Voucher Trading System Application Programming Interface (VTS-API)

Status of This Memo

Copyright Notice

IESG Note

   This document is not a candidate for any level of Internet Standard.
   This document specifies the Voucher Trading System Application
   Programming Interface (VTS-API), which assumes that the VTS plug-in
   is trusted by its user. The application making calls to VTS-API
   ought to authenticate the VTS plug-in and securely bind the plug-in
   with the VTS provider information specified in the Voucher Component.
   However, this document does not specify an approach to application
   authentication. The VTS-API should not be used without being
   augmented by an application authentication mechanism.

Abstract

   This document specifies the Voucher Trading System Application
   Programming Interface (VTS-API). The VTS-API allows a wallet or
   other application to issue, transfer, and redeem vouchers in a
   uniform manner independent of the VTS implementation. The VTS is a
   system for securely transferring vouchers; e.g., coupons, tickets,
   loyalty points, and gift certificates. This process is often
   necessary in the course of payment and/or delivery transactions.

Table of Contents

1. Introduction

   This document specifies the Voucher Trading System Application
   Programming Interface (VTS-API). The motivation and background of
   the Voucher Trading System (VTS) are described in Requirements for
   Generic Voucher Trading [VTS].

   A voucher is a logical entity that represents a certain right, and it
   is logically managed by the VTS. A voucher is generated by the
   issuer, traded among users, and finally collected using VTS. The
   terminology and model of the VTS are also described in [VTS].

   VTSes can be implemented in different ways, such as a centralized
   VTS, which uses a centralized online server to store and manage all
   vouchers, or a distributed VTS, which uses per-user smartcards to
   maintain the vouchers owned by each user. However, the VTS-API
   allows a caller application to issue, transfer, and redeem vouchers
   in a uniform manner independent of the VTS implementation. Several
   attempts have been made to provide a generic payment API. Java
   Commerce Client [JCC] and Generic Payment Service Framework [GPSF],
   for example, introduce a modular wallet architecture that permits
   diverse types of payment modules to be added as plug-ins and supports
   both check-like/cash-like payment models. This document is inspired
   by these approaches but its scope is limited to the VTS model, in
   which the cash-like payment model is assumed and vouchers are
   directly or indirectly transferred between the sender (transferor)
   and receiver (transferee) via the VTS. This document is not intended
   to support API for SET, e-check, or other payment schemes that do not
   fit the VTS model.

   Unlike the APIs provided in JCC and GPSF, which are designed to
   transfer only monetary values, this API enables the transfer of a
   wide range of values through the use of XML-based Generic Voucher
   Language [GVL]. The monetary meaning of the voucher is interpreted
   by the upper application layer using the information described in the
   language. This approach makes it possible to provide a simpler API
   in the voucher-transfer layer and enhances runtime efficiency. The

API specification in this document is described in the Java language
syntax. Bindings for other programming languages may be completed in
a future version of this document or in separate related
specifications.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in [RFC2119]

2. Processing Model

This section provides the processing model in which the VTS-API is
used. A part of the text in this section has been taken from the
Generic Voucher Language specification [GVL].

There are several ways to implement VTS. For discount coupons or
event tickets, for example, a smartcard-based distributed offline VTS
is often preferred, whereas for bonds or securities, a centralized
online VTS is preferred. While distributed VTSes would utilize
public (asymmetric) key-based or shared (symmetric) key-based
cryptographic challenge-and-response protocols to trade vouchers
securely, centralized VTSes would utilize transactions that rewrite
ownerships of vouchers on their database. Therefore, it is
impractical to define standard protocols for issuing, transferring,
or redeeming vouchers at this time.

To provide implementation flexibility, this document assumes a
modular wallet architecture that allows multiple VTSes to be added as
plug-ins. In this architecture, instead of specifying a standard
voucher transfer protocol, two specifications, Voucher Component and
VTS-API, are standardized (Figure 1).

```
Sender wallet/Issuing system     Receiver wallet/Collecting system
+------------------------+       +--------------------------+
|                        |       |                          |
| |                 Voucher Component                    | |
| |            (Specifies VTS Provider and Promise)      | |
| |--------------------------------------------------------->| |
| |                      |       |                       | |
| |           Intention to receive and payment (option)  | |
| |<- - - - - - - - - - - - - - - - - - - - - - - - - - - | |
| |                      |       |                       | |
| |                      |       |                       | |
| |  Issue/transfer/ VTS |       | VTS      Register     | |
| |  redeem request  plug-in     | plug-in  Listener(*1) | |
| |----------------->|   |       | |<------------------  | |
| |  (VTS API)       |<- - - - - ->|          (VTS API)  | |
| |                  | VTS-specific |                    | |
| |                  | protocol if VTS |                 | |
| |                  | is distributed  |                 | |
| |  Result          |<- - - - - ->|      Notify(*2)     | |
| |<-----------------|   |       | |------------------>| |
+------------------------+       +--------------------------+
```
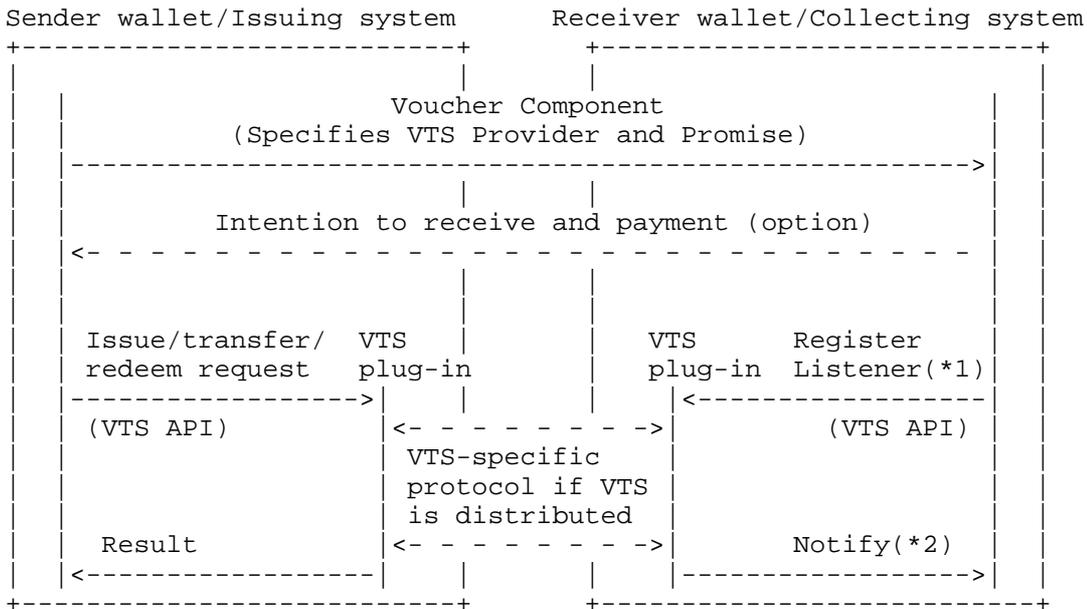
(*1) Registration is optional. Note also that the VTS plug-ins are
     usually pre-registered when the wallet or collecting system
     is started.

(*2) If a listener is registered.

              Figure 1. Wallet architecture with VTS plug-ins

In this architecture, a VTS provides a logical view of vouchers
called a Valid Voucher Set (VVS), which is a set that includes the
vouchers <I,P,H> managed by the VTS [VTS]. A user's wallet can
access (e.g., view, transfer, and redeem) the subset of the VVS that
includes a set of vouchers owned by the user by interacting with the
VTS plug-in via the VTS-API. Likewise, an issuing system can issue a
voucher and add it to the VVS, and a collecting system can be
notified of the redemption of vouchers via the VTS-API.

After a sender and a receiver agree on what vouchers are to be traded
and which VTS is to be used, the issuing system or wallet system
requests the corresponding VTS plug-in to permit the issue, transfer,
or redemption transactions to be performed via the VTS-API. The VTS
then logically rewrites the ownership of the vouchers on the VVS
using the VTS-specific protocol. Since the VTS is responsible for
preventing illegal acts on vouchers like forgery or reproduction, as
required in [VTS], the protocol would include a cryptographic
challenge-and-response (in a distributed VTS) or a transactional

database manipulation with adequate access controls (in a centralized
VTS). Finally, a completion event is sent to the wallet systems or
issuing/collecting systems.

This document describes the VTS-API specification. See [GVL] for the
Voucher Component specification that gives the syntax and semantics
for describing and interpreting the meaning of vouchers.

3. Design Overview

We have adopted the following approach to specify the VTS-API.

     1) Provide an abstract and uniform API that encapsulates the VTS
        implementation. For example, a common API is provided for both
        centralized and distributed VTSes. Issuers and application
        developers have more freedom in VTS selection.

     2) To provide an abstract and uniform API, this document
        introduces an interface called VTSAgent that is associated with
        a holder and provides methods to manipulate vouchers held by
        its holder. Vouchers are accessed through the methods provided
        by the VTSAgent.

     3) Use existing standards for the VTS branding mechanism
        (negotiation). This document assumes that the VTS to be used
        for sending a voucher has settled the VTS-APIs are called.
        Negotiation can be done within the upper application layer
        using other standards (e.g., [IOTP] or [ECML]), if necessary.

     4) Support only the push-type voucher transfer interface, in which
        the voucher transfer session is initiated by the transferor
        side. A pull-type voucher transfer interface can be
        implemented on top of the push-type VTS interface at the
        application level.

4. Concepts

The VTS-API consists of the following interfaces. A VTS is required
to implement all of the interfaces except ReceptionListener, which is
intended to be implemented by wallets or other applications that use
VTS.

     VTSManager
        Provides the starting point for using a VTS plug-in. All of
        the objects needed to manipulate vouchers can be directly or
        indirectly acquired via the VTSManager. A VTSManager maintains
        the two repositories: a ParticipantRepository and a
        VoucherComponentRepository, both of which are described below.

ParticipantRepository
     Provides the access points of participants that are to be
     trading partners. A ParticipantRepository maintains
     Participants and acts as an "address book" of trading partners.

Participant
     Represents a participant (such as an issuer, a holder, or a
     collector). A Participant interface knows how to obtain the
     corresponding VTSAgent described below.

VTSAgent (extends Participant)
     Provides the access point of vouchers in the Valid Voucher Set
     (VVS) that is logically managed by the VTS. A VTSAgent
     provides a means of manipulating vouchers held by its holder
     according to basic trading methods; i.e., issue, transfer,
     consume, and present. Before calling trading methods, the
     application must create a Session, which is described below.

Session
     Represents the logical connection established by the trade. A
     Session has references to two Participant interfaces; i.e.,
     those of the sender and the receiver. After trading methods
     are called using a Session, the Session holds a reference to
     the Vouchers to be traded.

Voucher
     Represents one or more vouchers in which all of the issuer and
     promise parts of the vouchers are the same. A Voucher holds
     references to the Participant interface who issued the voucher
     (issuer) and to a VoucherComponent (promise), which is
     described below.

VoucherComponent
     Represents a Voucher Component, described in [GVL]. It defines
     the promise part of the voucher.

VoucherComponentRepository
     Provides the access points of VoucherComponents. A
     VoucherComponentRepository maintains VoucherComponents and acts
     as a "voucher type book" managed by the VTS. This document
     assumes that a set of VoucherComponents has been acquired and
     stored in this repository. Delivery of VoucherComponents is
     beyond the scope of this document. It may be delivered within
     the VTS from the trading partners or manually acquired from a
     trusted third party (see Section 3 of [GVL]).

ReceptionListener
    Provides a listener function with regard to the receipt of a
    voucher by a VTSAgent to wallets or other applications that
    implement this interface. (This interface may not be
    implemented as part of the VTS.)

5. Interface Definitions

   The interfaces defined in this document reside in the package named
   "org.ietf.vts". Wallets or other applications that use this API,
   should import this package as "import org.ietf.vts.*;".

5.1. VTSManager

   public interface VTSManager

      Provides the starting point for using a VTS plug-in.

      All of the objects needed to manipulate vouchers can be directly
      or indirectly acquired via a VTSManager so that wallets or other
      applications can make the VTS available by instantiating an object
      implementing this interface.

      A class that implements the VTSManager interface must have a
      public default constructor (a constructor without any parameters).
      The VTS provides a name for such a constructor so that the
      implementation class can bootstrap the interface.

5.1.1. getParticipantRepository

   public ParticipantRepository getParticipantRepository()

      Returns a repository that maintains Participants.

   Returns:

      the ParticipantRepository of the VTS, or null if no
      ParticipantRepository is available.

5.1.2. getVoucherComponentRepository

   public VoucherComponentRepository getVoucherComponentRepository()

      Returns a repository that maintains VoucherComponents.

   Returns:

      the VoucherComponentRepository of the VTS, or null if no
      VoucherComponentRepository is available.

5.2. ParticipantRepository

   public interface ParticipantRepository

      Provides the access points of Participants. A
      ParticipantRepository maintains Participants and acts as an
      "address book" of trading partners.

      The object implementing this interface maintains Participants (or
      holds a reference to an object maintaining Participants), which
      are to be trading partners.

      The implementation of a ParticipantRepository may be either (an
      adaptor to) "yellow pages", which is a network-wide directory
      service like LDAP, or "pocket address book", which maintains only
      personal acquaintances.

5.2.1. lookup

   public Participant lookup(String id)

      Retrieves the participant that has the specified id.

   Returns:

      the participant associated with the specified id, or null if the
      id is null or the corresponding participant cannot be found.

5.3. Participant

   public interface Participant

      Represents the participants (such as issuers, holders, and
      collectors).

      This interface is used as a representation of the trade partners
      and issuers of vouchers. Anyone can retrieve objects that
      implement Participants from the participant repository.

5.3.1. getIdentifier

   public String getIdentifier()

      Returns the identifier of the participant. Each participant must
      have a unique identifier.

      The identifier can be used for looking up and retrieving the
      participant via the ParticipantRepository.

      The format of the identifier is implementation-specific.

   Returns:

      the identifier string of the participant.

5.3.2. getVTSAgent

   VTSAgent getVTSAgent()

      Returns a VTSAgent, whose identifier is the same as the identifier
      of the participant.

   Returns:

      an object that implements the VTSAgent.

5.4. VTSAgent

   public interface VTSAgent extends Participant

      Represents contact points to access vouchers in a Valid Voucher
      Set (VVS) that is managed by the VTS.

      Each VTSAgent is associated with a holder and provides a means for
      managing vouchers owned by the holder. The holder must be
      authenticated using the login() method before being called by any
      other method, otherwise, a VTSSecurityException will be issued.

      Before any trading method is called, e.g., issue(), transfer(),
      consume(), and present(), the application must establish a session
      by the prepare() method.

      Due to network failure, sessions may often be suspended when the
      voucher is sent via a network. The suspended sessions can be
      restarted by the resume() method. Details on the state management
      of a session are described in Section 5.5.

Some VTSAgents may not have all of the trading methods; a voucher
collecting system doesn't require its VTSAgent to provide a method
for issuing or creating vouchers. A VTSAgent returns a
FeatureNotAvailableException when an unsupported method is
invoked.

5.4.1. login

   public void login(String passphrase)
         throws VTSException

   Authenticates the VTSAgent. The passphrase is specified if the
   VTS requires it for authentication, otherwise it must be null.
   Nothing is performed if the VTSAgent has already been logged-in.
   The authentication scheme is implementation-specific. Examples of
   the implementation are as follows:

   1) Vouchers are managed on a remote centralized server
      (centralized VTS), which requires a password to login. In this
      case, the application may prompt the user to input the password
      and the password can be given to the VTSAgent through this
      method. For further information, see the Implementation Notes
      below.

   2) Vouchers are managed on a remote centralized server
      (centralized VTS), which requires challenge-and-response
      authentication using smartcards held by users. In this case,
      the passphrase may be null because access to the smartcard can
      be done without contacting the application or user (i.e., the
      VTSAgent receives the challenge from the server, sends the
      challenge to the smartcard (within the VTS), and returns the
      response from the smartcard to the server). Note that a PIN to
      unlock the smartcard may be given through this method,
      depending on the implementation.

   3) Each user holds their own smartcard in which their own vouchers
      are stored (distributed VTS). In this case, the passphrase may
      be null because no authentication is required. Note that a PIN
      to unlock the smartcard may be given, though this depends on
      the implementation.

   Implementation Notes:

      A VTS is responsible for providing secure ways for users to
      login(). It is strongly recommended that secure communication
      channels such as [TLS] be used if secret or private information
      is sent via networks. Fake server attacks, including the so-
      called MITM (man-in-the-middle), must be considered as well.

   Throws:

      VTSSecurityException - if authentication fails.

5.4.2. logout

   public void logout()
         throws VTSException

      Voids the authentication performed by the login() method.

      After this method is called, calling any other method (except
      login()) will cause a VTSSecurityException.

      The VTSAgent can login again by the login() method.

   Throws:

      VTSSecurityException - if the VTSAgent is not authenticated
      correctly.

5.4.3. prepare

   public Session prepare(Participant receiver)
         throws VTSException

      Establishes a session that is required for trading vouchers. The
      trading partner who receives the vouchers is specified as the
      receiver. The vouchers to be traded will be specified later (when
      a trading method is called).

      The establishment of a session is implementation-specific. A
      centralized VTS implementation may start a transaction, while a
      distributed VTS implementation may get the challenge needed to
      create an authentic response from the receiver in the following
      trading method.

      If the VTSAgent does not have the ability to establish a session
      with the specified receiver (permanent error), the VTSAgent throws
      an InvalidParticipantExeption. If the VTSAgent cannot establish a
      session due to network failure (transient error), the VTSAgent
      throws a CannotProceedException.

   Parameters:

      receiver - the trading partner who receives vouchers.

   Returns:

      an established session whose state is "prepared" (see Section
      5.5).

   Throws:

      CannotProceedException - if the preparation of the session is
         aborted (e.g., network failures).

      FeatureNotAvailableException - if the VTSAgent does not provide
         any trading methods.

      InvalidParticipantException - if the specified participant is
         invalid.

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.4. issue

   public void issue(Session session,
                     VoucherComponent promise,
                     java.lang.Number num)
         throws VTSException

      Issues vouchers. This method creates the specified number of
      vouchers <this, promise, receiver> and adds them to the VVS. If
      the VTS is distributed, this method would create a "response" that
      corresponds to the challenge received in the prepare() method and
      send it to the receiver. Note that the receiver is specified when
      prepare() is called. Nothing is performed if the specified number
      is 0.

      The session MUST be "prepared" when calling this method. The
      state of the session will be "activated" when the vouchers are
      created, and it will be "completed" when the transaction is
      successfully completed or "suspended" if the transaction is
      interrupted abnormally (e.g., network failures).

   Parameters:

      session - the session used by the issue transaction.

      promise - the promise part of the voucher.

      num - the number of vouchers to be issued.

   Throws:

      CannotProceedException - if the transaction cannot be successfully
         completed.

      FeatureNotAvailableException - if the VTSAgent does not provide a
         means of issuing vouchers.

      InvalidStateException - if the session is not "prepared".

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.5. transfer

   public void transfer(Session session,
                        Participant issuer,
                        VoucherComponent promise,
                        java.lang.Number num)
         throws VTSException

      Transfers vouchers. This method rewrites the specified number of
      vouchers <issuer, promise, this> to <issuer, promise, receiver> in
      the VVS; i.e., deletes the vouchers from the sender and stores
      them for the receiver. Similar to issue(), this method would
      create and send the response to the receiver if the VTS is
      distributed. The VTSAgent must have sufficient vouchers in the
      VVS. Nothing is performed if the specified number is 0.

      The session MUST be "prepared" when calling this method. The
      state of the session will be "activated" when the voucher are
      retrieved from the sender, and it will be "completed" when the
      transaction is successfully completed or "suspended" if the
      transaction is interrupted abnormally (e.g., network failures).

      If null is specified for the issuer parameter, it indicates "any
      issuer". This method selects vouchers to be transferred from the
      set of vouchers returned by the getContents(null, promise).

Parameters:

   session - the session used by the transfer transaction.

   issuer - the issuer part of the voucher, or null.

   promise - the promise part of the voucher.

   num - the number of vouchers to be transferred.

 Throws:

   CannotProceedException - if the transaction cannot be successfully
      completed.

   FeatureNotAvailableException - if the VTSAgent does not provide a
      means of transferring vouchers.

   InsufficientVoucherException - if the VTSAgent does not have a
      sufficient number of vouchers to transfer.

   InvalidStateException - if the session is not "prepared".

   VTSSecurityException - if the VTSAgent cannot be authenticated
      correctly.

5.4.6. consume

```
public void consume(Session session,
                    Participant issuer,
                    VoucherComponent promise,
                    java.lang.Number num)
     throws VTSException
```

Consumes vouchers. This method deletes the specified number of
vouchers <issuer, promise, this> from the VVS and notifies the
receiver of the deletion. Similar to issue() and transfer(), the
response would be created and sent to the receiver if the VTS is
distributed so that the receiver can obtain proof of the deletion.
The VTSAgent must have a sufficient number of vouchers in the VVS.
Nothing is performed if the specified number is 0.

The session MUST be "prepared" when this method is called. The
state of the session will be "activated" when the vouchers are
deleted, and it will be "completed" when the transaction is
successfully completed or "suspended" if the transaction is
interrupted abnormally (e.g., network failures).

If null is specified for the issuer parameter, it indicates "any
issuer". This method selects vouchers to be consumed from the set
of vouchers returned by the getContents(null, promise).

   Parameters:

      session - the session used by the consume transaction.

      issuer - the issuer part of the voucher, or null.

      promise - the promise part of the voucher.

      num - the number of vouchers to be consumed.

   Throws:

      CannotProceedException - if the transaction cannot be successfully
         completed.

      FeatureNotAvailableException - if the VTSAgent does not provide a
         means of consuming vouchers.

      InsufficientVoucherException - if the VTSAgent does not have a
         sufficient number of vouchers to consume.

      InvalidStateException - if the session is not "prepared".

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.7. present

   public void present(Session session,
                       Participant issuer,
                       VoucherComponent promise,
                       java.lang.Number num)
          throws VTSException

      Presents vouchers. This method shows that the sender has the
      specified number of vouchers <issuer, promise, this> in the VVS to
      the receiver of the session; no modification is performed to the
      VVS. However, the response would be sent to the receiver as well
      as consume() in order to prove that the VTS has been distributed.
      The VTSAgent must have a sufficient number of vouchers in the VVS.
      Nothing is performed if the specified number is 0.

      The session MUST be "prepared" when this method is called. The
      state of the session will be "activated" when the vouchers are

retrieved, and it will be "completed" when the transaction is
successfully completed or "suspended" if the transaction is
interrupted abnormally (e.g., by network failures).

If null is specified for the issuer parameter, it indicates "any
issuer". This method selects vouchers to be presented from the
set of vouchers returned by the getContents(null, promise).

   Parameters:

      session - the session used by the present transaction.

      issuer - the issuer part of the voucher, or null.

      promise - the promise part of the voucher.

      num - the number of the voucher to be presented.

   Throws:

      CannotProceedException - if the transaction cannot be successfully
         completed.

      InsufficientVoucherException - if the VTSAgent does not have a
         sufficient number of vouchers to present.

      InvalidStateException - if the session is not "prepared".

      FeatureNotAvailableException - if the VTSAgent does not provide a
         means of presenting vouchers.

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.8. cancel

   public void cancel(Session session)
           throws VTSException

      Releases the session. "Prepared" sessions MUST be canceled. An
      implementation MAY be permitted to cancel "activated" or
      "suspended" sessions.

   Throws:

      InvalidStateException - if the state of the session cannot be
         canceled.

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.9. resume

   public void resume(Session session)
        throws VTSException

      Restarts the session. Only "suspended" sessions can be resumed.
      The state of the session will be re-"activated" immediately, and
      it will be "completed" when the transaction is successfully
      completed or "suspended" again if the transaction is interrupted
      abnormally (e.g., network failures).

   Throws:

      CannotProceedException - if the transaction cannot be successfully
         completed.

      InvalidStateException - if the session is not "suspended".

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.10. create

   public void create(VoucherComponent promise, java.lang.Number num)
        throws VTSException

      Creates vouchers where the issuer is the VTSAgent itself. This
      method creates the specified number of vouchers <this, promise,
      this> and adds them to the VVS. Nothing is performed if the
      specified number is 0.

   Throws:

      FeatureNotAvailableException - if the VTSAgent does not provide a
         means of creating vouchers.

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.11. delete

    public void delete(Participant issuer, VoucherComponent promise,
                       java.lang.Number num)
          throws VTSException

       Deletes vouchers. This method deletes the specified number of
       vouchers <issuer, promise, this> from the VVS. The VTSAgent must
       have sufficient vouchers in the VVS. Nothing is performed if the
       specified number is 0.

    Throws:

       InsufficientVoucherException - if the VTSAgent does not have a
          sufficient number of vouchers to delete.

       VTSSecurityException - if the VTSAgent cannot be authenticated
          correctly.

5.4.12. getContents

    public java.util.Set getContents(Participant issuer,
                                     VoucherComponent promise)
           throws VTSException

       Returns the set of vouchers whose issuer and promise both match
       the issuer and promise specified in the parameters.

       If null is specified for the issuer or promise parameter, it
       indicates "any issuer" or "any promise", respectively. If null is
       specified for both parameters, this method selects all vouchers
       owned by the holder from the VVS.

    Returns:

       the set of vouchers held by the holder of the VTSAgent.

    Throws:

       VTSSecurityException - if the VTSAgent cannot be authenticated
          correctly.

5.4.13. getSessions

    public java.util.Set getSessions()
           throws VTSException

       Returns a set of incomplete sessions prepared by the VTSAgent.

   Returns:

      the set of sessions prepared by the VTSAgent that are not yet
      completed.

   Throws:

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.14. getLog

   public java.util.Set getLog()
         throws VTSException

   Returns a set of completed sessions prepared or received by the
   VTSAgent. This set represents the trading log of the VTSAgent. A
   VTS may delete an old log eventually, so that the entire log may
   not be returned; the amount of the log kept by the VTSAgent is
   implementation-specific.

   Returns:

      the set of completed sessions prepared or received by the
      VTSAgent.

   Throws:

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.15. addReceptionListener

   public void addReceptionListener(ReceptionListener l)
         throws VTSException

   Adds a ReceptionListener to the listener list.

   After a ReceptionListener l is registered by this method,
   l.arrive() will be called whenever the VTSAgent receives a
   voucher.

   Nothing is performed if the specified listener is null.

   Throws:

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.4.16. removeReceptionListener

   public void removeReceptionListener(ReceptionListener l)
         throws VTSException

      Removes a ReceptionListener from the listener list.

      Nothing is performed when the specified listener is null or not
      registered.

   Throws:

      VTSSecurityException - if the VTSAgent cannot be authenticated
         correctly.

5.5. Session

   public interface Session

      Represents the logical connection established by the trade.
      Sessions are established by VTSAgent#prepare().

      A session has four states: prepared, activated, suspended, and
      completed. The initial state of a session is "prepared", and the
      session will be "activated" immediately when any of the trading
      methods of VTSAgent is called. The "activated" session will be
      "completed" after the trading method is successfully completed.
      If the trading method fails transiently (e.g., network failure),
      the session will be "suspended". Suspended sessions can be re-
      "activated" and restarted by calling VTSAgent#resume().

      A completed session may disappear from the VTSAgent; the session
      will be collected by the GC unless other objects keep its
      reference.

5.5.1. getIdentifier

   public String getIdentifier()

      Returns the identifier of the session. The generation scheme of
      the identifier is implementation-specific. An implementation may
      use a transaction ID as the identifier of the session.

   Returns:

      the string of the identifier of the session.

5.5.2. getVoucher

    public Voucher getVoucher()

       Returns the voucher to be traded using the session, or returns
       null if the session has not been activated.

    Returns:

       the voucher to be traded, or null if the state of the session is
       "prepared".

5.5.3. getSender

    public Participant getSender()

       Returns the sender of the session (i.e., the creator who prepared
       the session).

    Returns:

       the sender of the session.

5.5.4. getReceiver

    public Participant getReceiver()

       Returns the receiver of the session (i.e., the participant
       specified when preparing the session (by the VTSAgent#prepare()
       method)).

    Returns:

       the receiver of the session.

5.5.5. isPrepared

    public boolean isPrepared()

       Verifies if the session is "prepared".

    Returns:

       true if the session is in the "prepared" state, otherwise, false.

5.5.6. isActivated

   public boolean isActivated()

      Verifies if the session is "activated".

   Returns:

      true if the session is in the "activated" state, otherwise, false.

5.5.7. isSuspended

   public boolean isSuspended()

      Verifies if the session is "suspended".

   Returns:

      true if the session is in the "suspended" state, otherwise, false.

5.5.8. isCompleted

   public boolean isCompleted()

      Verifies if the session is "completed".

   Returns:

      true if the session is in the "completed" state, otherwise, false.

5.6. Voucher

   public interface Voucher

      Represents voucher(s) described in [VTS]. An object implementing
      this interface can represent more than one voucher if all of the
      issuer part and the promise part of the vouchers are the same.

5.6.1. getIssuer

   public Participant getIssuer()

      Returns the issuer part of the voucher(s).

   Returns:

      the participant who issued the voucher(s).

5.6.2. getPromise

   public VoucherComponent getPromise()

      Returns the promise part of the voucher(s).

   Returns:

      the voucher component that defines the promise of the voucher.

5.6.3. getCount

   public java.lang.Number getCount()

      Returns the number of the voucher(s).

   Returns:

      the positive (>0) number of the voucher(s).

5.7. VoucherComponentRepository

   public interface VoucherComponentRepository

      Maintains VoucherComponents.

      An object implementing VoucherComponentRepository provides a means
      of retrieving the voucher components that are the promises of
      vouchers in the VVS.

      Before issuing a voucher, the promise of the voucher must be
      registered with this repository. The repository can be
      implemented as either a network-wide directory service or personal
      storage like the ParticipantRepository.

5.7.1. register

   public VoucherComponent register(org.w3c.dom.Document document)

      Creates a voucher component associated with the specified DOM
      object and registers the voucher component with the repository.

      A voucher component of the voucher to be issued must be registered
      using this method.

      Nothing is performed (and the method returns null) if the
      specified document is null or the syntax of the document does not
      conform to the VTS.

The method returns the registered voucher component if the
specified DOM object has been already registered (no new voucher
component is created in this case).

   Returns:

   a registered voucher component associated with the specified
   document, or null if the document is null or has wrong syntax.

5.8. VoucherComponent

   public interface VoucherComponent

   Represents the voucher component that defines the promise of the
   voucher.

   Each VoucherComponent object has its own unique identifier and is
   associated with an XML document that describes the promise made by
   the issuer of the voucher (e.g., goods or services can be claimed
   in exchange for redeeming the voucher).

   This interface can be implemented as sort of a "smart pointer" to
   the XML document. An implementation may have a reference to a
   voucher component repository instead of the voucher component, and
   it may retrieve the document dynamically from the repository when
   the getDocument() method is called.

5.8.1. getIdentifier

   public String getIdentifier()

   Returns the identifier of the voucher component. Each voucher
   component must have a unique identifier. The identifier may be
   used to check for equivalence of voucher components.

   The format of the identifier is implementation-specific, however,
   it is RECOMMENDED that the hash value of the voucher component in
   the identifier be included to assure uniqueness. For generating
   the hash value, it is desirable to use a secure hash function
   (e.g., [SHA-1]) and to apply a canonicalization function (e.g.,
   [EXC-C14N]) before applying the hash function to minimize the
   impact of insignificant format changes to the voucher component,
   (e.g., line breaks or character encoding).

   Returns:

   the identifier string of the voucher component.

5.8.2. getDocument

   public org.w3c.dom.Document getDocument()

      Returns a Document Object Model [DOM] representation of the
      document associated with the voucher component by the
      VoucherComponentRepository#register() method.

      The DOM object to be returned may be retrieved from a
      VoucherComponentRepository on demand, instead of the
      VoucherComponent always keeping a reference to the DOM object.

      The VTS must guarantee that the getDocument method will eventually
      return the DOM object, provided that the voucher associated with
      the corresponding voucher component exists in the VVS.

   Returns:

      a DOM representation of the document associated with the voucher
      component.

   Throws:

      DocumentNotFoundException - if the associated DOM object cannot be
         retrieved.

5.9. ReceptionListener

   public interface ReceptionListener extends java.util.EventListener

      Provides a listener interface with a notification that a VTSAgent
      has received a voucher.

      When a voucher arrives at the VTSAgent, the VTSAgent invokes the
      arrive() method of each registered ReceptionListener.
      ReceptionListeners can obtain a Session object, which contains
      information about the received voucher and the sender of the
      voucher.

      This interface is intended to provide a means of notifying a
      wallet that "You have new vouchers", so that this interface may be
      implemented by wallets or other applications that use VTS.

5.9.1. arrive

   public void arrive(Session session)

      Provides notification of the arrival of a voucher.

After the listener is registered to a VTSAgent (by the
VTSAgent#addReceptionListener() method), the VTSAgent invokes this
method whenever it receives a voucher.

The specified session is equivalent to the session used by the
sender to trade the voucher. The state of the session is
"completed" when this method is called.

5.10. Exceptions

```
java.lang.Exception
  +-- VTSException
       +-- CannotProceedException
       +-- DocumentNotFoundException
       +-- FeatureNotAvailableException
       +-- InsufficientVoucherException
       +-- InvalidParticipantException
       +-- InvalidStateException
       +-- VTSSecurityException
```

VTSException
   This is the superclass of all exceptions thrown by the methods in
   the interfaces that construct the VTS-API.

CannotProceedException
   This exception is thrown when a trading is interrupted by network
   failures or other errors.

DocumentNotFoundException
   This exception is thrown when the document associated with a
   voucher component cannot be found.

FeatureNotAvailableException
   This exception is thrown when the invoked method is not supported.

InsufficientVoucherException
   This exception is thrown when the number of the voucher is less
   than the number specified for trading.

InvalidParticipantException
   This exception is thrown when the specified participant cannot be
   located.

InvalidStateException
   This exception is thrown when the state of the session is invalid
   and the operation cannot proceed.

```
   VTSSecurityException
      This exception is thrown when authentication fails, or when a
      method that requires authentication in advance is called without
      authentication.
```

6. Example Code

```
   // Issue a voucher

   VTSManager vts = new FooVTSManager();
   ParticipantRepository addrBook = vts.getParticipantRepository();
   VoucherComponentRepository vcr = vts.getVoucherComponentRepository();

   Participant you = addrBook.lookup("http://example.org/foo");
     // looks up a trading partner identified as
     // "http://example.org/foo".
   VTSAgent me = addrBook.lookup("myName").getVTSAgent();
     // a short-cut name may be used if VTS implementation allows.

   VoucherComponent promise = vcr.register(anXMLVoucherDocument);
     // registers a voucher component that corresponds to the voucher
     // to be issued.

   try {
     me.login();
       // sets up the issuer's smartcard (assuming distributed VTS).
     s = me.prepare(you);
       // receives a challenge from the partner.
     me.issue(s, promise, 1);
       // sends a voucher using the received challenge.
     me.logout();
   } catch (VTSException e) {
       // if an error (e.g., a network trouble) occurs...
     System.err.println("Sorry.");
     e.printStackTrace();
       // this example simply prints a stack trace, but a real wallet
       // may prompt the user to retry (or cancel).
   }

   // Transfer all my vouchers

   VTSManager vts = new FooVTSManager();
   ParticipantRepository addrBook = vts.getParticipantRepository();

   Participant you = addrBook.lookup("8f42 5aab ffff cafe babe...");
     // some VTS implementations would use a hash value of a public key
     // (aka fingerprint) as an identifier of a participant.
   VTSAgent me = addrBook.lookup("myName").getVTSAgent();
```

```
   try {
     me.login();
     Iterator i = me.getContents(null, null).iterator();

     while (i.hasNext()) {
       Voucher v = (Voucher) i.next();
       s = me.prepare(you);
       me.transfer(s, v.getIssuer(), v.getPromise(), v.getCount());
     }

     me.logout();
   } catch (VTSException e) {
     System.err.println("Sorry.");
     e.printStackTrace();
   }

   // Register an incoming voucher notifier (biff)

   VTSManager vts = new FooVTSManager();

   ParticipantRepository addrBook = vts.getParticipantRepository();
   VTSAgent me = addrBook.lookup("myName").getVTSAgent();

   ReceptionListener listener = new ReceptionListener() {
     public void arrive(Session s) {
       System.out.println("You got a new voucher.");
     }
   };

   try {
     me.login();
     me.addReceptionListener(listener);
     me.logout();
   } catch (VTSException e) {
     System.err.println("Sorry.");
     e.printStackTrace();
   }
```

7. Security Considerations

   Security is very important for trading vouchers. VTS implementations
   are responsible for preventing illegal acts upon vouchers (as
   described in [VTS]), as well as preventing malicious access from
   invalid users and fake server attacks, including man-in-the-middle
   attacks.

   The means to achieve the above requirements are not specified in this
   document because they depend on VTS implementation. However,

Terada & Fujimura   Informational   [Page 29]

   securing communication channels (e.g., using TLS) between client VTS
   plug-ins and the central server in a centralized VTS (as described in
   5.4.1 login()), and applying cryptographic challenge-and-response
   techniques in a distributed VTS are likely to be helpful and are
   strongly recommended to implement a secure VTS.

   This document assumes that the VTS plug-in is trusted by its user.
   The caller application of a VTS should authenticate the VTS plug-in
   and bind it securely using the VTS Provider information specified in
   the Voucher Component. This document, however, does not specify any
   application authentication scheme and it is assumed to be specified
   by other related standards. Until various VTS systems are deployed,
   it is enough to manually check and install VTS plug-ins like other
   download applications.

8. Acknowledgements

   The following persons, in alphabetic order, contributed substantially
   to the material herein:

      Donald Eastlake 3rd
      Iguchi Makoto
      Yoshitaka Nakamura
      Ryuji Shoda

9. Normative References

   [DOM]   V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs,
               A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and
               L. Wood. "Document Object Model (DOM) Level 1
               Specification", W3C Recommendation, October 1998,
               <http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/>

   [GVL]   Fujimura, K. and M. Terada, "XML Voucher: Generic Voucher
               Language", RFC 4153, September 2005.

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119, March 1997.

10. Informative References

   [ECML]   Eastlake 3rd, D., "Electronic Commerce Modeling Language
               (ECML) Version 2 Specification", RFC 4112, June 2005.

   [EXC-C14N] J. Boyer, D. Eastlake, and J. Reagle, "Exclusive XML
               Canonicalization Version 1.0", W3C Recommendation, July
               2002, <http://www.w3.org/TR/2002/REC-xml-exc-c14n-
               20020718/>

   [GPSF]   G. Lacoste, B. Pfitzmann, M. Steiner, and M. Waidner
                (Eds.), "SEMPER - Secure Electronic Marketplace for
                Europe," LNCS 1854, Springer-Verlag, 2000.

   [IOTP]   Burdett, D., "Internet Open Trading Protocol - IOTP
                Version 1.0", RFC 2801, April 2000.

   [JCC]   T. Goldstein, "The Gateway Security Model in the Java
                Electronic Commerce Framework", Proc. of Financial
                Cryptography '97, 1997.

   [SHA-1]   Department of Commerce/National Institute of Standards and
                Technology, "FIPS PUB 180-1. Secure Hash Standard. U.S.",
                <http://csrc.nist.gov/publications/fips/fips180-2/
                fips180-2withchangenotice.pdf>

   [TLS]   Dierks, T. and C. Allen, "The TLS Protocol Version 1.0",
                RFC 2246, January 1999.

   [VTS]   Fujimura, K. and D. Eastlake, "Requirements and Design for
                Voucher Trading System (VTS)", RFC 3506, March 2003.

Authors' Addresses

   Masayuki Terada
   NTT DoCoMo, Inc.
   3-5 Hikari-no-oka, Yokosuka-shi, Kanagawa, 239-8536 JAPAN

   Phone: +81-(0)46-840-3809
   Fax: +81-(0)46-840-3705
   EMail: te@rex.yrp.nttdocomo.co.jp

   Ko Fujimura
   NTT Corporation
   1-1 Hikari-no-oka, Yokosuka-shi, Kanagawa, 239-0847 JAPAN

   Phone: +81-(0)46-859-3053
   Fax: +81-(0)46-859-1730
   EMail: fujimura.ko@lab.ntt.co.jp

Full Copyright Statement

Intellectual Property

Acknowledgement