Designing a generic payment service

by J. L. Abad Peiro
N. Asokan
M. Steiner
M. Waidner

The growing importance of electronic commerce has resulted in the introduction of a variety of different and incompatible payment systems. For business application developers, this variety implies the need to understand the details of different systems, to adapt the code as soon as new payment systems are introduced, and also to provide a way of picking a suitable payment instrument for every transaction. In our work, we unify the different mechanisms in a common framework with application programming interfaces. 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.

Ever since money was invented as an abstract way of representing value, systems for making payments have been in place. In the 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, checks, 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 financial clearing systems. The financial clearing systems are built on the closed, strictly controlled networks of financial institutions that 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.[1] Currently, many different, incompatible Internet payment systems compete with one another. Most shops accept, at best, only a subset of them.

Consider a business application that 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 make sure that the application knows how to use all the different payment systems its users are likely to have available, and, in case multiple payment instruments are available, provide a way to choose one of them. Our primary motivation was to develop a framework that frees the business application from addressing these issues.

We describe the design and implementation of a generic payment service that provides such a framework for enabling applications to use different payment systems in a transparent manner. The primary component of this generic service is a coherent hierarchy of application programming interfaces (APIs) for the transfer of monetary value. The flow of information during transactions leads to a classification of actual payment systems into a set of payment

0018-8670/98/$5.00 © 1998 IBM

models. The API hierarchy reflects this separation into different payment models. It consists of a base 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; enables applications to be concerned just with the questions of "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)
- Tools and framework necessary for the incorporation of actual payment systems into the generic payment service

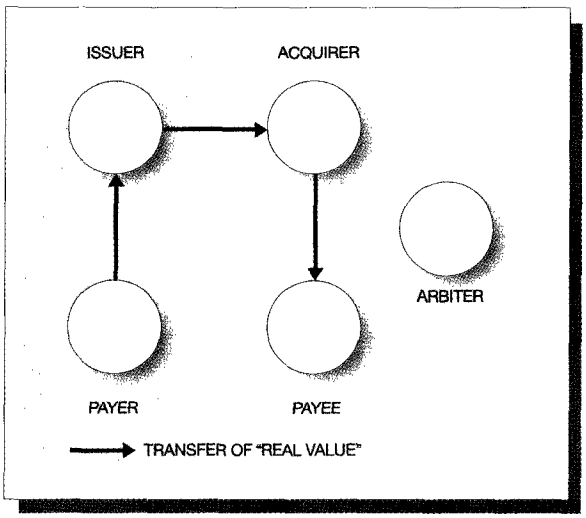
This work is being carried out as part of the SEMPER (Secure Electronic MarketPlace for EuRope) project aimed at building a secure electronic marketplace.[2] However, the architecture of the generic payment service is independent of the specific environment of SEMPER: a stand-alone implementation of the service is possible. In fact, the intermediate results of our work were used as a basis for IBM's Internet payment framework called *SuperSET* and led to IBM's CommercePoint* eTill product. Earlier reports on this work appeared as public project reports[3,4] of SEMPER.

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 at 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. In Asokan et al.[1] a survey of existing payment systems is presented. In this section, we present an intuitive model of electronic payment systems as a first step in the design of the generic payment service.

Players. Electronic payments involve a *payer* and a *payee*. The intent of the payment is to transfer monetary value from the payer to the payee. Transfer is accomplished by an electronic payment protocol. The process also requires at least one financial institution that links the data exchanged in the payment protocol to transfers of monetary value. The financial institution may be a bank that deals with monetary value represented in terms of "real value," or it may be some organization that issues and controls other forms of representation (e.g., loyalty points). Here we use the word *bank* to mean all different types of financial institutions and the phrase "real value" 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.

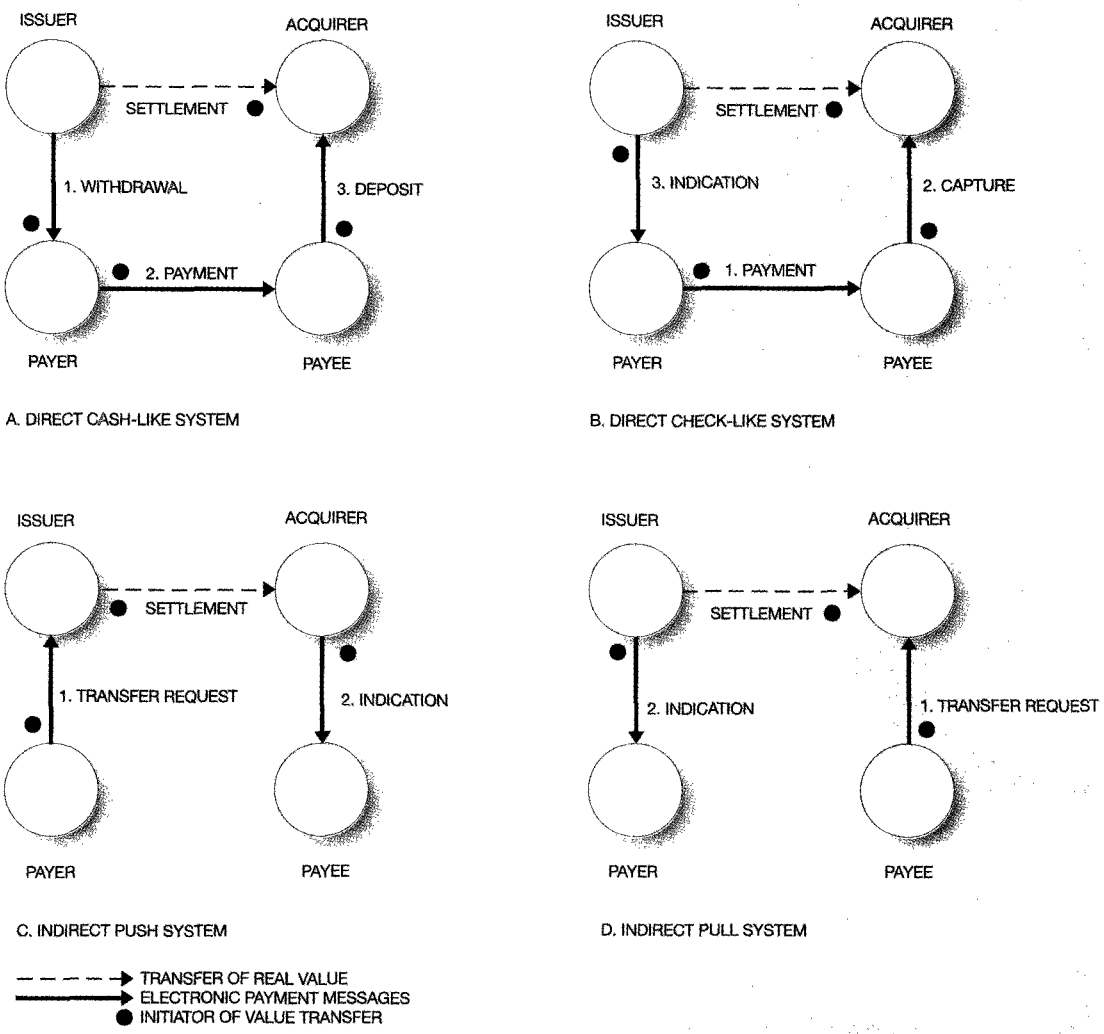
Figure 1 Players of a payment system



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 pieces 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.)

Certain payment systems might involve more players, e.g., registration and certification authorities, or

Figure 2 Payment models



other trusted third parties that provide anonymity[5,6] or enforce receipts for payments.[7–9]

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

One criterion for distinction is whether the communication between the payer and the payee is *direct* or *indirect*. In the latter case, the payment operation is initiated by one player and involves only the initiator and the bank(s). The other player is notified by its bank at the completion of the transaction. An example of direct payment is paying by cash or check. An example of indirect payment is paying by means

of a standing order or wire transfer. Most currently proposed Internet payment systems implement direct payments. Consequently, we will focus only on those systems.

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 among: (1) prepaid (or cash-like) payment systems, (2) pay-now payment systems, and (3) pay-later payment systems. The latter two are quite similar; in both cases, the user must have some sort of an "account" with the bank, and a payment is always done by sending some sort of "form" from payer to payee (check, credit-card slip, etc.). Thus, we consider these two cases to belong to the same model, which we call the check-like model.[10] In the rest of this paper, we use the terms *cash-like* and *check-like* to refer to the two models. Similar informal models of payment systems have been used by various others.[7,11,12] A large number of proposed or existing payment systems can be grouped into these two categories. Examples of cash-like payment systems include ecash**,[13] NetCash,[14] CAFE,[15] and Mondex**.[16] Examples of check-like systems include credit-card protocols such as SET**,[17] iKP,[18] CyberCash**,[19] and electronic check schemes such as the scheme proposed by FSTC**.[20] 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. Additional work in this direction has already begun.[21,22]

Design of the generic payment service

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 players may specify certain security attributes for this value transfer.
- 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.

In the simplest case, the transfer of value happens between two end points. We call such an end point a *purse*.[23] 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, and an ecash 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 authorization (e.g., insert a smart card or enter a pass phrase, or both, for each withdrawal), whereas others may not have any access control.

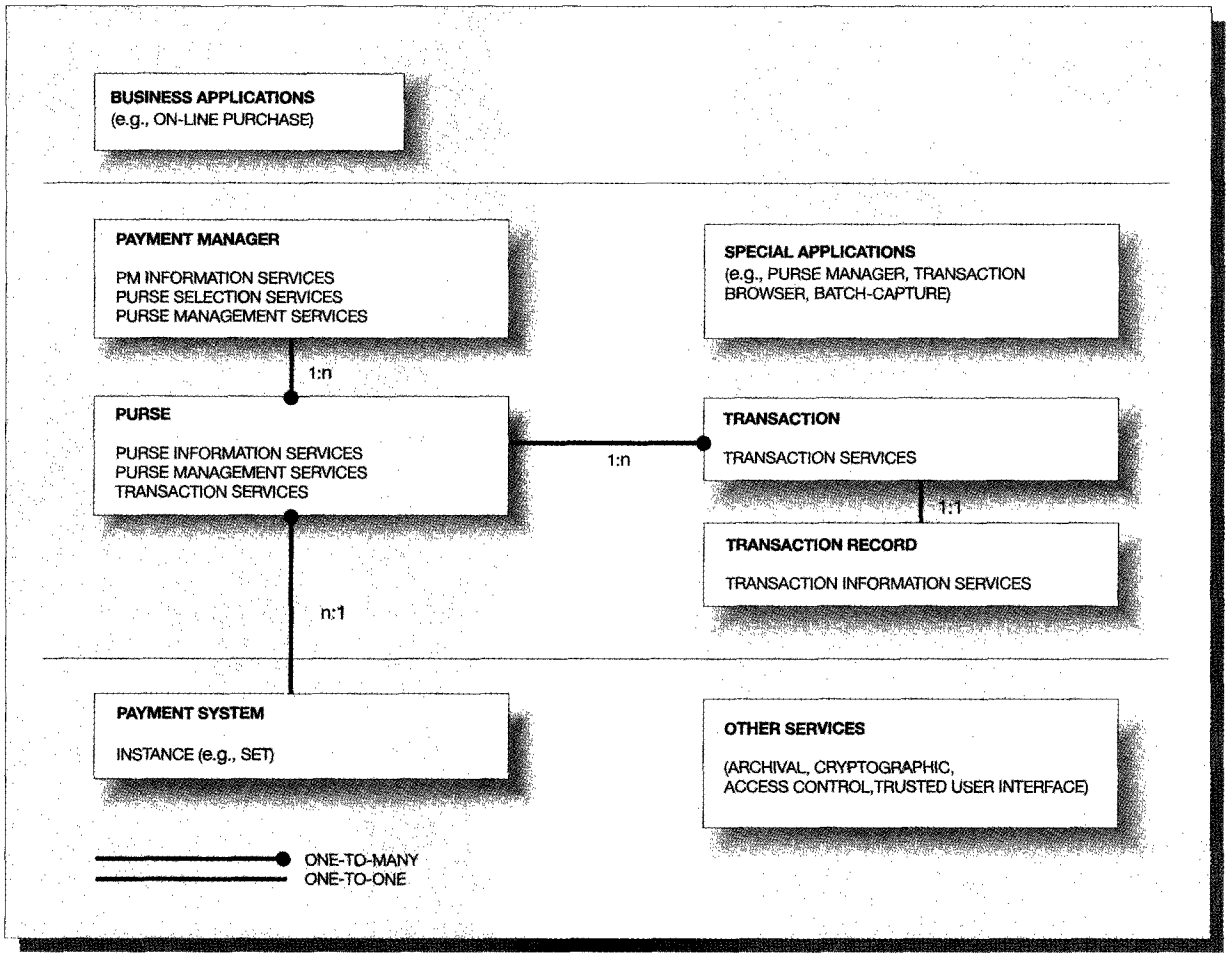
Each value transfer in progress is embodied in a separate *transaction*. A purse may be involved in several concurrent transactions. *Transaction management services* allow transactions to be queried for their status, canceled, or recovered from a crash. Before beginning a transaction, each player must 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 are 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 managing its purses and transactions. *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 or statistics on all payments received and made in a certain period of time.

Finally, *dispute management services* allow the user to make claims about (alleged) past transactions as well as prove or disprove them to an arbiter. As we mentioned earlier, none of the payment systems introduced so far has integrated dispute-handling features. Most limit themselves to the collection of evidence alone. Thus, in the original design, we did not address the dispute management issue. We discuss the problem and outline an approach to a solution in a later section. The detailed treatment of dispute handling is a focus of our current work and will be described in a forthcoming paper.

Figure 3 shows the entities in the generic payment service and the services they provide. We treat transactions as transient entities. Each transaction is associated with a longer-lived transaction record where all relevant information about the transaction is

Figure 3 Generic payment service in action (entities in a typical instance)



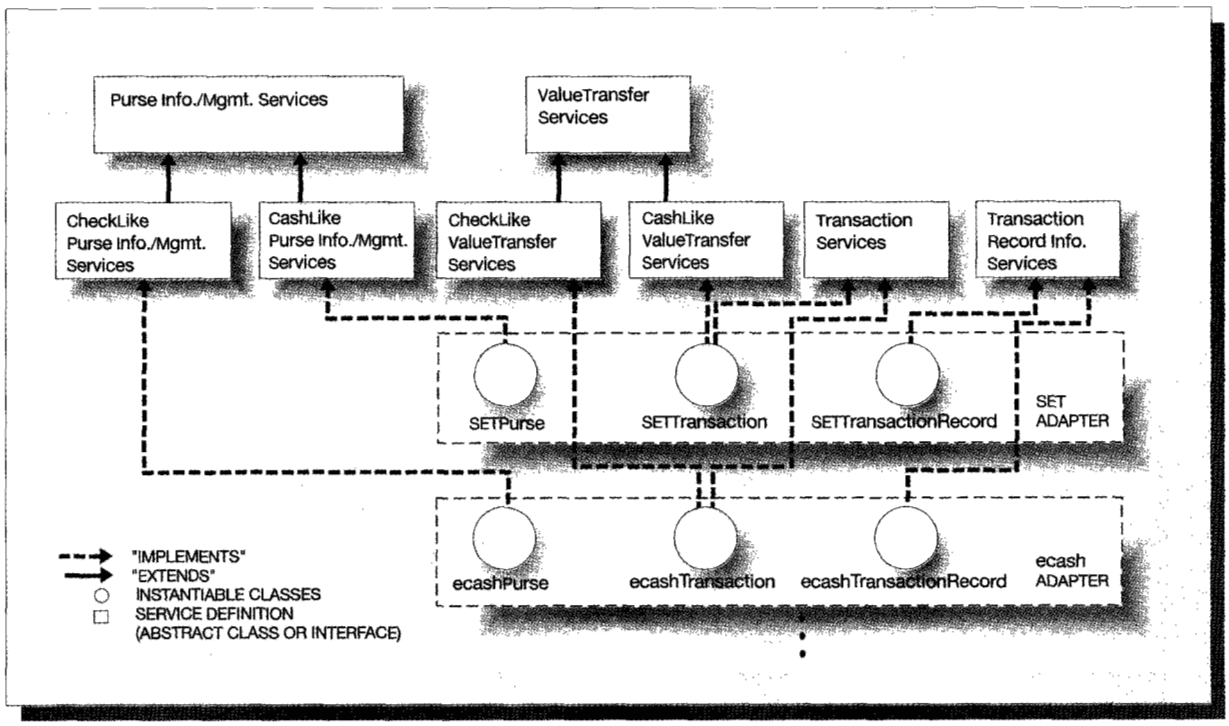
maintained. Some of the services are distributed over more than one entity (e.g., information services are provided jointly by the payment manager, purses, and transaction records).

Design overview. To define each of the above services more concretely, we adopted the following approach. For a given class of services (e.g., value transfer services):

1. Identify the primitives for this service that are common to most payment systems. Describe these in the form of a *base* service interface. For example, the ValueTransferServices[24] interface contains primitives such as pay.
2. Then for each payment model, identify any additional primitives not already covered in the base interface but common to all payment systems of that model. Describe these in the form of a subinterface. For example, the subinterface CashLikeValueTransferServices for the cash-like model defines primitives such as withdraw( ) that are relevant only in the cash-like model.

Some services, such as purse selection, are provided by the generic payment service itself. Other services are provided by the various payment systems. To incorporate a specific payment system into the generic payment service, a system-specific *adapter* must be built. The adapter uses the services provided by the payment system to implement services defined in the generic payment service interface. To introduce a

Figure 4 Generic payment service (classes and interfaces)



new model, a new (possibly empty) subinterface will have to be defined for each service interface. In the next subsection, we describe the services interfaces in detail.

The high-level design described so far can be implemented in a variety of ways. We opted for an object-oriented approach. We describe the entities and services required in the generic payment service in terms of base classes and interfaces. The four main types of entities identified in the previous section (purses, transactions, transaction records, and the payment manager) are described by four different classes. Each service interface corresponds to an interface or abstract class. Concrete implementations for the services that are independent of payment systems are provided by the payment manager or related classes. Adapters for specific payment systems can then provide implementations for the remaining interface and abstract class methods. For example, the Transaction class in an adapter is expected to implement the services defined in the ValueTransferServices interface as well as the TransactionServices interface.

The ValueTransferServices interface has model-specific extensions. The adapter for a given payment system should implement the branch of the ValueTransferServices interface corresponding to the model of that payment system. Figure 4 illustrates the classes that constitute an adapter and the services they implement.

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 base classes. In the following subsections, we describe the services of the generic payment service and the objects that provide them (shown in Figure 3) in more detail.

Services. The primitives of the value transfer services interface are described briefly in Table 1. Primitives for other services are similar. We do not show them here for lack of space. Square parentheses ([ ]) indicate optional parameters. We do not show exceptions and errors. Concrete Java** bindings of the service descriptions can be found in Reference 25.

Table 1 Generic payment service: Value transfer services

Primitive	Input	Output	Description
Base services			
pay	*payee, amount, options, ref.**		Send a payment
receivePayment		*payee, amount, options, ref.*	Receive a payment
reversePayment	*transaction record*		Ask/get a refund
reverseReceivedPayment	*transaction record*		Make a refund
Additions for cash-like model			
withdraw	*amount, options, ref.*		Load money into purse
deposit	*amount, options, ref.*		Unload money from purse
Additions for check-like model			
receiveRawPayment	*payer, amount, options, ref.*		Receive a payment (defer authorization)
authorize			Authorize a previous raw payment
capture	[*amount*]		Capture a previous raw payment
multiCapture	*list of transaction records*		Capture a set of previous raw payments

*ref. allows this payment to be linked to its context, e.g., an order and its description.

Purses. A purse is an abstraction of an instance of a payment system that is available to the user. It is necessary to have services for:

- Creating a purse (i.e., a constructor to instantiate a purse object)
- Configuration and setup, which will be used by purse management applications (e.g., to associate a purse with a credit card and to register with a certification authority)
- Initialization, which is invoked during startup to activate the purse
- Creating transactions (see the next subsection)
- Information (e.g., answers to questions such as "Does this purse provide nonrepudiatable receipts for payments?")

These services are part of the purse management, purse information, and transaction services. We use a Purse class hierarchy. The base Purse class defines the aforementioned services and provides default implementations for some of them. For each payment model, we extend the base Purse class to a model-specific subclass (e.g., CheckLikePurse class). Adapter-writers shall extend a model-specific Purse class and override or extend default implementations as necessary. For example, to adapt the SET payment system (SET, or Secure Electronic Transactions**,[17] is a protocol for making credit-card transactions over the Internet), we define a class SETPurse that extends CheckLikePurse.

Additionally, the Purse class hierarchy also provides services for information management corresponding to these purses (e.g., answers to questions such as "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 as follows: a *pay-only* purse can be used to make but not to receive payments, a *receive-only*[26] purse can be used to receive but not to make payments, and a *pay-and-receive* purse can be used for making and receiving payments.

Mondex and ecash are examples of payment systems that do support pay-and-receive purses. Most other payment systems do not.

Transactions and transaction records. As mentioned, the base ValueTransferServices 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 subinterfaces may define additional services. For example, the subinterface 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 ValueTransferServices interface hierarchy. Information associated with a transaction (both transient information such as state 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 information can be used in crash recovery and dispute management as well as for informational purposes.

The base PaymentTransaction defines general transaction services such as trying to abort an ongoing transaction or retrieving its current status. Each subclass of the base class implements a leaf interface of the ValueTransferServices interface hierarchy (e.g., SETTransaction extends PaymentTransaction and implements the CheckLikeValueTransferServices interface). Each leaf Purse class provides a startTransaction( ) method that creates a new transaction of the appropriate type (e.g., in the SETPurse class, the startTransaction( ) method will instantiate a SETTransaction object).

Payment manager. The payment manager provides services for purse selection as well as to retrieve management information. It keeps track of the 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, and registration of a new adapter. 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 initializing all the relevant components on startup. The current design does not yet address fault-tolerance issues. The manager will be the entity providing services for shut-down and fault-tolerance mechanisms such as crash recovery.

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 a user. Except negotiation, 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)

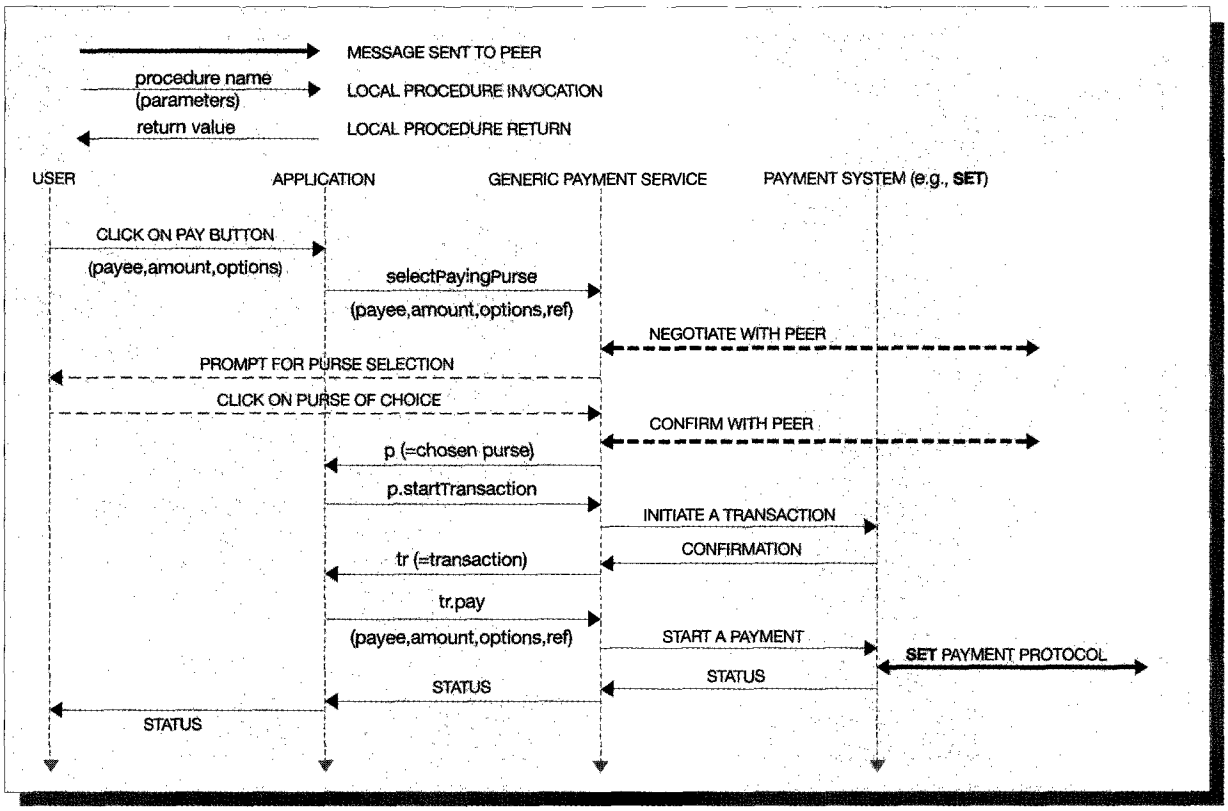
We have designed and implemented a simple negotiation protocol that can support various negotiation policies. Two example methods, selectPayingPurse and selectReceivingPurse implementing a default policy, are provided: the payer is the initiator of negotiation, the payee is allowed to adjust the amounts in its reply (e.g., the merchant may add a surcharge for using a credit card or give a discount for using ecash). It is also possible to enforce other negotiation policies.

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:

1. Identify the model to which the payment system belongs (e.g., SET belongs to the check-like model).
2. Implement a subclass of the Purse class corresponding to the payment model identified (e.g., SETPurse extends CheckLikePurse). This step im-

Figure 5 Interactions during a payment (dotted lines indicate optional flows)



plies providing implementations for all abstract services defined in the ancestor Purse classes (e.g., Purse and CheckLikePurse) 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 configurations necessary for the payment system.

3. Implement a subclass of the PaymentTransaction class that implements the value transfer services defined in the leaf of the ValueTransferServices interface hierarchy corresponding to the payment model identified (e.g., SETTransaction implements CheckLikeValueTransferServices and inherits from PaymentTransaction).

In addition, if any special action needs to be taken during the installation of the adapter, a suitable installation hook must be provided. A standard installation application is available as part of SEMPER. It performs two actions: installing the contents of the adapter module in the correct locations, and registering the name of the new purse class with the payment manager. If there are any adapter-specific installation procedures, they must be implemented in the form of an installation hook defined in the ModuleInstallHook interface.

Using the generic payment service

Once a user has installed one or more payment instruments along with the adapters on his or her system, two kinds of usage are possible: making payment transactions via business applications or using special applications for various purposes.

Payment transactions. The primary use of the generic payment service is via business applications making payment transactions.

A user will initiate payment transactions using some sort of a high-level business application (e.g., a Web

browser or a CD-catalog reader). Figure 5 shows the object interactions that take place at the payer end during execution of a typical payment transaction. The important things to note are:

- The user need not specify the payment instrument to use if he or she does not want to; the payment service can be configured to prompt him or her for selection of a payment instrument if it cannot do so by itself.
- The application is not aware of the specific payment instrument being used; it deals with generic Purse and PaymentTransaction objects. It need not even know the model to which the chosen purse belongs.

The sequence of events at the payee side is similar, with minor differences. The payee application is probably an unattended merchant server. Thus, there will be no user interaction. There may be interactions with third parties during the transaction. For example, in a check-like system, the payee's adapter may contact the acquirer for authorization. One can also imagine a payment system where the payer's adapter has to obtain some sort of a credential from the issuer before each payment. All such communication with third parties is carried out within the adapter—the calling applications are typically unaware of them.

This example is also intended to give an idea about how the generic payment service enables business application development. The primary services used by the business applications are purse selection and value transfer between payer and payee. Both of these are common to all payment systems. Thus, a large class of applications using the generic payment service need not be aware of system- or model-specific details. Certain special applications (see next subsection) will make use of the model-specific components of the generic payment service.

Special applications. The second category of usage is via special applications. The most important special application is a purse management tool.

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.) Purse management makes use of a setup() method provided by the Purse class in an adapter. This method must implement all the necessary configuration for that payment system. For example, the setup() method of the SETPurse allows the user to enter the credit-card information (cardholder name, brand, number, expiry date) that is then stored as part of the purse state.

Other applications. There can be a number of other special applications. Some of these are model-specific. A *batch capture* application can be used by the merchant to capture a set of received payments for check-like purses; typically this will be used as part of end-of-day processing. A *withdrawal* application can be used to load money into cash-like purses. The SEMPER prototype implementation comes with two model-independent special applications: a *transaction browser* allows the user to browse through accumulated transaction records; a *module installer* allows a user to install a new payment instrument along with its adapter.

Extending the design

During the first half of the SEMPER project (fall 1995 to the end of 1996), we designed the generic payment service as described so far and built a prototype. Since then, we have been focusing on extending the design by adding functionality, and by revising some aspects by using better techniques. In this section, we describe these extensions.

Dispute management. Support for handling disputes is a crucial aspect of any system providing accountability. Consider some typical claims that users of a generic service might want to make or deny:

- ABC paid $100 to XYZ on 12/29/97, 10:32 GMT
- ABC paid $100 to XYZ with external reference "Order #432"
- ABC did *not* pay $100 to XYZ
- ABC deposited $100 at bank B on 12/29/1997
- ABC deposited $100 at bank B on 12/29/1997, before noon GMT

A transaction may result in several items of evidence. Only a subset of these items 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.

The problem of dealing with disputes in the generic payment service has three aspects: (1) how to express dispute claims, (2) how to map evidence collected during a transaction to subsequent dispute claims, and (3) since a dispute involves the interaction of more than one player, how to define a multiparty dispute protocol.

In general, disputes can be expressed in terms of *statements* about a (possibly alleged) transaction. We use the following structure for dispute statements:

[not] <TRANSACTION> [<*role*> = <player>] *
[<*attr*><*OP*><value>] *

The examples above would then correspond to statements like:

- PAYMENT *payer* = ABC, *payee* = XYZ, *amount* = $100, *time* = "12/29/97, 10:32 GMT"
- PAYMENT *payer* = ABC, *payee* = XYZ, *amount* = $100, *reference* = "Order #432"
- **not** PAYMENT *payer* = ABC, *payee* = XYZ, *amount* = $100
- DEPOSIT *user* = ABC, *bank* = B, *amount* = $100, *time* = "12/29/97"
- DEPOSIT *user* = ABC, *bank* = B, *amount* = $100, *time* < "12/29/97"

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

The dispute management interface provides services to construct dispute claims and prove them to a verifier. In the simplest case, it is enough to extract the right pieces of evidence (such as receipts) and present them to the verifier. In other cases, it may be necessary to interact with the verifier using a complex proof protocol. The Purse subclasses in the adapters will be required to implement the dispute management interface.

Typically, disputes are about payment transactions. These disputes will be started by the applications that initiated the payment in the first place (e.g., a Web browser on the payer's side and a merchant server on the payee's side). Disputes between a user and the bank (e.g., wrong entry in a bank statement) will require special bank-specific applications to drive the dispute.

Currently, we are expanding on these ideas in building a framework for handling disputes in the generic payment service.

Payment security policies. A number of considerations apply to payment security policies. We describe them here.

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 of the types of limits are:

- Each payment from a specified purse (say P1) should not exceed 100 CHF (Swiss francs).
- 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 authorization.
- No more than four payments without explicit user authorization can be made 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 authorized by the user.

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

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

Access control is required in the following cases:

- Access to secret information required to use the underlying payment system (e.g., personal identification numbers [PINs], pass phrases, credit-card numbers, etc.). There may be several different pieces of such information.
- Access to purse operations.

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 pass phrase to authorize high-value payments. The underlying payment instrument may or may not support such granularity.

Figure 6 Incorporating policy in a purse

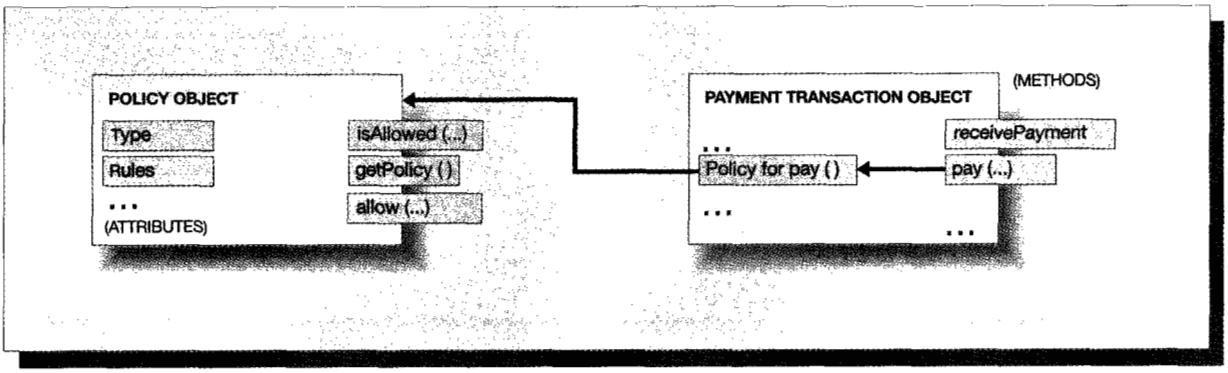
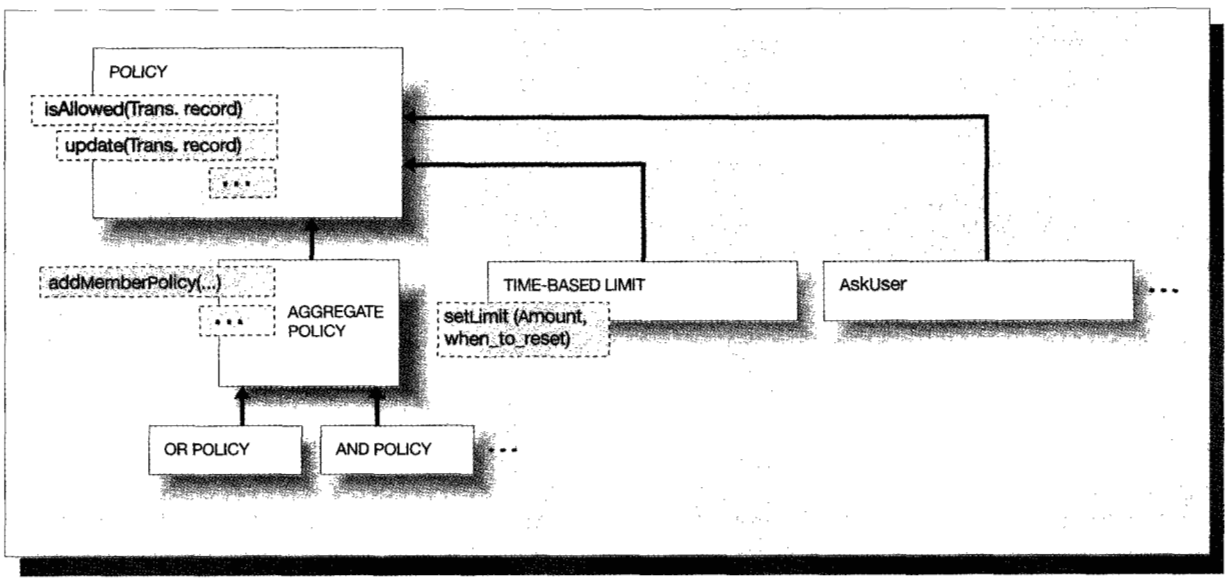


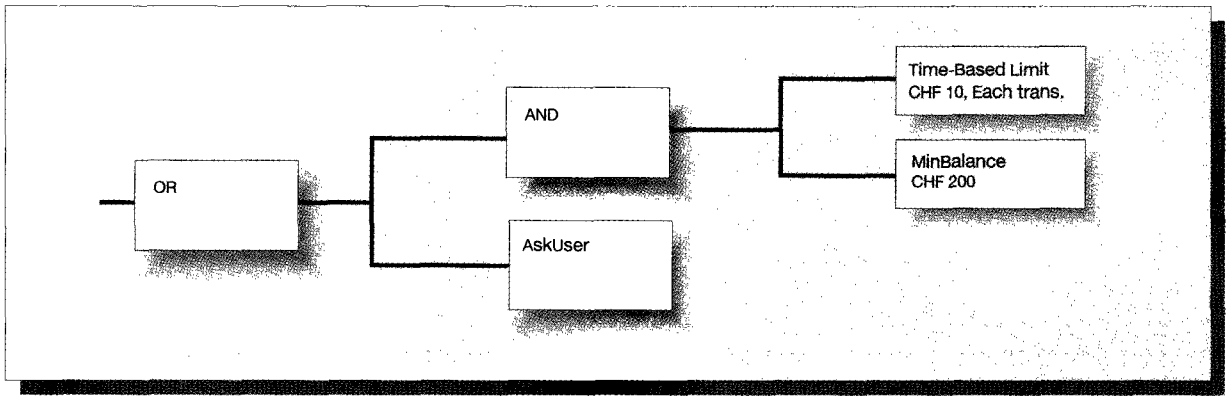
Figure 7 Class hierarchy for policy objects



A common solution: Policy framework. 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 authorization for the service (Figure 6). All policy objects provide ways to check for current availability of a service (the isAllowed( ) method), and to indicate that an authorized 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 above are stateful).

These methods can be used by purse services (such as pay( ) and receivePayment( ) methods) to manage authorization. A reference to the transaction record is provided as an 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.

Figure 8 An example policy



The policy class hierarchy is shown in Figure 7. 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:

- The *AskUser* policy class displays relevant information about the transaction to the user and asks for his or her approval.
- The *MinBalance* policy class makes sure that the minimum balance is above a specified value.
- The *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: An *OR* policy allows the service if any of its constituent policies do so; an *AND* policy allows the service if all of its constituent policies do so.

With these policy objects 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 shown in Figure 8. (When a policy object has other constituent policy objects in it, a top-to-bottom evaluation order for the constituent objects is assumed.)

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 6, the pay method in the transaction class of an adapter must query the payment policy object in its purse before proceeding with the payment.

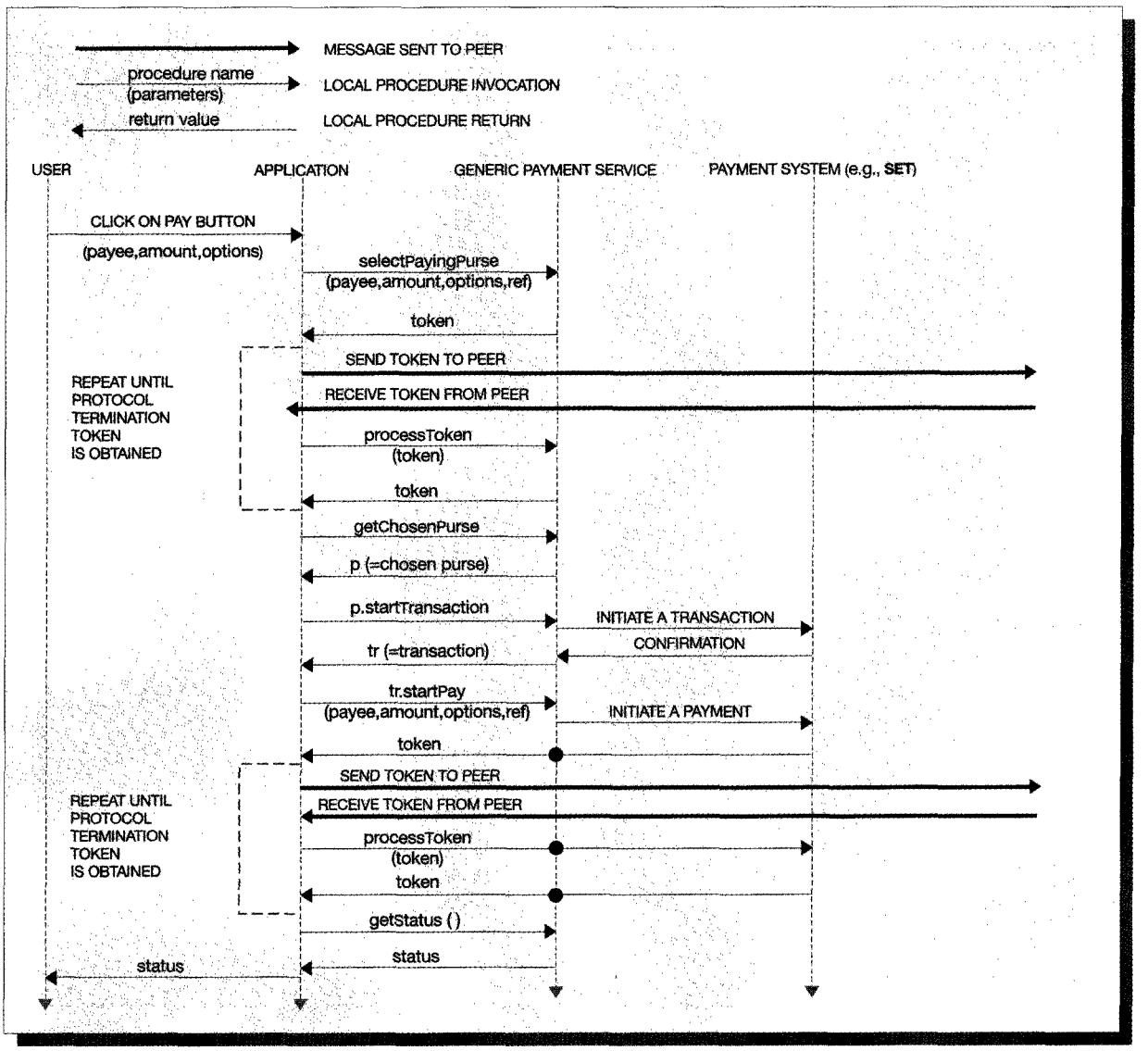
Clearly, several issues need to be resolved. We require a language to express policies and efficient techniques for evaluating and updating policies. A similar approach is given by the PolicyMaker framework.[27] However, policies in this model cannot be defined in terms of pure contextual information such as the total amount spent from a purse.

The description here is intended only to give a flavor of the issues involved. We are currently working on addressing these issues in depth.

Token-based interface definition. The original design assumed a synchronous model since the first version of the SEMPER architecture did the same.[25] However, we have defined a "token-based" interface that can support an asynchronous model in a straightforward manner. Our token-based interface is inspired by the GSS-API[28] approach. It has two types of methods: (1) one "starter" method for each different type of protocol; the starter methods return a token containing the first message of the protocol, and (2) 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.

In the token-based model, the payment service does not engage in any direct communication with the

Figure 9 Interactions during a payment in the token-based model (dotted lines indicate optional flows)



peer. Instead, the caller is expected to take care of the 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 application is expected to communicate this token to its peer entity, the responding caller application. 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 TValueTrans-

ferServices parallel to the ValueTransferServices 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 TValueTransferServices interface. Figure 9 illustrates object interactions in the same scenario depicted in Figure 5, but with a token-based interface for negotiation and value transfer.

Since no peer-to-peer communication is taking place inside the generic payment service, the caller does not have to block on service invocations. The designer of the calling application has the freedom to use an asynchronous implementation architecture. More importantly, the token-based approach can allow an application to supplement the level of security provided by a payment system by transporting the tokens via a channel with particular security attributes. For example, even though payment protocol messages in SET are encrypted, an eavesdropper may be able to determine and link the identity of the payer and payee by watching the network addresses in the payment messages. With a token-based interface, if the applications were able to establish an untraceable communication channel between them, they could extend the untraceability to SET payments as well.

Currently, the interface TValueTransferServices is optionally implemented by subclasses of the PaymentTransaction class. Since the token-based version is more general than the synchronous version, we plan to make the former the default value transfer 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.[29]

Related work

U-PAI[30] is being developed as part of the Stanford Digital Libraries project.[31] 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 (Common Object Request Broker Architecture) and do not 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).[32] 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 that can be used in our work.

The E-CO System project had roughly the same scope[29] as our work, although their main focus so far seemed to be on establishing APIs and mechanisms for payment negotiation.[33] Additional information was not public, and the project appears to have been discontinued.

Status and conclusions

We have presented the design of a generic payment service. Complete Java bindings of our payment service interfaces can be found in the home page of SEMPER deliverable D03.[25] 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. Our work served as a basis for the design of IBM's Internet payment framework, SuperSET. Adapters for a variety of other payment systems are under development by various partners in the SEMPER consortium. For example, DigiCash is developing an adapter for ecash, Royal PTT Nederland NV (KPN) integrates Chipper,**[34] and SET is adapted by IBM.

Acknowledgments

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/.

Interesting discussions with several people helped us develop and refine the ideas presented in this paper; in particular we 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 Jay Black and Matthias Schunter and the anonymous referees for their valuable comments on previous versions of this paper.

Cited references and notes

1. N. Asokan, P. Janson, M. Steiner, and M. Waidner, "State of the Art in Electronic Payment Systems," *Computer* **30**, No. 9, 28–35 (September 1997).
2. M. Waidner, "Development of a Secure Electronic Marketplace for Europe," *Proceedings of the Fourth European Symposium on Research in Computer Security* (*ESORICS*), Rome, Italy, E. Bertino, H. Kurth, G. Martella, and E. Montolivo, Editors, number 1146 in *Lecture Notes in Computer Science*, Springer-Verlag, Berlin (September 1996); also published in *EDI Forum* **9**, No. 2, 98–106 (1996).
3. J. L. Abad Peiro, N. Asokan, and M. Waidner, *Payment Manager—Overview*, 212ZR054, SEMPER Consortium (March 1996).
4. J. L. Abad Peiro, N. Asokan, M. Steiner, and M. Waidner, *Designing a Generic Payment Service*, 212ZR055, SEMPER Consortium (September 1996).
5. S. H. Low, N. F. Maxemchuk, and S. Paul, *Anonymous Credit Cards*, Technical Report, AT&T Bell Laboratories, Murray Hill, NJ (1993); submitted to *1993 IEEE Symposium on Research in Security and Privacy*, Oakland, CA.
6. D. L. Chaum, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Communications of the ACM* **24**, No. 2, 84–88 (February 1981).
7. H. Bürk and A. Pfitzmann, "Payment Systems Enabling Security and Unobservability," *Computers and Security* **8**, No. 5, 399–416 (August 1989).
8. B. Cox, J. D. Tygar, and M. Sirbu, "NetBill Security and Transaction Protocol," *Proceedings of the First USENIX Electronic Commerce Workshop*, USENIX, New York (July 1995), pp. 77–88.
9. B. Pfitzmann, M. Waidner, and A. Pfitzmann, "Recthssicherheit trotz Anonymität in offenen digitalen Systamen," *Computer und Recht* **3,** No. 10, 712–717 (October 1987), No. 11, 796–803 (November 1987), No. 12, 898–904 (December 1987); also published in *Datenschutz und Datensicherung DuD* **14,** No. 5, 243–253 (1990) and No. 6, 305–315 (1990).
10. In our prototype implementation, we used the phrase "account-based." It was somewhat confusing because certain practical implementations of cash-like payment systems, such as DigiCash's ecash also have a notion of an "account" in the bank. Thus, in the interest of avoiding confusion, we use the phrase "check-like" here.
11. C. Neuman and G. Medvinsky, "Requirements for Network Payment: The NetCheque Perspective," *Proceedings of IEEE Compcon '95*, San Francisco (March 1995).
12. F. F. Masaguer, "Security in Electronic Trading Over Open Networks: A Detailed Analysis and Comparison," *14th Worldwide Congress on Computer and Communications Security Protection*, C.N.I.T Paris-La Defense, France (June 1996), pp. 39–66.
13. See http://www.digicash.com/ for more information.
14. G. Medvinsky and B. C. Neuman, "NetCash: A Design for Practical Electronic Currency on the Internet," *1st ACM Conference on Computer and Communications Security*, V. Ashby, Editor, ACM Press, Fairfax, VA (November 1993), pp. 102–106.
15. 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, and M. Waidner, "The ESPRIT Project CAFE—High Security Digital Payment Systems," *Proceedings of the Third European Symposium on Research in Computer Security* (*ESORICS*), Brighton, UK, D. Gollmann, Editor, number 875 in *Lecture Notes in Computer Science*, Springer-Verlag, Berlin (November 1994).
16. See http://www.mondex.com/ for more information.
17. MasterCard and Visa, *SET Secure Electronic Transactions Protocol*, Version 1.0 edition (May 1997); Book One: Business Specifications, Book Two: Technical Specification, Book Three: Formal Protocol Definition; available from http://www.mastercard.com/set/.
18. M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, and M. Waidner, "iKP—A Family of Secure Electronic Payment Protocols," *First USENIX Workshop on Electronic Commerce*, New York (July 1995), pp. 89–106.
19. D. E. Eastlake, B. Boesch, S. Crocker, and M. Yesil, *CyberCash Credit Card Protocol Version 0.8*, Internet Draft (July 1995).
20. *Electronic Check Proposal*, Technical Report, Financial Services Technology Consortium (1995).
21. B. Pfitzmann and M. Waidner, *Properties of Payment Systems—General Definition Sketch and Classification*, Research Report RZ 2823 (#90126), IBM Research (May 1996).
22. B. Pfitzmann and M. Waidner, "Integrity Properties of Payment Systems," private communication of work in progress (December 1996); contact the authors for the current status of the work.
23. Other names to denote the same concept have been used in the literature. The word *pocket* appears to be gaining favor.
24. In the prototype implementation, this hierarchy was named PurseServices. Here we opt for a more intuitive name.
25. SEMPER Consortium, *Basic Services: Architecture and Design*, Deliverable D03 of ACTS Project AC026, Public Specification (September 24, 1996); available from http://www.semper.org/info/index.html#deliverables.
26. Sometimes the word *till* is used to denote a receive-only purse.
27. M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized Trust Management," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA (May 1996).
28. J. Linn, *Generic Security Service Application Program Interface, Version 2*, Internet Network Working Group, Standards Track, Request for Comments: RFC 2078 (January 1997); obsoletes RFC 1508.
29. A. Bahreman, "Generic Electronic Payment Services: Framework and Functional Specification," *Second USENIX Work-*

shop on Electronic Commerce, USENIX, Oakland, CA (November 1996), pp. 87–103.
30. S. P. Ketchpel, H. Garcia-Molina, A. Paepcke, S. Hassan, and S. Cousins, "U-PAI: A Universal Payment Application Interface," *Second USENIX Workshop on Electronic Commerce*, USENIX, Oakland, CA (November 1996), pp. 105–121.
31. See http://www-diglib.stanford.edu for more information.
32. See http://www.javasoft.com/commerce for more information.
33. A. Bahreman and R. Narayanaswamy, "Payment Method Negotiation Service," *Second USENIX Workshop on Electronic Commerce*, USENIX, Oakland, CA (November 1996), pp. 299–314.
34. See http://www.chipper.com/ for more information.

General reference

For a collection of WWW pointers on electronic commerce, see http://www.semper.org/sirene/outsideworld/ecommerce.html.

Accepted for publication August 20, 1997.

José L. Abad Peiro *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (electronic mail: jla@zurich.ibm.com).* Mr. Abad Peiro is a member of the network security research group at the IBM Zurich Research Laboratory. He received an M.S. in computer science from the Universidad Politécnica de Valencia, Spain, and finished the last year of telecommunications engineering in the École Nationale Supérieure des Télécommunications de Bretagne, France. Prior to joining IBM, he was a network manager in the communications division of the European Space Agency in Darmstadt, Germany. He is pursuing a Ph.D. in computer science from the Catholic University of Louvain. Mr. Abad Peiro is a member of the ACM.

N. Asokan *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (electronic mail: aso@zurich.ibm.com).* Mr. Asokan has been a member of the network security research group at the IBM Zurich Laboratory since 1995. His research interests include network security, electronic commerce, and mobile computing. He received his B.Tech. in computer science and engineering from the Indian Institute of Technology, Kharagpur, an M.S. in computer science from Syracuse University, and is a candidate for a Ph.D. in computer science from the University of Waterloo. From 1990 to 1995, he was a software systems specialist at the University of Waterloo, working primarily on network security issues. Mr. Asokan is a member of the ACM.

Michael Steiner *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (electronic mail: sti@zurich.ibm.com).* Mr. Steiner is a research staff member of the network security research group at the IBM Zurich Research Laboratory, where he works on security in network management and electronic commerce. His interests include secure and reliable systems as well as cryptography. He received a Diploma in computer science from the Swiss Federal Institute of Technology (ETH). Mr. Steiner is a member of the ACM.

Michael Waidner *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (electronic mail: wmi@zurich.ibm.com).* Dr. Waidner is the manager of the network security group at the IBM Zurich Research Laboratory. His research interests include cryptography, security, and all aspects of dependability in distributed systems. He has coauthored numerous publications in these fields. Dr. Waidner received his diploma and doctorate in computer science from the University of Karlsruhe, Germany. He is a member of the ACM, GI, IACR, and SIAM.

Reprint Order No. G321-5664.