SWAPEROO: A Simple Wallet Architecture for Payments, Exchanges, Refunds, and Other Operations

Neil Daswani, Dan Boneh, Hector Garcia-Molina, Steven Ketchpel, Andreas Paepcke
Stanford University
Computer Science Department
Stanford, CA 94305
{daswani, dabo, hector, ketchpel, paepcke}@cs.stanford.edu

Abstract

Most existing digital wallet implementations support a single or a limited set of proprietary financial instruments and protocols for electronic commerce transactions, preventing a user from having one consolidated digital wallet to manage all of his or her financial instruments. Commercial efforts to implement extensible digital wallets that are capable of inter-operating with multiple instruments and protocols are a step in the right direction, but these wallets have other limitations. In this paper, we propose a new digital wallet architecture that is extensible (can support multiple existing and newly developed instruments and protocols), symmetric (has common instrument management and protocol management interfaces across end-user, vendor, and bank applications), non-web-centric (can be implemented in non-web environments), and client-driven (the user initiates all operations, including wallet invocation).

1.0 Introduction

A number of electronic commerce applications allow end-users to purchase goods and services using digital wallets. Once a user decides to make an online purchase, a digital wallet should guide the user through the transaction by helping him or her choose a payment instrument that is acceptable to both the user and the vendor, and then hide the complexity of how the payment is executed. A number of wallet designs have recently been proposed, but we will argue they are typically targeted for particular financial instruments and operating environments. In this paper, we describe a wallet architecture that generalizes the functionality of existing wallets, and provides simple and crisp interfaces for each of its components.

In particular, the architecture we propose here has the following features. Existing proposals have some of these features, but we believe that none provides *all* of them in a comprehensive way.

Extensible. A wallet should be able to accommodate all of the user's different payment instruments, and inter-operate with multiple payment protocols. For example, a digital wallet should be able to "hold" a user's credit cards and digital coins, and be able to make payments with either of them, perhaps using SET [1] in the case of the credit card, and by using a digital coin payment protocol in the latter case. As banks and vendors develop new financial instruments, a digital wallet should be capable of holding new financial instruments and making payments with these instruments. For instance, vendors should be able to develop electronic coupons that offer discounts on products without requiring that users install a new wallet to hold these coupons and make payments with them. Similarly, airlines should be able to develop frequent-flyer-mile instruments so that users may pay for airline tickets with them.

Many existing commercial digital wallet implementations are not extensible. They support limited, fixed sets of payment instruments and protocols, or require extra coding effort to support each instrument and protocol combination. In this scenario, end-users may need to use different wallets depending upon the payment instrument they want to use, and may even need to use different wallets to make purchases from different vendors. The CyberCash wallet, for example, only supports payments using certain credit cards and "CyberCash Coins." [2] Similarly, DigiCash's ecash Purse only supports ecash issued by a set of issuer banks [3]. The Millicent wallet only supports scrip used to make micro-payments [4]. Furthermore, while most existing wallets support at least one protocol for issuing payments, few support protocols for other types of financial transactions such as refunds or exchanges.

There do exist efforts to build digital wallets that support multiple financial instruments and payment protocols such as the Java Wallet [5] and the Microsoft Wallet [6]. In addition, some of these efforts are beginning to gain support as evidenced by initiatives such as CyberCash's development of a CyberCoin Client Payment Component (CPC) for the Microsoft Wallet, allowing users to make payments with CyberCash coins using Microsoft's Wallet. Unfortunately, these wallet architectures do not provide all of the features we describe next.

Symmetric. Vendors and banks run software analogous to wallets, which manages their end of the financial operations. Since the functionality is so similar, it makes sense to re-use, whenever possible, the same infrastructure and interfaces within end-user, vendor, and bank wallets. For example, the component that manages financial instruments (recording account balances, authorized uses, etc.) can be shared across these different participants in the financial operations. If the wallet components that are re-used are extensible, then we automatically get extensibility at the bank or vendor. So, for instance, an extensible instrument manager will allow the bank or vendor to easily use new instruments as they become available.

Current wallet implementations are often not symmetric. For instance, the components that make up the Microsoft Wallet are client-side objects. Seemingly little infrastructure is shared between the server-side CGI-scripts that process electronic commerce transactions, and the client-side Active/X controls that make up the wallet.

Non-web-centric. Interfaces should be similar regardless of what type of device or computer that the user, bank, or vendor application is running on. A digital wallet running on an "alternative" device, such as a personal digital assistant (PDA) or a smart card, for example, has substantial functionality in common with a digital wallet built as an extension to a web browser. Thus, a digital wallet in these environments should re-use the same instrument and protocol management interfaces.

Many existing wallet architectures such as the Microsoft Wallet and the Java Wallet are heavily web-centric (as they are implemented as Active/X controls or plug-ins, respectively). With the exception of the JECF's (Java Electronic Commerce Framework's) [5] recent inclusion of a smart card API, these wallets do not even begin to address issues surrounding digital wallets running on "alternative" devices (such as PDAs), or in non-web environments.

Client-Driven. The interaction between the wallet and the vendor, we believe, should be driven by the client (i.e., the customer). Vendors should not be capable of invoking the client's digital wallet to do anything that the end-user may resent or consider an annoyance. For example, a vendor should not be able to automatically launch a client's digital wallet application every time the user visits a web page that offers the opportunity to buy a product. Imagine what life would be like if, simply by walking into someone's store, the store owner had the right to reach into your pocket, pull out your wallet, hold it in front of you, and ask you if you wanted to buy something! A client-driven approach for building a digital wallet is important because software which customers consider "intrusive" will hinder the success of electronic commerce for all participants involved.

Some commercial wallets are not purely client-driven, since some of them can allow vendors to invoke a user's wallet simply by either: 1) having the user visit the vendor's web site, or 2) having the vendor send the user email. (See Section 6 for details.) When a vendor invokes the Java Wallet, for example, the splash page screen of the wallet applet is brought up and the user is prompted to enter her wallet password.

The wallet architecture we propose here has the features we have described. Specifically, 1) it can inter-operate with multiple existing and newly developed instruments and protocols; 2) it defines standard APIs (Application Programming Interfaces) that can be used across commerce applications for instrument and protocol management; 3) it builds a foundation general enough to implement digital wallets on "alternative" devices in addition to wallets as extensions to web browsers; and 4) it ensures that electronic commerce operations, including wallet invocation, are initiated by the client. Our contribution is not a set of "new" services for wallets, but rather a flexible architecture that incorporates the best of existing ideas in a clean and extensible way. To verify some of our functionality claims, we have implemented this architecture in Java and C++. The Java version supports most of the features described in the body of the paper. In addition, while the C++ version implements only a subset of the features described here, it does provide support for digital wallets to run on non-traditional devices such as PDAs [17]. These implementations run on the Windows and PalmOS platforms, and implementation details are described in the body of the paper.

2.0 Terminology

In this section, we briefly define some terminology necessary for understanding our wallet architecture. The wallet architecture is described in the next section.

2.1 Instrument Instance & Instrument Class

An *instrument instance* (or, *instrument*, for simplicity) is a collection of state information representing economic value that a protocol can operate on as part of an electronic commerce transaction. For example, "Gary's Citibank Mastercard" is an instrument whose state is made up of his full name, credit card number, and expiration date. The instrument may also store other information such as his billing address.

It is important to note that an instrument, in the context of this paper, is a digital proxy for an instrument in the "physical" world. "Gary's Citibank Mastercard" is, in reality, an agreement or contract between Gary and Citibank which may be made up of a signed credit card application in addition to other documents such as a contract stating Gary's credit limit and the terms of the agreement. The digital representation of this instrument, however, only contains state parameters that are relevant for conducting commerce transactions with that instrument, and the instrument's digital representation need not contain the actual contract. In the case of "Gary's Citibank Mastercard", for example, the digital instrument may only contain those state parameters necessary for the SET protocol [1] to execute an online payment operation.

Each instrument belongs to an *instrument class*. An instrument class defines the structure of the state information necessary to store an instance of a given instrument, as well as behavior that is common to all instruments of that class. Examples of instrument classes are CyberCoin [2], ecash [3], or Mastercard. "Gary's Citibank Mastercard" is an instance of the Mastercard instrument class.

2.2 Protocol

A *protocol* defines a sequence of *steps* that accomplish a particular *operation* using a specified instrument. In each step, the protocol may send information to a peer, or process information locally. For example, in one step the protocol may create a certificate containing the user's account number and payment amount. In the next step, it may send the certificate to the peer.

The work of a protocol is to execute a correct sequence of steps to accomplish a requested operation; the sequence is not necessarily static and pre-determined, but may vary dynamically depending upon requests and responses sent between the parties executing the protocol. In general, a payment protocol is one that supports a PAY operation and whose sequence of steps results in a transfer of economic value between two or more parties. SET is a payment protocol that may be used to transfer monetary value from a bank to a vendor's account, while concurrently (and atomically) debiting the user's credit card account, under the condition that the resulting balance does not exceed the user's credit limit.

A protocol may be defined to be compatible with one or more instrument classes. The SET protocol is compatible with both credit card and debit card instrument classes, and may be used to execute payment operations with both credit cards and debit cards (see Figure 1). The CyberCash Protocol, on the other hand, may only be compatible with the CyberCoin instrument class.

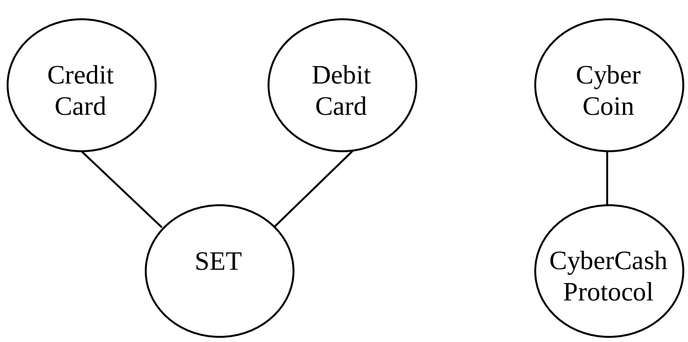


Figure 1: Protocol & Instrument Class Compatibility

2.3 Client

A *client* is a human user or a software agent. A software agent may allow the user to make online purchases or participate in online auctions, for instance.

2.4 Digital Wallet

A *digital wallet* is a software component that provides a client with instrument management and protocol management services. Instrument management and protocol management are defined in Section 3, but, in brief, are services that allow the wallet to 1) install and uninstall instrument classes and protocols, 2) create, update, and delete instruments and protocols, and 3) execute protocols. Digital wallets are capable of executing an operation using an instrument according to a protocol. A digital wallet presents its client with a standard interface of functions; in the case that the client is a human user, this standard interface of functions may be accessed through a graphical user interface (GUI).

A digital wallet is linked into an end-user, bank, or vendor application and provides the application with instrument management and protocol management services. The digital wallets that are linked into vendor and bank applications provide these management services in the same way that end-user digital wallets do. A vendor's digital wallet, however, may be part of a much larger software application that is integrated with order and fulfillment systems. Similarly, a bank's digital wallet may be part of a larger application that is integrated with general ledger, profit & loss, and reconciliation systems.

Furthermore, a wallet is not limited to being a plug-in or applet or some other extension of a web browser. A digital wallet with a graphical user interface may also run as an application on its own. A digital wallet may also run on computers that are not connected to the Internet such as smart cards or personal digital assistants. The user interface to the digital wallet may vary in such cases. In the case of a PDA, for example, the digital wallet may have a pen-based user interface. In the case of a smart card, the digital wallet may have no user interface at all. Nevertheless, in each case, the set of functions that the digital wallet's interface presents to its client should be the same.

2.5 Peer

A *peer* is a remote entity that may be an end-user, vendor, or bank wallet that is capable of performing operations on instruments according to a protocol.

2.6 Session

A *session* is an interaction between two peers, and state information that may be built up over time as a result of interaction between the two peers may be stored in a `Session` object. One peer is said to initiate a session, while the receiving peer is said to service the session. The `Session` object keeps track of which peer is the initiator and which is the servicer.

3.0 SWAPEROO Architecture

In the following, we present an *extensible*, *symmetric*, *non-web-centric*, and *client-driven* architecture for digital wallets.

In the SWAPEROO architecture, the interaction between a client wallet and a peer wallet roughly works as follows: Once a session is initiated by the client and the peer wallet prepares to service the client, the client can determine what instrument classes are available on the peer wallet, and then select an instrument class that is common to both peers. After an instrument class is selected, protocol management functions are called to determine what available protocols may be used to conduct operations on an instrument of the selected class. Depending upon what protocols are shared, a protocol is selected. The protocol supports certain operations for the selected instrument class, and the client may invoke those operations on an instrument instance. This interaction is described in detail in Section 5.

A digital wallet is an object that has four required key architectural component objects: a Profile Manager, an Instrument Manager, a Protocol Manager, and a Wallet Controller (see Figure 2).

In Figure 2, objects within the dotted lines are the core components of the wallet object[1]. We assume that communication between the core components of the wallet object is secure such that sensitive data structures containing private information about users and their instruments may be passed between objects within the wallet. In a real wallet implementation, this "boundary" around the secure components of a wallet may be supplied by the operating system by having the core components reside within the address space of a process. This approach, of course, assumes that the operating system is trusted and safe. A trusted operating system will not itself attempt to compromise the security of the wallet, and a safe operating system allegedly has no loopholes which would allow other malicious software it is running to compromise the security of the wallet by reaching into its address space.

Since code modules that implement instruments and protocols may be obtained from different sources, they cannot all be mutually trusted. In particular, it should not be possible for instruments or protocols developed by a software vendor to compromise the privacy, security, or functionality of instruments or protocols developed by another software vendor. At the same time, it is beneficial to have instruments and protocols

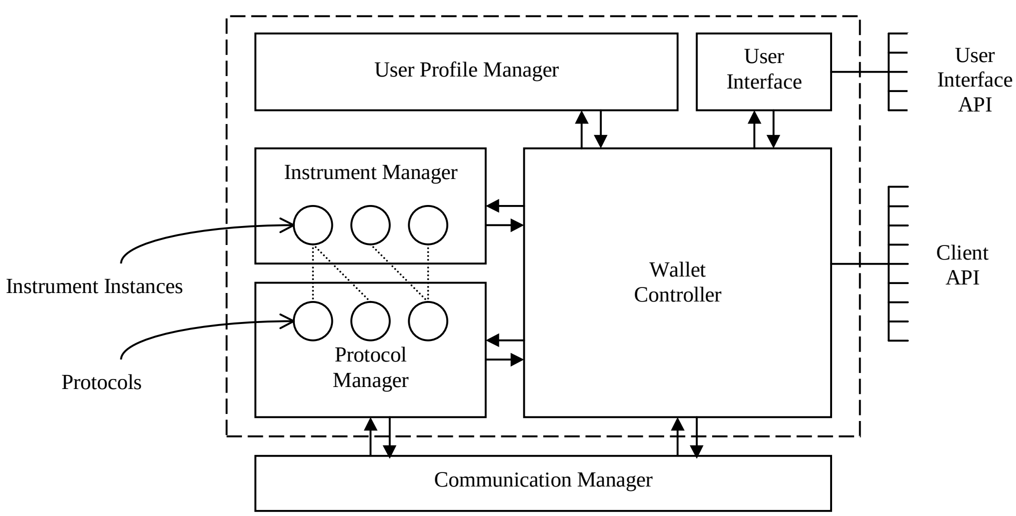


Figure 2: The SWAP Generalized Digital Wallet Architecture

[1] Although the User Interface is a core component when present, it is not a required component.

run within the same address space such that they can exchange private data efficiently.

Two approaches should both be taken to accomplish these conflicting goals. Code modules that implement instruments and protocols should, first of all, be digitally signed by their source (i.e. the software vendor that developed the module). As part of the installation process of such a code module into a wallet, the signature on the code module should be checked to determine if the module can be trusted. Secondly, after installation of the module, a capabilities-based security model needs to be employed to protect modules of code from each other. In such a model, a called object would be able to authenticate the object calling it by verifying that object's digital signature. The security model may also provide support to allow an object calling another object to verify the called object's digital signature. Under such a security model, objects that implement various instruments and protocols may authenticate each other at run-time to prevent potentially "malicious" calls from taking place. Implementing such a security model was beyond the scope of our work. A Java-version of such a security model is addressed by [16], and could be re-used within our work.

Objects that are outside the dotted lines reside in a different address space. However, under a capabilities-based security model in which these external objects are only granted the appropriate capabilities to access privileged data, they may optionally reside in the address space of the wallet process itself, or can be dynamically-linked into the wallet process.

All components of the wallet are briefly described below except for the wallet's Cryptographic Engine (which has been excluded from Figure 2, since all components of the architecture within the wallet may use the Cryptographic Engine to encrypt sensitive data). The Cryptographic Engine resides within the wallet's address space.

1. The *Instrument Manager* manages all of the instrument instances contained in the wallet, and may be queried to determine which instrument classes and instances are available to execute a given payment or other operation.

2. The *Protocol Manager* manages all of the protocols that the wallet may use to accomplish various operations, and invokes protocols to carry out the interaction between the digital wallet and the vendors and banks. The Protocol Manager relies on the Communication Manager to process low-level communications requests with other computers representing banks and vendors.

3. The *Wallet Controller* presents a consolidated interface for the entire wallet to the client by coordinating the series of interactions between the Profile Manager, Instrument Manager, and Protocol Manager necessary to carry out high-level requests received from the client, such as "purchase a product." The Wallet Controller hides the complexity of the other components of the wallet, and provides a high-level interface to the client. A non-human client, or software agent, can make method calls on the Wallet Controller's interface through the Client API. A human client may use a graphical user interface (GUI) which may make method calls on the Wallet Controller. The Wallet Controller also handles end-user authentication and access control for operations in the wallet.

4. The *User Profile Manager* manages information about clients and groups of clients of the wallet including their user names, passwords, ship-to and bill-to addresses, and potentially other profile information as well. In addition, the Profile Manager keeps access control information about what financial instruments each user has the authority to access, and the types of operations specific users have the privilege to execute with them.

5. The *Communication Manager* provides the wallet with an interface to send and receive messages between a wallet and a peer by setting up a "connection" with a remote Communication Manager. The Protocol Manager builds on top of the "connection" abstraction to support the concept of a session. A "connection" is typically asynchronous, while communications between peers in a session occur in (message,response) pairs where one peer sends a message, the other peer receives the message, executes some action, and returns a response. Depending upon the implementation of the Communication Manager, the messages may be sent over different types of networks using different communication protocols.

   For example, one implementation of a Communication Manager may send and receive messages over the Internet using HTTP requests and responses over a TCP/IP ethernet network. In this case, a session may be made up of a sequence of several HTTP GET messages and their corresponding responses. Another implementation of a Communication Manager may send and receive messages over an RS232 port.

Note that the Protocol Manager is responsible for making calls to the Cryptographic Engine to encrypt any data that is passed to the Communication Manager, such that the data can be securely transmitted over the communications medium. The Communication Manager cannot be responsible for encryption of sensitive data from the wallet because it is not a core component, and can be replaced by another Communication Manager to run the wallet on another device. If the Communication Manager is relied upon to encrypt sensitive data, then the Communication Manager might be replaced with a malicious Communication Manager that sends all sensitive data to an adversary.

6. The *Client API* is an interface provided by the Wallet Controller that may be used by a software agent acting on behalf of an end-user, vendor, or bank.

7. The *User Interface* provides a graphical interface to the services offered by the Wallet Controller's interface. The User Interface is an optional component of the wallet. Some devices, such as most smart cards, do not have the ability to display a graphical user interface, and hence the Wallet Controller interface must be accessed through the Client API. Note that the User Interface is a core component within the wallet because certain parts of the user interface have access to sensitive user data. For example, the edit box object into which a user enters the password to "unlock" the wallet should run within the wallet's protected address space. On the other hand, customization of the wallet's user interface presents an important branding opportunity for banks and vendors that distribute wallets.

8. The wallet's user interface exports parts of its interface as the *User Interface API* to satisfy both the privacy and customization requirements. Methods in the User Interface API may be overloaded by software vendors to render customized parts of the interface. The User Interface API also decouples the GUI so that the GUI can be run on a thin client, such as a network computer, while the core components of the wallet can be run on a server.

We will now describe each of the required core components of the digital wallet.

3.1 Instrument Management

The Instrument Manager is responsible for managing instrument instances, as well as information about classes of instruments. Instrument Management is made up of the following services: instrument capability determination, instrument installation, instrument storage and retrieval, and instrument negotiation. Before describing the Instrument Manager interface in detail, we will first briefly describe the instrument objects that the Instrument Manager is responsible for storing and retrieving.

An instrument may be a financial instrument that can be used to make a payment, such as a credit card, debit card, or electronic coin. More generally, however, an instrument is made up of state information representing economic value that a protocol can operate on. For example, a digital cash instrument's state can be made up of its dollar (or other currency) value digitally signed by its issuing bank. The protocol used between an end-user wallet and a vendor wallet supports a `VERIFY` operation which verifies that the cash is authentic by applying the issuing bank's public key to the coin.

While end-user, vendor, and bank wallets share many code modules, some specialization is appropriate. Instruments may have to be managed slightly differently in end-user, vendor, and bank wallets. To illustrate this, we consider a trivially simple digital cash instrument example; please note that in real systems such a naive digital cash scheme is not viable because of real-world security, efficiency, and performance considerations. In this "toy" digital cash example, every time that a vendor receives a digital coin signed by a client's private key, the vendor needs to keep track of that signature in addition to its dollar value in its digital cash instrument. On the other hand, the client may want to keep track of the vendor's signatures on coins she signed for purposes of non-repudiation in her digital cash instrument. Although this is a simple case, we can begin to see that the state information for the digital cash instrument may differ depending upon whether or not the digital cash is being stored in the end-user's wallet or in the vendor's wallet.

Consider a digital cash instrument whose instrument class is `Digital-Cash-Instrument`. We derive two subclasses from our `Digital-Cash-Instrument` to manage the different implementations of the digital cash instrument on the different peers. A `Vendor-Digital-Cash-Instrument` is stored on the vendor, and is able to store a list of client's digital coin signatures. A `Client-Digital-Cash-Instrument` is able to store the vendor's signature on the client-signed coins. Although the `Vendor-Digital-Cash-Instrument` and the `Client-Digital-Cash-Instrument`, for the most part, present similar interfaces since they both derive from `Digital-Cash-Instrument`, the subclasses present specialized interfaces to their respective callers to access client or vendor-specific instrument information. Note once again that this is a trivial example, and is provided simply to illustrate that the representation of the digital cash instrument may need to be specialized depending upon whether the peer is a user, vendor, or bank.

In addition to providing access to instrument information in the digital wallet's memory, the Instrument Manager provides interfaces to store and retrieve instruments to and from persistent storage. Note that the Instrument Manager may make calls, if necessary, to the wallet's Cryptographic Engine to encrypt instrument state information in preparation for writing this information to persistent storage, and for decrypting instrument state information when reading this information back from persistent storage.

Upon initialization, the Instrument Manager determines what instrument classes the wallet is capable of using by consulting a configuration file, dynamically determining this through introspection, or by accessing a Capabilities Management service [7]. Alternatively, the Instrument Manager can dynamically download an instrument class from a trusted third-party, and install it. The Instrument Manager may call the Cryptographic Engine to verify that the instrument class code is signed by the trusted third-party. Once the code supporting the appropriate instrument classes is loaded, instrument instances can be created by the user, but more often are loaded from encrypted files on the user's local hard disk, or even potentially from a file server on a network. Finally, the Instrument Manager supports methods to create, modify, commit changes to, and delete instrument instances under transactional semantics.

The Instrument Manager supports methods that query for available instrument classes to conduct instrument negotiation. Note that in our client-driven approach there is no way for a vendor to "Offer" instrument capabilities as there might be in [8], unless the vendor is explicitly queried.

3.2 Protocol Management

The Protocol Manager is responsible for managing protocol objects. Protocol Management is made up of the following services: protocol capability determination, protocol installation, and protocol negotiation.

Upon initialization, the Protocol Manager determines what protocols the wallet is capable of using. As with Instrument Managers, this information can typically be read from a configuration file, determined dynamically through introspection, or can be accessed through a Capabilities Management service [7]. Once this information is determined, a class representing each protocol is loaded into memory, and one instance of each protocol class is instantiated. The instantiation of each protocol instance can be delayed until the Protocol Manager needs to use that protocol.

A protocol is capable of performing operations with an instrument. The exact operations that the protocol can support during a particular session may depend upon the type of peer the wallet is connected to. Consider, for example, a digital coin instrument, and associated digital coin protocols. During a session in which the wallet is connected to a bank, the digital coin protocol may provide deposit and withdrawal operations. On the other hand, while connected to a vendor, the digital coin protocol may provide purchase and refund operations. Protocol objects are responsible for ensuring that such operations take place under transactional semantics.

Furthermore, the Protocol Manager is capable of conducting protocol negotiation with peers for a specified instrument class. That is, the Protocol Manager is capable of determining which protocols the local and remote peers share in common for the specified instrument class. The Protocol Manager accomplishes protocol negotiation with a peer through a Protocol Negotiation Protocol (PNP). Protocol Negotiation Protocols are subclasses of `Protocol` in our architecture, and are managed in the same way as other protocols are managed, with the exception that the Protocol Manager must successfully load a PNP upon initialization. At least one PNP must load successfully such that it may engage in protocol negotiations with peers. Additionally, although a wallet may load more than one PNP upon initialization, the particular PNP that will be used to conduct protocol negotiation must be fixed by both peers before a session is initiated, otherwise a PNP to decide which PNP to use becomes necessary, and so forth.

Protocols may be *compatible* with one or more instrument classes, and a protocol object is capable of determining whether or not it is compatible with an instrument class. Once an instrument class is chosen by the client, the Protocol Manager may query the protocol objects it manages to determine which ones are compatible with the chosen instrument class. If more than one protocol is compatible with the chosen instrument class, the Protocol Negotiation Protocol may pass the list of compatible protocols to the instrument class to allow the instrument class to determine which protocol is *optimal* for that instrument class.

To illustrate with an example, consider two protocols, `Clear-Text-HTTP-Post` and `SET-Protocol`, that are both compatible with a `VISA-Credit-Card` instrument. When the Protocol Manager calls the `Clear-Text-HTTP-Post` protocol's `compatible-With` method passing the `VISA-Credit-Card` instrument class as an argument, the return value will be `true`. Similarly, when the Protocol Manager calls the `SET-Protocol` protocol's `compatible-With` method passing the `VISA-Credit-Card` instrument class as an argument, the return value will also be `true`. Both protocols may not be available on both the client and the peer, but in the case that they are, the PNP may call the instrument class's `get-Preferred-Protocol` method passing both protocols as parameters. The instrument class is responsible for determining which protocol is optimal (using its own definition of optimal), and the `VISA-Credit-Card` instrument class may return `SET-Protocol` in favor of `Clear-Text-HTTP-Post` to obtain the best possible security for subsequent operations.

In summary, a protocol object is responsible for determining *compatibility* with an instrument class, the PNP is responsible for determining the *availability* of protocols between peers, and an instrument class can be called upon by a PNP for determining *optimality* amongst several compatible protocols available to both peers.

3.3 User Profile Management

The User Profile Manager stores information about clients and groups of clients of the wallet. The model we assume is one in which a client is a person or software agent that has authorization to use one or more financial instruments. It is important to note that most existing wallet implementations only allow for clients that are human users and not software agents. However, for the sake of discussion in this section, we will use the terms client and user interchangeably. A group is a set of clients that have access to a set of shared financial instruments, and each client within a group is authorized to conduct financial transactions with the set of shared financial instruments. When financial instruments may be shared between clients in a group, the instruments may or may not be used concurrently by two or more users in the group depending upon the type of the instrument. Although the User Profile Manager provides access to information about which clients are authorized to use which instruments, and who "owns" or may access which instruments, the Instrument Manager (described in Section 3.2) provides synchronized, concurrent access to use instruments, such that conflicting operations are prevented. For example, a wallet should ensure that digital cash owned by a group of clients does not get doubly spent because two clients attempt to concurrently make a payment using the same digital cash instrument instance.

An example of a group might be a corporate department. Each employee in the department may have access to a set of shared financial instruments, but depending upon an employee's position within the company, the employee may be authorized to only conduct transactions under a certain amount.

3.4 Wallet Controller

The Wallet Controller provides an interface to all of the services that the wallet may offer to external objects. The "outside world" cannot see, and does not have direct access to any of the components internal to the wallet such as the Instrument or Protocol Managers. In our implementation, the components of the wallet are all private data members of the `Wallet-Controller` class.

Once a Wallet Controller method is invoked, the Wallet Controller coordinates the steps that need to be carried out among the User Interface, Profile Manager, Instrument Manager, Protocol Manager, and Cryptographic Engine to execute a payment or other operation. Before accessing or carrying out an operation with instrument instance data, the Wallet Controller makes the appropriate calls to the User Profile Manager to ensure that the user involved has the appropriate privileges to carry out the operation.

4.0 Vendor and Bank Digital Wallets

The end-user wallet interacts with bank and vendor wallets to execute commerce transactions. The bank and vendor wallets have architectures symmetric to the user's digital wallet, as shown in Figure 3.

In place of a User Profile Manager, the bank wallet has an Account Profile Manager that allows the bank to manage non-financial information about the bank's clients. (Financial information about the bank's clients is stored in instruments in the bank's Instrument Manager.) Similar to the way in which the User Profile Manager maintains access control information about instrument instances, the Account Profile Manager does the same for the bank wallet. The Bank Controller queries the Account Profile Manager before accessing instrument instance data to determine whether or not the user has the appropriate authorization to conduct a transaction; in the case that a user's credit line has been overdrawn, or if the user's credit card has been cancelled, the Account Profile Manager would return an error response to the Bank Controller's query.

In the vendor wallet, the User Profile Manger is replaced with a Customer Profile Manager, which is used to store access control information about its customers in the case of non-anonymous transactions. For example, the Customer Profile Manager might store the user's age, and the Vendor Controller may query the Customer Profile Manager to determine if the user is above a certain age to purchase a product.

A key difference between the Wallet Controller running in the end-user application and the Bank and Vendor Controllers running in the bank and vendor applications, respectively, is that the end-user Wallet Controller drives the wallet interaction, and it is active, in that it generates requests and receives responses. The Bank and Vendor Controllers, on the other hand, are passive, in that they receive requests and generate responses. A Wallet Controller generates requests, and these requests are pushed down through its Communication Manager and out to the peer wallet. Peer wallets receive requests through their Communication Managers, and the requests are propagated to and are serviced by the Bank and Vendor Controllers.

The Instrument, Protocol, and Communication Managers used by the end-user, bank, and vendor are one and the same, and are re-used across the wallets.

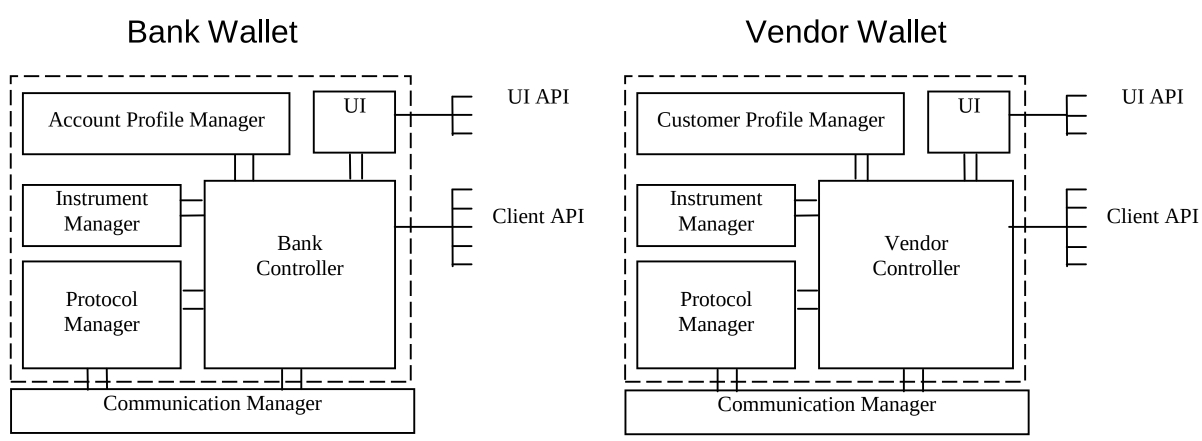


Figure 3: Symmetric Bank & Vendor Wallet Architectures

5.0 Wallet Operation & Interaction Model

In this section, we describe how our symmetric, client-driven wallets operate and interact during a session to execute a payment. (Executing any other operation such as a refund happens in a similar fashion, and may possibly require only a subset of the steps described below.) In our example here, we assume that the "ordering" phase of the shopping interaction is complete, and the necessary "invoice" (containing information about what products and/or services the end-user would like to purchase) is stored in a property list called `inv` as a set of (name, value) pairs. We will describe the process from the point of view of the end-user wallet, and we will use the diagram in Figure 4 as an aid. To the right of the states in Figure 4, we supplement the figure with the method names from our wallet implementation that an application would invoke on the Wallet Controller. Methods in boldface are public in the Wallet Controller's interface, while methods in regular font are private to the Wallet Controller. Finally, the careful reader will note that the methods seem single-threaded; it is because the wallets in our implementation are single-threaded. However, we can easily extend to the multi-threaded case by passing the Session object to each method (excluding `initiate-Session()`), or by instantiating one Wallet Controller per session.

Although we describe the interaction from the end-user wallet point of view, it is important to keep in mind that a vendor or bank's wallet can also be a client in a wallet interaction. For example, as part of a transaction with the end-user wallet, a vendor's wallet may act as a client and initiate a session with a bank's wallet to obtain credit authorization information for a purchase.

5.1 Initialization / Session Initiation.

When an application is launched, static initialization takes place before it starts executing. During static initialization, the wallet components including the Instrument Manager, Protocol Manager, and Profile Manager are constructed and initialized. After static initialization, the wallet may be "unlocked" by supplying login/password information and a session with a peer wallet may be initiated.

The Wallet Controller presents the user's login and password to the User Profile Manager to determine if the user should be allowed to use the wallet. The Wallet Controller also passes the user's password to the Instrument Manager so that it may use the password to decrypt instrument instances and/or the user's private key from persistent store.

To initiate a session, the `initiate-Session` method in the Wallet Controller is called, passing a `Peer` object as a parameter. (A `Peer` object contains the peer's name and details about how to set up a session with that peer.) The Wallet Controller's `initiate-Session`, in turn, calls the Protocol Manager's `initiate-Session` to initiate the session. The Protocol Manager makes calls to the Communication Manager to set up the session with the remote peer using the underlying communication mechanism.

5.2 Instrument Class Negotiation

The first step that takes place after session initiation is instrument class negotiation. In this step, the client's wallet can determine what instrument classes are known to both wallets by 1) determining what instrument classes are known to its Instrument Manager, 2) determining what instruments are known to the remote Instrument Manager, and 3) computing the intersection.

Those instrument classes that are available to both wallets may offer different terms and conditions for purchasing a given set of products or services. As an example, the price of a product may vary depending upon whether the customer decides to pay using a credit card or using ecash. Furthermore, extended warranties may be offered in the case that a credit card is used to make a purchase. For each available instrument class, the instrument class negotiation step also determines these terms and conditions.

To start instrument class negotiation, the application calls the Wallet Controller's `get-Instrument-Classes` method with the invoice information, `inv`, as an argument. (The invoice information is included to determine the available instrument classes because some instrument classes may not be applicable for certain types of purchases. Also, the `inv` property list is populated with the terms and conditions described above such that the terms and conditions become part of the invoice.) The Wallet Controller then calls its `get-Local-Available-Instrument-Classes` method to determine what instrument classes the local wallet supports and are available. The Wallet Controller makes this determination by, in turn, making a call to the Instrument Manager to determine what instrument classes are available to the wallet. Then, the Wallet Controller calls its `get-Remote-Available-Instrument-Classes` method to determine what instrument classes the remote peer is capable of dealing with.

The call to get-Remote-Available-Instrument-Classes results in a remote procedure call to the peer's Wallet Controller. To respond to the get-Remote-Available-Instrument-Classes procedure call, the remote Wallet Controller calls its get-Local-Available-Instrument-Classes. Other calls of the form get-Local... and get-Remote... described in the following sections work similarly. Also, in our implementation, the get-Remote-Available-Instruments call populates a "price" property stored in inv with a list of (instrument class, price) pairs to indicate the different prices that would be charged for using the corresponding instrument classes in addition to reporting the available instrument classes. The call chain for instrument class negotiation is depicted in Figure 5. Solid arrows in Figure 5 indicate method calls from the object from which the arrow originates, and dotted arrows indicates the return of control. (The Wallet Controller relies on the Communication Manager to handle the low-level details of executing the remote procedure call described above, but the actual calls that the Wallet Controller makes to the Communication Manager have been omitted from Figure 5 to keep the diagram simple. Also, arguments and return values for the methods have been omitted from Figure 5 for the same reason, but this information can be found in Figure 4.)

To digress momentarily from the end-user wallet's point of view, note that, in Figure 5, after the vendor's Wallet Controller calls its local Instrument Manager to determine what instrument classes are available, it may optionally "notify" the Vendor Application. The vendor Wallet Controller gives the Vendor Application the ability to subscribe to various events and possibly filter the results before they are returned to the end-user's Wallet Controller. Although this capability is not of crucial importance during instrument class negotiation, it is useful to notify the Vendor Application of other events, such as the successful execution of a payment. If the Vendor Application is, for example, a front-end for a vending machine, the vending machine would dispense the appropriate product after receiving notification that payment was successfully transferred.

To complete instrument class negotiation, the Wallet Controller calls get-Common-Instrument-Classes to compute the intersection of available instrument classes. The results received from get-Local-Available-Instrument-Classes and get-Remote-Available-Instrument-Classes are passed as parameters to get-Common-Instrument-Classes.

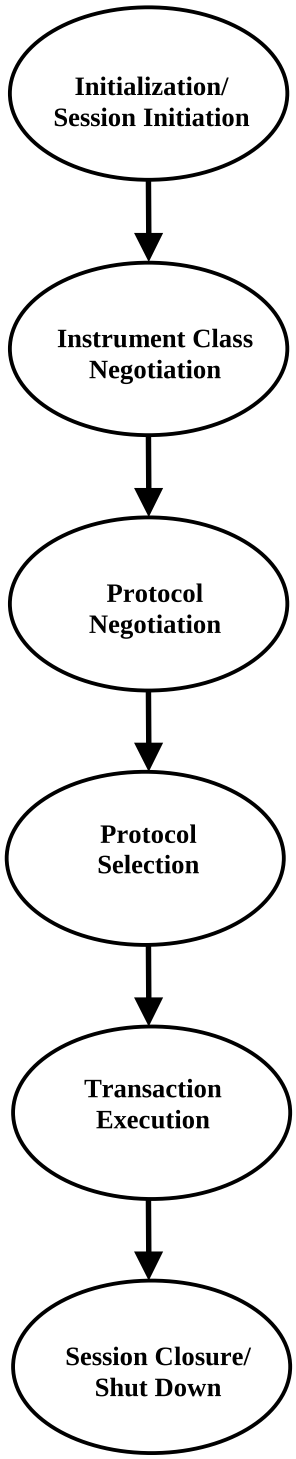
Once instrument class negotiation is completed, the application is presented with a list of instrument classes with which the user may execute a transaction. The user must select one or more instrument classes before the next step, protocol negotiation, can begin, since the choice of what protocols can be used to execute a transaction is dependent upon the instrument classes that the user selects. In a typical purchase, one instrument will be used to purchase the products and services specified in a specific inv record. However, multiple instruments, possibly of different instrument classes, may be used to make such a purchase, and protocol negotiation for each instrument class would need to be carried out.

5.3 Protocol Negotiation

Once the instrument class has been negotiated, the application may call the Wallet Controller's negotiate-Protocol method to start protocol negotiation. The Wallet Controller's negotiate-Protocol method, in turn, calls the Protocol Manager's negotiate-Protocol method. The Protocol Manager's negotiate-Protocol method then calls the doOperation method of the currently active Protocol Negotiation Protocol (PNP), and also sends a message to the peer informing it that the local wallet is entering the protocol negotiation step. The peer will symmetrically call its PNP's do-Operation.

The default PNP that we implemented calls the Protocol Manager's get-Local-Available-Protocols method to obtain a list of protocols available that are locally compatible with the selected instrument class, and then calls get-Remote-Available-Protocols to determine the protocols that the remote wallet supports for the selected instrument class. The PNP finally calls get-Common-Protocols to compute the intersection. This default protocol negotiation mechanism is similar to instrument negotiation.

Wallet Interaction Model | Wallet Controller Methods




```
Session initiateSession(Peer aPeer)

List getInstrumentClasses (PropertyList inv)
  List getLocalAvailableInstrumentClasses(PropertyList inv)
  List getRemoteAvailableInstrumentClasses(PropertyList inv)
  List getCommonInstrumentClasses(List iclist1, List iclist2)

ProtocolRecord negotiateProtocol (InstrumentClass anInstClass)
  List getLocalAvailableProtocols(InstrumentClass anInstClass)
  List getRemoteAvailableProtocols(InstrumentClass anInstClass)
  List getCommonProtocols(List plist1,List plist2)

void setLocalTransactionProtocol(InstrumentClass instclass,
                                 Protocol protocol)
void setRemoteTransactionProtocol(InstrumentClass instclass,
                                  Protocol protocol)
List getAvailableOperations(InstrumentClass instclass,
                            Protocol protocol)

List getInstruments (InstrumentClass instclass)
void executeTransaction (PropertyList pinfo,
                         Instrument anInstrument,
                         Operation anOperation)

void closeSession (Peer aPeer)
```

Figure 4: Wallet Interaction Model & Wallet Controller Interfaces

Although we implemented the simple PNP above, the remote peer could respond to `get-Remote-Available-Protocols` with a list of available protocol names and the location of the signed code for those protocol classes. Such an implementation of a PNP may decide to download, dynamically link, and install a protocol that is not locally supported by the Protocol Manager if the user agrees to permit the wallet to do so. Such a PNP may expedite the case in which the intersection of locally and remotely available protocols is the null set, which would prevent the wallets from executing a transaction. The Java Electronic Commerce Framework (see Section 6), for example, employs such a mechanism as the default behavior. Finally, since the notion of a Protocol Negotiation Protocol has been abstracted out in our architecture, a PNP such as JEPI's UPP protocol (see Section 6) could be used in place of our default PNP[2].

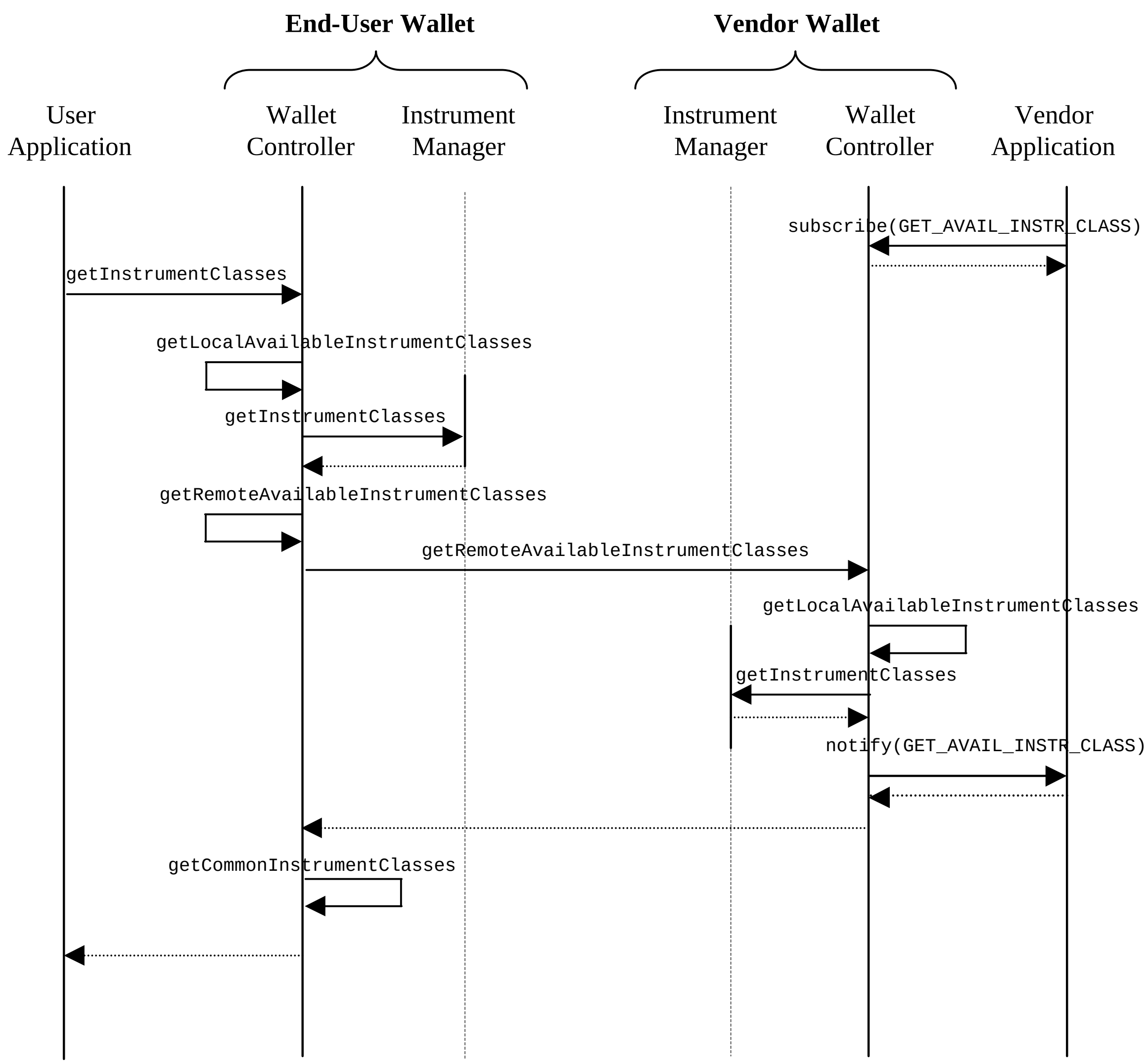


Figure 5: Instrument Class Negotiation Call Chain

[2] Note that Instrument Negotiation can also be abstracted out into an Instrument Negotiation Protocol, although for our implementation we felt that demonstrating this capability with Protocol Negotiation was sufficient.

5.4 Protocol Selection

If the result of protocol negotiation for a given instrument class yields one specific protocol, then protocol selection is automatic, and that protocol will be used to execute transactions with instruments of the given instrument class. On the other hand, if more than one protocol is available for a given instrument class, then a preferred protocol needs to be selected by the client's instrument class.

If the PNP reports more than one protocol in common, the Wallet Controller calls the instrument class's `get-Preferred-Protocol` method as mentioned in Section 3.2. The instrument class selects an appropriate protocol among the ones available based on a variety of parameters, such as the type of device on which the wallet is running, the level of network connectivity and bandwidth available, the dollar amount of the transaction, or user preferences.

For example, when executing a transaction using a wallet on a PDA with limited processing power but with a direct connection (via a docking port or cradle) to a vendor's cash register, an unencrypted session protocol might be preferred over an encrypted session protocol since the link could be assumed to be secure, and key exchange processing overhead need not be incurred to provide for a faster transaction. Such information about device characteristics and connectivity may be obtained from a system properties table similar to the table returned by Java's `System.getProperties` method. In general, policies regarding how to carry out protocol selection based on such parameters is beyond the scope of this paper; our architecture does, however, provide a framework and the appropriate "hooks" for such policies to be implemented by implementing and/or overloading the `get-Preferred-Protocol` method.

Once the protocol is selected for the given instrument class, the Wallet Controller invokes `set-Local-Transaction-Protocol` and `set-Remote-Transaction-Protocol` with the instrument class and selected protocol as arguments. Following protocol selection, the application may inquire what operations the selected protocol makes available by calling the Wallet Controller's `get-Available-Operations` method. Protocols offer different sets of operations depending upon the instrument class over which the protocol executes and, as described in Section 3.2, whether the peer wallet is a bank or vendor.

5.5 Transaction Execution

At this point, the application can present the user a list of the available operations and the user can select one of them for execution. After that, the user must select an instrument instance of a previously selected instrument class on which to execute the operation. To obtain a list of possible instrument instances that the user may choose from, the application calls the Wallet Controller's `get-Instruments` method passing the previously selected instrument class as a parameter. The application presents a list of the names of each of the returned instrument instances to the user, and the user can choose one of the instrument instances.

Once an operation and instrument instance are selected, the application calls the Wallet Controller's `execute-Transaction` method with these parameters, along with the invoice information stored in `inv`. After the Wallet Controller verifies that the user has the appropriate privileges to execute the transaction by querying the User Profile Manager, the Wallet Controller calls the Protocol Manager's `execute-Transaction` method. The Protocol Manager then calls the `do-Operation` method of the appropriate protocol object. The peer wallet is sent a message informing it of the name of the protocol and operation that is to be executed, and it then starts executing that protocol. The peer will symmetrically call the appropriate protocol object's `do-Operation` method on its side. The protocol objects then execute all of the necessary actions to accomplish the operation.

The operation may or may not execute successfully. An exception will be thrown by the `do-Operation` method if the operation fails. The remote vendor or bank application will typically subscribe to its wallet's `EXECUTE_TRANSACTION` events, and will be notified of a failed operation.

Until this point in the discussion, we have described the various wallet states in succession, but it is important to note that the application may decide to negotiate over a different instrument class or select a different protocol. For instance, after transaction execution, the application may return to the instrument negotiation step to select another instrument class for the next operation. As a second example, after selecting an instrument class, and after protocol negotiation and selection takes place, the application may decide that the range of operations available from the vendor for that instrument class and protocol combination is not acceptable. At that point, the application may decide to choose another protocol instead, and protocol negotiation will be conducted once again until a protocol is chosen and another set of operations can be presented to the user. In general, after instrument class negotiation the application can return to any previous step in the wallet interaction. (The extra arrows entailed by this have been left out of Figure 4 to keep the diagram uncluttered.)

This capability can also allow payments to be executed in multiple steps with different instruments. For example, if the price of making a given purchase is constant regardless of the instrument class, the vendor may accept receiving large payments as a combination of several smaller payments of different instruments. After instrument class negotiation takes place, the end-user selects multiple instrument classes, and protocol negotiation and selection is completed for each one. The end-user then specifies the amount to be paid with each instrument, and the transaction execution step occurs for each instrument. If the payment fails for any one of the instruments, the user may be prompted to select an alternate instrument, or `REFUND` operations need to be executed for all the other instruments to abort the payment. The wallets involved must take advantage of transaction management services to ensure that recovery and rollback are executed correctly in the face of machine or network failures.

Finally, as presented above, when the user chooses an instrument class, protocol negotiation and selection are done for that instrument class, and the user then selects an instrument instance and an operation to execute. If there are many different possible instrument class, protocol, and available operation options, this approach will best guide the user through the interaction. On the other hand, if there are relatively few instrument classes, protocols, and available operation options, then forcing the user to go through all those steps may be overkill. To avoid these extra steps, the application can "hide" some steps from the user even though both wallets must go through each of the steps.

Consider an example in which the user has only a few instrument instances, and wants to quickly make a payment. After initiating a session, the user application may call `get-Instrument-Classes` and then call `get-Instruments` for each instrument class returned to obtain all instrument instances, saving the user of having to select an instrument class. The user application can present the user with a choice of all instruments. After the user chooses an instrument with which to make the payment, the application determines the instrument class from the instrument object by calling the Instrument's `get-Instrument-Class` method, and then does protocol negotiation and selection for the instrument class. If it turns out that no protocol is available for that instrument class, an error is reported to the user. Similarly, if it turns out that a protocol can be selected, but that the `PAY` operation is not available, an error is reported to the user. If a suitable protocol can be selected, and the `PAY` operation is available, the user is saved from having to explicitly select an instrument class.

5.6 Close Session / Shut Down

In general, the application may close the session at any time. In the typical case, the application may do so after transaction execution, but may also do so after, for instance, finding that it does not share any instrument classes in common with a vendor. A peer application has the option of closing the session (but may just return an error if desired) if the application makes an invalid call, such as calling a method with an instrument class that is undefined or with an instrument class that it did not return as the result of `get-Remote-Available-Instruments`.

The application should "lock" the appropriate instrument instances upon closing a session. The results of all instrument negotiation, protocol negotiation, and protocol selection are forgotten upon close of the session, although the wallet architecture may be extended to include a feature to support caching of such information in the future.

After closing all sessions, the user may decide to shut down the wallet, at which point all wallet components save any unsaved information to persistent storage.

6.0 Related Work

Java Wallet / Java Electronic Commerce Framework. [5] The Java Wallet is not purely client-driven. In Sun's Java Electronic Commerce Framework, electronic commerce operations are initiated when a merchant server sends a Java Commerce Message (JCM) to a client's web browser. A JCM has a MIME-type of `application/x-java-commerce`, and the client's web browser will invoke the Java Wallet once the JCM message is received [9]. By convention, a vendor should not send a JCM to a client unless the client clicks a "PAY" button on a form on the vendor's web site. However, there is nothing preventing a vendor from sending a JCM to the client and invoking their wallet in response to the client simply visiting a page on the vendor's web site. These JCMs may be generated by CGI (Common Gateway Interface) scripts or servlets on the server-side, and sent to the client to invoke the Java Wallet. Alternatively, a JCM may also be generated by an applet that is downloaded to the client browser. In this case, the applet can make a method call on the JECF installed on the client to invoke the Java Wallet. Another way in which a vendor can invoke a user's Java Wallet in an unsolicited fashion is by sending them HTML email with such an applet embedded within it. Users that run HTML email readers that are integrated with their web browser, such as Netscape Navigator or Microsoft Internet Explorer and also have the JECF installed can have their wallet automatically invoked upon reading unsolicited email received from vendors. In order to render the HTML email message, the HTML will be parsed and the Java Virtual Machine will be started to render the applet the vendor embedded in the email. In this scenario, the applet on the page will start executing, generate a JCM, and make a `JECF.startOperation` method call passing the generated JCM as a parameter to invoke the user's Java Wallet.

Vendors can use these mechanism to urge customers to make impulse purchases simply by invoking an end-user's wallet when the end-user visits their web site or receives email from them. Customers may resent this feature since it gives vendors the ability to "take the customer's wallet out of his pocket." In our architecture, a user must *explicitly* launch the wallet to make a payment; this allows the user to make the payment when he or she pleases, and not when the vendor decides it is the appropriate time to pop-up the user's wallet.

Besides being client-driven, our digital wallet architecture supports the notion of a session while the JECF does not. In our model, once a client decides to initiate a session with a vendor, state information may be accumulated throughout the session, and multiple operations may take place during a single session. After initial instrument and protocol negotiation, the negotiated selections are retained as part of the state information in the session, and making additional payments thereafter does not require re-negotiation for each payment operation between the wallet and vendor. This mechanism allows us to implement lightweight instruments and protocols to execute micro-payments. However, in the JECF model, a separate JCM from the vendor is required to execute each commerce transaction, and all of the arbitration, negotiation, and selection may need to be done for each JCM. This can make the execution of micro-payments more costly in the JECF model.

Microsoft Wallet. [6] The Microsoft Wallet is composed of two Active/X controls: an Address Selector control, and a Payment Selector control. This model is also not purely client-driven, since vendors embed these controls on web pages on their web site to prompt the user to make a payment. Furthermore, the selection of the protocol is not client-driven. Within an HTML tag passed as a parameter to the Payment Selector control is an "accepted types" string which contains a list of (instrument class,protocol) pairs that are acceptable to the vendor. Upon choosing an instrument class, the accepted types are scanned from left to right searching for the first occurrence of the selected instrument class, and the corresponding protocol is chosen. Since the vendor orders the accepted types string, the vendor has the ability to choose a protocol that is advantageous to itself and disadvantageous to the client. For example, the vendor may choose a protocol that may lower its transaction cost, but that may take a longer time to execute over the network, costing the client more in network access charges. In our architecture, the Protocol Negotiation Protocol objects running on both the wallet and the vendor negotiate on the protocol to be used for a selected instrument class, and the client is not locked into using the fixed algorithm hard-coded within the Payment Selector control.

IBM Generic Payment Service. [15] IBM Zurich Research proposes a Generic Payment Service as part of the SEMPER (Secure Electronic Marketplace for Europe) project. The service gives the user the ability to have multiple "purses," each representing a different payment system. The concept of a "purse" in the contexts of the Generic Payment Service roughly corresponds to a combination of an instrument instance and an associated protocol.

Shopping Models. [10] The Shopping Models Architecture formalizes many different customer, merchant, and payment service interactions in terms of order, payment, and delivery event handlers. Our wallet architecture addresses payment in the context of Shopping Models. The wallet and vendor controllers, for example, expand on the interfaces that the `CustomerPayment` and `MerchantPayment` event handlers present in the context of that work.

U-PAI (Universal Payment Application Interface). [11] U-PAI proposes a standard interface to multiple payment mechanisms. A U-PAI `AccountHandle` fits into our architecture as a combination of an `Instrument` and a `Protocol` object since the `AccountHandle`'s interface provides methods to access instrument data, such as an account balance, as well as methods to execute a payment such as `startTransfer`.

JEPI (Joint Electronic Payments Initiative). [12] JEPI was a joint initiative between the W3 Consortium and CommerceNet whose goal was to develop a payment selection protocol as an extension to HTTP. JEPI's UPP [13] protocol is an example of a Protocol Negotiation Protocol in our architecture. Assuming a Communication Manager that is capable of sending HTTP messages (with the appropriate PEP extensions), UPP may be implemented within our architecture as a Protocol Negotiation Protocol.

NetBill. [14] In the NetBill protocol, payment is guaranteed to happen atomically with the delivery of goods by involving a trusted third party. To implement the NetBill payment mechanism in our architecture, the end-user wallet, vendor, and trusted third party applications would execute a NetBill payment protocol that would interact with NetBill money instrument objects.

SET. [1] SET is a secure electronic transaction protocol developed jointly by Visa and Mastercard. As mentioned previously in examples throughout the paper, SET is a protocol object within our architecture that can be used to make payments, as well as execute other operations defined in the SET protocol, with Mastercard and VISA credit card instrument classes.

GEPS (Generic Electronic Payment Services). [7] In the context of GEPS, the Instrument Manager takes advantage of Capabilities Management (CM) and Method Negotiation (MN) services. The Protocol Manager takes advantage of Capabilities Management (CM), Method Negotiation (MN), and Transaction Management (TM) services. The User Profile Manager takes advantage of Preferences Management (PM) services.

7.0 Conclusion

We propose a new generalized digital wallet architecture that is extensible, symmetric, non-web-centric, and client-driven. This architecture not only is extensible enough to inter-operate with multiple instruments and protocols as some existing wallet architectures are, but also incorporates other desirable features of a digital wallet architecture in a comprehensive way. In particular, the SWAPEROO wallet architecture also

- Factors out symmetric infrastructure and interfaces that may be common among end-user, vendor, and bank wallets,
- Generalizes to operating environments beyond the WorldWide Web, such that digital wallets can be developed for "alternative" devices such as PDAs and smart cards without re-inventing a new design for the wallet, and
- Ensures that the client is the proactive party in the payment phase of the shopping interaction.

Our proposed generalized digital wallet architecture has been implemented in Java and C++. Using our implementation, we built a digital wallet application for the PalmPilot personal digital assistant, and a vendor application that interfaces with a vending machine; the details of that particular implementation using the SWAPEROO wallet architecture is described in [17]. Complete Java and C++ APIs for the architecture are available at `http://www-db.stanford.edu/~daswani/wallets/`.

Acknowledgements

We would like to thank Craig Mudge for participating in several discussions with us which helped contribute to ideas in this paper. We also wish to thank the anonymous referees for their helpful comments which aided in various revisions of this paper.

References

[1] SET Secure Electronic Transaction (TM) LLC. SETCo website: http://www.setco.org/.

[2] CyberCash. CyberCash Home Page. CyberCash website: http://www.cybercash.com/.

[3] DigiCash. DigiCash: Solutions for Security and Privacy. DigiCash website: http://www.digicash.com/.

[4] Digital Equipment Corporation. MilliCent. MilliCent website: http://www.millicent.digital.com/.

[5] Sun Microsystems. Java Commerce Home Page. JavaSoft website: http://java.sun.com/commerce/.

[6] Microsoft Corporation. Microsoft Wallet. Microsoft wallet website: http://www.microsoft.com/wallet/.

[7] Alireza Bahreman. Generic Electronic Payment Services. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, 1996.

[8] Alireza Bahreman and Rajkumar Narayanaswamy. Payment Method Negotiation Service. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, 1996.

[9] Java Commerce Messages White Paper. Sun Microsystems website: http://java.sun.com/products/commerce/docs/whitepapers/jcm_whitepaper/jcm_whitepaper.html.

[10] Steven P. Ketchpel, Hector Garcia-Molina, and Andreas Paepcke. Shopping Models: A Flexible Architecture for Information Commerce. In *Proceedings of the Fourth Annual Conference on the Theory and Practice of Digital Libraries*, 1997. At http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1996-0052.

[11] Steven Ketchpel, Hector Garcia-Molina, Andreas Paepcke, Scott Hassan, and Steve Cousins. UPAI: A Universal Payment Application Interface. In *USENIX 2nd Electronic Commerce workshop*, 1996.

[12] W3C Joint Electronic Payments Initiative (JEPI). W3C website: http://www.w3.org/ECommerce/Overview-JEPI.html.

[13] D. Eastlake. Universal Payment Preamble Specification. W3C website: http://www.w3.org/ECommerce/specs/upp.txt.

[14] B. Cox, D. Tygar, and M. Sirbu. NetBill Security and Transaction Protocol. In *First USENIX Workshop of Electronic Commerce Proceedings*, 1995.

[15] J.L. Abad-Peiro, N. Asokan, M. Steiner, M. Waidner. Designing a Generic Payment Service. *IBM Systems Journal Vol. 37 No. 1*, 1998.

[16] T. Goldstein. The Gateway Security Model in the Java Electronic Commerce Framework. In *Proceedings of the Financial Cryptography First International Conference, FC '97, 1997.*

[17] N. Daswani, D. Boneh. Experimenting with Electronic Commerce on the PalmPilot. [*preprint*]