Payment Method Negotiation Service: Framework and Programming Specification

Alireza Bahreman and Rajkuman Narayanaswamy
EIT

PAYMENT METHOD NEGOTIATION SERVICE

Framework and Programming Specification

Alireza Bahreman
bahreman@eit.com

Rajkumar Narayanaswamy
rajkumar@eit.com

E-CO System Project[Ψ]
EIT/VeriFone

ABSTRACT

We propose a Payment Method Negotiation Service in this paper. This service enables transacting peers to determine and negotiate their common set of payment methods, protocols, systems or mechanisms, system providers, capabilities, accounts, and instruments. The service also enables the peers to select, among the negotiated choices, a particular payment system or mechanism to use to conduct commerce. We use a modular framework and define an object-based solution. The programming interface is modular and extensible and can interoperate with a wide range of payment systems. The service is transport independent and can support a variety of communication models. Therefore, all electronic commerce application developers would be able to accommodate this service in their applications. The Payment Method Negotiation service introduced here is part of the Generic Electronic Payment Services (GEPS) framework being designed and implemented in the E-CO System Project[1].

1 Introduction

The excitement surrounding electronic commerce has resulted in the introduction of numerous payment systems and mechanisms. Transacting peers must determine which of the many payment choices they will use in a commerce transaction. Due to this abundance of choice, end-users and applications alike face the challenges of negotiating and selecting a particular payment system or mechanism to use. Application developers face an increasing challenge of enabling such negotiation and selection with the introduction of a variety of new payment systems and mechanisms.

To address this problem, we propose a service called Payment Method Negotiation (MN). This service enables transacting peers to mutually agree on a common set of payment methods, protocols, systems or mechanisms, system providers, capabilities, accounts, and instruments. The service also enables the peers to select, among the negotiated choices, a particular payment system or mechanism to use to conduct commerce. The service is generic enough to allow integration into a variety of applications and is extensible enough as to interoperate with a variety of payment systems. The MN service introduced here is part of the Generic Electronic Payment Services framework defined in a companion paper [GEPS].

To better understand what the *negotiation* and *selection* processes entail, consider the following examples. Consider the scenario in which Alice walks into a store. At the entrance, the merchant might have displayed the Visa®, MasterCard®, and Discover® logos. This is an indication that the merchant accepts those payment instruments. In a sense, the merchant is *offering* its payment capabilities to the customer upon entering the store. Alternatively, Alice could *query* the merchant

Ψ This work has been funded by DARPA contract DABT63-95-C-0133.

1 For more information, please visit our [WebSite] at http://eco.eit.com.

and inquire about acceptable payment mechanisms or methods. For example, Alice could ask the merchant if it accepts Cash or Debit as a method of payment. Note that neither the offer nor the inquiry necessarily presents a complete set of capabilities. In other words, neither party is obliged to reveal all of its capabilities. As a result of these and other related exchanges, Alice and the merchant will be able to determine if there are any acceptable means of payment which they both share in common. The above interaction describes the negotiation process. Only information specific to the payment choices are exchanged during negotiation. For example, the process does not include discount negotiation for using Cash versus Credit. That is outside the scope of this paper. Finally, the selection process is the determination of a specific choice of payment from the negotiated list of choices. For example, Alice might select one of her Visa cards.

The negotiation process does not require human interaction when implemented in the online world. This is because a list of payment capabilities can be stored online and retrieved automatically when needed. The Capability Management service of the GEPS provides this functionality. The only intelligence involved is in determining the amount of information to reveal at any stage of the negotiation. The selection process, on the other hand, will likely require input from the human end-user. The maintenance, and usage of, end-user's preferences could replace some of this interaction. (The Preference Management provides this service in GEPS.) However, the involvement of the end-user in the final selection process is very important. To not allow the end-user to select the final payment choice is like forcing Alice to give her purse to the merchant and letting the merchant pick the payment method of choice. If that happens, it is likely that Alice would not visit that store again, anytime soon!

The above example of negotiation and selection is possible using the Payment Method Negotiation framework suggested in this paper. In order to explain how we accomplish this, we organize this paper using the following structure. We start by presenting the terminology which we use throughout this paper in Section 2. We present the payment negotiation requirements in Section 3. Section 4 covers the Payment Method Negotiation framework while in Section 5 we introduce the programming interface and implementation details. We discuss the security considerations in Section 6. In Section 7, we compare our work to, and report on, other related work in this area. We wrap up with concluding remarks in Section 8.

2 Terminology

In this section, we present definitions for some of the terminology we use consistently throughout this paper. We provide this as an aid to understanding the discussions in this paper.

2.1 Payment Method

Payment method is a high-level categorization of a payment transaction. A way of making a payment, if you will. Some known payment methods include: Cash, Credit, Debit, Electronic Check, and Electronic Cash. For example, Alice might pay Bob ten US dollars in paper currency; she is therefore using the Cash payment method.

2.2 Payment Protocol

Payment protocol refers to a collection of messages, used to carry electronic payment related information and instructions among the parties involved, and the flow or sequence of those messages. As an example of an electronic payment protocol, consider the Secure Electronic Transaction (SET) specification.

2.3 Payment Mechanism or System

Payment mechanism or system refers to a complete system designed to enable and execute payment transactions among parties. Some payment mechanisms require a third party intermediary such as NetBill, from Carnegie Mellon University.

2.4 Payment System Provider

Payment system provider refers to the party operating a payment system or mechanism. For example, CyberCash, Inc., is a payment system provider, for their credit card payment mechanism.

2.5 Payment Capability

Payment system providers often implement the software needed to use their payment mechanism. Other developers then use this software in their application to conduct electronic payments. We refer to these implementations as payment capabilities. The nature of their implementation can vary (for example, an applet, a shared library, or even a plug-in). The end result, however, remains the same (i.e., it enables an application to use the services provided by a payment system).

2.6 Payment Service

Payment service refers to a collection of payment functionality implemented as a module and used during a payment transaction by applications. There are five major payment services identified in [GEPS]. They are Payment Interface Management, Payment Method Negotiation, Preference Management, Capability Management, and Transaction Management. We dedicate this paper to covering the Payment Method Negotiation service.

2.7 Account and Account Proxy

An account is where one gathers value. Alternatively, an account may provide its owner with the privilege to use a payment system. In the physical world, we are very familiar to the concept of an account (for example, a Bank account). In the online world, there may be new types of accounts, all with similar purposes as in the physical world. An account proxy, however, is the online representation of an account in the physical world. The account proxy can cache the information in the account (for example, the running balance). However, the account proxy's data is not necessarily up to date. One can close or destroy an online account, and an account proxy. However, one cannot destroy a physical account in a pure online fashion; in order to permanently close a physical account, one would require out-of-band mechanisms.

2.8 Payment Instrument

A device or token used during the payment transaction. A particular Visa card with an account number and an expiration date is an example of a payment instrument to use in transactions of type Credit payment method.

3 Requirements

The payment negotiation process must meet the following requirements. This set of requirements is a subset of those covered in a related document [NegReq]. For a comprehensive set of requirements related to Payment Negotiation, please review that document. Here, we will express a few of those requirements that we feel are important and which we have addressed by our framework. We group these requirements under business, functional, and technical categories.

3.1 Business Requirements

The business requirements ensure the successful adoption of the payment negotiation mechanism by application developers and end-users. These dictate that, the negotiation mechanism must require the least amount of change from the existing systems; it must be easy and cheap to use and integrate with. In addition, the negotiation mechanism must facilitate payment system competition without stifling innovation. It should be convenient for customers and merchants to use (must particularly be easy to use for the customers). Finally, the use of the negotiation mechanism must ultimately drive down the cost of doing business

3.2 Functional Requirements

The functional requirements dictate that the negotiation mechanism must support all payment systems and mechanisms; provide for a smooth transition from pre- to post-negotiation (i.e., be well integrated); and be easy to use for both purchasers and merchants. In addition, the negotiation model must support both automated and manual selection.

3.3 Technical Requirements

We group the technical requirements into three sets: message format, protocol, and infrastructure requirements. The message format requirements dictate that the message format be extensible in order to support all payment systems. It must also allow for the conveyance of payment related auxiliary information. The messages must support mechanism to specify preferences and their strength (for example, optional, required, or refused). The message format must be transport independent (i.e., must be possible to carry it over as binary data as well as ASCII). As for the protocol requirements, the design of the payment negotiation must be symmetric, meaning that both the customer and the merchant can initiate the negotiation. It must also be transport mechanism independent, which in the case of a protocol means that it can be accomplished in both online and store-and-forward fashions. The protocol must also provide support for instances in which there may be proxies or intermediaries between the negotiating peers. The infrastructure requirements dictate the use of a framework such as GEPS in conjunction with the payment negotiation. This way, the negotiation module can take advantage of the integrated and extensible capability list management, and flexible preference management to accomplish its tasks.

4 Framework

4.1 Functionality

The Payment Method Negotiation (MN) service enables transacting peers to mutually agree on a common set of payment methods, protocols, systems or mechanisms, system providers, capabilities, accounts, and instruments. The service also enables the peers to select, among the negotiated choices, a particular payment system or mechanism to use to conduct commerce.

The MN service is generic enough to allow integration into a variety of applications and is extensible enough as to interoperate with a variety of payment systems. There are three reasons for this. First, the negotiation logic may occur at any time during an application process. For example, parties can negotiate just prior to a payment. Alternatively, parties can inquire into the other peer's capabilities at any time. For example, the customer and merchant may wish to determine if they share any payment capability before beginning the shopping process. This is to ensure that they will not fail to complete the payment transaction later and waste all the effort made during the shopping. Second, either party can initiate the negotiation resulting in a symmetric process. Either the customer or the merchant (client or server in a distributed application) could initiate the negotiation process. Third, the MN service is extensible and allows for the integration of all payment systems. The MN protocol uses extensible message syntax and can carry relevant information on all payment systems between peers.

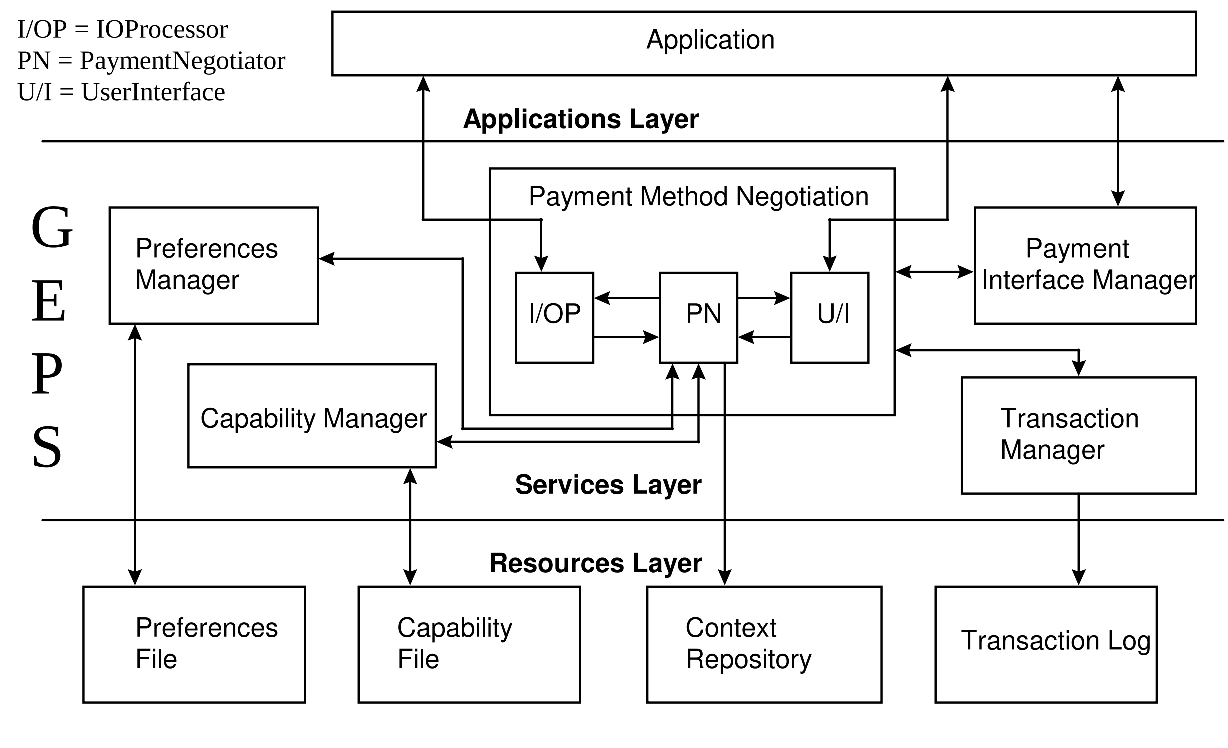
The MN has three components—IOProcessor (I/OP), PaymentNegotiator (PN), and UserInterface (U/I). The next section explains, in greater detail, the implementation of each component. Figure 1 depicts the MN framework and components. The framework calls for each of the transacting peers to have their individual installation of the MN service. In order to conduct the negotiation, the MN uses peer-to-peer communication between these two installations through exchanging tokens. The applications that use the MN service are responsible for the transport of these tokens. The content of the tokens is opaque to the applications.

The I/OP translates MN tokens into a representation that the application can transport and vice versa. There can be several implementations of this object leading to support for different transport mechanisms. Using this extensible mechanism, we have made the MN service transport independent. Section 5.2.1 provides an example of the use of this mechanism.

The PN maintains the state of the negotiation and implements the negotiation logic and policy. It is the primary object responsible for making the negotiation possible. The PN maintains and enforces the negotiation policy. Different implementations of PN are necessary to implement varying policies. There are four basic phases during the negotiation process. They are Query, Offer, Process, and Selection. In the Query phase, a party is asking the other for a listing of its preferred payment capabilities. In other words, it is asking the other party to list its acceptable payment mechanisms. Either party can initiate the negotiation by entering the Query phase. During the Offer phase, one party is listing its capabilities to the other party. A party can begin the Offer phase either as a result of a query, or voluntarily. Either party can initiate the negotiation by entering the Offer phase voluntarily. Note that in all cases, neither party is obliged to reveal all of their capabilities to the other. The policy implemented by the PN will dictate the level of openness in the exchange. During the Process phase, the parties will exchange messages back and forth in order to determine a set of common and acceptable payment capabilities. At the end, there may be one or more such capabilities identified. For each capability, the parties can negotiate acceptable usage criteria or hash specific details of that capability. For example, the parties might start negotiating on what capabilities they share that support the Credit payment method. Once both parties determine that they have such a capability, the parties might narrow the choices by listing the payment instruments used in a Credit transaction. This might further reduce the number of choices. Narrowing the choices in the negotiation might require some information other than those directly specific to payments (for example, profile information). For instance, one of the parties involved in the negotiation might need the residence country of the other to choose a payment mechanism. We achieve this by passing such information as additional details for the use of a payment capability. If there is no common payment mechanism or capability, the negotiation is over and the parties cannot complete the payment transaction. If there are several common capabilities, then the parties enter the Selection phase to select one of those choices as the means for payment. If only a single common payment capability is negotiated, the selection is automatic, but may still require end-user confirmation.

The U/I provides a consistent way to obtain end-user input when needed. The goal of the MN service is to make the negotiation process as automatic as possible (i.e., with minimum interactions required from the end-user). This is partly to avoid end-user confusion and

Figure 1- Payment Method Negotiation service in relationship to GEPS framework



partly for ease of use. The end-user does not always know all of the intricate details regarding payment capability choices. For example, while the end-user knows the difference between their Visa card and their Discover card, the end-user might not be aware of the differences between SET and CyberCash (two different payment protocols). It is easier for the end-user to make one choice at the Selection phase instead of being called upon several times during the Process phase. The U/I provides the end-user input, when required. In most cases, the Selection phase will require the final end-user input to complete the negotiation with the selection of a payment capability.

4.2 Relationship to GEPS

Figure 1 illustrates the schematic of the MN service and how it relates to the other modules in the Generic Electronic Payment Services [GEPS] framework. The MN service introduced here is an integral part of the GEPS framework. Both negotiating peers (the server and the client) will have a GEPS environment implemented. The input message to the PN is a negotiation token wrapped in a transport specific format. The I/OP converts the input into a negotiation token. The PN object processes the token using the preferences given by the Preference Manager over the list of payment capabilities generated by the Capability Manager. The PN may also need input from the end-user. In which case, the U/I opens an application level window (part of the Payment Interface Management's Wallet) to indicate the status to the end-user and/or receive input from him/her. The I/OP receives the output from the PN (a negotiation token), bundles it in a transport dependent form, and delivers it to the application for transmission. The Transaction Manager might log individual transaction messages and operations for later use in trace and audit trail operations.

5 Implementation

In this section, we present a more detailed look at the Payment Method Negotiation framework by introducing its programming building blocks. To present the Application Programming Interface (API) of the MN service, we describe the main classes and their methods.

5.1 Resources

The MN uses a set of data-types (NegotiationContext and NegotiationToken) in order to implement its services. These resources help in maintaining negotiation state and conveying negotiation messages.

5.1.1 NegotiationContext

The context repository stores an object of this type for every session. It contains information about the buyer and seller profiles, details of the transaction, list of payment capabilities (both rejected and accepted), and any additional payment related information. A session may include several transactions. Once the payment capability of choice has been negotiated, all transactions in that session can use the results. Alternatively, each transaction could initiate a negotiation, possibly building upon the results of any previous negotiation made in that session.

5.1.2 NegotiationToken

This object contains the negotiation information and information about each stage in the payment negotiation process. A representation of this object travels on the wire between the negotiating parties. Applications do not know anything about its content. Only the peers at the MN service level know how to process these tokens. Specifically, only the PN module uses these tokens. Each NegotiationToken has a *type* field that indicates what processing function can the recipient perform on that token. The valid types already defined for NegotiationTokens include: `Inquiry`, `Request`, `Intermediate`, `Final`, `Selected`, `Failed` and `Invalid`. The use of these types will become clear in Section 5.2 when we explain the processing functions. The I/OP module also is able to parse these tokens as it has to wrap (unwrap) them in (from) application specific format.

We define additional data types to hold transaction related details, such as payment amount information and party profiles. There are also utility functions that provide information about the NegotiationToken and NegotiationContext. For example, there will be a function to extract the final selected capability or the list of negotiated payment capabilities from a NegotiationToken (i.e., get_payment_capability and get_capability_list). See Appendix A for more details.

5.2 Processing

The MN service is part of the GEPS framework. From a structural perspective, it consists of three components—I/OP, PN, and U/I. Appendix B contains more details on each of the classes implementing these components.

5.2.1 IOProcessor (I/OP)

This component takes care of the interface between the applications and the MN tokens. Recall that the MN uses peer-to-peer communication by passing tokens. The application is responsible for transporting these tokens. The I/OP enables the wrapping and unwrapping of the NegotiationTokens generated by the MN into a format suitable for the applications to transport. The I/OP functionality includes receiving input messages from the transacting peer, converting them into NegotiationTokens, and embedding negotiation intentions from NegotiationTokens into outgoing messages. This is an abstract class, which could be subclassed to work with different transport protocols. Functions needed by the I/OP include:

to_external_form: converts the input token (NegotiationToken) into a message that is suitable for transport (TransportToken) to a remote peer by the calling application; and

to_internal_form: converts the input message (TransportToken) into a NegotiationToken that is suitable for internal processing.

The ability to subclass the I/OP provides the means to achieve transport independence for the MN service. We illustrate this by an example using a subclass of the I/OP intended for use within the HTTP transport and specifically using the Protocol Extension Protocol (PEP). We call this class HTTP-IOProcessor, to denote that it is a derived subclass of the I/OP.

In order to understand the example, we must first give a brief summary of the message formats in the PEP protocol. For detailed information, please refer to the documentation [PEP]. Essentially, all PEP messages are HTTP headers to be exchanged between a client and a server. There are four header types: Protocol, Protocol-Request, Protocol-Query, and Protocol-Info. The Protocol header indicates the extension protocol used in that message, while the Protocol-Request expresses the interest in a protocol in the response message. The Protocol-Query header allows one party to ask the other if it supports a particular extension; the response comes back in a subsequent message using the Protocol-Info header. The value for each of these PEP headers is one or more *bags*. The bag syntax is a curly parentheses-delimited *bagname* and *bagitem* pair. The bagname is a name of a protocol used in creating the bag. The bagitem can be one or more bags leading to a recursive structure. The bagitem could also be null or a token.

Using the PEP syntax, it is easy to construct a message containing all the information needed in any protocol (for example, a payment negotiation protocol). Specifically, the Universal Payment Preamble [UPP] is the vocabulary used to encode payment related information over PEP. This is indeed the approach taken by the Joint Electronic Payment Initiative (JEPI) project. In the E-CO System project, we have adopted that same syntax in encoding the necessary information for transport using HTTP-IOProcessor.

The HTTP-IOProcessor would have to parse HTTP headers using the PEP extension and translate the value of those headers into a NegotiationToken object when receiving data from the peer. When sending data to the peer, the reverse process is used (i.e., the NegotiationToken data are converted into bags suitable for transmission using the PEP headers).

For example, a sample input to the HTTP-IOProcessor (subclassed from I/OP) employed in an HTTP/PEP combined environment could be a header like:

Protocol-Request: {GEPS-MN Bag-A Bag-B Bag-C}

The output would be a NegotiationToken, representing the negotiation inputs {Bag-A} {Bag-B} {Bag-C}. The PN object processes this token. Similar data conversion happens when the input is a NegotiationToken; the token is converted into one or more PEP header(s) to be embedded in HTTP messages.

5.2.2 PaymentNegotiator (PN)

The PN is the primary component of the MN that is responsible for understanding the negotiation requests and generating appropriate responses. The PN creates a NegotiationContext and stores it with an appropriate handler in the context repository for every transaction session. Once assigned, the handler information travels in all future NegotiationTokens generated in that session. The PN receives a NegotiationToken as input, performs a series of steps, and outputs another NegotiationToken and a status value. As a side effect, the PN also updates the NegotiationContext. The status indicates if the negotiation is completed, failed, or needs to continue (i.e., COMPLETED, FAILED, and CONTINUE). Some of the functionality performed by the PN are enumerated below:

- Obtain the corresponding context from the database; if there is none, create a new context and assign it the handler from the NegotiationToken.
- Receive list of capabilities from the current context and NegotiationToken.
- Validate the capability list with the help of GEPS Capability Manager service.
- Process end-user's preferences with the help of GEPS Preference Manager service.
- Prompt the end-user, in case of difficulty processing negotiation decisions.
- Create an output NegotiationToken, representing negotiation intentions of the end-user.

There are five main methods of the PN which when called in different sequences allow applications to perform all possible payment negotiation scenarios. In Section 5.3, we explain these scenarios in greater detail. We explain these five methods below:

begin_negotiation_context: One party calls this function to create the initial NegotiationContext and NegotiationToken thereby beginning the negotiation dialog. There is no need for an input NegotiationToken to make this call. If succeeded, the function generates a NegotiationToken of type `Inquiry`, which does not contain any payment capability list. In the event of a failure, the output token type is `Failed`. The only valid status codes generated from a call to this function are FAILED or CONTINUE. The former indicates that the function failed to process the call, while the latter indicates that the negotiation must continue.

offer_negotiation_context: Another way to begin the payment negotiation dialog is with a call to this function. There is no need for an input NegotiationToken to make this call. If successful, it generates a NegotiationToken, of type `Request`, that contains a list of all valid payment capabilities that exist in the caller's GEPS environment. These capabilities may or may not be fully detailed yet. If this function fails, it returns a token of type `Failed`. Valid status codes generated are FAILED and CONTINUE. The application policy might dictate a degree of exposure that is more restricting than that enforced by the current implementation. In that case, a different implementation of the PN must be used.

process_negotiation_context: This is one of the most commonly used functions during the negotiation dialog. This function requires a valid NegotiationToken as input. Only tokens of type `Inquiry`, `Request`, and `Intermediate` are valid as inputs to this function. The function either delivers a list of applicable payment capabilities, or returns an error condition (if the transaction is not possible due to a lack of common payment capabilities). Valid status code returns are FAILED or CONTINUE. The output token is either of type `Intermediate` (meaning that the token includes a list of payment capabilities), or of type `Failed` (meaning that the negotiation has failed). The capability list included in the output token includes all of the payment capabilities of the input token (marked rejected or accepted possibly with additional detail). It also includes any additional payment capability the caller wishes to offer for negotiation. This function uses the NegotiationContext as well as the GEPS Capability Management (CM) service to process the list of capabilities. It uses the CM to reject or accept (possibly with additional detail) the payment choices. The NegotiationContext records the result of the negotiation; it is used to avoid duplicate processing by the CM (i.e., if a payment capability has already been rejected, it will not be further processed again). This function probes all local payment capabilities that have not been previously rejected to offer their capability for inclusion in the outgoing payment capability list of the output NegotiationToken. Since both parties maintain the negotiation context locally, there is no need to repeat any payment capability in the token's list (unless there are changes).

finalize_negotiation: The use of this function indicates the caller's intention to end the negotiation while still offering the recipient the opportunity to make the final selection. The recipient must respond by selecting one of the payment capabilities listed in the output NegotiationToken generated by this function or end the negotiation with a token of type `Failed` if none are acceptable. Valid input tokens are of type `Intermediate`. The resulting token is of type `Final` or `Failed`. The GEPS Preference Management service and the user's preferences help sort the capability list in the output token; the list includes all payment capabilities negotiated so far. The valid status codes generated as a result of this call are FAILED or CONTINUE.

accept_negotiation: Either peer uses this function to end the negotiation or respond to a finalize_negotiation call by the other peer. The function selects a common payment capability that has been negotiated so far. Only tokens of type `Final` or `Intermediate` are valid inputs. If successful, the output token is of type `Selected` and includes the selected payment capability. Otherwise, the token type is `Failed`. Valid status codes generated are COMPLETED or FAILED.

When necessary, the functions above will use the U/I module to prompt the end-user for some input to help make negotiation decisions. All functions process the input tokens using the corresponding negotiation context from the context repository and update the context as well. All functions also generate error messages if their processing fails.

5.2.3 UserInterface (U/I)

The U/I component allows the PN to receive input from the end-user when needed during the negotiation phases. It also ensures that there is a consistent interface to the end-user which ultimately leads to ease of use and acceptability of the MN framework. Not all applications could use the same graphical user interface. Therefore, we have specified the U/I module as an Interface, so that the applications can independently implement it providing their own unique look and feel. (In the GEPS framework, the Wallet in the Payment Interface Management service implements the U/I.) At the very minimum, however, all implementations must support the following functionality.

get_choices: Allows the end-user to identify a list of choices, possibly ordered according to their preference. For example, one can ask the end-user to place a check mark against all acceptable payment methods. The end-user might choose Cash and Credit. If ordering is allowed, the end-user might place the number one next to Cash and the number two next to Credit. This indicates that while the end-user would like to negotiate both Cash and Credit payments, the end-user prefers Cash. The Preference Management service of the GEPS framework can automate the ordering aspects.

select: This helps the end-user select a payment capability from a list of choices. This is used during the Selection phase. For the most part, this is not automated in order to empower the end-user. In some cases (as in small value transactions) it may be automated using the end-user's preferences and the Preference Management service of the GEPS framework.

fill_details: This is a mechanism to ask the end-user to fill in the details for a payment capability. For example, to determine the taxes on a Credit transaction one might require the shipping address. The Wallet functionality of the GEPS Payment Interface Management service can automatically provide most of the required data.

confirm: This is used to ask the end-user to confirm a payment choice or to authorize the payment transaction.

display_message: This is a generic functionality to display a message to the end-user.

add_system_messages: This function is used to display progress status to the end-user with visual effects. It could also be used for transaction logging.

Each of the methods presented above must allow the end-user to abort the transaction at that point. The application developers may provide additional methods to give the end-user more control over the negotiation process.

5.3 Message Flow

What we have described so far is the foundation and framework for the Payment Method Negotiation service. The resources (NegotiationContext and NegotiationToken) and the MN building components (the I/OP, PN, and U/I) presented here describe the implementation essence of MN. To help place things in perspective, it is instructional to see how two parties can negotiate and select a payment capability using these basic primitives. How these primitives support a variety of possible negotiation and selection scenarios? When can one use the U/I functionality? How do these primitives support the Offer, Query, Process, and Selection phases of negotiation? Finally, what is an example of the sequence of function calls in a typical negotiation and selection process?

We hope to be able to answer these and similar questions in this section by providing an example of a simple negotiation and selection between Alice and Bob. We discuss each stage of the process at length along the way. We strongly believe that specifying the message flow is essential to understanding the Payment Method Negotiation service (that is, we need more than just specifying the message syntax or the API).

There are numerous possible scenarios one could imagine between two parties, Alice and Bob. We describe one such message flow between Alice and Bob in stages (see Figure 2). Since the MN is symmetric, it is not important to distinguish between the party roles. Therefore, either Alice or Bob could be the customer (or the merchant). We assume they are both negotiating peers running the GEPS environment on their systems. They both have an implementation of the MN service as part of their GEPS implementation. They might each have different negotiation policies implemented. We also assume that each party has independently acquired and installed one or more payment capabilities on their system.

5.3.1 Stage One: Initiate Negotiation

Let's begin the example with Alice initiating the payment negotiation process. There are still several possible scenarios caused by two variables. The first variable is with respect to the length of the negotiation itself. Alice could just be curious and would like to know Bob's payment capabilities; that is, she is not at the point where she must pay Bob. She could wait until she reaches that point later, but she has decided to make sure early on that Bob is able to receive her payment. The difference is in the criticality of the time needed to complete the negotiation process. If one urgently needs the result, the number of messages they can afford to exchange is less than the case in which there is no urgency.

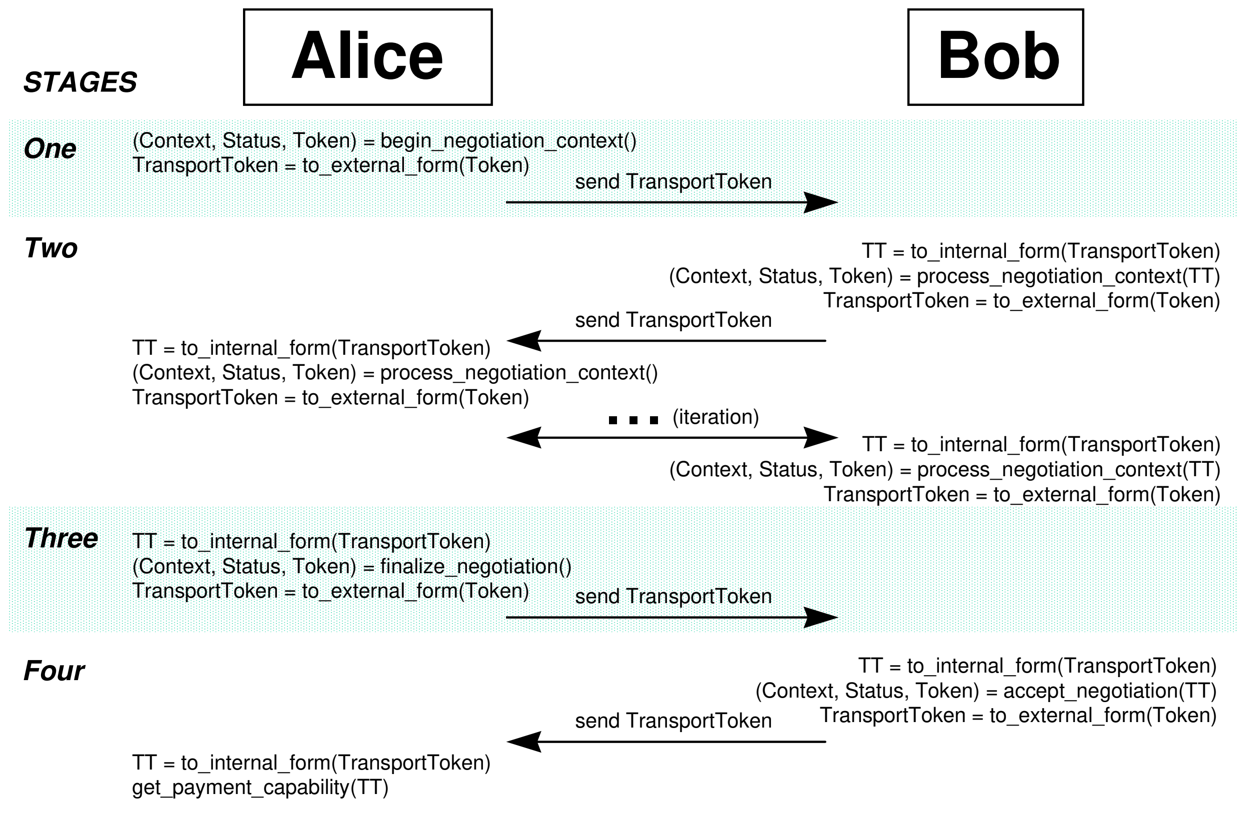
The second variable is with respect to the amount of information revealed by the party who starts the negotiation. Alice can initiate the process in one of several ways: she could offer all (or some of) her payment capabilities and ask Bob, which of those he supported; she could simply ask Bob what he supported without revealing any of her capabilities; and she could simply pick her favorite payment capability and offer it to Bob and wait and see what his response is. The function calls that can support these are offer_negotiation_context, begin_negotiation_context, and offer_negotiation_context, respectively. The difference is in the amount of information being revealed by the party who goes first.

The above scenarios are examples of the Offer and Query phases. In our example, we have chosen the scenario in which Alice asks Bob to reveal his payment capabilities without revealing any of her own. This is depicted as stage one in Figure 2. In our example, we assume that the negotiation is started well in advance of the actual payment. Therefore, there are no immediate limits on the number of the message exchanges that can be afforded.

5.3.2 Stage Two: Processing Negotiation

This stage corresponds to the Process phase. It could involve as many message exchanges as the two parties are willing to engage in. Each time, one party offers some or all of his/her capabilities for the first time; rejects any of the capabilities of the other party (offered in a previous message); or enumerates additional detail on a previously offered payment capability. Since each party maintains the negotiation context, it is not necessary for the parties to repeat themselves by resending already negotiated payment capabilities unless they have been changed. The negotiation messages can be piggybacked on other application communications that the parties might be engaged in. For example, the messages can be exchanged while a customer is browsing a web-based online catalog.

Figure 2- Sample Message Flow Between Negotiating Peers



In our example, this is the first opportunity to use the U/I in order to make the negotiation process more efficient. For example, the PN can more efficiently process the negotiation if it first found Alice's payment method of choice. Assuming that her choice is to pay with a credit card, one only needs to negotiate Credit payment capabilities. This avoids the time spent negotiating over other payment methods.

5.3.3 Stage Three: Finalizing Negotiation

This stage represents an attempt by one party, Alice, to put an end to the Process phase. However, she is still allowing Bob to make the final selection. Alice uses the finalize_negotiation function to stop the negotiation loop started in the previous stage. This still corresponds to the Process phase as neither party is making a selection. The ability for either party to call the finalize_negotiation function supports numerous negotiation scenarios empowering both parties.

One can also use the U/I in this stage to allow the end-user to approve the use of payment capabilities negotiated so far or even manually sort them in their preferred order. Most of this, of course, could be done automatically as well using the GEPS Preference Management service.

5.3.4 Stage Four: Selecting Payment

Another way to put an end to the negotiation process is to enter the Selection phase by calling accept_negotiation. This is either in response to a finalize_negotiation from the other party, or voluntarily. Either way, one party is taking the initiative to make a selection therefore ending the negotiation process. The ability for either party to call the accept_negotiation function supports numerous negotiation scenarios.

To allow the final selection, one would most likely use the U/I. Except for cases such as small value transactions, in which the preferences might dictate an automated selection. Also, of course, if the party making the call is a server and does not represent a human, the selection might take place without the help of the U/I.

The example above and the PN functions assume that there will be a number of messages exchanged between the parties during the Process phase and at least one for completing the Selection phase. In fact, there is no predefined maximum number of exchanges. If the application requires a minimum number of exchanges to process negotiation and selection, it could call the following sequence of function calls instead. Alice would have to initiate with an offer_negotiation_context function that fully describes her desired payment capability or capabilities. Bob follows by calling the process_negotiation_context function returning only the selected payment capability. In this example, both parties must be aware that the intention is to minimize the number of messages exchanged. To programmatically enforce and convey this behavior, Alice must instead begin the negotiation with a call to the finalize_negotiation. We have decided not to allow the use of finalize_negotiation as the first function called in the negotiation process. This is in order to avoid overloading the function with the initialization logic performed in the begin_negotiation_context and offer_negotiation_context function calls.

5.4 Policy Implementation

Another very important aspect of the payment negotiation to consider is the policy implementation. To fully describe the negotiation process, one must do more than just specify the message syntax, or the message flow. One must define the various policies that impact the message flow and to understand how to implement those policies. We devote this section to a discussion of the relevant policies and how and where in the system to implement them. There are a number of policy issues that we can identify. Table 1 summarizes the results. For each policy issue, it describes where to implement a solution—at the calling application level, at the PN level, or at the U/I level.

Overall, we believe in not following the complicated path of policy choices. That path will likely require end-user policy management as another GEPS service. We instead have decided to concentrate on a few limited policies enforced as different implementation of the PN and the U/I. This gives us a valuable experience on which to build future generic policy management services.

5.4.1 Negotiation Sequence

This issue deals with the sequence of negotiation functions being called by the application. This affects the length of the negotiation process and the end-points. Using the tokens' type field, we can encode part of the intended negotiation logic. For example, the token generated as a result of the finalize_negotiation call is of type `Final`. This ensures that the receiving party calls accept_negotiation on that token. The type field effectively controls what function to call in order to process what type of tokens. In addition to the token type field, the calling application must be aware of the negotiation process. Through making the function calls, an application can initiate, process, and terminate payment negotiation. For example, unless the application is aware of the negotiation process, the two parties could loop forever each calling the function process_negotiation_context. The token type is not sufficient to break the loop; the application must call either the accept_negotiation or the finalize_negotiation function. NegotiationContext maintains a counter that helps applications understand the negotiation progress. During every session, applications can use the counter to determine the length of the negotiation process.

5.4.2 Negotiation Efficiency

The goal of the negotiation process is to determine common payment methods, protocols, systems or mechanisms, system providers, capabilities, accounts, and instruments. To improve the efficiency of the negotiation, the peers can negotiate in that order instead of any random order. This is because the above order ensures that the parties will not waste time negotiating on payment capabilities that they do not have or wish to use. For example, assume that Alice wants to pay using her credit cards. Had the parties first agreed on the payment method, they would not have wasted time negotiating all common payment instruments and protocols of type Cash method. Even if they both agree on the payment method of Credit, they should not negotiate the payment instrument (i.e., what specific credit card) before they first negotiate a payment protocol to conduct the transaction in. If one party has the SET protocol implementation while the other has the CyberCash credit card payment, they will still not be able to perform the transaction. Both the GEPS Preference Manager and the MN's U/I are involved in allowing the end-user make his or her preferences known early. The application has the ultimate control in determining the order of the choices to negotiate.

5.4.3 Exposure Control

By making an offer, each party is revealing their payment capabilities. It might be against one's policy to reveal all their payment capabilities at any time. Either party might want to reveal the minimum amount of required information. Also the party who starts first, will be the first to reveal their capabilities. There are two control mechanisms implemented with respect to this issue resulting in different policies. First, either party can decide to be the first to reveal by calling the offer_negotiation_context function. Second, using

Table 1 - Policy considerations and where in the framework they are implemented and/or enforced

Policy	Calling Application	PaymentNegotiator	UserInterface
Negotiation Sequence	■ Sequence and order of function calls ■ Terminate negotiation loop	■ Token type field enforces use of functions ■ Counter maintained in the NegotiationContext	—
Negotiation Efficiency	■ Specify the order of negotiating choices when making a function call	■ Follow orders provided by the calling application	■ Ask payment method first if not specified in the preferences
Exposure Control	■ Choice of revealing first by calling offer_neg… instead of begin_negotiation_context	■ Different implementations of PN provide various exposure policies: e.g., all at once or gradual	—
Preference Ordering	■ Specify preference criteria used ordering	■ Order by preference criteria ■ Sort using GEPS Preference Management	■ When preference not specified, ask end-user
Solicit Input	■ Application implements the U/I and controls policy	■ Call U/I functions for all end-user input	■ U/I implemented by the application

different implementations of the PN functions, one can control the amount of exposure. Some will reveal all choices when probed, while other implementations might hold back on some information and apply a gradual release of information.

We also considered the idea of maintaining *filters* by the PN. The applications implement these filters and use them to monitor and control the exposure level. For example, a filter might remove half of the items in a token's payment capability list before passing that token to the other peer. By implementing the filters, the applications could enforce all the policies they would like. We have decided that, at least initially, the use of different implementations of the PN would be sufficient to achieve the same objectives while being more realistic.

5.4.4 Preference Ordering

At any instance during the negotiation, one party is offering a list of choices to the other party. The issue is whether or not to require that the list be sorted based on end-user preferences? If sorted, the process is guaranteed to meet the end-user preferences faster. This is because, the negotiation could first determine the common choices that are preferred before spending time on less preferred options. To enable this we require that the calling application specify the ordering by indicating the preference criteria. The PN uses the GEPS Preference Management service to sort the capability list using those criteria. The U/I is responsible to obtain preference orderings when such information is not stored in the GEPS preference files.

5.4.5 Soliciting Input

The issue of when to ask the end-user for input is a complicated one. Probably there are no two applications that share the exact policy. Nonetheless we want to enable minimum conformance by implementing a user interface that is consistent and yet allows flexible application design. To do this, we have specified the U/I as an Interface with minimum functionality requirements. The calling application is required to implement the U/I and can therefore enforce its own interface.

5.5 Error Handling and Reporting

Several types of errors may occur while the PN is processing the input tokens. There are several ways to handle errors. First, a signal corresponding to the error is sent to the parent application upon occurrence of the error. Second, an exception is thrown upon an occurrence of an error and caught by the application that later consults an exception table to perform the recovery. Third, error related information is included within the token, as a status code; an application can read the status code from the token object and invoke a handler for that error type. Errors and their corresponding recovery steps are stored in a static error handler table, which is visible to all parts of the application and payment framework. Common errors that might occur include:

- Requested payment method is not available.
- Payment method is available, but no payment capability is installed which supports it.

- Payment method is available, but no available payment protocol supports it.
- Payment method, capability, and protocol are supported, but payment instrument is not accepted.
- The syntax of the token is malformed.
- The end-user performs abort, cancel, or retry operations.
- Other runtime errors that may occur.

When the application receives signal of a kind or an exception of a type, then it has to perform the sequence of steps specified in the global error table to recover from the error.

6 Security Consideration

There are two types of possible security attacks: denial of service attacks by corrupting the messages exchanged between the two parties and disclosure of capabilities resulting from eavesdropping the communications. Using a secure channel, it is possible to curtail these attacks. However, a malicious party can still cause the denial of service by interrupting the flow of messages between the two parties.

If the confidentiality of the payment capabilities is of paramount importance, then the parties could establish a secure channel over which they could conduct their communications. This way, the negotiation occurs over the secure channel and is not subject to eavesdropping.

7 Related Work

The payment negotiation framework presented in this paper supports a variety of negotiation and selection scenarios required by applications today. To the best of our knowledge it is the only work in this area addressing the complete range of issues including generic framework, common user interface, transport independence, end-user preferences, and generic programming interface. We intend to implement the service presented here as part of the Generic Electronic Payment Services [GEPS] framework. The U/I and I/OP components provide the common user interface and transport independence. The GEPS Preference Management service maintains the end-users' preferences and provides them for use by the Payment Method Negotiation service. While we have not specified a complete set of APIs in this paper, we have indicated the main set of required functions.

There is only one other related work that attempts to address part of the payment negotiation space. That is the work performed in the Joint Electronic Payment Initiative [JEPI] project sponsored jointly by CommerceNet Consortium and the World Wide Web Consortium. We participated in JEPI during the first six month of this year as a cofounder and member of the design team. We made sure that our work builds on top and over the work already started by that team. The JEPI team is primarily concerned with specifying the syntax of the negotiation messages between the parties. They use UPP and PEP to form the messages and transport them. Our work uses that syntax but does more by building a supportive framework around it.

An implementation of the Payment Method Negotiation service has begun at the E-CO System project. You can access our [WebSite] for additional information about the project.

8 Conclusion

The excitement surrounding electronic commerce has created a lot of creative solutions for electronic payments. Transacting peers require a solid framework with which to negotiate and select among these choices. We believe that our Payment Method Negotiation service provides such a framework. Our service supports all phases of negotiation as specified in Section 4.1. Along with the GEPS, these services represent a complete set using which one can implement any application using electronic payments. Only time will tell if the implementation of all the functionality is possible. However, based on our early findings and the result of our implementation so far, all indications are in favor of such achievements.

We are addressing the technical challenges presented by the introduction of a variety of electronic payment solutions. However, there are other non-technical hurdles one has to overcome in order to successfully advocate the MN framework. These hurdles are related to the business acceptance of such a framework and the incorporation of such a framework into industry products. Currently, most businesses engage in competitive developments that prevents them from adopting open frameworks such as that advocated within this paper. In addition, perhaps it is too early in the development stages of the electronic payment industry to know enough what the future will hold and what framework is the right all encompassing and open model. We are positioning our work to address some of these issues, but only time will tell if we can achieve the industry acceptance required to make the framework ubiquitous.

Acknowledgments

The author would like to acknowledge the dedication and team work of the E-CO System project members without whom the implementations and many aspects of this work would not have been possible. They are Rajkumar Narayanaswamy, James Galvin, Andrea Stirrup, and Nick Zhang.

In writing this document we reviewed the ideas presented in existing work. These include the Protocol Extension Protocol (PEP), Universal Payment Preamble (UPP) work, as well as any material available from the mailing lists of the Joint Electronic Payment Initiative (JEPI).

In closing, many thanks belong to Donald Eastlake of CyberCash. The author worked closely with him while in the JEPI Design Team and has profited from many of his insights and fruitful discussions.

References

[WebSite] See the E-CO System Project website at http://eco.eit.com.

[GEPS] Alireza Bahreman, "Generic Electronic Payment Services: Framework and Functional Specification." Proceedings of the Second USENIX Electronic Commerce Workshop, November 1996.

[NegReq] Alireza Bahreman, "Electronic payment negotiation mechanism: Requirements document." VeriFone, January 9, 1996. Available at: http://eco.eit.com/negotiation/NR.html.

[PEP] Rohit Khare, "HTTP/1.2 Extension Protocol." Internet Draft <draft-ietf-http-pep-00>, August 1996.

[UPP] Donald E. Eastlake 3rd, "Universal Payment Preamble." Internet-Draft <draft-eastlake-universal-payment-03.txt>, August 1996.

[JEPI] See http://www.w3.org/pub/WWW/Payments and www.commerce.net/work/taskforces/payments/jepi.html.

Contact Information

You can reach the authors at:

Alireza Bahreman & Rajkumar Narayanaswamy
bahreman@eit.com & rajkumar@eit.com
EIT/VeriFone, 530 Lytton Avenue
Palo Alto, CA 94301-1539, USA
TEL: +1 415-617-9730; FAX: +1 415-617-9746

Appendix A - Payment Method Negotiation Service Resources, Draft 0.1

Please note that this is a work in progress and therefore, the specifications might change.

```
class NegotiationContext {
    •  Buyer/Seller Profile Information
    •  TransactionDetails
    •  Payment Capability List
    •  Additional Payment Information
    •  Negotiation Loop Counter
    •  Utility functions such as:
       create_negotiation_context()
       update_negotiation_context()
       delete_negotiation_context()
       get_peer_information()
       get_negotiation_loop_count()
}

class NegotiationToken {
    •  Negotiation Status Code
    •  TransactionDetails
    •  Sender/Receiver Profile Information
    •  Payment Capability List
    •  Additional Payment Information
    •  Utility functions such as:
       get_payment_capability()
       get_capability_list()
}

class PartyProfile {
    •  Hashtable of name, value pairs with
       information about the following:
    •  name and identification details
    •  affiliation
    •  residence country
    •  personal information
    •  other relevant information
    •  Utility functions
}

class TransactionDetails {
    •  Transaction Handler
    •  Price/Amount Details
    •  Business Information
    •  Miscellaneous parameters
    •  Utility functions
}
```

Appendix B - Blueprint for Classes and Functions in GEPS Payment Method Negotiation Service, Draft Version 0.1

Please note that this is a work in progress and therefore, the specifications might change.

```
abstract class IOProcessor {      // implement subclasses for each different transports

   public abstract TransportToken   to_external_form();
   public abstract NegotiationToken to_internal_form();
}

public class PaymentNegotiator {            // different version for different policies

   public NegotiationToken begin_negotiation_context() {
      // similar to offer_negotiation_context except no capability list is formed
      // and output token type is Inquiry
   }

   public NegotiationToken offer_negotiation_context() {
      // Input: Caller preferred negotiation order and preference criteria for sorting
      // Output: NegotiationToken of type Request, Status, and NegotiationContext
      //          (if there already, the NegotiationContext is updated)

      - Get list of capabilities based on caller preferred negotiation order
        (i.e., payment method first, then payment protocol, and so on)
        use GEPS Capability Management that probes all payment capabilities installed
      - Sort them using caller's preference criteria and GEPS Preference Management
      - If NegotiationContext exists update it, otherwise create and update
      - Create new token holding the capability list with type Request
      - If successful, set status as Continue and return token
      - Otherwise, set status and token type to Failed
   }

   public NegotiationToken process_negotiation_context() {
      // Input: Token of type Inquiry, Request, or Intermediate
      //        Caller preferred negotiation order and preference criteria for sorting
      // Output: NegotiationToken of type Request, Status, updated NegotiationContext

      - If token of different type, return error
      - Get list of payment capabilities from the input token and NegotiationContext
      - Use GEPS Capability Management to validate all new offers
      - Reject any capability which does not meet needs or has been rejected before
      - Update NegotiationContext with new offers
      - Get list of new capabilities based on caller preferred negotiation order
        (i.e., payment method first, then payment protocol, and so on)
      - Create list of new offers and rejected offers and updates capabilities
      - Sort the list using caller's preference criteria and GEPS Preference
        Management if desired
      - Create token of type Intermediate that includes the new capability list
      - Return the token
   }

   public NegotiationToken finalize_negotiation() {
      // similar to process_negotiation_context except output token is of type Final
      // and input token is of type Intermediate
   }

   public NegotiationToken accept_negotiation () {
      // Input: Token of type Intermediate, or Final
      //        Caller preferred negotiation order and preference criteria for sorting
      // Output: NegotiationToken of type Selected, Status, updated NegotiationContext

      - If token of different type, return error
      - Get list of payment capabilities from the input token and NegotiationContext
      - Use GEPS Capability Management to validate all new offers
      - Reject any capability which does not meet needs or has been rejected before
      - Update NegotiationContext with new offers
      - Get list of new capabilities based on caller preferred negotiation order
        (i.e., payment method first, then payment protocol, and so on)
      - Create list of new offers and rejected offers and updates capabilities
      - Sort the list using caller's preference criteria and GEPS Preference
```

```
         Management if desired
       - Use GEPS Preference Management to select one of the payment capabilities
       - If succeeded, return token of type Selected including the payment capability
       - If failed, throw an exception and set status and token type to Failed
    }

}

public interface UserInterface {  // Uses Device Objects to interact with the end-user

    // For manually selecting a list of choices
    public PaymentBagArray get_choices(PaymentBagArray list) {
       DeviceHolder dh = new DeviceHolder(list, "SelectionDevice");
       dh.waitForResult();
       return dh.getAnswer();
    }

    // To ask the end-user to select a specific capability
    public PaymentBag select(PaymentBagArray list) {
                    DeviceHolder cd = new DeviceHolder(list,  "CheckboxDevice");
                    cd.waitForResult();
                    return cd.getAnswer();
    }

    // To ask the end-user to fill in the details manually
    public PaymentBag fill_details() {
       DeviceHolder fill = new DeviceHolder(null, "FillDevice");
       fill.waitForResult();
       return (PaymentBag fill.getAnswer());
    }

    // To ask the end-user to confirm the current selection or proceedings of the
    negotiation
    public String confirm(String question) {
       DeviceHolder dh = new DeviceHolder(question, "ConfirmationDevice");
       dh.waitForResult();
       return (String )dh.getAnswer();
    }

    // display some message to the end-user
    public void display_message(String mesg) {
       DeviceHolder dd = new DeviceHolder(mesg, "DisplayDevice");
       dd.waitForResult();
       return;
    }

    // add to the visual event viewer of the process
    public void add_system_messages(String mesg) {
                 messages_area.add(mesg);
    }
}
```