Designing a Generic Payment Service*

J. L. Abad-Peiro† , N. Asokan‡ , M. Steiner, M. Waidner
IBM Research Division
Zürich Research Laboratory, CH-8803 Rüschlikon, Switzerland
{jla,aso,sti,wmi}@zurich.ibm.com

11/26/96

ABSTRACT

The growing importance of electronic commerce has resulted in the birth of a plethora of different and incompatible payment systems. For business application writers this implies the need to understand the details of different mechanisms, to adapt the code as soon new payment schemes arrive and also to provide a way of finding a common mechanism for every transaction. In our work, we unify the different mechanisms in a common framework with APIs. Our framework provides services for transparent negotiation and selection of payment instruments as well. This allows applications to be developed independent of specific payment systems with the additional benefit of providing a central point of control for payment information and policies.

Keywords

Electronic commerce, payment systems, open networks, inter-operability, secure transactions, security, privacy.

1. Introduction

1.1 Background

Ever since money was invented (around 1700 B.C) as an abstract way of representing value, systems for making payments have been in place. In course of time, new and increasingly abstract representations of value were introduced. A corresponding progression of value transfer systems starting from barter, through bank notes, payment orders, cheques, and later credit-cards has finally culminated in "electronic" payment systems. Mapping between these abstract payments and the transfer of "real value" is still guaranteed by banks through the financial clearing systems. The financial

* This work was partially supported by the Swiss Federal Department for Education and Science in the context of the ACTS Project AC026, *SEMPER*; however, it represents the view of the authors. *SEMPER* is part of the *Advanced Communication Technologies and Services* (ACTS) research program established by the *European Commission Directorate General XIII*. For more information on *SEMPER,* see *http://www.semper.org/.*

† Also affiliated to the Swiss Ferderal Institute of Technology (ETH), Zürich.

‡ Also affiliated to the University of Waterloo, Canada.

clearing systems are built on closed, strictly controlled networks of financial institutions which are hence considered comparatively more secure than open networks.

Recently, there has been a great deal of interest in facilitating commercial transactions over open computer networks, such as the Internet. Several electronic payment systems have been proposed and implemented in the past few years [AJSW 96]. The current state is that many different, incompatible Internet payment systems compete with each other. Most shops accept, at best, only a subset of them.

1.2 Motivation

Consider a *business application* which implements some business scenario such as on-line purchase of goods. Ideally, the application should be able to make use of any available means of payment. Currently, the developer of such an application has to worry about:

- making sure that the application knows how to use all the various different payment systems its users are likely to have available, and
- in case multiple payment instruments are available, providing a way to choose one of them.

Our primary motivation was to develop a framework which frees the business application from addressing these issues.

We describe the design and implementation of a generic payment service which provides such a framework for enabling applications to use different payment systems in a transparent manner. Earlier reports on this work appeared in a public project reports of SEMPER [AAW 96,D03 96] The primary component of this generic service is a coherent hierarchy of application program interfaces (APIs) for the transfer of monetary value. The flow of information during transactions determines a classification of actual payment systems into a set of *payment models.* The API hierarchy reflects this separation into different payment models: it consists of a root API common to all payment models and extensions specific to each model.

In addition to the unified interfaces, the generic service has the following features and facilities:

- mechanisms for automatic selection of the specific payment instrument to be used in a transaction; this will enable the applications to be concerned just with the questions “how much to pay?” and “to whom?” but not with “using what payment instrument?”
- information, management, and control services to enable the development of applications that use the generic payment service (e.g. inquiring what payment systems are available), and
- tools and framework necessary for the incorporation of actual payment systems into the generic payment service;

This work is being carried out as part of the SEMPER project aimed at building a secure electronic marketplace [Waid 96]. However, the architecture of the generic payment service is independent of the specific environment of SEMPER: a stand-alone implementation of the service is possible.

In summary, the motivation for a generic payment service is twofold:

- with a generic payment API and mechanisms for transparent negotiation and selection of payment instrument, applications can be developed independent of specific payment systems. This interface constitutes a central point of control of payment information and policies,
- the process of defining a generic payment service goes hand in hand with the development of a formal definition of a secure payment system and the properties it should possess. Such a formal definition will be a useful framework for verification and comparison of security properties of payment systems. We have begun additional work in this direction [PfWa96].

2. Models of Electronic Payment Systems

There are several different electronic payment systems. All of them have the same basic purpose of facilitating the transfer of value among different parties. They differ in various aspects such as the point in which an electronic transaction is linked to the movement of real monetary value in the financial clearing system, and the degree of security provided by the system. First, we present a classification of different models of electronic payment systems [AJSW 96] on which we base the design of the generic payment service.

2.1 Players

Electronic payments involves a **payer** and a **payee**. In several payment protocols, the "payer" and the "payee" are referred to by more specific terms like "customer" or "buyer" and "merchant" or "seller," respectively. We prefer the terms "payer" and "payee" since on the payment system level there is no reason to limit the application to just purchases. The intent of the payment is to transfer monetary value from the payer to the payee. This is accomplished by a payment protocol which results in the exchange of data. The process also requires at least one financial institution which links the exchanged data to transfers of economic value.

The financial institution may be a bank which deals with economic value represented in terms of "real money;" or it may be some organisation that issues and controls other forms of representation (e.g. frequent flyer miles). In this paper, we use the term **bank** to mean all different types of financial institutions and the term "real money" to cover all forms of value representations used by financial institutions. Typically, banks participate in payment protocols in two roles: as an **issuer** (interacting with the payer) and as an **acquirer** (interacting with the payee). Finally, an **arbiter** may be involved in resolving disputes in the payment system. The basic set of players involved in a payment system is illustrated in Figure 1. In most systems, the presence of the arbiter is not explicit: even if the necessary proofs of evidence are produced, dispute handling is done outside the payment protocol and often not even specified. Sometimes, it is not even possible to define dispute handling at the protocol level since the resolution of disputes may be subject to policy decisions of the users and financial institutions (a full-fledged payment system built on top of a given payment protocol should however provide appropriate dispute management services).

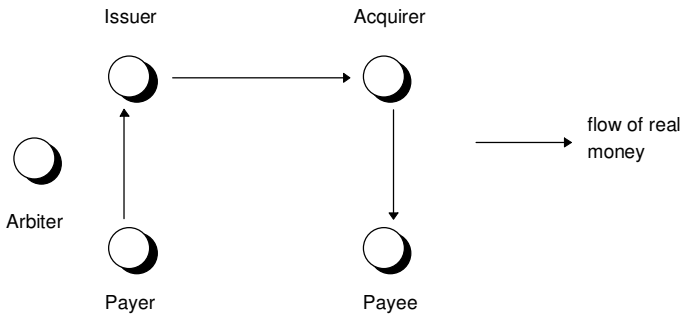

Figure 1 **Players of a Payment System**

Certain payment systems might involve more players, e.g.,

- registration and certification authorities, or
- other trusted third parties that enforce receipts for payments, clearings, and money transfers in general.

2.2 Payment Models

We classify payment systems according to the necessary flows of information between the players. Figure 2 lists, without claiming completeness, the four most common payment models and their information flows.

One criteria for distinction is whether there is a **direct** communication between the payer and the payee or whether the communication is only **indirect.** In the latter case, the payment operation is initiated by one party and involves only the initiator and the bank(s). The other party is notified by its bank at the completion of the transaction. An example of the former is paying by cash or cheque. An example of the latter is paying by means of a standing order or wire transfer.

Most currently proposed Internet payment systems implement direct payments. Consequently, in this paper we focus only on these systems. However, the framework presented can be easily adapted to indirect payment scenarios.

According to the relationship between the time the payment initiator considers the payment as finished, and the time the value is actually taken from the payer, one can distinguish between:

- pre-paid (or "cash-like") payment systems, see Figure 2a,
- pay-now payment systems, and
- pay-later payment systems, see Figure 2b.

The latter two are quite similar: in both cases, the user must have some sort of an "account" with the bank. Thus, we consider these two cases to belong to the same model, which we call the **account-based**[1] model. In the rest of this document, we will use the terms **cash-like** and **account-based** to refer to the two models.

In an **on-line payment system,** the payer and the payee are connected on-line with a third party (typically issuer or acquirer) while in an **off-line payment system** the

[1] Certain products of the cash-like model (e.g., Digicash' e-cash[Digi 94]) may also happen to have corresponding accounts. However, this is an implementation "add-on." A barebones cash-like system need not have a corresponding account in a bank.

payment is a transaction involving only payer and the payee. On-line payment systems are often considered more secure since the third party can ensure that the payer actually possesses the money they want to transfer to the payee. In most off-line electronic payment systems, double spending of money is prevented by tamper-resistant hardware (smartcards or "electronic wallets"). As a second line of defence, some systems ensure that double spending would be provable, subject to the usual cryptographic assumptions, in case tamper-resistance is broken [BBCM 94].

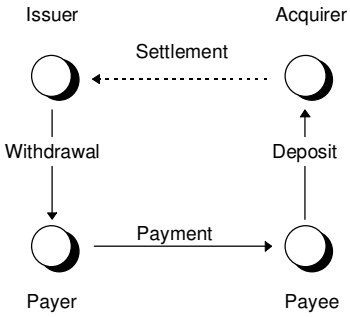

Figure 2a **Direct cash-like system**

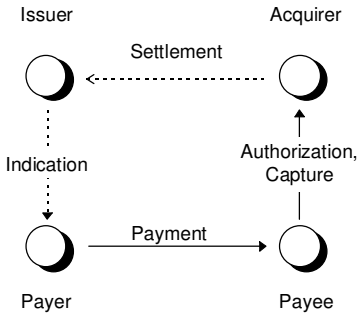

Figure 2b **Direct account-based system**

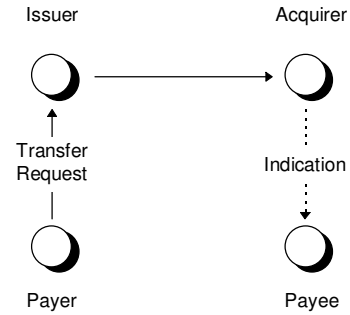

Figure 2c **Indirect push system**

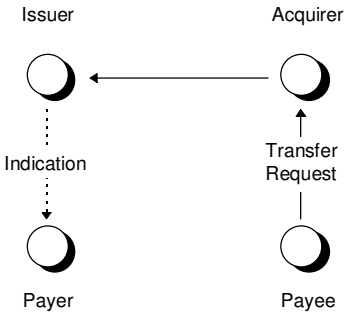

Figure 2d **Indirect pull system**

Figure 2 **Payment Models** (dotted lines indicate flows that may be external to the digital system)

3. The Generic Payment Service

3.1 Scope and Terminology

The main functionality of any payment system is to provide **value transfer services**consisting of the following:

- Moving electronic value from a payer to a payee. The payer might request a receipt from the payee or from the bank, depending on the selected payment model and level of security.
- Moving electronic value back from payee to payer in a payment reversal.
- Converting "real money" into electronic value ("loading") or vice versa ("deposit"); the former is relevant only in the cash-like model.
- Converting one type of electronic value to another: for example, currency exchange. Note that currency exchange is not a unilateral process but should be carried out according to the terms of another party, such as a bank. This process typically involves running a protocol with that party.

In the simplest case, the transfer of value happens between two end points. We call such an end point a **purse**. A purse corresponds to a single instance of a specific **payment system** and contains all the user information related to that instance. For example, a user who has a credit card account, an instance of a stored-value card, a *DigiCash* e-cash account will have three separate purses associated with each of the above. **Purse management services** allow a user to set-up, configure, manage, and delete purses. Some purse-related services might require authorisation (e.g., insert a smartcard and/or enter a passphrase for each withdrawal) while others may not have any access control.

Each value transfer in progress is embodied in a separate **transaction**. A purse may be involved in several concurrent transactions. **Transaction management services** allow transactions to be queried for their status, cancelled, or recovered from a crash. Before beginning a transaction, each party must choose a suitable purse. This selection may have two parts: a *local* decision based on preferences and requirements and a mutual decision based on *negotiations*. The services that enable this decision making a collectively known as **purse selection services**.

In addition to purses and transactions, we use a separate entity called the **payment manager** to manage the overall operation of the generic payment service. Each player will have one active payment manager mastering an instance of the generic payment service. **Information services** permit the retrieval of information on the state of the payment manager, or a specific purse: for example, a list of previous transactions, the last statement of account received, or statistics on all payments received and made in a certain period of time.

Finally, **dispute management services** allow the user to retrieve or verify proofs of evidence (like signed receipts or statements of account) that may need to be shown to a third party in the event of a dispute.

3.2 Design

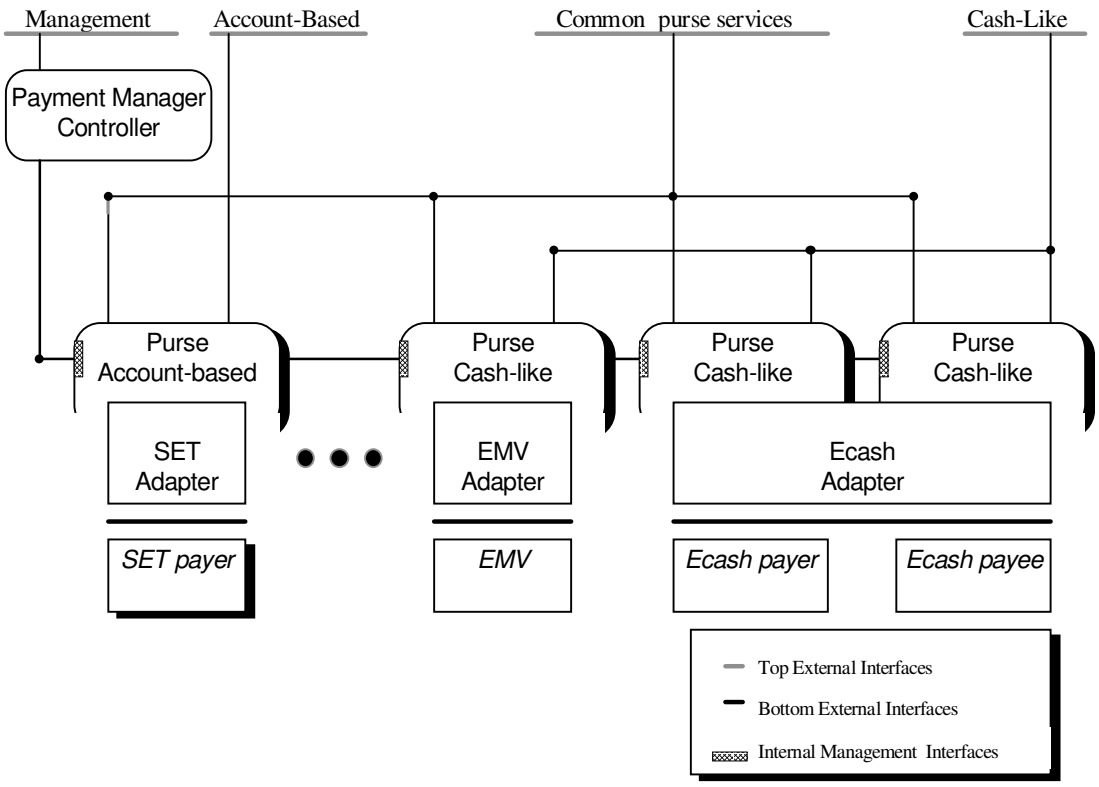

Figure 3 **Generic Payment Service - Architecture**

We describe the value transfer services in terms of a hierarchy of interfaces. The root interface, called `PurseServices`, defines common payment services expected of a purse of any payment system. Sub-interfaces define additional services provided by different payment models. A sub-interface corresponding to each model (e.g. `AccountBasedPurseServices` for the account-based model) extends the root `PurseServices` interface. Additional models can be included later. To incorporate a specific payment instrument into the generic payment service, a system-specific *adapter* mapping the services of the payment system to the interface of one of the defined payment models needs to be implemented. Figure 3 depicts the architecture of the generic payment service.

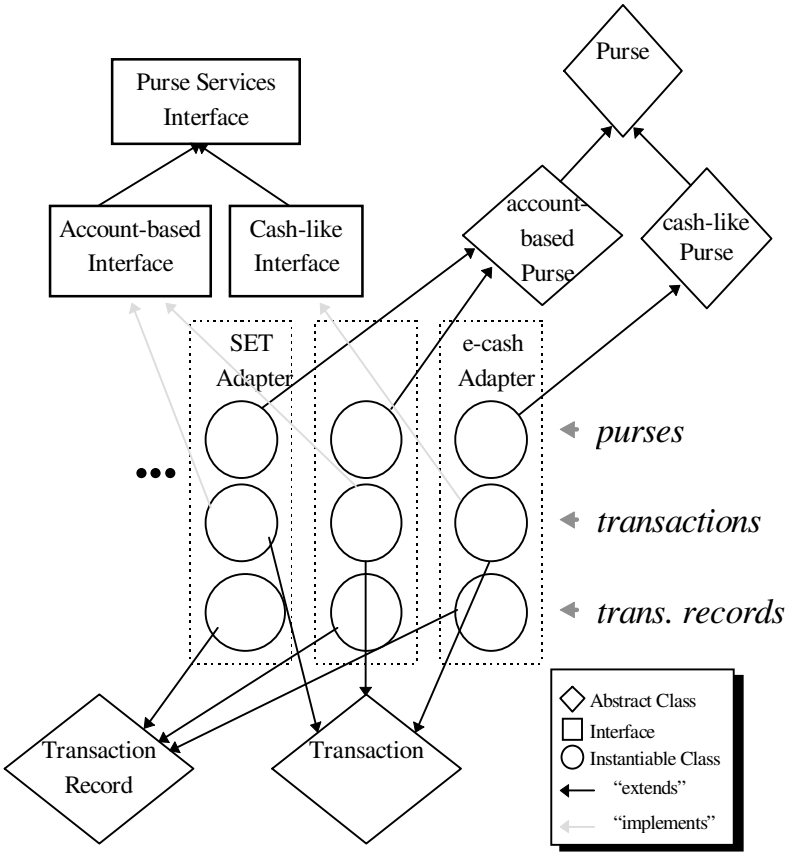

Figure 4 **Generic Payment Service — Classes and Interfaces**

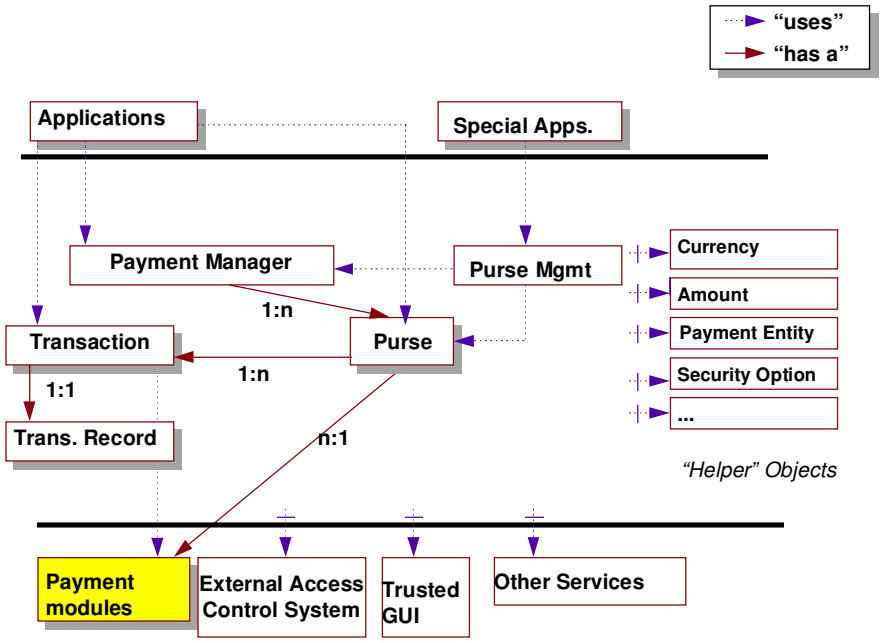

Figure 5 **Generic Payment Service in Action — Objects in a Typical Instance**

The high-level design described so far can be implemented in a variety of ways. We opted for an object-oriented approach, as shown in Figure 4. This allows us to describe objects and services required in the generic payment service in terms of root classes and interfaces. Adapters for specific payment systems can then provide implementations for the various interface methods and, where necessary, extend (or "subclass") the root classes. The `Transaction` objects are expected to implement the services defined in the `PurseServices` interface. `Transaction` objects are transient. Each `Transaction` object is associated with a longer-lived `TransactionRecord` object where the information about the transaction is maintained. The users of the generic payment service (e.g., business applications) can treat the various objects (such as purses and transactions) as instantiations of the generic root classes. Objects involved in a typical instance of the generic payment service is shown in Figure 5. In the following sections, we describe these objects in more detail.

3.2.1 Purses

A purse is an abstraction of an instance of a payment system that is available to the user. A purse should provide services

- for creating a purse (i.e. a constructor to instantiate a purse object),
- for configuration and set-up, which will be used by purse management applications (e.g. to associate a purse with a credit card and to register with a SET CA),
- for initialisation, using which it could be activated during start-up,
- for creating transactions (see Section 3.2.2), and
- for information services (identified in the management interface — e.g. answers to questions like "does this purse provide non-repudiable receipts for payments?").

We define a `Purse` class hierarchy. The root `Purse` class provides default implementations of some of these services. For each payment model, we extend the root `Purse` class if necessary (e.g. to `AccountBasedPurse`). Adapter-writers shall extend a high-level `Purse` class and override/extend default implementations as necessary (e.g. `SETPurse` extends `AccountBasedPurse`).

Additionally, the `Purse` class hierarchy also provides services for information management corresponding to these purses (e.g. answers to questions like "what is the user name associated with this purse?" or "what amount is associated with this purse (where applicable)?")

A further classification of purses can be made based on the subset of operations supported by the purse[2]:

- **pay-only purse** for a purse that can be used to make but not to receive payments,
- **receive-only purse** for a purse that can be used to receive but not to make payments, and
- **pay-and-receive purse** or just **purse** for a purse that can be used for making and receiving payments.

[2] Sometimes different names with the same meaning have been used elsewhere in the literature: e.g. a receive-only purse is sometimes referred to as a *till*. We feel that our nomenclature is less ambiguous.

Note that most current payment systems do not support pay-and-receive purses.

3.2.2 Transactions and Transaction Records

As mentioned, the root `PurseServices` interface defines value transfer services that are common to all payment models. Some example services defined in this interface are:

- `pay` makes a payment from a purse to a designated recipient,
- `receivePayment` is the counterpart of `pay`; it receives an incoming payment.

Model-specific sub-interfaces may define additional services. For example, the sub-interface for the cash-like model has a service to `withdraw` money from the bank into the purse.

Every instance of a value transfer service is abstracted by a transaction. The `PaymentTransaction` class implements the value transfer services described in one branch of the `PurseServices` interface hierarchy. Information associated with a transaction (both transient information such as state keeping that is relevant only while the transaction is active and "permanent" information such as receipts or other evidence that is relevant long after the transaction is completed) is kept in a related `PaymentTransactionRecord` object. This can be used in crash recovery and dispute management as well as for informational purposes.

The root `PaymentTransaction` defines general transaction services such as trying to abort an on-going transaction or retrieving its current status. Each sub-class of the root class implements (e.g. `SETTransaction` extends `PaymentTransaction`). Each leaf `Purse` class provides a `startTransaction()` method which creates a new transaction of the appropriate type (e.g. in the `SETPurse` class, the `startTransaction()` method will instantiate a `SETTransaction` object).

3.2.3 Payment Manager

The payment manager provides the following services:
- Management and information
- Purse selection

3.2.3.1 Management and Information Services

The payment manager keeps track of:

- currently active transactions,
- links to stored information about past transactions,
- currently available purses,
- known payment module adapters, etc.

To maintain and manage this information, the payment manager provides various services such as creation and registration of a purse, deletion of a purse, registration of a payment module adapter, registration and de-registration of active transactions. Additional services are provided to make this information available to other objects and applications in a variety of useful ways.

The manager is also responsible for initialising all the relevant components on start-up. Eventually, it will provide services for shut-down and fault tolerance mechanisms such as crash recovery.

3.2.3.2 Purse selection

Selection of a purse to be used in a transaction is based on several factors:

- requirements for the transaction (e.g. security requirements),
- static user preferences,
- negotiation with a peer payment manager, and
- manual selection by user.

Except the third, the remaining factors are all local. The payment manager provides various services to facilitate this local selection.

Negotiation with the peer for selection of the payment instrument can be done in several ways. But all negotiation protocols consist of simple request-response exchanges. Currently, we restrict negotiation for tuples containing two parameters:

- *Payment System Name*: We define "payment system name" as follows: two purses that report the same payment system name can potentially engage in a payment transaction between themselves. Typically, the payment system name corresponds to a single (protocol, brandname) pair; e.g. SET:MasterCard and SET:Visa will be two different payment systems. It is up to the adapter to determine the payment system name associated with a purse as long as it satisfies the definition above.
- *Amount (value and currency)*

Negotiation may take one of the following forms:

1. The initiator proposes a list of payment means and associated amounts; the responder picks a suitable subset. Figure 6 illustrates this type of negotiation. The figure shows both the interface events (`requestPSChoice` and `choosePS`) together with the protocol flows.
2. The initiator selects a single purse and asks the responder whether the corresponding payment system is acceptable; if it is, the responder selects a suitable purse for the subsequent transaction and sends a confirmation to the initiator. Otherwise, the responder sends an error message to the initiator and throws an exception to its invoker. This is illustrated in Figure 7.

The first form is for the negotiation of payment systems whereas the second form is a hand-shake process. Both are optional — if all the information about the payment endpoints are known *a priori*, then no negotiation is necessary. If the first form is used, the handshake will almost always be necessary.

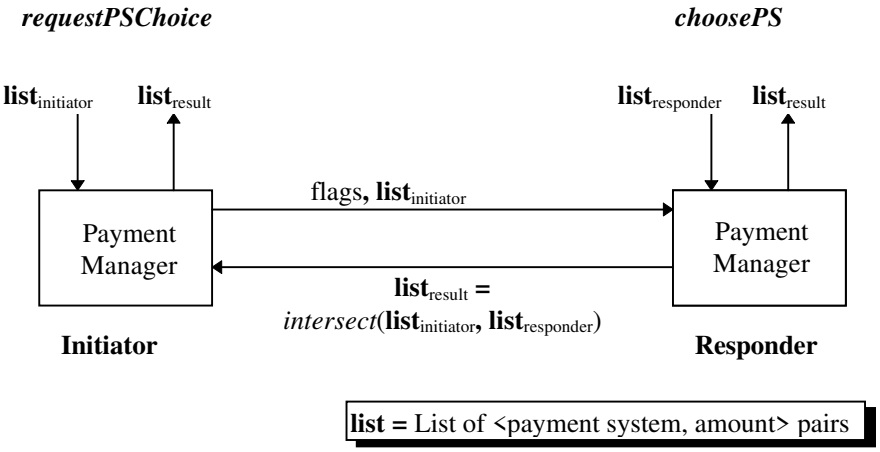

Figure 6 **Negotiation protocol for selection**

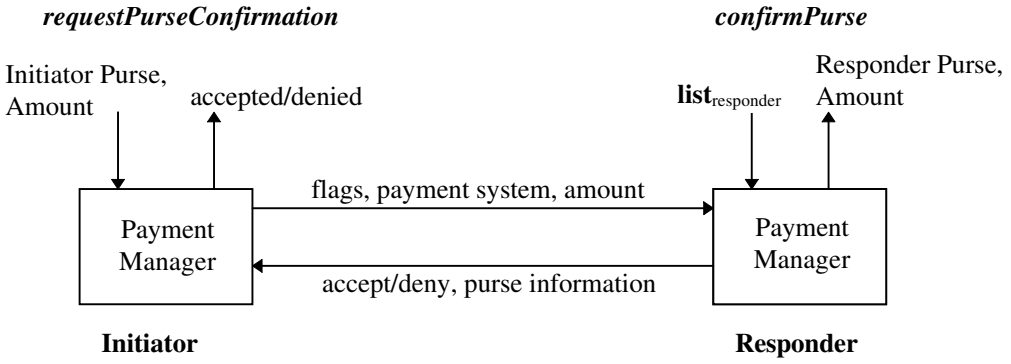

Figure 7 **Negotiation protocol for confirmation**

Flags in the initiator's request messages indicate the nature of the selection sought from the responder. For example, the responder may be allowed to suggest a revision to the proposed amount; this is useful when the price varies depending on the payment method.

Using these services, various negotiation policies can be implemented. Methods implementing purse selection based on a default policy is provided as an example in the form of the `selectPayingPurse` and `selectReceivingPurse` methods for purse selection.

3.2.4 Helper Objects

In addition to the entities described in the previous sections, there are a number of "helper" objects used in the generic payment service. Figure 5 shows some of them which appear at the interface level:

- `Currency` objects represent the type of electronic value. It can be a real currency (identified by a three letter ISO code) or "funny money" currency such as frequent flyer miles,
- `Amount` objects consist of a currency and value,

- `PaymentEntity` objects are used to represent the identity of a player in the payment model. Currently, it consists of a "name" and a communication address,
- `SecurityOption` objects denote the type of security services required for a transaction. Typically, value transfer services take a list of `SecurityOption` objects as one parameter,

3.3 Extending the Design

3.3.1 Dispute Management

Support for handling disputes is a crucial aspect of any system providing accountability. To motivate our design for the dispute management component, consider some typical claims that users of a generic service might want to make or deny:

- ABC paid $100 to XYZ on 10/29/96, 10:32 hours
- ABC paid $100 to XYZ with external reference "Order #432"
- ABC did **not** pay $100 to XYZ
- ABC deposited $100 at bank B on 10/29/96
- ABC deposited $100 at bank B on 10/29/96 before noon

A transaction may result in several items of evidence. Only a subset of these may be relevant to a particular dispute. Hence, it is useful to have a way of indicating the nature of the dispute to the underlying payment system so that it can produce the minimal amount of evidence relevant to the dispute. In general, disputes can be expressed in terms of *statements* that a party wants to prove or deny. We use the following structure for dispute statements:

[**not**] **service**=<service> [<*role*>=<party>]* [<*attr*> <*OP*> <value>]*

The services provided by the generic payment service, and hence the values of the attribute **service**, are well defined. Each method in the `PurseServices` interface hierarchy has a well-defined, finite set of services that it is associated with. For example,.

- `pay()` → *payment*
- `receivePayment()` → *payment*, *authorisation, capture*
- `capture()` → *capture*
- `deposit()` → deposit

Each service should have

1) a finite, well-defined set of *roles* associated with it. For example,
   - *payment* → *payer*, *payee*, *bank*
   - *deposit* → *depositor*, *bank*
2) a finite, well-defined set of *attributes* relevant for disputes. The definition of each attribute should include what operations can be performed on its values.. For example,
   - *payment* → *amount*, *time*, *place*

The identity of the party playing each role and the values for various relevant attributes can be inferred from either input/output parameters of the associated methods or the

environment in which the methods are invoked. The environment consists of the state of the purses involved and external factors such as time and place.

Using this representation, the examples at the beginning of this section can be expressed as:

- **service**=*payment*, *payer*=ABC, *payee*=XYZ, *amount*=$100, *time*="10/29/96 10:32"
- **service**=*payment*, payer=ABC, *payee*=XYZ, *amount*=$100, *ext−ref*="Order #432"
- **not service**=*payment*, payer=ABC, *payee*=XYZ, *amount*=$100
- **service**=*deposit*, *depositor*=ABC, *bank*=B, *amount*=$100, time="10/29/96"
- **service**=*deposit*, *depositor*=ABC, *bank*=B, *amount*=$100, time<10/29/96 noon"

The dispute management interface provides services to extract evidence to support statements. The evidence may require an interactive proof. In this case, we require a token-based version (see Section 3.6) of the evidence extraction service. In addition, it also provides a service to verify that a given set of evidence proves the statement associated with them. Sub-classes of the `Purse` class are required to implement the dispute management interface.

Every user will have a special dispute management application. To initiate a dispute, she will construct the dispute statement with the help of this application and send it to an appropriate verifier. The verifier decides then which involved party has to provide evidence (e.g. if the dispute statement is of the form 'not service=payment...' then normally the payee or the bank would have to prove the statement 'service=payment..' to win the dispute). The verifier can be either another party in the alleged transaction or an arbiter. Typically, one handles a dispute by first trying to resolve it among the parties involved and proceeding to involve the arbiter if this resolution did not succeed.

3.3.2 Payment Security Policies

3.3.2.1 Limits on Value Transfer

A user of the generic payment service may wish to associate several types of limits to the purses available. Some examples for the types of limits are:

- Each payment from a specified purse (say **P1**) should not exceed 100 CHF,
- Total payments from all purses taken together should not exceed 1000 CHF in any 24-hour period,
- Total payments from all purses should not exceed 10,000 CHF in a given calendar month,
- Payments below 10 CHF do not need explicit user authorisation.,
- No more than 4 payments without explicit user authorisation in any 24-hour period,
- If a payment will bring the balance in a specified purse (say **P2**) below 200 CHF, it must be explicitly authorised by the user.

Clearly, the limits may involve complex computations and may require several different pieces of information during the computation.

3.3.2.2 Access Control

Access control is a critical functionality of the generic payment service. We note the following in order to motivate our design:

1) Access control is required in the following cases:
   - Access to secret information required to use the underlying payment system (e.g., PINs, passphrases, credit card numbers, etc.). There may be several different pieces of such information.
   - Access to purse operations.

2) Even for the same purse operation, it may be necessary to control access differently depending on the parameters. For example, a user may decide to have no access control for payments of small amounts or have a different passphrase to authorise high value payments. The underlying payment instrument may or may not support such granularity.

3.3.2.3 A Common Solution: Policy Objects

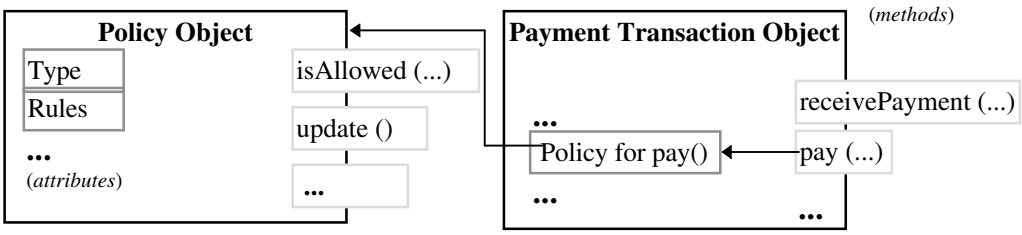

Figure 8 **Incorporating policy in a Purse**

We have taken a common approach to address both limits and access control requirements by using the notion of `Policy` objects. A purse can associate one policy object with each service it provides. Whenever a service is requested from a purse, the corresponding policy object will be queried to determine authorisation for the service (Figure 8). All policy objects provide ways

- to check for current availability of a service (the `isAllowed()` method), and
- to indicate that an authorised service is being provided (the `update()` method) so the `Policy` object can change any relevant internal state parameters (note that policies as in the second last example in Section 3.3.2.1 are stateful).

These methods can be used by purse services (such as `pay()` and `receivePayment()` methods) to manage authorisation. A reference to the transaction record is provided as argument to the `isAllowed()` and `update()` methods. Through the transaction record, it is possible to access the purse(s) involved in an operation. Thus, different implementations of these methods can access all the information they need in order to make the policy decisions.

The `Policy` class hierarchy is shown in Figure 9. A policy may be *simple* or *aggregate*. Simple policy objects are self-contained and make their decisions independently of other policy objects. There may be several kinds of simple policy objects. Some examples are:

- `AskUser` policy class displays relevant information about the transaction to the user and asks for his approval,

- `MinBalance` policy class makes sure that the minimum balance is above a specified value, and
- `TimebasedLimit` policy class provides a way to set simple time-based limits.

Aggregate policy objects have a list of constituent policy objects. The policy decision of the aggregate object is a function of the policy decisions of its constituent objects. Some examples are:

- OR policy allows the service if *any* of its constituent policies do so,
- AND policy allows the service if *all* of its constituent policies do so.

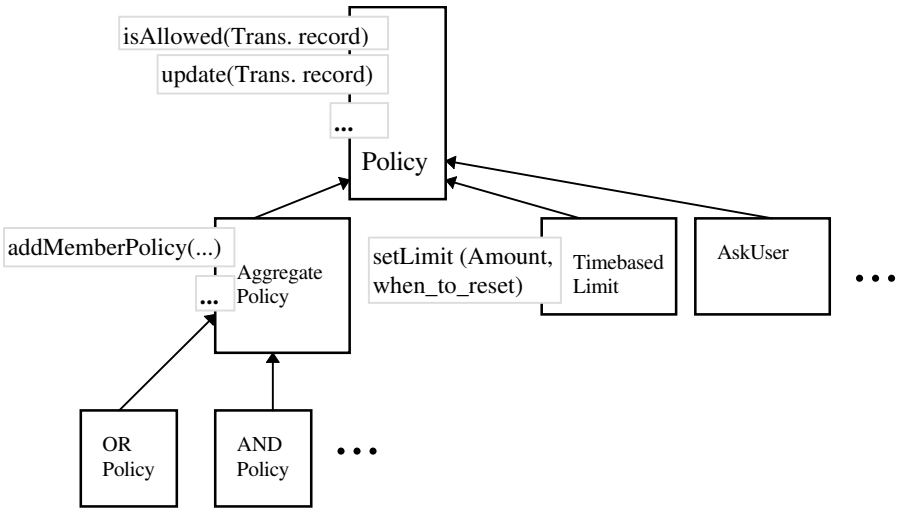

Figure 9 **Class Hierarchy for Policy Objects**

With these we can express complex policies. For example the policy "if the amount is less than CHF 10 and the balance afterwards is going to be above CHF 200, allow the payment, otherwise ask the user" will correspond to a policy object network

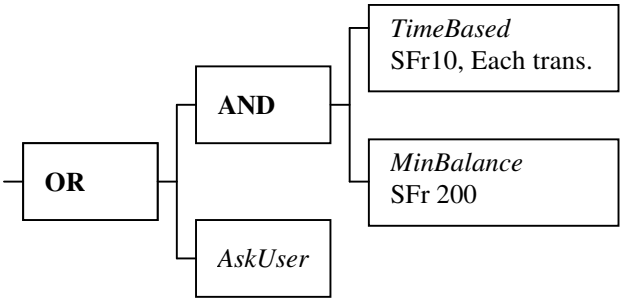

Figure 10 **An Example Policy**

Additional policy classes may be defined and incorporated into this hierarchy. Users of policies (e.g. the `pay()` and `receivePayment()` methods) will have a single access point. Notice that policy objects are intended as a mechanism to *express* policies. The enforcement of these policies is up to the implementations of the services: for example, as shown in Figure 8, the `pay` method in the transaction class of an adapter must query the payment policy object in its purse before proceeding with the payment.

3.4 Adapting a Payment System

In order to incorporate a new payment system into the generic payment service, a suitable adapter has to be designed (Figure 4 indicates what constitutes an adapter). The following steps are required in this process:

- Identify the model to which the payment system belongs (e.g. SET [SET 96] belongs to the account-based model).
- Implement a sub-class of the `Purse` class corresponding to the payment model identified (e.g. `SETPurse` extends `AccountBasedPurse`). This implies providing implementations for all abstract services defined in the top-level Purse classes (e.g. `Purse` and `AccountBasedPurse`) and overriding default implementations therein, where necessary. In particular, the new class must provide a proper implementation of the `setup()` method: this method should allow the user to carry out all configuration necessary for the payment system.
- Implement a sub-class of `PaymentTransaction` class which implements the value transfer services defined in the leaf of the `PurseServices` interface hierarchy corresponding to the payment model identified (e.g. `SETTransaction` implements `AccountBasedPurseServices` and inherits from `Payment-Transaction`). This sub-class can also implement additional optional value transfer service interfaces as `TAccountBasedPurseServices` (see Section 3.6) and `MicropaymentServices` for repeated low-value payments (e.g. SET enhanced with a mechanism like μ-*i*KP [HSW 96]).

In addition, if any special action needs to be taken during the installation of the adapter, a suitable installation application must be provided. A standard installation application is available as part of the generic payment service. It performs two actions:

- Install the adapter module in the correct location,
- register the name of the new purse class with the payment manager.

3.5 Using the Generic Payment Service

Assuming that a user has installed one or more payment instruments along with their adapters on her system, there are two kinds of usage:

3.5.1 Purse Management

Before being able to use an installed payment instrument, a purse corresponding to it must be created and configured. A special "purse management application" is provided for this purpose. Changes to purses are written out to stable storage. Purse management is an infrequent activity — typically, once a purse is created and configured, it can be used in several subsequent payment transactions.

3.5.2 Payment Transactions

A user will initiate payment transactions using some sort of a high-level application (e.g. a web-browser or a CD-catalogue reader). As an example scenario, consider the case in Table 1. It only shows the events that take place on the payer side. The important things to note are:

- The user need not specify the payment instrument to use if he does not want to; the payment service can be configured to prompt him for selection of payment instrument if it cannot do so by itself,
- The application is not aware of the specific payment instrument being used; it deals with a generic `Purse` and `PaymentTransaction` objects.

User	Application	Generic Payment Service
Click the "pay" button specifying an `amount` and identity of `payee`		

	`Purse chosen_purse`
	`= PaymentManager.selectPayingPurse`
	`(payee,amount,options,reference)`	

		Start negotiation with peer: - send a list of payment systems
		- receive a subset chosen by peer
		If it is not possible to make an automatic choice, present a list of purses to the user
Click on the purse of choice		
		- Ask the peer to confirm the payment system chosen. - return the chosen purse (e.g. `SETPurse`) to the application
	`PaymentTransaction tr =` `chosen_purse.startTransaction()`	
		Create a `SETTransaction` object
	`tr.pay` `(payee, amount, options, reference)`	
		`pay()` method in `SETTransaction` invokes appropriate native SET procedures to carry out the payment.

Table 1 **An example payment transaction on the payer side**

3.6 Support for an Asynchronous Model

The original design assumed a synchronous model since the first version of the SEMPER architecture did the same[D03 96]. However, we have defined a "token-based" interface which can support an asynchronous model in a straight-forward manner.

Our token-based interface is inspired by the GSS-API [GSS 96] approach. It has two types of methods:

- one "starter" method for each different type of protocol; the starter methods returns a token containing the first message of the protocol, and
- a common "processor" method; this takes a token as input and depending on the internal state of the protocol run, may return another token as output.

The payment service does not engage in any direct communication with the peer. Instead, the caller is expected to take care of the communication and can benefit from already established channels and has additionaly the choice of compensating missing security requirements not fulfilled by the payment mechanism (e.g. anonymous communication). The payment service is still responsible for maintaining the state of a protocol run. The initiating caller invokes an appropriate starter method in the payment service API to start a protocol. Typically, these starter methods will return a "token" as output. The initiating caller is expected to communicate this token to its peer entity, the responding caller. The latter in turn will invoke the processor method on its instance of the payment service and give the received token as input. From this point on, whenever a caller entity receives a token as output from the processor method, it will send the token to its peer; whenever a caller entity receives a token from its peer, it will invoke the processor method on its payment service giving the received token as input.

We define a token-based version of value transfer services in an interface hierarchy called `TPurseServices` parallel to the `PurseServices` interface hierarchy. For each method (e.g. `pay()`) in the latter, we define a corresponding starter method (e.g. `startPay()`) in the former. In addition, a common processor method `processToken()` is defined in the `TPurseServices` interface.

Currently, the interface `TPurseServices` is optionally implemented by sub-classes of the `PaymentTransaction` class. Since the token-based version is more general than the synchronous version, we plan to make the former the default purse services interface and deprecate the latter.

We are in the process of defining a token-based interface for the negotiation of payment system as well. The designers of the E-CO System have used a similar token-based API for the negotiation of payment system [Bahr 96].

4. Related Work

U-PAI [KGPH 96] being developed as part of the Stanford Digital Libraries project (URL: *http://www-diglib.stanford.edu*). Their focus is on providing a unified interface to payment services — they do not address negotiation for parameters before a payment transaction begins; nor do they explicitly address issues like refunds. They

also appear to assume a distributed object infrastructure such as CORBA and don't have a very clear security and trust model. The **Joint Electronic Payments Initiative** (JEPI) focuses only on defining the protocol for the negotiation of various payment related parameters such as the payment system. The scope of our work roughly corresponds to the scope of these two projects taken together.

Sun recently announced their **Java Electronic Commerce Framework** (JECF). The framework is still in the process of being defined. Their emphasis appears to be on the payer side: payers will be able to download different "payment cassettes" (roughly corresponds to a payment instrument and its adapter in our terminology) and integrate them into their JECF installation. They also propose a sophisticated general access-control scheme which can be used in our work.

The **E-CO System** project (URL: *http://eco.eit.com/*) has roughly the same scope [Bahr 96] as our work although their main focus so far seemed to be on establishing APIs and mechanisms for payment negotiation[BaNa 96]. Additional information is not public at the time of writing.

5. Status and Conclusions

We have presented the design of a generic payment service. A prototype of the generic payment service with all the basic functionality has been implemented as part of the SEMPER project and tested using a "dummy" payment system and an adopter for *i*KP [BGHH 95]. Adapters for SET [SET 96] and DigiCash e-cash [Digi 94] are under development by partners in the SEMPER consortium.

6. Acknowledgements

Interesting discussions with several people helped us develop and refine the ideas presented in this paper; in particular we like to thank Ali Bahreman, Mark Linehan, Birgit Pfitzmann, Tom Scanlan, John Schey, Berry Schoenmakers, Julia Sime, Els van Herreweghen, and John West. We are grateful to Matthias Schunter for his valuable comments on previous versions of this paper.

7. References

For a collection of WWW pointers on electronic commerce, see <http://www.zurich.ibm.com/Technology/Security/sirene/outsideworld/ecommerce.html>.

AAW 96 *J.L. Abad-Peiro, N. Asokan, M. Waidner:* **Payment Manager Overview**, SEMPER activity paper 212ZR054, March 1996. Available in: <http://www.semper.org/info/.>

AJSW 96 *N. Asokan, P. Janson, M. Steiner, M. Waidner.* **Electronic Payment Systems**, August 1996. In submission. An earlier version appeared in *Informatik - Zeitschrift der Schweizer Informatikorganisation,* June 19995. Available in: <http://www.zurich.ibm.ch/Technology/Security/publications/1996/ajsw96.ps.gz>

Bahr 96 *A. Bahreman:* **Generic Electronic Payment Services;** presented at the USENIX E-Commerce Workshop, Oakland CA., 1996

BaNa 96 *A. Bahreman, R. Narayanaswamy:* **Payment Method Negotiation Services;** presented at the USENIX E-Commerce Workshop, Oakland CA., 1996

BBCM 94 *J.-P. Boly, A. Bosselaers, R. Cramer, R. Michelsen, S. Mjølsnes, F. Muller, T. Pedersen, B. Pfitzmann, P. de Rooij, B. Schoenmakers, M. Schunter, L. Vallée, M. Waidner:* **The ESPRIT Project CAFE - High Security Digital Payment Systems;** ESORICS '94, LNCS 875, Springer-Verlag, Berlin 1994, 217-230.

BGHH 95 *M. Bellare, J. A. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, M. Waidner:* **iKP — A Family of Secure Electronic Payment Protocols;** USENIX E-Commerce Workshop, New York, 1995.

Digi 94 *DigiCash b.v.:* **DigiCash — Numbers that Are Money**; available at *http://www.digicash.com/publish/digibroc.html*, March 1994.

D03 96 *SEMPER Consortium.* Basic Services: Architecture and Design, **SEMPER Deliverable D03**, October 1996. Available in: <http://www.semper.org/deliver/index.html>

GSS 96 *J. Linn:* **Generic Security Service Application Program Interface, Version 2**. Internet-draft <draft-ietf-cat-gssv2-08.txt>, August 1996**.**

HSW 96 *R. Hauser, M. Steiner and M. Waidner:* **Micro-Payments based on *i*KP;** IBM Research Report RZ2791, February 1996.

KGPH 96 *S. Ketchpel et al:* **UPAI: A Universal Payment Application Interface v0.9,** to be presented at the Usenix Workshop in Electronic Commerce, 1996.

PfWa96 *B. Pfitzmann, M. Waidner:* **Properties of Payment Systems - General Definition Sketch and Classification**, IBM Research Report RZ 2823, May 1996.

SET 96 *SET Working Group:* **Secure Electronic Transactions;** Mastercard and Visa, August 1996, Draft. Available from <http://www.mastercard.com/set/set.htm>

Waid 96 *M. Waidner:* **Development of a Secure Electronic Marketplace for Europe;** ESORICS '96, Rome; LNCS 1146, Springer-Verlag, Berlin, 1996, 1-14.
Also *SEMPER* report 431ZR031, February 19, 1996. Available from <http://www.semper.org/info/431ZR031.ps.gz>

8. Appendix A: API for the Generic Payment Service

For a complete set of JAVA bindings for the generic payment service API, see *http://www.semper.org/deliver/d03/javadoc/semper.payment.html*

1) Purse Selection Services

Local

PaymentManager: `selectPurse`
`(PaymentEntity peer, Flag operation_type, Amount amount,`
`List candidate_purses, List negotiation_results)`

Select a single purse from the specified list of purses based on the result of a previous negotiation (negotiation results are in the form of a list of [payment system name, amount] pairs). The operation type flag indicates the direction of the proposed movement of value (in/out).

PaymentManager: `selectCandidatePurses`
`(PaymentEntity peer, String my_alias, Amount amount,`
`Flag operation_type, List payment_systems,`
`List purse_names, List security_options)`

Select a single purse locally (no negotiation) from among the specified list of purses or payment systems given certain user requirements (such as the security services required and the amount type). The operation type flag indicates the direction of the proposed movement of value (in/out).

Negotiations

PaymentManager: choosePS
(ComPoint c, Flag choice_type, List payment_systems,
Amount amount)

Receive a proposed list of payment systems and corresponding amounts from the peer; construct a new list by selecting an acceptable subset and revising amounts where applicable. Send the new list to the peer and return the same to the caller.

PaymentManager: requestPSChoice
(ComPoint c, Flag choice_type, List payment_systems,
Amount amount)

Propose a list of payment systems and an amount to the peer and request them to choose a subset of the payment systems. The choice type flag indicates what sort of modified proposals are acceptable from the peer.

PaymentManager: confirmPurse
(ComPoint c, Flag op_type, Amount amount,
List candidate_purses,
PaymentEntity peer)

Receive a confirmation request from the peer containing a single payment system and a corresponding amount. Indicate to the peer whether their choice is acceptable, given the list of candidate purses. If it is acceptable, select a single purse locally from the list of candidate purses.

PaymentManager: requestPurseConfirmation(
ComPoint c, PurseReference my_purseref, List short_list,
PaymentEntity peer)

Request confirmation from the peer after having selected a purse.

Cover

PaymentManager: selectPayingPurse
(PaymentEntity peer, String my_alias, Amount amount,
List payment_systems, List purse_names,
List security_options, String ext_ref)

An example use of the payment selection methods.

PaymentManager: selectReceivingPurse
(PaymentEntity peer, PaymentEntity me, Amount amount,
List payment_systems, List purse_names,
List security_options, String ext_ref)

An example use of purse selection for receiving payments. The payee is allowed to choose a subset of the proposed payment systems and to reset the proposed amount to pay.

2) Value Transfer Services

PurseServices interface hierarchy

Common

PaymentTransaction: `pay`
`(PaymentEntity payee, Amount amount,`
`List security_options, String ext_ref)`

PaymentTransaction: `receivePayment`
`(PaymentEntity payer, Amount amount,`
`List security_options, String ext_ref)`

PaymentTransaction: `reversePayment`
`(PaymentTransactionRecord ref)`
Request reversal of a previously made payment.

PaymentTransaction: `reverseReceivedPayment`
`(PaymentTransactionRecord ref)`
Process request for reversing a previously received payment.

Account-based Model

PaymentTransaction: `authorise(Amount amount)`
Request authorisation on a previously received payment.

PaymentTransaction: `capture(Amount amount)`
Capture a previously authorised payment.

PaymentTransaction: `multiAuthorise`
`(List transactions)`
Request simultaneous authorisation for several received payments.

PaymentTransaction: `multiCapture`
`(List transactions)`
Simultaneously capture several received payments.

PaymentTransaction: `receiveRawPayment`
`(PaymentEntity payer, Amount amount,`
`List security_options, String ext_ref)`
Receive payment information but defer authorisation/capture.

Cash-like Model

PaymentTransaction: `withdraw`
`(Amount amount, List security_options, String ext_ref)`
Load electronic value into the purse.

PaymentTransaction: `deposit`
`(Amount amount, List security_options, String ext_ref)`
Move electronic cash from a purse to a bank account.

Token-based PurseServices interface hierarchy

Common

PaymentTransaction: `startPay`
`(PaymentEntity payee, Amount amount,`
`List security_options, Role role, String ext_ref)`
Start the payment protocol and depending on the role (initiator/responder), return the first token to be sent to the payee.

PaymentTransaction: `startReceivePayment`
`(PaymentEntity payer, Amount amount,`
`List security_options, Role role, String ext_ref)`
Start the receive-payment protocol and depending on the role (initiator/responder), return the first token to be sent to the payee.

PaymentTransaction: `startReversePayment`
`(Role role, PaymentTransactionRecord ref)`

PaymentTransaction: `startReverseReceivedPayment`
`(Role role, PaymentTransactionRecord ref)`

PaymentTransaction: `processToken(TokenCapsule mesg)`
Process an intermediate token in a protocol. Optionally return a token to be sent to the peer.

Account-based Model

PaymentTransaction: `startAuthorise`
`(Role role, Amount amount)`

PaymentTransaction: `startCapture`
`(Role role, Amount amount)`

PaymentTransaction: `startMultiAuthorise`
`(Role role,` List `transaction_recs)`

PaymentTransaction: `startMultiCapture`
`(Role role,` List `transaction_recs)`

PaymentTransaction: `startReceiveRawPayment`
`(PaymentEntity payer, Amount amount,`
`List security_options, Role role, String ext_ref)`

Cash-like Model

PaymentTransaction: `startWithdraw`
`(Amount amount, List security_options, Role role,`
`String ext_ref)`

PaymentTransaction: `startDeposit`
`(Amount amount, List security_options, Role role,`
`String ext_ref)`

Optional interfaces

Repeated low-value Payment Services

PaymentTransaction: `startMicropayment (PaymentEntity payee, Amount max_amount, List security_options, String ext_ref)`
Prepare for a series of subsequent micropayment operations.

PaymentTransaction: `startReceiveMicropayment (PaymentEntity payer, List security_options, String ext_ref)`
Prepare to receive a series of subsequent micropayment operations.

PaymentTransaction: `doMicropayment(Amount amount)`
Make a single micropayment operation.

PaymentTransaction: `receiveMicropayment (Amount amount)`
Receive a single micropayment.

PaymentTransaction: `stopMicropayment()`
Stop making micropayments.

PaymentTransaction: `stopReceiveMicropayment()`
Stop receiving micropayments.

3) Transaction Management Services

Purse: `startTransaction()`
Start a transaction on this purse.

PaymentTransaction: `getStatus()`

PaymentTransaction: `abort()`
Abort the current transaction, if possible.

PaymentManager: `registerTransaction (PaymentTransaction tr, String ext_ref)`
Register an active transaction with the Payment Manager.

PaymentManager: `deregisterTransaction (long transaction_ID)`

4) Purse Management Services

PaymentManager: `createPurse (String purse_class_name, String purse_name)`
Create a new purse and register with the payment manager.

Purse: `deletePurse(String purse_name)`
Remove a purse from the payment manager's list.

Purse: `registerWithBank()`

Register or re-register a purse_in the bank.

Purse: `setup()`
Carry out all the necessary configuration for this purse (prompting the user for necessary information)

5) Information Services

PaymentManager: `getListOfPurses (Flag return_type_indicator)`
Return the list of all purses registered with the payment manager. The flag indicates the form of the returned list (as a list of purse names, payment system names etc.)

6) Dispute Management Services

Purse: `getEvidence(Statement statement)`
Returns an Evidence object containing the evidence necessary to support a claimed statement.

Evidence: `verifyEvidence()`
Verify whether this evidence proves the statement associated with it.