
class PayWord_Broker: public PayMethod::Broker
{
  Orblite::Long verify(const AuthInfo &,
                       PayWord_RedemptionList &);
};
```

7.3.12 PayWord_Info

The following shows part of the class declaration of PayWord_Info:

```
class PayWord_Info: public PayMethod::Info
{
public:
  void max_amount(const AuthInfo &, Orblite::ULong);
  Orblite::Long max_amount(const AuthInfo &);
  void denomination(const AuthInfo &, Orblite::ULong);
  Orblite::Long denomination(const AuthInfo &);
  void cmnt_valid_time(const AuthInfo &, Orblite::UShort);
  Orblite::Short cmnt_valid_time(const AuthInfo &);
};
```

PayWord_Info provides the capability to configure the timestamp of each newly created commitment, so the person who issues the commitment can decides its duration of validity. PayWord_Info also allows users to configure the maximum amount and the denomination of the chain. The two parameters are used to tailor the credit limit for each commitment. Another advantage of having a configurable denomination is to provide flexibility for customers to use chains of various units; however, the current implementation only allows each wallet to commit one chain to a vendor. The extension to supporting multiple commitments for each vendor is required to exploit this feature.

Figure 7-2 provides a module dependency diagram of the eleven PayWord classes. It is extended from the module dependency diagram of PayMethod to show the interaction between PayMethod's interfaces and the actual implementation of PayWord. Note that relationships between the internal classes and the rest of the classes are not drawn.

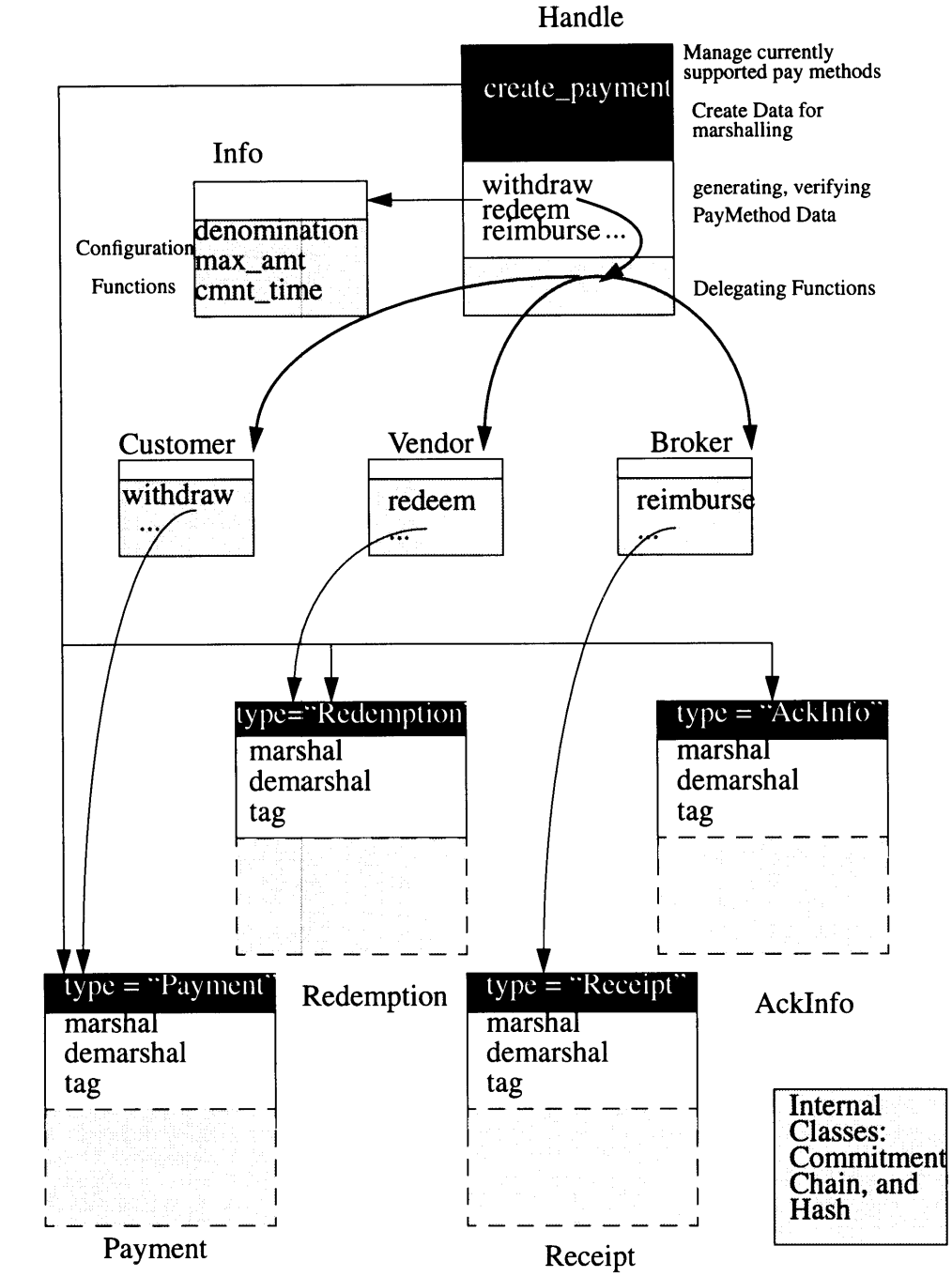

Figure 7-2: Module Dependency Diagram for PayWord classes.

Chapter 8

Wallet

The wallet of a Clearinghouse user has two responsibilities. First, it manages instances of derived payment methods belonging to the user. To maintain the evolvability for each payment method, the wallet has to interact with them through the paymethod abstraction layer. The wallet also needs to communicate with other internal data structures in Clearinghouse. The functionalities of managing the payment methods and interacting with the Clearinghouse core should not be exposed to a Clearinghouse user. Second, the wallet provides a channel for configuring default parameters for both the payment methods and a transaction; these functionalities are provided to authenticated users. In addition to the basic functions, a user wallet should provide room for future extensions.

To separate the access rights and allow extensions to a user wallet, the wallet of a Clearinghouse user is represented by three classes: Wallet, EWallet, and C/V/BWallet. Wallet is responsible for interacting with the internal data structures, including the paymethod abstraction layer. It is an abstract base class and its function declarations are published. EWallet derives from Wallet. It provides a set of functions for configuring default parameters for each transaction and providing access to the derived classes of PayMethod::Info through the object space of Wallet. CWallet, VWallet, and BWallet each derives from EWallet. They are used for providing future extensions of which the implementations may be different for different entities.

Figure 8-1 consists of three parts. Wallet is responsible for all the interaction with the paymethod abstraction and other core data types in Clearinghouse (not shown). EWallet makes use of Wallet's interface to provide the channel for configuring individual payment methods. C/V/BWallet are used for future extension.

8.1 Wallet

8.1.1 Design Issues

Wallet has three functions. First, it manages different payment methods of a user wallet. Second, it creates, unpacks, and dispatches the contents of request_slips and ack_slips to a payment method through the paymethod abstraction layer. Third, it provides a searching mechanism for the user wallet.

Wallet stores a set of user-authenticated payment methods in an singly linked list. Each entry of the list has a pointer to PayMethod::Handle, the abstract base class serving as the core of the paymethod abstraction layer. When a wallet dispatches actions to a specific payment method, it first locates the instance for the payment method and issues one of the calls from the uniform set of interfaces defined in PayMethod::Handle. If the requested payment method is not found in the singly linked list, Wallet calls the payment abstraction layer to create one, given that it is supported by the paymethod abstraction layer. Currently, Wallet always attempts to create a new paymethod handle if the requested paymethod handle has not yet existed in the wallet. However, an ORBlite user should be allowed to configure this option in the future.

Wallet is responsible for creating and unpacking request_slips and ack_slips. Wallet will create a valid request_slip or ack_slip based on the results of the actions dispatched to payment methods.

Wallet searches for instances of its own type. It provides a static function for Clearinghouse users to lookup or create wallet in the Clearinghouse process. Each instance of Wallet is authenticated by a unique authinfo of a Clearinghouse user, and

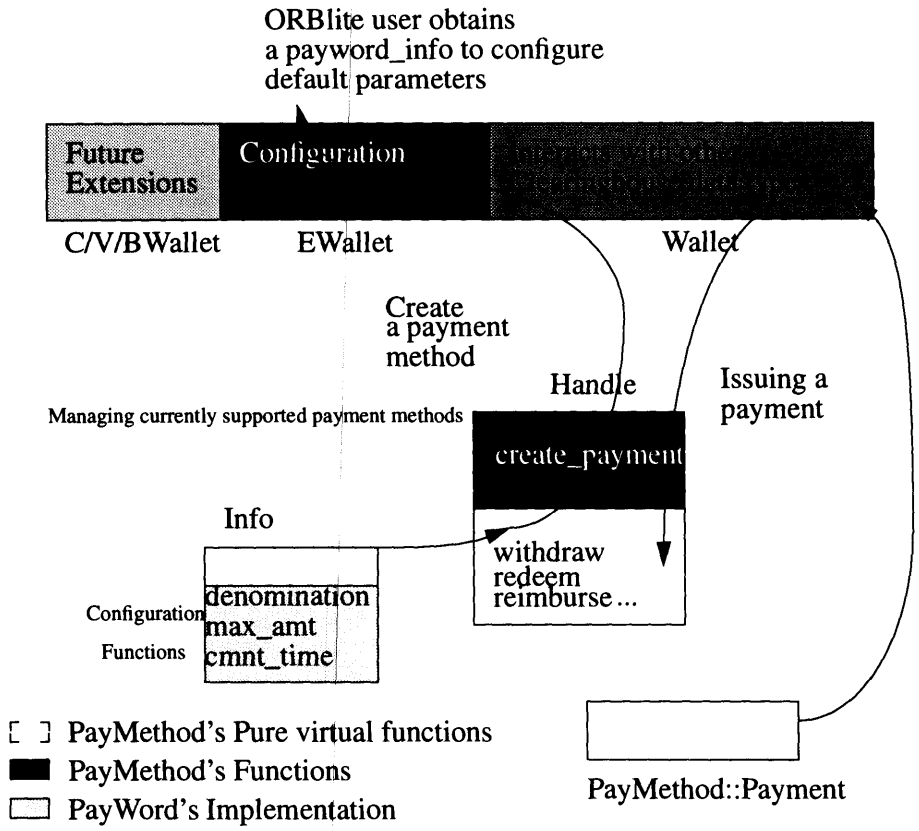

Figure 8-1: A user wallet interacts with PayMethod.

the authinfo has to be provided when the wallet is created.

8.1.2 Implementation Details

The following is part of the class declaration of Wallet:

```
class Wallet
{
public:
  static Request_Slip *issue_payment(PaySlip &);
  static Request_Slip *issue_redemption(RedeemSlip &);

  static Orblite::Boolean verify(const Ack_Slip &, _Orblite_Slip &);
  Ack_Slip *verify(const Orblite::Identifier &/* xport_id */,
                   const Request_Slip &, const Orblite::Long);

  Ack_Slip *revert(const Orblite::Identifier &/* xport_id */,
                   const Request_Slip &, const Ack_Slip &);

  virtual const Orblite::Identifier &wallet_type() const = 0;
};
```

Wallet::issue_payment creates a request_slip. The function is called by a pay_slip. It is a static function so that the pay_slip does not need to locate an instance of Wallet before invoking Wallet::issue_payment. The function uses the authinfo in the pay_slip to locate the user's wallet. If the wallet is found, issue_payment gets the bid from the _orblite_slip. The wallet then calls PayMethod::Handle::withdraw to get a payment and calls PayMethod::Handle::ackinfo to get a change.[1] The wallet then creates a request_slip and assembles the results into it.

Wallet::issue_redemption works similarly to Wallet::issue_redemption, except that is called by a redeem_slip to create a request_slip, and PayMethod::Handle::redeem is called instead of PayMethod::Handle::withdraw.

There are two overloaded Wallet::verify functions. The first one is called by the price module (at the server's side) to verify an incoming request_slip. Since the

[1] In the current implementation, the change method must be specified. This restriction should be relaxed in the future.

price module should locate the instance of a wallet before verifying the request_slip, Wallet::verify is not a static function. Wallet::verify retrieves the contents from the request_slip and invokes the PayMethod::Handle::verify to verify either the incoming payment or a redemption. If the incoming data is a payment, and the amount of the payment covers the price quoted by the server, a valid ack_slip (optionally includes the change) is returned. Otherwise, the payment is voided by calling PayMethod::Handle::revert of the same payment method's handle, and the payment is put in an ack_slip and bounced back to the client. An incoming redemption is handled in the same way.

The second overloaded Wallet::verify is invoked at the client side after an ack_slip is received by a pay_slip or a redeem_slip. The status of the request_slip is examined. If change is included, it will be verified by a payment method's handle specified by the request_slip.

Wallet::revert is used for reverting a payment at the server side. Wallet::revert is initiated by PP_Server through the price module. Wallet::revert is called when other modules in ORBlite causes an exception. For example, an object invocation of ORBlite has failed. Wallet::revert simply delegates the reversion to PayMethod::Handle::revert. Currently, the implementation of this function has not yet been tested.

8.2 EWallet

EWallet is an abstract base class derived from Wallet. The existence of this class is to provide common configuration operations for wallets of Clearinghouse users. These functions include adding payment methods, configuring default values for each payment method, specifying default payment or change methods to be used, querying all payment methods currently supported by the paymethod abstraction layer, locating all payment methods residing in the wallet, etc. Currently, functions for adding, removing, and configuring individual payment methods have been implemented. This specification of EWallet is given to Clearinghouse users.

EWallet::info takes in as the parameter the name of a payment method and returns

a derived instance of PayMethod::Info in a user's wallet. Thus, the user can configure default parameters for specific payment methods. EWallet::info accesses the list of paymethod handles located in class Wallet. The following shows the declaration of EWallet::info:

```
class EWallet: public Wallet
{
  PayMethod::Info *info(const AuthInfo &,
const Orblite::Identifier &mthd);
};
```

8.3 CWallet, VWallet, BWallet

CWallet, VWallet, and BWallet are derived from EWallet. C,V,B stand for Customer, Vendor, and Broker respectively. Currently, each ORBlite user can own only one wallet since an instance of a wallet is identified by authinfo which is kept within Wallet.

The three classes are relatively empty at this moment. They all have to provide a function called lookup_or_add to lookup or create a user's wallet. These classes can be independently extended to provide more functionalities to customers, vendors, and brokers. For example, these classes may keep track of the balances and the transaction logs for the users, and the functions may be defined differently in each class. In addition, the type of the wallet may restrict the access right to certain functions of payment methods. For instance, only BWallet can verify a redemption and issue a reimbursement.

Currently, only CWallet is supported. The specification of the three classes should be publicly available for ORBlite users. The following shows part of the class declaration of CWallet:

```
class CWallet: public EWallet
{
public:
```

```
  static CWallet *lookup_or_add(const AuthInfo &au_info);
  const Orblite::Identifier &wallet_type() const;
};
```

8.4 Wallets

Wallets is a container class of instances of Wallet. It is instantiated as a static member of class Wallet. Wallets is represented as a double link list, with each entry containing a reference to an instance of Wallet.

Chapter 9

_Orblite_Slip, PaySlip, & RedeemSlip

9.1 _Orblite_Slip

9.1.1 Design Issues

_Orblite_Slip is used by an ORBlite user to indicate the interest of sending a payment or a redemption. It is derived from a base class in ORBlite. Objects derived from this base class are used as attributes for other modules within the ORBlite framework. When an ORBlite user wants to issue a payment or a redemption, an _orblite_slip is included as a component in the instance of the base class and will be accessed by a commerce-enabled communication protocol.[1] The inclusion of an _orblite_slip in the attribute is entirely transparent to other parts of ORBlite.

Wallet could have served the role of Slip. However, the entire wallet has to be blocked from the moment that an ORBlite user issues the remote invocation to the time that the call is returned to the user. This is because some of the parameters, like the bidding price, have to be locked to ensure that the configuration for the object invocation will stay the same until the invocation is finished. This restriction will cause a bottleneck in a multi-threaded environment such as ORBlite. Therefore,

[1] In the current implementation, only PP (described in section 11) is available.

_Orblite_Slip is used to contain the configuration for each transaction, and instances of _Orblite_Slip can concurrently access a multi-threaded wallet.

A Clearinghouse user cannot provide more than one specification for a payment or a redemption for each object invocation. This restriction has been imposed by the base class of _Orblite_Slip.

9.1.2 Implementation Details

The following shows part of the class declaration of _Orblite_Slip:

```
class _Orblite_Slip :
         public _Orblite_CallInfo::Component
{
public:
  static _Orblite_Slip* lookup(const _Orblite_CallInfo &val);

  virtual Request_Slip *issue(const Orblite::Identifier
                              &transport_tag) = 0;
  virtual Orblite::Boolean verify(Ack_Slip &) = 0;

  void xaction_completed();

  // this price() declares the query interface
  // for PaySlip and RedeemSlip.
  virtual const Orblite::Long price() const = 0;

protected:
  // this price() sets the price in an _orblite_slip.
  void price(const Orblite::Long);
};
```

_Orblite_Slip defines the set of functions which have been declared in its abstract base class. These functions allow an instance of _Orblite_Slip to be accessed from the ORBlite attribute list (CallInfo). The addition function, however, is not defined in _Orblite_Slip since Clearinghouse or other parts of ORBlite should not be given permission to add the _orblite_slip into the list of attributes. This function is defined in PaySlip and RedeemSlip, so that the addition of slips can only be done by an ORBlite user.

Slip defines two pure virtual functions: Slip::issue and Slip::verify. The first one is called by a payment-enabled communication protocol to a issue a request_slip, and the second one is called to verify an ack_slip. However, they have to be defined by the derived classes since issuing a payment is different from a redemption. Based on the result of Slip::verify, the communication protocol sets the status of the transaction in the _orblite_slip, and this is done by calling _orblite_slip::xaction_completed().

Slip contains the authinfo of an ORBlite user, and it is used by Wallet to locate the user's wallet. Slip::price is used to set the price of the transaction by Wallet. This function is called regardless of whether the transaction was completed successfully. This policy allows the user to include an empty or an invalid _orblite_slip to query the price of an object invocation.

9.2 PaySlip

The following shows part of the class declaration of PaySlip:

```
class PaySlip: public _Orblite_Slip
{
public:
  Request_Slip *issue(const Orblite::Identifier &transport_tag);
  Orblite::Boolean verify(Ack_Slip &);

  static PaySlip *lookup_or_add(const AuthInfo &au_info,
Orblite::CallInfo &val);

  const Orblite::Long price() const;
  Orblite::Boolean xaction_is_completed();
  void refresh(AuthInfo &au_info);

  void buy(const AuthInfo &,
           const Certificate &from,
           Orblite::Long amt,
           _Orblite_Boolean set_wallet = _Orblite_FALSE);
  const Orblite::Long buy_amount(const AuthInfo &);
  const Certificate *buy_from(const AuthInfo &);

  Orblite::Identifier &pay_method();
```

```
  void pay_method(const AuthInfo &au_info,
                  const Orblite::Identifier,
                  Orblite::Boolean set_wallet = _Orblite_FALSE);

  Orblite::Identifier &change_method();
  void change_method(const AuthInfo &au_info,
                     const Orblite::Identifier,
                     Orblite::Boolean set_wallet = _Orblite_FALSE);
};
```

PaySlip is an _orblite_slip, and it is not intended to be subclassed. It provides the implementation for the interfaces of the base classes. The implementation includes PaySlip::issue and PaySlip::verify. PaySlip::issue invokes Wallet::issue_payment to get a request_slip, while PaySlip::verify calls Wallet::verify to verify the contents of a received ack_slip from a server. In addition to functions issue and verify, PaySlip also needs to define functions required to add a payslip to the ORBlite attribute list.[2]

Besides overriding the interfaces of the base classes, PaySlip offers an ORBlite user a set of functions to configure a payment transaction. In the current implementation, PaySlip supports the methods that specify the amount of purchase, the payment method, and the change method. Additional functions such as specifying a replenishment method or enabling auto-replenishment can also be implemented. Currently, Wallet does not support changing default values. Therefore, to make a purchase in the current implementation, an ORBlite user has to specify the payment method, change method, and the amount purchased in a pay_slip.

In addition to configuration, PaySlip provides query functions to the status of payment transaction. The status is set through the Slip's interfaces once the object invocation is completed. PaySlip::xaction_is_completed lets the user know if the payment is processed successfully, and PaySlip::price is used for getting the price quote.

PaySlip::refresh allows a pay_slip to be re-used. It clears all configuration parameters and also the result of the previous payment.

[2]The deletion of an instance of PaySlip is done by other ORBlite functions, and the action is not authenticated.

9.3 RedeemSlip

RedeemSlip is the other derived class from _Orblite_Slip. It allows an ORBlite user to configure a redemption transaction. The structure and interface of RedeemSlip is very similar to that of PaySlip.

Chapter 10

Request_Slip & Ack_Slip

10.1 Request_Slip

10.1.1 Design Issues

Request_Slip contains the information that will be sent between the client and the server of a communication protocol. This information includes a field containing either a payment or redemption and another field containing an ackinfo. The reason of sending a request_slip rather than sending the paymethod data individually is that the marshaling and demarshalling of the data is done by a communication protocol, which should not be exposed to the representation of the PayMethod data. Request_Slip is therefore used to convert derived PayMethod data into a transmittable data at the client side and buffer the transmittable data and convert it back to its original form at the server side.

Since the reason of having Request_Slip is solely for marshaling and demarshalling data, the data representation of Request_Slip is a IDL data type, and its implementation is generated by an IDL compiler.

10.1.2 Implementation Details

The following shows part of the class declaration of PaySlip:

```
class Request_Slip
{
public:
  Request_Slip();
  Request_Slip(const Request_Slip &);
  Request_Slip(PayMethod::Payment *,
               PayMethod::AckInfo *);
  Request_Slip(PayMethod::Redemption *,
               PayMethod::AckInfo *);

  Orblite::Boolean is_empty() const;
  Orblite::Boolean is_payment() const;
  PayMethod::Payment *payment() const;
  PayMethod::Redemption *redemption() const;

  PayMethod::AckInfo *ackinfo() const;

  Orblite::Boolean marshal(_Orblite_Transport_OutStream &os);
  Orblite::Boolean demarshal(_Orblite_Transport_InStream &is);
};
```

The data representation requires a set of functions to perform the conversion of data formats. The constructor of Request_Slip takes in PayMethod data as parameters and converts them into the internal data representation of Request_Slip, while Request_Slip::payment(), Request_Slip::redemption(), and Request_Slip::ackinfo() converts the internal data back to PayMethod data. All these functions are called by Wallet which should only know about PayMethod (the abstract base class of all payment methods). Therefore, the interpreted payment, redemption, or ack_info are returned as pointers to their abstract base classes (a nested class in PayMethod). Request_Slip::is_payment() allows Wallet to check if the content in the request_slip is a payment or a redemption.

To convert from PayMethod data into a sequence of characters, Request_Slip creates a character stream and marshals a derived PayMethod data onto the stream. As mentioned in section 6.2.4, each derived PayMethod data has to define both marshal and demarshal functions. Request_Slip puts the contents of the character stream into the corresponding data fields of the internal data representation. The paymethod data are stored as sequences of characters which do not convey any information about

the specific payment method of the marshaled paymethod data. Thus, Request_Slip has to additionally marshal the name of the payment method into a separate field in Request_Slip. This information is obtained from tag(), which is defined in the derived payment method data. The existence of a payment and redemption in a request_slip should be mutually exclusive; thus, they share the same data field in Request_Slip. To allow the server to determine if the data contains a payment or a redemption, Request_Slip marshals the type of the data (namely, "Payment" and "Redemption") into the internal data representation. The type is obtained from type(), which is defined in all classes of PayMethod data.

At the server side, a request_slip is demarshaled. To convert the internal data representation back to its original data form, Request_Slip gets the name and the type of the derived PayMethod data from the demarshaled data. It uses them as parameters to call PayMethod::Handle, which creates an instance of the original derived PayMethod data. As mentioned in section 6.2.1, PayMethod::Handle provides an abstraction layer which delegates the creation of the paymethod data to the appropriate derived PayMethod handle. Once the instance of the paymethod data has been created, the sequence of characters will be converted back into the instance of the derived paymethod data. The conversion is done in the same way (but in reversed manner) as converting derived paymethod data into sequences of characters.

In addition to the data converting functions, Request_Slip defines marshal and demarshal functions which allows the communication protocol to marshal and demarshal the internal data structure to and from a communication protocol.

10.2 Ack_Slip

Ack_Slip contains the reply for a request_slip. It uses the same data conversion techniques as Request_Slip, and its internal data representation is very similar to that of Request_Slip. Ack_Slip contains a field which mutually excludes a bounced payment or redemption, and it contains another field for the receipt. In the current implementation, a valid ack_slip only has one of the two fields specified at a time.

However, PayMethod receipt and bounced PayMethod data are intentionally designed not to share the same field in Ack_Slip, since a receipt may be extended to include more information in the future.[1]

In addition to the derived PayMethod data, the internal data representation of Ack_Slip includes data members providing general information about the status of the transaction. Currently, these data fields include the price quoted for invoking the object and also a boolean flag for reporting the status of the transaction. The price is considered as a mandatory information for the customer, and this restriction is imposed by making the price a required parameter for the constructors of Ack_Slip.

[1]For example, the bounced payment may be caused by reasons specific to the change method, and it should be stated in a receipt.

Chapter 11

The Communication Protocol: A Modified IIOP

In the current design, the payment framework can only use one RPC communication protocol. This RPC protocol is a modified IIOP. The restriction of using only one communication protocol is not an ultimate design solution for Clearinghouse, and modifying the actual implementation of a communication protocol seems to have violated the design goal of the ORBlite architecture: building evolvable systems [6]. However, it is much simpler to add the commerce capability directly to an existing communication protocol than designing another abstraction layer. Enhancing an existing communication protocol provides a quick way to bring the payment framework into operation within a restricted time frame, which is a prime issue for the first stage completion of the project. Moreover, using a modified RPC protocol does not change the external specification of ORBlite, i.e. any ORBlite-compliant applications can use the commerce-enabled communication protocol in the same way as one without the commerce capability.

In Clearinghouse, the commerce-enabled, modified, IIOP will be called PP (which stands for Payment communication Protocol). PP has added an implementation to the IIOP client. The added code locates an orblite_slip from the list of attributes in an object invocation, calls Orblite_Slip::issues to issue a request_slip, and invokes Orblite_Slip::verify to a received ack_slip from the server. PP_Server has added code to

an IIOP server. It gets a pricemodule which locates the price for an object invocation and the wallet of the object provider. The price module triggers the wallet to verify the received request_slip and return the result (an ack_slip) back to the client. PP does not need to know other details of the payment framework besides the interfaces to _Orblite_Slip and PriceModule, and the existence of Request_Slip and Ack_Slip.

When the PP Client[1] receives a request for making a remote object invocation, it looks up an _orblite_slip from the list of attributes in the ORBlite invocation. The actual lookup is done by a static function (_Orblite_Slip::lookup). If an _orblite_slip is included, PP Client invokes _Orblite_Slip::issue() to get a request_slip. If the _orblite_slip fails to issue a request_slip, PP Client aborts and sends an exception to the user. This is because the payment framework cannot satisfy the ORBlite user's request, and there is no need to send out a request to a server.

If an _orblite_slip is not found in the callinfo, it indicates that an ORBlite user does not wish to send a payment or redemption; in this case, an empty _orblite_slip will be created to fulfill the communication protocol.

To send the request_slip to the server, PP added a field for the request_slip to the original IIOP message header. The advantage of including request_slip in the message header is that the operations for marshaling and demarshalling the request_slip are triggered by the original IIOP implementation.

When the PP Server receives a request_slip, it gets the price module. PriceModule has the role that is symmetrical to _Orblite_Slip in PP Client. PP Server calls PriceModule::verify with the request_slip as a parameter (the interface to PriceModule will be discussed in section 12). The result of the invocation will be an ack_slip. If the verification of the request_slip does not succeed (indicated by the result returned from Ack_Slip::valid_xaction()), the PP Server will not make an object invocation; instead, it will send an exception with the ack_slip to the client.

If the verification succeeds, the object is invoked. If the object is not available, PP Server calls the PriceModule::revert to revert any actions previously done by

[1]PP Client is not the name of a C++ class. This name refers to a *group* of C++ classes which provide the functionalities for a RPC Client. PP Server conveys the similar meaning.

PriceModule::verify. For example, if a payment has been verified, the action will be rolled back as if the payment was never made, and a new ack_slip which includes a bounced payment will be issued by the price module. If the object invocation succeeds, the ack_slip will be sent back to PP::Client through a modified IIOP ReplyHeader.

Once the PP::Client receives the ack_slip, the PP::Client delegates the verification process to the _orblite_slip. A boolean function is returned to indicate the status of the verification.

Chapter 12

PriceModule

12.0.1 Design Issues

PriceModule provides the pricing information for ORBlite-compliant objects. There are two ways of providing the pricing information. The first way is to include the price within an object; the second way is to provide a separate repository for this information.

The first approach would require the change in the object modeling and would very likely to require a change to the current architecture of ORBlite. The second approach provides the pricing as a separate piece from the ORBlite framework and is thus preferred. Moreover, it provides the possibility of having dynamic pricing for an object. For example, an object can be priced differently based on the identity of the caller, allowing for subscriptions, contracts, etc.

12.0.2 Implementation Details

The following shows part of the class declaration of PriceModule:

```
class PriceModule {
public:
  Ack_Slip *verify(const Orblite::Identifier &, const Request_Slip &,
   const Orblite::Identifier &, const Orblite::Identifier &,
   const Orblite::Identifier &, const Orblite::ArgList*);
```

```
  Ack_Slip *revert(const Orblite::Identifier &, const Request_Slip &,
   const Ack_Slip &,
   const Orblite::Identifier &, const Orblite::Identifier &,
   const Orblite::Identifier &, const Orblite::ArgList*);

  static PriceModule &get_price_module();
};
```

PriceModule::verify is called by PP Server. The function takes in as parameters the request slip, the object key, operation, and an optional argument list of the ORBlite invocation. The object key and the operation id are keys to locate the price of an object invocation and the wallet of the object provider. The price module will in turn invoke the corresponding wallet with the price and the request_slip. The optional arglist can help determine if an object should be charged based on additional information from the object invocation. For example, a customer who provides a valid VIP number will not be charged for the invocation.

PriceModule::revert works similarly to verify. The only difference is that PriceModule::revert will call the wallet of the object provider to unroll the action done for the request_slip. PriceModule::verify is always called before the invocation of an object, while PriceModule::revert is called only if the invocation fails.

Chapter 13

Pay_Util

Pay_Util is a group of classes serving as the common utilities for other data types in Clearinghouse. There are three categories of classes: Date, AuthInfo and Certificate, and PGPGlue and MDGlue.[1] Date is used for issuing timestamps in Clearinghouse. AuthInfo and Certificate authenticate ORBlite users to Clearinghouse. PGPGlue provides a public key cryptography algorithm, and MDGlue provides a one-way, collision-resistant hashing algorithm.

13.1 Date

Date is responsible for creating a timestamp and checks if that timestamp has been expired. It uses the system time handling functions. The internal representation of this timestamp is in Coordinated Universal Time (UTC) standard, as a local time is inadequate for global communications.

13.2 AuthInfo and Certificate

AuthInfo and Certificate are units of data used for authentication and authorization within Clearinghouse. Most functions require a valid authinfo to authenticate the

[1] PGPGlue and MD5Glue should have been renamed to reflect that the two classes serve more general purposes

invocation. Instances of Authinfo are stored in many objects, e.g. wallet and derived payment methods. Currently, an authinfo only contains a certificate.

Certificate contains the information for obtaining user's signatures from PGPGlue. Currently, the data member of a certificate is PGP-oriented. It contains the user's keyID, the certifier's (broker's) keyID, and the user's passphrase. The user's keyID and the passphrase are used for creating signature certificates for the user. The certifier's keyID is used by a vendor to assemble a redemption. The certifier's keyID is used as a parameter to locate all the payments certified by the broker. Currently, the vendor does not verify this broker's keyID with the one in the user's public key certificate.

Representation of User Identities

There have been problems in finding a proper representation for the user identities in Clearinghouse. It seemed that a virtual process id (*vp_id*) would be a good choice for user identification. However, it was found that *vp_id* could refer to different parties at different time. Object keys provide both uniqueness and permanency, but an object key corresponds to only one object, while each entity is usually responsible for groups of objects. Thus, using object keys does not fit the ownership model.

The current solution is to assign a unique identity to each party. However, there is a question of how the identity of an object's owner can be found by a caller. In the current design, the caller has to provide the identity of the object's owner in a payslip or a redeemslip. An alternative solution is to publish this identity with the ORBlite's object reference. However, this may require an extension to the ORBlite architecture.

13.3 PGPGlue and MD5Glue

PGPGlue provides an abstraction layer between Clearinghouse and the actual digital signature and encryption mechanisms. PGPGlue uses PGP 2.6.2 [18] to generate and verify signatures and encryption of data. Currently, PGPGlue interacts with PGP

through a system call.

PGP was chosen since it offers public key cryptography, key management, and a user interface. It allows users to generate public and private keys in advance through its own user interface, and it also allows Clearinghouse to access the key rings. Using PGP through a system call is a quick way to bring the framework into operation. However, the current interface (a system call) should be re-engineered in the future, or even replaced with a different public key generation package which provides a better interface.

MD5Glue provides an abstraction layer between Clearinghouse and a one-way, collision-resistant hashing algorithm. Currently, as the name suggested, MD5Glue uses MD5 [19], a message digest algorithm to provide the hashing functionalities. The MD5 binary comes from rsaref library in the PGP 2.6.2 source tree [18].

Chapter 14

Putting It All Together

Here is a revisit of the simple example in section 1.3.

> NetStock is a company that provides on-line stock quotes through the Internet.
>
> NetStock has a set of electronic agents which sends out instant stock quotes. To make these electronic agents available to the public, a NetStock software developer ''publishes'' them using the ORBlite framework.

The electronic agents are basically objects obtaining the most updated stock quotes. The "publishing" step is required for any object made accessible through any object bus.

> Since the company is ''selling'' on-line accesses to stock quotes, the developer also needs to register each agent, its access price, and the NetStock's account information with a special repository.

The "special repository" is the price module for an ORBlite process. "The NetStock account" is actually a wallet. To create the wallet, NetStock needs a public and private key pair. In the current payment framework, this is done by PGP. The public key has to be certified by an authority which the customer trusts.

A customer wants to get a stock quote.

Just as what NetStock did, the customer needs a public key and a private key, and his public key has to be certified by an authority which NetStock trusts. The customer also needs a wallet.

At a front panel (e.g a web browser), he issues a ''get-stock-quote'' command and an authorization to pay an ''electronic token.''

The front panel is a CORBA-compliant application that bridges gap between the customer and ORBlite. The panel transforms the authorization from the customer into a payslip. Besides the authorization information, the payslip also contains the recipient, the amount of the payment, and the pay method. Suppose the NetStock uses paywords as electronic tokens, the pay method will be PayWord [20]. The payslip is a specification for generating a payment. The panel application puts the payslip into the callinfo, which is treated as an attribute of a remote object invocation using ORBlite.

The front panel sends the ''get-stock-quote'' command to a NetStock server through the ORBlite framework.

The panel application invokes the "get-stock-quote" operation of an electronic agent. It is done in the same manner as any remote object invocation that does not have the payment option.

The ORBlite framework handles the payment and forwards the command to a stock-quote electronic agent at the NetStock server.

The "get-stock-quote" remote invocation calls the ORBlite infrastructure, and eventually the payment communication protocol is invoked to handle the preparation and the transmission of the payment and the "get-stock-quote" command. The payment protocol finds the slip in the attribute of the remote invocation and invokes the Slip::issue operation to get a request_slip. Since the slip is actually a payslip,

Slip::issue calls Wallet::issue_payment. Wallet::issue_payment locates the customer's wallet (cwallet) and tries to get its instance of the PayMethod::Handle (handle) if the pay method specified in the payslip is supported by the payment framework. If handle is available, it will dispatch the withdraw operation to get the payment.

Wallet::issue_payment also needs to get ack_info for the specified change method, and it is obtained in the same way as the payment. Wallet::issue_payment then packs the payment and ack_info into a request_slip which will be forwarded to the payment protocol. The payment protocol sends out the request_slip and the "get-stock-quote" command through a reliable communication protocol to the ORBlite process residing at the NetStock server.

The payment protocol at the NetStock server receives the message and converts the bit-stream into the request_slip and the "get-stock-quote" command. Then the payment protocol gets the price module. The price module mirrors the role of the payslip at the customer's site, and it contains the price for the "get-stock-quote" command and the NetStock's wallet (vwallet). The price module calls the verify operation of vwallet to verify the contents in the request_slip. The vwallet.verify operation (like Wallet::issue_payment) looks up the handles for the pay method and the change method specified in the request_slip. The payment handle, if supported by the NetStock's payment framework, dispatches the verify operation for the payment, and the change handle works in the same way. If the payment is valid, vwallet creates an ack_slip which may contain a receipt for the change of the transaction.

```
Once the payment is cleared, the customer will get the
requested stock quote.
```

Once the transaction is cleared, the "get-stock-quote" command is sent to the electronic agent. If the invocation succeeds, the ack_slip and the permission to get the stock quote are both sent back to the payment protocol at the customer's site.

Similar to the mechanism of issuing the request_slip, the payment protocol calls the original pay_slip with the ack_slip. The pay_slip in turn calls Wallet::verify with the ack_slip and the pay_slip. In Wallet::verify, the pay_slip locates cwallet which

gets the handle to verify the receipt, if any. The result is forwarded to the payment protocol.

The payment protocol puts the price quoted by the server into the payslip. If the transaction succeeds, the payment protocol also sets a flag in the payslip. All this information will be propagated to the front panel through the ORBlite infrastructure. The front panel gets the result of the invocation and can query the state of the payment transaction and the price quote from the attribute containing the payslip.

```
The customer may continue polling the stock quotes of the same
company.
```

If the customer is using PayWord (a micropayment scheme), he does not need to create another commitment for subsequent payments (unless the commitment has expired, or the authenticated chain of hashes has been completely consumed). He simply sends a hash value (a payword) to the NetStock server. The more the customer uses the commitment, the less the average computation- and network-overhead cost for each payment becomes. Other micropayment protocols, such as [4, 20], have used the same technique to reduce the average overhead costs for each payment.

Chapter 15

Threats

The following is a list of possible threats which may cause harm to users of Clearinghouse. This list is composed with the assumption that the trust model proposed in section 5.3 is intact.

- The failure in the network communication link, or any possible attacks at the network communication (e.g. interception, tampering, or sniffing) will cause modification or loss of Clearinghouse data.
- Tampering the public and private key rings of PGP 2.6.2 (pubring.pgp and secring.pgp) will invalidate the public and private keys used by Clearinghouse.
- Bugs or execution failures of Clearinghouse and ORBlite may cause inadvertent modification or loss of data.
- Creating derived classes of Clearinghouse data types to create a side-channel to modify the data.

Chapter 16

Complete Work and Test Results

In this thesis project, a payment framework has been designed and a preliminary implementation (which enables a payment capability) has been completed. The following provided a more detailed account on what has or has not been completed.

- PayMethod: A paymethod abstraction layer has been designed and implemented to allow multiple payment methods to be concurrently supported within Clearinghouse.

- PayWord: A micropayment protocol has been implemented as a derived class of PayMethod to demonstrate the use of the paymethod abstraction layer. The ability to issue payments and changes of PayWord have been tested.

- User Wallet: Wallet has been implemented to manage instances of payment methods belonging to a user. It dispatches the issuing and verification of payments and their corresponding receipts to specified payment methods.[1] Wallet, together with EWallet and CWallet, have provided the functionalities for searching and creating a user wallet.

- _Orblite_Slip & PaySlip: _Orblite_Slip has been implemented as an attribute transparently included in an ORBlite invocation. It has provided the ability to

[1] only PayWord is supported.

connect PP and Wallet. PaySlip has provided the ability to issue a payment and the functions to query the price and the status of a transaction.

- Request_Slip & Ack_Slip: They are both implemented to transport a payment (or redemption) and a receipt through a communication protocol.

- PP: PP has added implementation to IIOP so that it can interact with Clearinghouse. It has provided the end-to-end connection for Clearinghouse.

- PriceModule: A simple, container class has been implemented to provide pricing information for ORBlite-compliant objects. It has also provided functions to forward PP_Server's requests to Wallet.

- PayUtil: A set of simple data structures have been implemented to provide common utilities to other data structures in Clearinghouse.

The following is the set of test cases successfully verified the payment framework. PayWord is used as both the payment method and the change method, as it is the only payment method having been implemented for Clearinghouse at the moment.

- No payslip: In this case, the user does not wish to pay at all. An empty request_slip will be created internally in Clearinghouse. The object (requires payment) is not invoked and an exception is returned to the client indicating that the specification for payment is not valid.

- A payslip with insufficient funding: Money is bounced. An exception is returned. Since a payslip is included, the price quote can be queried from the payslip after the attempt of making the remote invocation.

- Exact payment : The payment is verified by the server, and the object is invoked. A "transaction complete" status and the price is returned to the client.

- Excessive payment : Payment is verified. Change (modeled as a payment in PayWord) is generated, and object is invoked. The client gets the same result as in the case of exact payment, plus a change which is successfully verified.

- Subsequent payments : Subsequent payments between the same entities do not generate further commitments. Only payword hashes in the existing chains are retrieved and sent to the other end. The object will be invoked. A "transaction complete" status and a price quote will be returned to the customer.

Chapter 17

Future Work

This thesis project only completes the first stage of the payment framework. The following is a list of future work for Clearinghouse.

- *Removing the dependency on PP:* Currently, PP is the only communication protocol that is capable of handling the commerce-capability for an object invocation. It is necessary to remove this constraint in the future to turn Clearinghouse into an entirely independent and evolvable piece in ORBlite.

- *Removing restriction on verification model:* Currently, PP indirectly calls PayMethod::verify to verify a payment (or a redemption) before an object is invoked, and it calls PayMethod::revert only if the ORBlite invocation does not succeed. PP has actually defined the specification of PayMethod::verify and PayMethod::revert. It has imposed a "charged-and-undo-later" policy to all payment methods being supported in the framework. This policy can be inadequate for some payment methods which require a pre-validation process, e.g. an on-line credit-card payment process. The restriction will no longer exist if PayMethod::revert will always be called after an object invocation. This is because each payment method will always get the control of processing the paymethod data before and after the object is invoked, so the payment method can decide when the object invocation is actually charged.

- *More payment methods:* Only PayWord has been implemented. Other payment protocols (e.g. credit card and token-based) should be implemented to examine other dimensions of the paymethod abstraction layer.

- *Better exception handling:* Currently ORBlite user exception BAD_PARAM is used to signal customers of any errors occurred in a payment or a redemption. This is because the current ORBlite architecture does not allow more user exceptions to be added to the core. Another way of handling exception of a slip for an object invocation should be considered.

- *Revocation of wallets:* The revocation of a wallet does not invalidate its reference in the price module. It is not a difficult task to be done in the first stage; however, the current price module requires a better interface and incorporates more advanced searching algorithms. The invalidation may as well be done together with the redesign. Also, it is found that merging Wallets and PriceModule into one single class may ease the job of revocation, since then it can be done through one interface.

- *Better interaction with PGP:* System calls are used to interact with PGP. This poses a bottleneck in the performance in Clearinghouse. The interaction needs to be changed to a normal function call to PGP code; however, this will require re-engineering in the existing PGP implementation. A replacement of PGP with another mechanism may be a better solution.

- *Balances:* Balances and transaction logging should be considered to provide more accounting capability to Clearinghouse users.

- *Auto-replenishment and -redemption:* Currently, auto-replenishment and -redemption have not been implemented and their designs have not been thoroughly examined. This feature can be added by letting wallets upcall for another _orblite_slip. The actual internal transfer can be done either by local bypass mechanism in ORBlite or a direct transfer within a wallet. However, local bypass mechanism has not been explored in Clearinghouse either.

- *Sharing:* Sharing payslips, redeemslips, wallets, or paymethods within wallets can be made possible in the framework. This requires a proper design for establishing, determining, and revoking ownership. Currently an instance of AuthInfo class is modeled as a unique information for a user. The same model can be used to do the sharing trick, but a modification of AuthInfo may simplify the design.

- *Data recovery measures:* There are no data recovery measures used at the moment. If Clearinghouse crashes, users will lose all the money residing in the wallets.

- *Vendor authentication:* Currently, vendors are not required to authenticate themselves to customers in each transaction. In other words, customers sometimes cannot ensure if they are communicating with the right entity.

- *Concurrency Control Verification:* Multi-threaded environment has not been tested in Clearinghouse.

- *Transporting public keys:* The recipient of an encrypted data, such as a PayWord's commitment, requires a public key of the sender to verify the authenticity of the data. Currently, there are no mechanisms to transfer public keys. An out-of-band mechanism is required to obtain and transfer public key certificates.

Chapter 18

Conclusion

This thesis has documented the design and the details of the preliminary implementation of Clearinghouse, a "per-call" payment framework for ORBlite. The development of a payment framework in ORBlite was motivated by the belief that some objects in an open object environment would be presented as services and would require a payment framework to provide electronic commerce capability to those objects. Currently, most payment frameworks provide an aggregated model which are inadequate for individual objects being charged in small amount of money. A "per-call" payment model is therefore chosen for Clearinghouse to facilitate an environment which is more suitable for micropayment.

As a service built for an open, heterogeneous environment, Clearinghouse has to provide evolvability for individual payment methods. Therefore, a paymethod abstraction layer for Clearinghouse has been designed and implemented to allow each payment method to evolve independently. In addition, this paymethod abstraction layer has allowed Clearinghouse to simultaneously support multiple payment methods. Clearinghouse wallets have taken the advantage of the paymethod abstraction layer and provide users an option to select and configure individual payment methods in their wallets, yet without having the wallets to expose to the data structures of the payment methods.

In the first-stage development of Clearinghouse, PayMethod has been implemented to serve the functions of the paymethod abstraction layer. PayWord, together with

other data types in Clearinghouse, have been implemented to demonstrate the use of the paymethod abstraction layer to handle payments and changes. A set of test cases have been performed to verify Clearinghouse's capability in handling payments and changes, and the test cases and results have been listed in this thesis.

Appendix A

Example Code for Clearinghouse Users

This section includes a sample client and server code to demonstrate the use of Clearinghouse for the example shown in Section 1.3 and Chapter 14. In that example, a customer is polling stock quotes, and the server (NetStock) publishes electronic agents responsible for giving out stock quotes. Assume that each of these electronic agents is responsible for providing stock quotes for only one company. The following server and client code show how a customer can poll HP's stock quotes from an HP-Stock agent which gives out HP's stock quotes.

A.1 Server Code

A.1.1 Creating an IDL interface

The following is an IDL description specifying the interface of accessing the HP-Stock agent.

```
interface HPAgent {
  exception Ex
    {   // For returning to caller any exceptions raised for
        // invoking an hp-stock agent.
string reason;
```

```
    };
  long get_stock_quote() raises(Ex);
};
```

A.1.2 Generating the Stub and the Skeleton

A NetStock software developer then uses an IDL compiler to generate a stub for customers and a skeleton for the implementation of the HP-Stock agent. Then the software developer implements the agent based on the skeleton. The following shows part of a stub, a skeleton, and the definition of HP-Stock agent (HPAgentImpl), they are all presented in C++.

Stub

```
class Agent: public virtual ORBlite::Object
{
public:
virtual ORBlite::Long
   get_stock_quote(ORBlite::Environment & _ev) const;
   ...
};
```

Skeleton

```
class _BOA_HPAgent: public virtual ORBlite::ImplBase
{
public:
  virtual ORBlite::Long get_stock_quote(ORBlite::Environment & _ev) = 0;
  ...
};
```

A.1.3 Object Definition of the Stock Agent

The following shows the definition of HPAgentImpl, an object to be implemented by NetStock.

```
class HPAgentImpl : public virtual _BOA_HPAgent
{
 public:
  HPAgentImpl();
  ~HPAgentImpl();
  ORBlite::Long get_stock_quote(/* inout */ ORBlite::Environment & _ev);

  // .. and some private memebers.
};
```

A.1.4 A Simple NetStock Server

The following is an implementation for a simple NetStock server providing HP's stock quotes.

```
#include <iostream.h>
#include <stdlib.h>
#include <signal.h>
#include <example_impl.h>
#include <hpl_orb/orb.h>
#include <hpl_orb/environment.h>
#include <hpl_orb/object.h>
#include <soa/soa.h>

// add for the payment option
#include <pay_util/certificate.h>
#include <pay_util/auth_info.h>

// cwallet is the only user wallet being
// supported currently. bwallet.h should be
// been used instead.
#include <wallet/cwallet.h>
#include <price_module/price_module.h>

static
void
sighandler(int)
{
  SOA::shutdown();
  exit(1);
}
```

```
int
main(int argc,
     char *argv[])
{
  // Initialize the ORBlite process
  signal(SIGTERM, sighandler);
  signal(SIGINT,sighandler);
  SOA::init(argc, argv);

  // Create an HP-Stock agent (which is basically an object) and
  // publishes its reference to the ORBlite framework.
  ORBlite::Environment ev;
  HPAgentImpl impl;
  cout << endl << "Publishing NetStock's HP-Stock Agent." << endl;
  ORBlite::Object obj = impl._self();

  assert( SOA::publish_reference("HWP", obj) == _Orblite_TRUE);

  // create a NetStock Certificate
  // This is a relatively one-time event.
  cout << "Creating Certificate for NetStock" << endl;
  const Certificate netstock_cert("Broker2",
  "NetStock", "NetStockPassPhrase");
  // Create an authorzation information for NetStock.
  // Basically put the NetStock's certificate in a new authinfo.
  cout << "create and put NetStock's certificate in AuthInfo" << endl;
  AuthInfo ns_ainfo(netstock_cert);

  // create a wallet
  cout << "Create NetStock's wallet" << endl;
  CWallet *ns_wallet = CWallet::lookup_or_add(ns_ainfo);
  if (!ns_wallet) {
    cout << "Sorry, you need a valid NetStock's wallet." << endl;
    return 0;
  }

  // Register the HP-Stock agent to the price module
  cout << "Register the HP-Stock agent to the price module ." << endl;
  PriceModule::Entry *hwp = new PriceModule::Entry(*ns_wallet);
  hwp->obj = impl._object_key();
  hwp->op = Orblite::Identifier("get_stock_quote");
  hwp->price = 1;  // one cent for getting HP price quote. The
   // price can be determined dynamically.

  PriceModule &pm = PriceModule::get_price_module();
```

```
  pm.olst.add_entry(hwp);

  // Now NetStock is ready to hand out the HP-Stock agent.
  cout << "Server is running..." << endl <<endl;
  SOA::run();
  return 0;
}
```

A.2 Client Code

The following is an implementation of polling HP's stock quotes at the customer's site.

```
#include <unistd.h>
#include <iostream.h>

#include <assert.h>
#include <example_types.h>
#include <hpl_orb/hpl_fwd.h>
#include <hpl_orb/orb.h>

#include <hpl_orb/object.h>
#include <hpl_orb/environment.h>

// add for the payment option
#include <pay_util/certificate.h>
#include <pay_util/auth_info.h>
#include <wallet/cwallet.h>
#include <orblite/callinfo.h>
#include <slip/pay_slip.h>
#include <payword/payword_info.h>

extern char *optarg;
extern int optopt;

main(int argc,
     char *argv[])
{
  // Initializing the Client...
  SOA::init(argc, argv);
```

```
// get object reference for HP-Stock HPAgent.
// ''HWP'' is an official short form for Hewlett-Packard
// in New York Stock Exchange.
Orblite::Object obj = SOA::initial_reference("HWP");
ORBlite::Environment ev;
HPAgent ns_hpagent = HPAgent::_narrow(obj, ev);
assert(ev.check_exception() == _Orblite_FALSE);

// Create authinfo and certificate
cout << endl << "Create Certificate for customer" << endl;
const Certificate ccert("Broker", "Customer", "customer");
cout << "Create and put certificate in AuthInfo" << endl;
AuthInfo ainfo(ccert);

// Create a wallet.
cout << "Create a wallet." << endl;
CWallet *cw = CWallet::lookup_or_add(ainfo);
assert(cw != NULL);

int bid = 1; // 1 cent
ORBlite::Boolean continue_to_poll = _Orblite_TRUE;

while (continue_to_poll == _Orblite_TRUE) {

// Create/lookup and refresh a payslip
cout << "Create/lookup and refresh a payslip" << endl;
PaySlip *pslip = PaySlip::lookup_or_add(ainfo, ev);
assert(pslip != NULL);
pslip->refresh(ainfo);

// Configure the slip: setting payment method and change method.
// PayWord is used in this demo for both  payment and change.
cout << "Configure the slip (pay_method and ch_method) " << endl;
pslip->pay_method(ainfo, PayWord_Info::tag());
assert(pslip->pay_method() == PayWord_Info::tag());
pslip->must_use_pay_method(ainfo, _Orblite_TRUE);
assert(pslip->must_use_pay_method() == _Orblite_TRUE);

pslip->change_method(ainfo, PayWord_Info::tag());
assert(pslip->change_method() == PayWord_Info::tag());

// Obtain NetStock's certificate: currently transporting public
// key is not supported. In the future, it should be handled
// by an off-the-band mechanism.
const Certificate ns_cert("Broker2", "NetStock",
```

```
                              "NetStockPassPhrase");
 pslip->buy(ainfo, ns_cert, bid);
 assert(pslip->buy_from(ainfo) != NULL
&& *pslip->buy_from(ainfo) == ns_cert);

 // Making a normal ORBlite invocation: calling HP-stock HPAgent
 cout << "Polling HWP price quote" << endl << endl;
 ORBlite::Long price_quote = ns_hpagent.get_stock_quote(ev);

 // Query the status of the return call. This includes any
 // exceptions raised for the payment process.
 if (ev.check_exception()) {
   const ORBlite::Exception *exc = ev.exception_value();
   const HPAgent::Ex *ex = HPAgent::Ex::_cast(exc);
   if (ex != NULL) {
     cout << "Got an Ex exception" << endl;
     cout << "  Reason: '" << ex->reason() << "'" << endl;
   } else {
     cout << "Got some other exception" << endl;
   }
 } else {
   cout << "ORBlite invocation is fine" << endl;
 }

 // Get the price quote.
 cout << endl << "HWP current stock price : " << price_quote;

 // Querying the status of the payment.
 cout << endl << "Payment Status:" << endl;
 PaySlip *result_pslip = PaySlip::lookup_or_add(ainfo, ev);

 cout << "Query the status of the transaction (payment) " << endl;
 cout << "Price quoted from server: " <<
         result_pslip->price() << endl;
 if (result_pslip->xaction_is_completed() == _Orblite_TRUE)
   cout << "Payment completed sucessfully." << endl;
 else {
   cout << "Sorry. Transaction Failed!!!!" << endl;
   cout << "You paid "<< bid << " cents only." << endl;
 }

 // Continue to poll?
 cout << endl << "enter non-q to continue > " << endl << endl;
 char c;
 cin >> c;
```

```
  if (c == 'q')
    continue_to_poll = _Orblite_FALSE;
  }

  return 0;
}
```

Bibliography

[1] E. Chi. Evaluation of Micropayment Schemes. *HP Laboratories Techncial Report,* HPL-97-14, January, 1997.

[2] DigiCash Inc. http://www.digicash.com/.

[3] S. Garfinkel. *PGP: Pretty Good Privacy* O'Reilly & Associates, Inc. 1995.

[4] S. Glassman, M. Manasse, et. al. *The Millicent Protocol for Inexpensive Electronic Commerce.* http://www.research.digital.com/SRC/millicent/papers/millicent-w3c4/ millicent.html.

[5] IBM Cryptolope. http://www.internet.ibm.com/ secureway/ cryptolopes.html.

[6] K. Moore and E. Kirshenbaum. Building Evolvable Systems: The ORBlite Project. *Hewlett-Packard Journal* Feb 1997: 63-73.

[7] K. Moore. "CORBA/OLE Tutorial". Presentation for *IEEE HotInterconnect-III* by Hewlett-Packard Laboratories, Palo Alto, CA.

[8] Netscape Communications Corporation. http://home.netscape.com/.

[9] K. Moore, E. Kirshenbaum, Walter Underwood. ORBlite RPC Transport Abstraction Layer. *HP Laboratories Techncial Report,* Internal Draft. July, 1995.

[10] Netscape Communications Corporation. *Netscape LivePayment Data Sheet.* http://home.netscape.com/comprod/products/iapps/platform/livepay_data_sheet.html.

[11] Netscape Communications Corporation. *Guidelines for Developing Multi-Merchant.* http://home.netscape.com/eng/LivePayment/index.html#merchant.

[12] Netscape Communications Corporation. *Netscape LivePayment White Paper.* http://home.netscape.com/comprod/products/iapps/platform/livepay_white_paper.html.

[13] Object Management Group Background Information. http://www.omg.org/bacgrnd.htm

[14] Open Market. http://www.openmarket.com.

[15] Open Market. *Merchant Solution^{TM} Technical White Paper.* 1995.

[16] Open Market. *OM-Transact^{TM} Technical White Paper.* 1996.

[17] R. Orfail, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide* Wiley 1996.

[18] PGP 2.6.2. ftp::/net-dist.mit.edu.

[19] R. Rivest. *The MD5 Message Digest Algorithm,* Internet RFC 1321. http:// theory.lcs.mit.edu/ rivest/Rivest-MD5.txt.

[20] R. Rivest and A. Shamir. *PayWord and MicroMint: Two simple micropayment schemes.* May 7, 1996. http://theory.lcs.mit.edu/ rivest/RivestShamir-mpay.ps

[21] L. Stein, E. Stefferud, N. Borenstein, M. Rose. *The Green Commerce Model.* First Virtual Holdings Incorporated. http://ftp.fv.com/pubdocs/green-model.txt.