Developer Guide

iPlanet Trustbase Transaction Manager

Version 3.0

October 2001

THIS DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

CETTE DOCUMENTATION EST FOURNIE « EN L'ÉTAT », ET TOUTES CONDITIONS EXPRESSES OU IMPLICITES, TOUTES REPRÉSENTATIONS ET TOUTES GARANTIES, Y COMPRIS TOUTE GARANTIE IMPLICITE D'APTITUDE À LA VENTE, OU À UN BUT PARTICULIER OU DE NON CONTREFAÇON SONT EXCLUES, EXCEPTÉ DANS LA MESURE OÙ DE TELLES EXCLUSIONS SERAIENT CONTRAIRES À LA LOI.

Contents

List of Figures

Introduction

This document forms one of the iPlanet Trustbase Transaction Manager framework documentation set. This document is aimed at designers and developers looking to produce applications that utilise the iPlanet Trustbase Transaction Manager framework. The guide is divided into three main sections:

- Introduction to the environment iPlanet Trustbase Transaction Manager has been designed for
- An overview of the Framework and how the components interact
- Development of a iPlanet Trustbase Transaction Manager application

Overall Layout

The complete documentation set comprises of:

- iTTM3.0-Install-Configuration-Guide.pdf is designed for operators looking to produce applications that utilise the iPlanet Trustbase Transaction Manager framework. It is designed to provide information for operators looking to install the iPlanet Trustbase Transaction Manager platform. This guide identifies hardware and software required prior to installation, how to install iPlanet Trustbase Transaction Manager from CD-ROM
- iTTM3.0-Developer-Guide.pdf (this Document) that indicates how to build and deploy your own services. At the end of this manual there are two worked examples on how to build your own applications and an appendix on how to operate your own PKI using the command line tool TokenKeyTool
- API reference <install_dir> / Trustbase / TTM / current / apidocs is a softcopy Java documentation set that is provided as part of the iPlanet Trustbase Transaction Manager installation. It is designed to provide application developers with the information to utilise the framework and tools provided within the iPlanet Trustbase Transaction Manager framework.

This manual assumes the reader is familiar with Java standards described in `http://www.javasoft.com` and XML described in `http://www.w3.org/TR/REC-xml`.

The manual also assumes that the reader has attended the iPlanet Trustbase Transaction Manager Developer Training course.

Related Documents

- Solaris 8 and Java Development Kit 1.2.1

  `http://docs.sun.com`

  `http://java.sun.com/products/jdk/1.1/docs/index.html`

- iPlanet Application Server 4.1

  `http://docs.iplanet.com/docs/manuals/ias.html`

- iPlanet Web Server 6.0

  `http://docs.iplanet.com/docs/manuals/enterprise.html`

- iPlanet Certificate Management System

  `http://docs.iplanet.com/docs/manuals/cms.html`

- Oracle 8i Installation and Configuration Guides

  `http://www.oracle.com`

- Hardware Security nCipher KeySafe 1.0 and CAFast

  `http://www.ncipher.com`

- Identrus Message Specifications

  `http://www.identrus.com`

  Transaction Coordinator requirements (IT-TCFUNC)

  Core messaging specification (IT-TCMPD)

  Certificate Status Check Messaging specification (IT-TCCSC)

Introduction

The iPlanet Trustbase Transaction Manager platform provides a message oriented middleware platform capable of supporting a variety of banking and trade facilitation applications. This platform is specifically designed to enable Financial Institutions to make use of the Identrus Network by providing all of the function required to process messages that conform to the Identrus messaging specification.

iPlanet Trustbase Transaction Manager Platform

The iPlanet Trustbase Transaction Manager Platform operates at a high level as follows:

- Incoming messages are received from dedicated security services that support secure channel communication - including HTTP over SSLv3 and SMIMEv2 over SMTP.
- Requests are passed through a message parsing and verification engine that verifies that the incoming message is requesting a service that is offered by the platform, is complete and correctly formatted and builds the service request message request for processing through the system. The platform supports a range of standards for message coding including XML and HTML. A naming service identifies the application logic that the message object will be routed to for processing.
- Application Logic. The platform can support a range of application logic and can be extended by both the deploying and third party organisations.
- Third party services. Applications executed on the iPlanet Trustbase Transaction Manager platform may apply to third party systems during the decision making process. These include an organisation's operational systems and third party services.
- Response Management. A response to the relying party is constructed and returned to the requesting party based on the policy defined for each particular transaction supported by the platform.

All stages of the transaction processing process can be recorded in the repository's audit facilities.

Identrus Transaction Coordinator

The iPlanet Trustbase Transaction Manager provides an implementation of the Identrus Transaction Coordinator as specified in the following Identrus documents:

Title	Description and Document Reference
Transaction Coordinator	Transaction Coordinator requirements (IT-TCFUNC)
Transaction Coordinator Messaging Protocol Definition	Core messaging specification (IT-TCMPD)
Transaction Coordinator Certificate Status Check Protocol Definition	Certificate Status Check Messaging specification (IT-TCCSC)

The iPlanet Trustbase Transaction Manager is designed to be an extensible platform that not only performs the core CSC function of an Identrus Transaction Coordinator, but also allows the developer to produce applications that conform to the Identrus messaging specification.

The iPlanet Trustbase Transaction Manager makes producing Identrus compliant applications simple for the Developer by providing the following functionality:

- Standard presentation of messages independent of the transport mechanism.
- Standard Authentication and Authorisation of requests.
- Common validation and error handling of all Identrus Network layer information within messages.
- Tools for generation of new application message types that conform to the Identrus specifications.

The early sections of this guide identify these common components and the functions supported by each. The later sections of the guide show how an Identrus application may be produced and deployed on the iPlanet Trustbase Transaction Manager infrastructure.

Product Features

iPlanet Trustbase Transaction Manager provides a platform for the delivery of business to business E-Commerce solutions. The principal challenges to be addressed are as follows:

Product features

Identity	The solution recognises different types of credential and uses them as the basis for authentication requests, sometimes to remote and/or third party services. It supports the establishment of multi-party trust relationships, including those based on iPlanet Trustbase Transaction Manager Third Parties and other external CA services.
Entitlement	Once the identity of trading partners has been established, the platform provides a means for a business to determine which services it should provide to a potential customer. The e-commerce platform specifies and enforces terms for the provision of services based on the user type, credential type, access channel, credit worthiness, and so on.
Dispute Resolution	Using certificate-based technology, the platform provides services for transaction signing, audit logging and non-repudiation.
Integration	iPlanet Trustbase Transaction Manager integrates with existing back and mid-office solutions such as Customer Relationship Management and Cash Management solutions. It supports a flexible e-commerce model, catering for '2-to-*n*'-party transactions.
Availability	It provides Business to business commerce on a global basis that allows 24x7x365 availability.
Scalability	It services an increasing customer base catering for unpredictable performance requirements.
Security	E-commerce solutions must provide total security for all parties. This includes identification and authentication services, data confidentiality and integrity, transactional security, and security management services.

Chapter 1

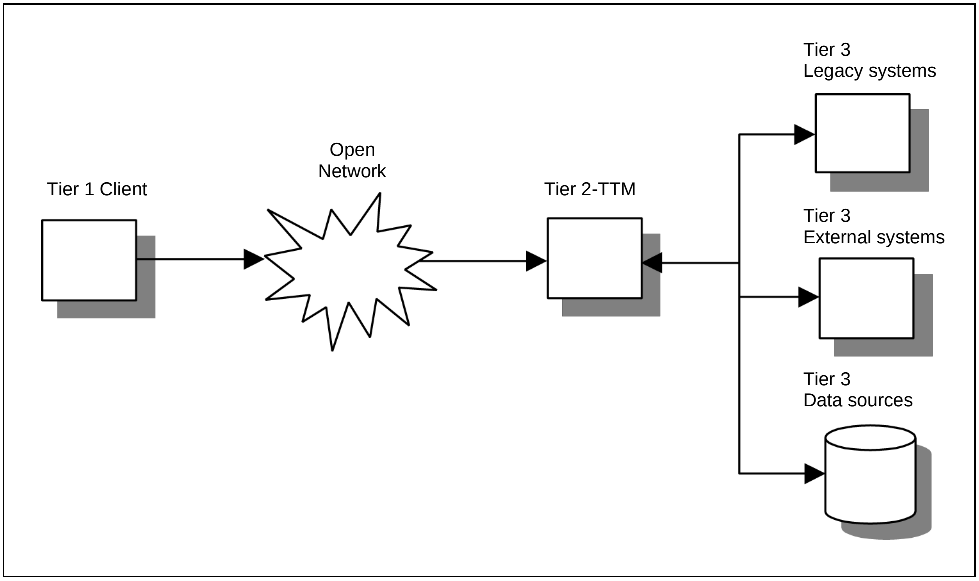
iPlanet Trustbase Transaction Manager Architecture

The iPlanet Trustbase Transaction Manager is designed to fulfil the need for Identrus enabled message oriented middleware for Financial Institutions. The platform provides a means of offering Financial Services Applications over the Internet that are consistent and re-useable.

Overview

Within a multi-tiered architecture, the iPlanet Trustbase Transaction Manager component is the middle tier infrastructure that provides the ability to offer new function, but shields first tier clients from the complexity of the existing enterprise infrastructure (see Figure 1-1).

Figure 1-1 Three Tier Architecture



In order to achieve this, the iPlanet Trustbase Transaction Manager is designed to be:

- Highly available
- Secure
- Reliable and scalable

It achieves this by using server side standards such as Servlets and EJB and leverages the existing reliability and scalability functions of the iPlanet Application Server.

The iPlanet Trustbase Transaction Manager Platform extends the iPlanet Application Server function by providing a message-handling pipeline that may be extended to process specific message types and formats. This message-handling pipeline contains four major components:

- Transport listeners
- Presentation and formatting handlers
- Routing and authorisation management
- Business logic plug ins

The message pipeline is populated with a set of components that provide all of the necessary pre-processing to support applications using the Identrus messaging protocol and HTML based communication.

The operational side of the iPlanet Trustbase Transaction Manager platform is augmented by a toolkit that allows the developer to generate message classes and deploy the completed application. This is designed to reduce the development effort required to produce applications that conform to the Identrus requirements to an absolute minimum.

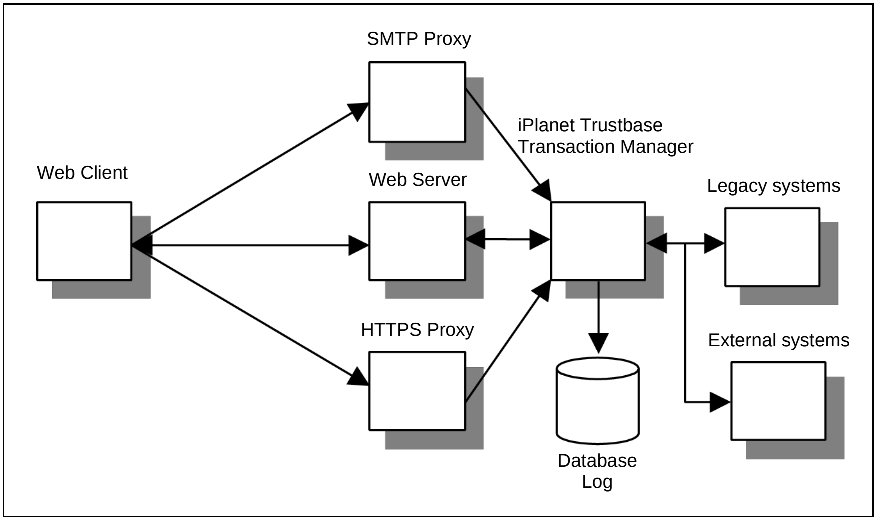
External interfaces

The iPlanet Trustbase Transaction Manager is a middleware component that provides a means of accessing and using existing legacy data sources over the Internet. In order to provide this function the platform must be capable of:

- Listening and processing web protocols - HTTP, HTTPS, SMIME
- Accessing existing legacy systems via JDBC, CORBA, RMI
- Using resources provided by other servers on the Web

Figure 1-2 shows how these external interfaces relate to the iPlanet Trustbase Transaction Manager component.

Figure 1-2 iPlanet Trustbase Transaction Manager Interfaces



Transport protocols

The iPlanet Trustbase Transaction Manager supports three basic transport protocols:

- SMTP - Asynchronous mailed based communication
- HTTP - Synchronous insecure communication
- SSL - Secure synchronous communication

iPlanet Trustbase Transaction Manager listeners proxy the SMTP and SSL transport protocols to provide both a means of processing, and a means of logging transport specific data. Any HTTP data is listened for directly by the Web Server.

These three transport protocols provide a means of carrying a variety of application level messaging. The iPlanet Trustbase Transaction Manager contains presentation components that deal directly with:

- SMIME wrapped XML or HTML
- HTML
- Identrus compliant XML application messages

The iPlanet Trustbase Transaction Manager platform may be extended to support other application messaging protocols as required.

Enterprise connectivity

The iPlanet Trustbase Transaction Manager business logic is designed so that the business logic implemented by the developer may use all of the standard connectivity components available within the J2EE platform. Standard J2EE connectivity components include:

- JDBC - Access to relational databases
- RMI and CORBA - Access to remote objects and EJB's
- JNDI - Access to directory and naming services
- JMS - Message oriented interfaces for use with message queues e.g. MQ Series

The iPlanet Trustbase Transaction Manager business logic is also capable of utilising the underlying connectivity components provided by the iPlanet Application Server for access to internal systems. These enterprise connectors include interfaces for:

- R/3 - Enterprise Resource Planning
- CICS - IBM Mainframe integration
- BEA Tuxedo - Transactional data systems
- Peoplesoft - Enterprise Resource Planning

Server to server connectivity

The iPlanet Trustbase Transaction Manager platform provides a means of abstracting the transport and presentation formats used by the incoming message within the presentation component. The same mechanism (through a different API) is used to allow a business service to make requests of other servers.

The Identrus Certificate Status Check (CSC) service uses this connector functionality to provide a means of determining the validity of a certificate at a particular point in time. (See Chapter 2, "Presentation logic".)

Routing

The router provides a mechanism for imposing structure and ordering on the execution of services in a secure way. It acts as a gatekeeper to ensure that services are only executed by authorised individuals and in an appropriate context. A user of the system will connect to the server and then exchange messages. At the highest level, the user will be trying to accomplish a task. Some tasks will require authorisation (and therefore authentication) prior to being performed; services may also perform tasks in a slightly different fashion depending on the identity of the user making the request.

The Router has been designed with the following in mind:

- Authentication and authorisation is kept separate from business logic.
- Configuration and management of the routing table is easily implementable and not error prone.
- Complex solutions can be built where required
- Implementing a simple solution is not difficult
- Services can implement atomic business level functions and are independent of one another.

The function of the router is central to the iPlanet Trustbase Transaction Manager platform. All messages are passed through a router, and, based on the current context of the message and its contents, the router will accept or reject the message for processing.

In order to define a flexible mechanism for routing, capable of working within a variety of complex environments, iPlanet Trustbase Transaction Manager provides rule based routing. This allows a means of modifying, and extending, the behaviour of the iPlanet Trustbase Transaction Manager installation over a period of time without the need to modify existing modules or services. See also Chapter 3, “Routing” for more details on this.

Authorisation

The basic requirement of being able to gate service access is met by the ability to route a message based not only on the message type, but also on its current level of authorisation. Within iPlanet Trustbase Transaction Manager, authorisation is considered an extension of authentication i.e. in understanding who a person is, we can determine what they are allowed to do.

The authentication mechanisms of iPlanet Trustbase Transaction Manager are not a separate component. Authentication data is gathered by the default iPlanet Trustbase Transaction Manager framework. This can then be added toby domain specific services. The platform provides a default authorisation service that is capable of mapping both a username and password, or a digital certificate onto a user group or *role*. The router then ensures that when a service is accessed, the role has been authorised for access to that service.

Developers are at liberty to replace the default authorisation service with a mechanism that maps user information onto existing repositories such as an enterprise directory service. See also Chapter 3, “Routing” for more information on this.

Services

Business services are at the heart of an e-commerce application, and the iPlanet Trustbase Transaction Manager provides a means of registering services written by the developer into the platform. These services need not be concerned with processing transport specific information, presentation specific information, authentication of the user, or authorisation of a users request. This allows the developer to concentrate on the function of the application, and integration of existing systems into a web enabled infrastructure. See also Chapter 5, “Standard Services” for more information on this.

Chapter 2

Presentation logic

When an iPlanet Trustbase Transaction Manager implementation, on receiving a message, routes the message through to its appropriate service, and the service cannot directly act upon this message, iPlanet Trustbase Transaction Manager utilises the Connection Manager component to process messages appropriately.

Overview

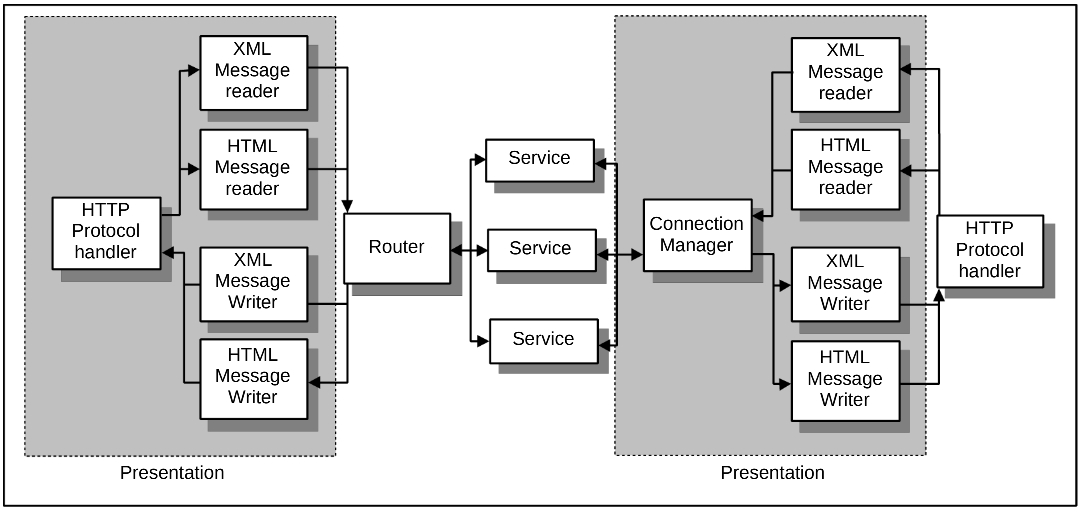
The iPlanet Trustbase Transaction Manager presentation components comprise of a set of protocol handlers, message readers and message writers. The purpose of the protocol handlers is to extract the transport specific headers and footers, determine the MIME type and message type, and to forward the message content to an appropriate message handler.

The message handlers are selected on the basis of the MIME and Message type determined by the protocol handlers. The iPlanet Trustbase Transaction Manager provides two default message handlers, these are:

- Identrus XML messages
- HTML messages

In the case of the Identrus XML message, the appropriate Identrus network processing is performed as described in the following sections.

Figure 2-1 Presentation layer components



In the case of HTML requests, the message reader converts all of the fields in the HTML form into an XML structure represented by a set of DOM nodes.

In both the cases of XML and HTML the output to the router is wrapped with the session information into an iPlanet Trustbase Transaction Manager message. These iPlanet Trustbase Transaction Manager messages are then sent, by the router, to the appropriate service.

The services perform the appropriate message transformation or construction and return an iPlanet Trustbase Transaction Manager message back to the router. The router will then send the iPlanet Trustbase Transaction Manager message back to an appropriate message writer. In the case of the HTML message writer this will extract the DOM values in the message into an HTML template determined by the message type. The template may contain a set of directives that allow conditional or recursive construction of form data.

Protocol handlers

When the platform receives a message, the protocol handler is selected on the basis of the MIME type of the incoming message and is then invoked. The function of this component is to determine the application protocol information, specifically:

- Message type
- Context ID

Each MIME type, and therefore protocol handler, represents a class of messages e.g. application/OCSP. The client must be capable of generating the appropriate MIME type.

- HTML Protocol Handler If the client is a Web browser it normally generates a MIME type of application/x-www-form-url-encoded. iPlanet Trustbase Transaction Manager comes supplied with the default Protocol Handler for use in this situation. Its default actions are to:
- Search the form fields for:
    - Message Type - Identifying the message type
    - Context ID - Identifying the Context Identifier
- To set the response MIME type to text/html indicating to the Browser that the reply is a standard HTML page.

Identrus protocol handler

The protocol handler does the initial message processing for all messages arriving at the TC with a mime-type that begins with the following string: "application/identrus-".

Once at the protocol handler the following actions are carried out in the following order:

- The raw XML stream is read from the client.
- The raw XML is parsed into the internal iPlanet object structure (known as the message tree)
- The message tree is then validated against the DTD for that message that is specified by the DOCTYPE in the XML message.

- The message tree is traversed and any ID attributes are checked to ensure that there are no duplicates. This is a mandatory XML processing step, and is especially important where XML structures will be digitally signed.
- The system identifier from the DOCTYPE tag is placed into a iPlanet Trustbase Transaction Manager Message Attribute identified by IdentrusConstants.SYSTEM_ID_ATTR.
- Elements from the NIB are extracted ready for Raw Message Logging.
- The Certificate Bundle is extracted from the message and all of the certificates are logged in the Certificate Log.
- The message, minus its Certificate Bundle, is sent to the Raw Message Log.
- The Raw Log Id is returned from the log and placed into a iPlanet Trustbase Transaction Manager Message Attribute identified by IdentrusConstants.RAW_RECORD_MARKER
- The iPlanet Trustbase Transaction Manager Message type is set to be IdentrusConstants.IDENTRUS_MESSAGE
- The message tree is attached to the iPlanet Trustbase Transaction Manager Message as serialised content, ready to be sent to the appropriate message reader.

NOTE The variable DOCTYPE is an XML concept. See for instance `http://www.w3.org/TR/REC-xml`. The constants SYSTEM_ID_ATTR, RAW_RECORD_MARKER and IDENTRUS_MESSAGE are Identrus Constants. See com.iplanet.trustbase.identrus.IdentrusConstants for more information on Identrus constants.

Message Readers

The appropriate message reader is invoked based on the information gathered by the protocol handler. Each Message Reader will have an implicit understanding of the class of messages it can convert into an internal iPlanet Trustbase Transaction Manager message.

Default Message Reader, HTTP/HTML Reader

iPlanet Trustbase Transaction Manager comes with a pre-registered HTTP / HTML Message Reader that takes an `application/x-www-form-url-encoded` type. This is registered as a default message reader, meaning that if the message reader registry is unable to make a specific match to the message type detected by the host environment adapter then the default HTTP / HTML message reader will be returned. If the MIME type is anything other than `application/x-www-form-url-encoded` an error is returned.

Once the appropriate message reader has been found it is then passed an iPlanet Trustbase Transaction Manager Message object and the input stream containing the remaining unprocessed message content. The message reader reads and parses this message content from the input stream and creates a data block containing an internal representation of the message content. This is then stored in the iPlanet Trustbase Transaction Manager Message object for use by the appropriate service(s).

Once the message reader has parsed the input stream, and completed construction of the iPlanet Trustbase Transaction Manager Message, the message analyser passes the iPlanet Trustbase Transaction Manager Message on to the Router, and handles any errors or exceptions that result from the subsequent processing of the iPlanet Trustbase Transaction Manager Message.

Using the default HTTP Reader

The HTTP reader performs a relatively simple extraction of information from the input stream. Each input field is extracted from the HTTP Post or Get string and placed into a DOM structure. Once all the fields have been extracted this DOM Object is stored within the iPlanet Trustbase Transaction Manager Message. If the user does not specify the structure of the DOM object then each field is stored as an attribute of a node HTTP_VALUES,

```
HTTP
HTTP_VALUES Attributes [<field_name> = <value>]*
```

However it is possible for the developer to force the fields to be stored in a more complex structure using certain escape sequences.

In order to create your own DOM structure the HTTPReader class has the ability to recognise an escape sequence contained in place of an input field name: <input name="$|xml:<node1>…<noden>|$">. For example:

```
<input name="$|xml:message.user.name|$">
<input name="$|xml:message.user.address|$">
<input name="$|xml:message.user?jobtitle|$">
```

This will lead to a set of nodes being created under the root HTTP node containing the following data:

```
HTTP
HTTP_VALUES Attributes [<field_name> = <value>]*
message
user Attribute jobtitle=<value>
name = <value>
address = <value>
```

Default Message Reader, Identrus Reader

This message reader will handle messages with a mime-type that begins with "application/identrus-" and a message type of IdentrusConstants.IDENTRUS_MESSAGE.

Once at the message reader the following actions are carried out:

- Signature check the mandatory DSIG signature
- Verify the certificate chain in the message, and ensure that the root certificate of the chain is in the database. Ensure that the root of the chain has a Certificate Store attribute of IdentrusConstants.IR_CA.
- Set the security context on the iPlanet Trustbase Transaction Manager Message ready for the authorisation phase in the router.
- Make a billing log entry for the message

The message is then sent for routing.

Default Message Reader, Identrus Error Reader

This message reader will handle messages with a mime-type of "application/identrus-identrustransporterror" and a message type of IdentrusConstants.IDENTRUS_TRANSPORT_ERROR

Once at the message reader the following actions are carried out:

- There is no signature to check on an incoming "Identrus Transport Error", so the security context on the iPlanet Trustbase Transaction Manager Message is left unset. Messages arriving through this reader should only be coming through the connector, so they should never go to the router.

Message Writers

Once a message has been routed through to one or more services and returned to the Router, it is passed back to the Message Analyser (this assumes no exceptions or errors are thrown). At this point the message analyser is required to determine which Message Writer to be used to process the returned iPlanet Trustbase Transaction Manager Message into a suitable form for return to the user. As with message readers the analyser uses the Messages returned type and response MIME type values to select the appropriate message writer.

The message writer is responsible for extracting information contained in the iPlanet Trustbase Transaction Manager Message and processing it into a format that can be written back to the client.

To register the message writer with the registry iPlanet Trustbase Transaction Manager is supplied with an HTML management interface. This allows the developer to register new Message Writer sub-classes in the internal configuration store associated with iPlanet Trustbase Transaction Manager. Alternatively registration may be undertaken by adding the appropriate entries to the tbase.properties file. To achieve this, new Message Writers should be added to the [`MessageWriter`] section and take the form:

```
message.writer=<name>:<classpath>
message.writer=
Script:uk.co.jcp.tbaseimpl.parse.message.http.ScriptWriter
```

where the <name> is a name given by the developer and the <classpath> is the fully qualified classpath of the new MessageWriter instance. It is important to note that if you add new Message Writers through the management interface and at a later date it becomes necessary to restart iPlanet Trustbase Transaction Manager using the properties file then all changes will be lost. So it is as well to add entries to the properties file even if you are using the management interface.

Default HTML Message Writer

iPlanet Trustbase Transaction Manager comes with a pre-registered HTML message writer that takes a text/html MIME type. This is registered as a default message writer, meaning that if the message writer registry is unable to make a specific match to the message type and MIME type detected by the host environment adapter then the default HTML message reader will be returned. If the MIME type is anything other than text/HTML an error is returned.

This MessageWriter is capable of translating XML fields contained in the message into an HTML form using pattern matching and templates similar to many other Dynamic HTML writers.

The ScriptWriter, the default HTML Message Writer, uses a scripting language that is intended to provide a simple way to display uncomplicated information quickly and efficiently without having to recourse to a full XSL.

This is intended for use where the response is expected to contain text (usually HTML), and the information returned by the router is simple enough to allow straightforward substitution of values from the returned XML into the Template.

The ScriptWriter class works by taking the DOM object contained in the returned message and extracting all the fields therein. These fields are then substituted into the template that is selected by using the type and format of the returned message type. The substitution is achieved using a series of Script Tags that allow the developer to specify which parts of the returned message are to be placed where in the template. The templates used by the ScriptWriter may be configured in one of two ways, either through the iPlanet Trustbase Transaction Manager HTML management interface or by adding the appropriate entries to the tbase.properties file in the [`ScriptWriter`] section as illustrated below:

```
writer.typeandformat=<format>:<type>:<template>
writer.typeandformat=text/html:TimeServiceTimeResponse:Config\Templates\
TimeService\TimeServiceTimeResponse.html
```

Where the <format> is text/html, <type> is the message type that will have been set by the service that last processed the message before it was returned, and <template> is the name of the template file. This file may either be an absolute path or a relative path, if it is a relative path then an entry is required to specify the location of the 'root':template.directory=.\ (note the trailing '\' which is required)

Script Tags

There are four main tags used in the script templates:

- $$xml:<XML Variable Name>$$

This represents the single substitution of a variable from the XML Document into the template. <XML Variable Name> is the fully qualified 'location' of a node in the XML Document returned by the router which is expected to contain a string value that may be substituted into the template. For instance, to retrieve the name from an XML Document one might specify the tag,

```
$$xml:message.content.user.name$$
```

This tag is also capable of specifying an attribute from a given XML node using the "?" escape sequence to select the given attribute. For example, in order to retrieve users middle initials that are stored as an attribute of their name, one might specify the following tag,

```
$$xml:message.content.user.name?middle_initials$$
```

- $$repeat:<XML Array Name>[<UserDefined Tag Name>:<Iterations>]$$…$$/repeat$$

Represents the start point for a repeating array of values contained in XML Document.

<XML Array Name> is the fully qualified 'location' of an array in the XML Document.

<UserDefined Tag> Name allows the template author to provide a shorthand name for this array location, e.g. If the array represents the address of a user, Response.Content.User.Address, then the author could tag this as address for ease of use within the repetition block.

<Iterations> tag allows the author to specify which 'parts' of the array are to be displayed.

Table 2-1 XML Repeat Iterators

Iterator	Description	Example
*	Iterate over all items contained in the specified array	*
<n>	Select the nth element of the array, if the array has less than n elements then nothing is displayed	2
<m>-<n>	Select from the mth to the nth elements (inclusive), if the array has less than n elements then this selects from m to the end, if the array has less than m elements then nothing is displayed	3-5
<m>-*	Select all elements from the mth up to the last element in the array	2-*
<attribute>=<value>	Select ALL the elements in the array where the specified attribute equals the given value	location= London
,	Allow multiple iterators to be specified together	0,3-5,9,postcode=EC1

Every repeat block is terminated with a $$/repeat$$ statement.

- $$xif:<XML Variable Name>[="<value>"]$$ ... $$/xif$$

Represents a conditional tag. In its simplest form this tag says iterate everything contained within the conditional statement if the node (or attribute - using the same "?" escape sequence) is found to exist within the XML.

If the "=" is used however, not only does the node or attribute have to exist but the value has to be an exact (case sensitive) match of the one specified. For example one might specify a conditional block say only print the user details if the job title matches "SysAdmin" as follows:

```
$$xif:message.content.user?jobtitle="SysAdmin"$$
```

Associated with the `$$xif:…$$` tag there is also an `$$xelse:…$$` tag to provide optional switching. This tag has two modes of use:

```
$$xelse$$
```

Simple usage, if the `$$xif:…$$` condition is unfulfilled then this condition is used instead. One might use this sequence if it is necessary to use two different display formats based on the contents of an XML node.

```
$$xelse:<XML Variable Name>[="<value>"]$$
```

Uses the same format as the xif statement to determine if this branch should be executed. This provides extra conditional switching which can be used to make more complex decisions about which sections of the template are to be used.

- $$xprop:[<section>]<property>[=<default_value>]$$

This is a special tag that is used to incorporate property's values from the tbase.properties file into the template. This allows different application server environments to be supported by the same template by changing the property value for each server type. The `$$xprop:…$$` tag is also special in that it is pre-processed before any of the other tags are dealt with, thus it is possible to include other tag information (except `$$xprop:…$$` itself) in the properties.

- `<section>` the name of the section that contains the property, if this is left out, the section used is the ScriptWriter section.
- `<property>` the name of the property that is to be used.
- `<default_value>` a default value to be used if the property is not found, if this is not specified and the property is not found then an empty string is returned.

Using the ScriptWriter tags

- `$$xml:…$$` substitution

As documented above this tag represents a straightforward substitution of a single value contained in the returned XML Document, into the HTML Template. In the following template snippet, we can see that the author expects the XML Document returned from the router to contain data indicating an administrative support mail address.

```
<P>
<FONT SIZE=-2>
Please try our
<A HREF="mailto:$$xml:response.content.supportaddress$$">Support
Desk</A>
if you have any enquiries
</FONT>
</P>
```

- `$$xml:…$$` filtering

It is also possible to introduce an attribute filter into the qualified path, as shown in the code snippet below:

```
<P>
Email the
<A
HREF="mailto:$$xml:response.content.user?title="SysAdmin".email$$"
>System Administrator</A>
if you have any enquiries regarding your login details.
</P>
```

Here the XML Document contains an array of one or more user's details and each user contains a node that contains their email address. We use the user title attribute to determine which of the users job description is System Administrator and use the resultant user to extract the email address.

- `$$repeat:…$$` loop

The following example shows the complete usage of a repeat block, including the `$$/repeat$$`. Here the author is indicating the XML Document response will contain an array of transactions, under the node http_response.content.transaction. This section of the XML Document tree is then tagged as txn.

Inside the repeat block, one then uses the standard `$$xml:…$$` notation to select data contained in each member of the array returned.

Each transaction will contain three nodes date, description and amount. These will be filled into a single table row and the table will contain as many rows as there are transactions returned.

```
<table BORDER="1" BGCOLOR="#fff0a0">
     <th COLSPAN="3" BGCOLOR="ffd080"><b><i><font SIZE="+1"
COLOR="#00a000">Statement Details</font></i></b></th>
     <tr>
          <td WIDTH="50" ALIGN="center"><B>Date</B></td>
          <td WIDTH="400" ><B>Description</B></td>
          <td WIDTH="200" ALIGN="center"><B>Amount</B></td>
     </tr>
     $$repeat:message.content.transaction[txn:*]$$
          <tr>
          <td>$$xml:[txn].date$$</td>
          <td>$$xml:[txn].description$$</td>
          <td ALIGN="right">$$xml:[txn].amount$$</td>
          </tr>
     $$/repeat$$
</table>
```

- Nesting `$$repeat:…$$` blocks

It is also possible to nest repeat blocks, if we consider the users example again it might be that each user node contains an array of address nodes (one per line of their address) and hence we might wish to iterate over the address lines for each user. The following template snippet shows how this might be done:

```
<table BORDER="1" BGCOLOR="#fff0a0">
     <th COLSPAN="3" BGCOLOR="ffd080"><b><i><font SIZE="+1"
COLOR="#00a000">System Users</font></i></b></th>
     <tr>
          <td WIDTH="100" ALIGN="center"><B>Name</B></td>
          <td WIDTH="400" ALIGN="center"><B>Address</B></td>
          <td WIDTH="100" ALIGN="center"><B>Telephone</B></td>
     </tr>
     $$repeat:message.content.user[user:*]$$
          <tr>
          <td>$$xml:[user].name$$</td>
          <td>
          $$repeat:[user].address[addr:*]$$
               $$xml:[addr]$$<br>
          $$/repeat$$
          </td>
          <td>$$xml:[user].telno$$</td>
          </tr>
     $$/repeat$$
</table>
```

- `$$xif:…$$`

In this case, we will extend the repeat example to show how we might need to perform a conditional display. In this case the phone number column either displays a home phone number for all teleworkers or an internal extension for those based in the office.

```
<table BORDER="1" BGCOLOR="#fff0a0">
     <th COLSPAN="3" BGCOLOR="ffd080"><b><i><font SIZE="+1"
COLOR="#00a000">System Users</font></i></b></th>
     <tr>
          <td WIDTH="100" ALIGN="center"><B>Name</B></td>
          <td WIDTH="400" ALIGN="center"><B>Address</B></td>
          <td WIDTH="100" ALIGN="center"><B>Telephone</B></td>
     </tr>
     $$repeat:http_response.content.user[user:*]$$
          <tr>
          <td>$$xml:[user].name$$</td>
          <td>
          $$repeat:[user].address[addr:*]$$
               $$xml:[addr]$$<br>
          $$/repeat$$
          </td>
          <td>
               $$xif:[user].workType="Teleworker"$$
                    $$xml:[user].homePhone$$
               $$xelse$$
                    $$xml:[user].workExtn$$
               $$/xif$$
          </td>
          </tr>
     $$/repeat$$
</table>
```

- `$$else:…$$`

Lastly, it is possible to perform a multiple conditional statement using the extendedlse construct. Taking the previous example we can extend the switch to include an extra case for salesmen.

```
<td>
     $$xif:[user].workType="Teleworker"$$
                    $$xml:[user].homePhone$$
     $$xelse:[user].workType="Sales"$$
                    $$xml:[user].mobile$$
     $$xelse$$
                    $$xml:[user].workExtn$$
     $$/xif$$
</td>
```

- `$$xprop:…$$` property substitution

The code snippet below shows how to get a property from the tbase.properties file into a template:

```
<hr>
<form METHOD=post
ACTION="$$xprop:[ApplicationServcer]form.string$$/appservlet">
     <h3>
Please enter your card details so that we may confirm your
identity:
</h3>
```

Default Identrus Message Writer

This message writer will handle messages with a mime-type that begins with "application/identrus-" and a message type of IdentrusConstants.IDENTRUS_MESSAGE.

Once at the message writer the following actions are carried out:

- Sign the message with the IdentrusConstants.L1_IP_SC certificate, unless the iPlanet Trustbase Transaction Manager Message attribute identified by IdentrusConstants. SIGNING_CERT_ATTR is present. If this attribute is present then its value is taken as the purpose Id of the certificate/key used to sign the outgoing message.

- The outgoing message is DTD validated according to its DOCTYPE, this prevents malformed messages from leaving the TC.
- The DOCTYPE tag is set and the mime-type of the message is determined from the DOCTYPE
- The Certificate Bundle is extracted from the message and logged with the Certificate Log
- The XML of the outgoing message, minus the Certificate Bundle, is recorded in the message log
- The message is written back to the client.

Default Identrus Error Writer

This message writer will handle messages with a mime-type of "application/identrus-identrustransporterror" and a message type of IdentrusConstants.IDENTRUS_TRANSPORT_ERROR

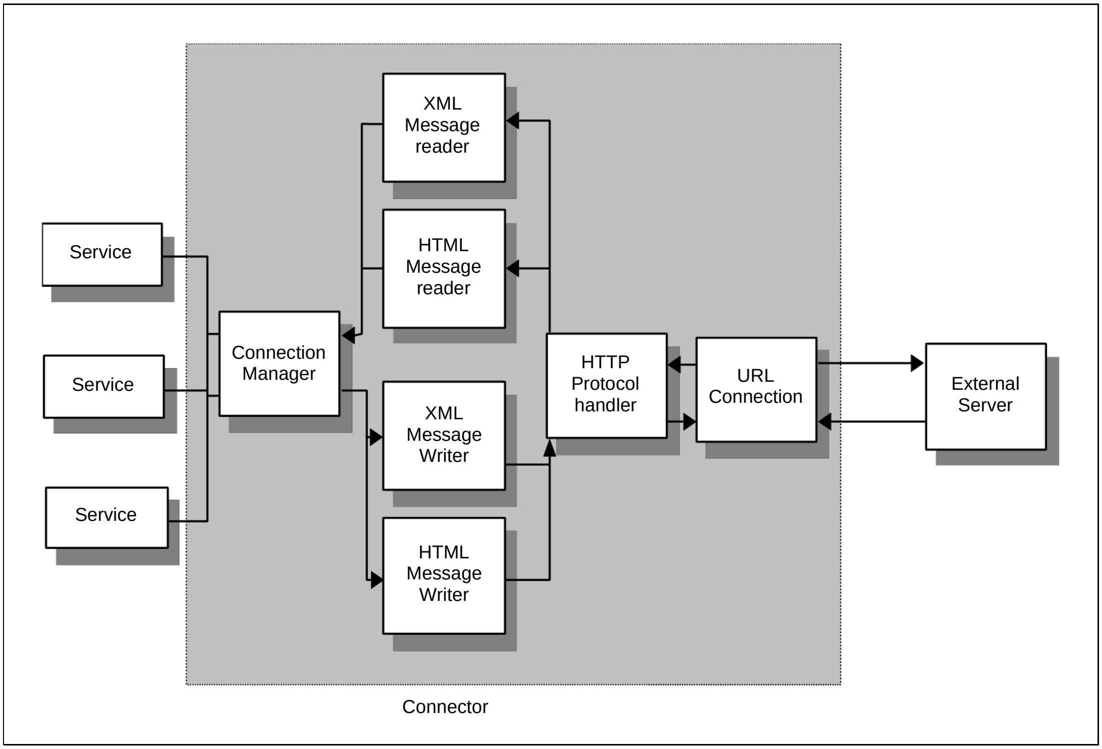
Once at the message writer the following actions are carried out:

- The outgoing message is DTD validated according to its DOCTYPE, this prevents malformed messages from leaving the TC.
- The DOCTYPE tag is set and the mime-type of the message is determined from the DOCTYPE
- The XML of the outgoing message is recorded in the raw log
- The message is written back to the client.

Connection Manager

On receiving a message iPlanet Trustbase Transaction Manager routes this message to an appropriate service which can process the message accordingly, and can, if required return a message in response. A problem arises when the particular iPlanet Trustbase Transaction Manager implementation, on receiving a message, routes the message through to its appropriate service, and the service cannot directly act upon this message. It needs additional information from another source in order to complete the task presented to it by the received message. Hence, iPlanet Trustbase Transaction Manager needs the facility to create new messages using specific message transport protocols and send these messages to specific external destinations. To accomplish this, the service needs to call the "Connection Manager".

Figure 2-2 Connection Manager Architecture



NOTE Please consult your API for more information about this:

- uk.co.jcp.tbase.connector
- uk.co.jcp.tbase.xurl
- uk.co.jcp.tbaseimpl.connector

The Connection Manager is essentially a service that can be used to send and receive messages to and from external entities. The Connection Manager is passed a iPlanet Trustbase Transaction Manager message and Destination Object. It uses this information to send and receive messages related to the specific message/destination object supplied. The Connection Manager Message Process works as follows:

- The Service calls the Connector, passing the iPlanet Trustbase Transaction Manager Message containing the request to the external resource, and a Destination Object describing the external resource. Destination Objects are application-defined implementations, and are recognised by ProtocolMap objects that are application defined plug-in components to the Connector. A null Message parameter may be passed, in which case no attempt will be made to write data to the URLConnection object when a connection has been established.
- The Connector calls the ProtocolMapManager to translate the Destination Object it was supplied with into a ProtocolDescriptor Object that specifies a URL for the connections, and a mime-type for the request content. The ProtocolMapManager determines which ProtocolMap from those registered with it (an administration activity) is responsible for dealing with the supplied Destination implementation, and passes the Destination Object to the ProtocolMap.
- The ProtocolMap translates the Destination Object into a ProtocolDescriptor . This action may be simple and local, or it may be complex and involve further external requests.
- The Connector makes a connection to the external resource using the URL in the ProtocolDescriptor to form a URLConnection.
- If a Message was passed to the Connector, it sets the doOutput variable on the URLConnection to true, to signify it's intention to write data to the connection. The Connector then selects a MessageWriter from its registry of MessageWriters, according to the type of the Message to be written (in the message Type attribute), and the required format, as specified in the

ProtocolDescriptor. The MessageWriter is called to translate the Message to the appropriate format, and write it to the OutputStream of the URLConnection. Should no Message have been passed to the Connector, the doOutputfield of the URLConnection is set to false, then nothing is written to the URLConnection..

- The Connector hands the InputStream of the URLConnection to the ConnectionProtocolAnalyser, along with the mime-type of the response [determined from the URLConnection]. The ConnectionProtocolAnalyser determines a ProtocolHandler to call, using the mime-type of the response.
- The ConnectionProtocolAnalyser gives the selected ProtocolHandler a new iPlanet Trustbase Transaction Manager Message, and the InputStream . The ProtocolHandler reads from the InputStream to determine the Message type of the response, which it sets on the Message (it may also set a Context identifier, although this will be ignored in this setting), before returning.
- The Connector selects a MessageReader based on the response Message type, and the response InputStream mime-type. The MessageReader is called to complete the parsing of the response. After the MessageReader completes, the Connector releases any resource used in making the external connection, and returns the parsed Message to the requesting Service.

NOTE iPlanet Trustbase Transaction Manager provides a default connector that implements the connector interface defined in uk.co.jcp.tbaseimpl.connector.DefaultConnector. Most of these classes are supplied as part iPlanet Trustbase Transaction Manager. Some are not. For instance, java.net.URLConnection is part of the core Java API.

Protocol Map Manager

The Connection Manager provides the ability for Services in the iPlanet Trustbase Transaction Manager framework to communicate with other external entities. To do this the Service passes a message and Destination to the Connector (Connection Manager), which routes the message to the specific external entity. To find the address of the communicating party, the supplied Destination is passed to the ProtocolMapManager. This ProtocolMapManager then passes the Destination to its associated ProtocolMap registered with the ProtocolMapManager. The specific ProtocolMap then returns a ProtocolDescriptor object associated with this Destination. Each specific ProtocolMap decides which ProtocolDescriptor to return for any supplied Destination.

To create a new application specific protocol mapping system, the following steps have to be followed:

- Creation of a Specific ProtocolMap.
- Creation of a Specific Destination
- Registering the new ProtocolMap with the ProtocolMapManager.
- Populating the ProtocolMap with data.

NOTE Please consult the framework uk.co.jcp.tbase.connector and the default connection manager class implementation can be found in uk.co.jcp.tbaseimpl.connector.

These are now described in turn.

- Creation of a Specific ProtocolMap

All ProtocolMap objects must implement the ProtocolMap interface. They must also provide an empty / default constructor, due to the fact that the ProtocolMapManager will dynamically instantiate the ProtocolMap object in order to register the class with itself.

The ProtocolMap enforces the following methods:

```
public Enumeration getDestinationTypes()
```

Returns a list of the package qualified class names of the Destination implementations that this ProtocolMap recognises.

```
public ProtocolDescriptor getProtocolDescriptor(Destination
destination)
     throws InvalidDestinationException
```

Translates an application's specified Destination object into a ProtocolDescriptor, specifying the URL and mime type for the connection.

NOTE The application specific ProtocolMap must not contain any objects that are not serializable, as the ProtocolMapManager has persistence. Consult your API uk.co.jcp.tbase.connector.ProtocolMap for more information on this.

- Creation of a Specific Destination

All Destinations must implement the Destination interface. This interface does not enforce any methods, leaving all methods and data purely application specific, thus relying on its associated ProtocolMap to have prior knowledge of its format.

NOTE The application specific Destination must not contain any objects that are not serializable, as the ProtocolMapManager has persistence. Consult your API uk.co.jcp.tbase.connector.Destination for more information on this.

- Registering the new ProtocolMap with the ProtocolMapManager.

Once an application specific ProtocolMap and Destination have been constructed, the ProtocolMap has be registered with the ProtocolMapManager before any Service in iPlanet Trustbase Transaction Manager can utilise its function.

NOTE If a new application specific ProtocolMap contains a reference to a specific Destination used by an existing ProtocolMap registered with the ProtocolMapManager then the new ProtocolMap will be prevented from registering with the ProtocolMapManager.

Protocol Maps can be registered in the tbase.properties file that contains a property section called "ProtocolMapManager". In this section the application specific ProtocolMap objects can be supplied. For instance,

```
[ProtocolMapManager]
protocol.map=uk.co.jcp.tbase.connector.SimpleProtocolMap
```

NOTE This `tbase.properties` file will only be read once unless the configuration object associated with the instance of iPlanet Trustbase Transaction Manager is removed. Note also, the full package name must be supplied.

URL Connection Implementation

Internally in the Connection Manager, once the ProtocolMapManager returns a ProtocolDescriptor from a supplied Destination, the URL string representation is obtained from the ProtocolDescriptor. From this string representation of a URL, an URLConnection object is obtained, from which the output and input streams are used to send and receive the message from the Connection Manager (Connector). This conversion of a string representation of a URL to a URLConnection object is carried out through the XURLxxxxx classes. The classes used are:

- XURL
- XURLStreamHandler
- XURLStreamHandlerFactory

These classes are needed, due to the fact that the existing URL class hierarchy supplied by javasoft is not extensible in an EJB format. This is because in the existing framework, if the URL class does not recognise an URL format then it looks to see if a XURLStreamHandlerFactory is loaded, which could supply the unknown URL format. This factory can only be set once, so in an EJB environment it is impossible to ascertain whether URLStreamHandlerFactory has been instantiated, so making this framework unusable. Hence, we have produced an extensible framework that can work in any environment. These objects are identified as XURL, XURLStreamHandler, and XURLStreamHandlerFactory.

The process flow is as follows:

- A XURL object is constructed using the string representation of the URL supplied by the ProtocolDescriptor.
- To obtain the URLConnection object the XURL's method openConnection() is called.
- Inside this method the XURLStreamHandlerFactory is called.
- The XURLStreamHandlerFactory, if not called before initialises itself from the tbase.properties file, registering XURLStreamHandler classes in the section "XURLStreamHandlerFactory", each class mapped against the value "url.stream.protocol".
- The XURLStreamHandlerFactory is searched using the protocol that will be used to connect to the specified URL as the key.
- If the protocol supplied maps to a XURLStreamHandler registered with the XURLStreamHandlerFactory, then the XURLStreamHandler object is called in order to provide the URLConnection object required.
- If the protocol does not match any XURLStreamHandlers stored within the XURLStreamHandlerFactory, then an URL object is created from the supplied string representation in the XURL, and the URLConnection object is obtained directly from that.

Providing this extra framework, JCP supplies the ability to override existing protocol formats, in order to introduce any specific enhancements, i.e. HTTPS, where our SSL is directly tied in with our Certificate Storage interfaces.

NOTE Please consult your API for more information this: uk.co.jcp.tbase.xurl.

Chapter 3

Routing

The router component has two specific functions:

- Provide a means of authorising requests to particular services
- Provide a means of controlling the flow of transactions

The router has a rule based architecture that allows the developer to modify the behaviour of a transaction in many cases without recourse to writing Java Code. This is achieved by producing a set of XML rules that determine the flow of a message through the system, and comparing these rules to the values held in the v's message.

The following sections define the concepts used in the router architecture, the function of the router, and the default implementation for Identrus message processing.

Messages

An iPlanet Trustbase Transaction Manager message is an entity that moves from one part of iPlanet Trustbase Transaction Manager to another. The message is an encapsulation of the message sent to iPlanet Trustbase Transaction Manager together with internal data about the message. The basic components of an iPlanet Trustbase Transaction Manager message are as follows:

- Message Type
- Attributes
- Data Content

NOTE Further details of the content of an iPlanet Trustbase Transaction Manager message can be found in the API documentation for uk.co.jcp.tbase.environment.Message.

The Message Type and Attributes determine how the framework itself will handle the iPlanet Trustbase Transaction Manager Message. Anything in the Data Content is treated as opaque by the iPlanet Trustbase Transaction Manager framework; i.e. any changes to the Data Content will not be visible to the iPlanet Trustbase Transaction Manager framework.

The Router determines the state of a message by interacting with the message's attributes. The router uses a set of pre-conditions based on attribute values to determine the state of the message and therefore which actions should be applied to that message. These pre-conditions and actions are stored in the form of router rules that are described later.

Message Attributes

Message Attributes are just name-value pairs in the same way as Java properties. Attribute names and values are always string types.

A protocol handler, message reader, message writer, the router or a service may read and write message attributes. This allows each stage in the message processing pipeline to add its own state information.

Attributes may be defined by any user application code, but a number of standard attributes are used by the iPlanet Trustbase Transaction Manager framework. These are defined below:

- messageType
  This attribute is treated specially by the Message API class, the message type is used in conjunction with the message MIME type to determine which Message Readers, Message Writers and Protocol Handlers should be used in the processing of this message. It is the responsibility of the Protocol Handler for the MIME type to set this attribute correctly. The messageType attribute may change during the processing of a message.
- security.role
  This attribute contains the security role to which this message has been attributed. The authenticator service sets this attribute to one of the following values: the role name for successful authentication, default when no role can be assigned
- security.cert.dn.<n> [1 <= n <= cert chain length]

  security.cert.sn.<n> [1 <= n <= cert chain length]

  These attributes describe the certificate chain authenticating a message. Each certificate in the chain [starting with the subject cert, numbered 1] is identified by it's issuer Distinguished Name and serial number
- security.cert_encoding
  This is always BASE64.
- security.cert_present
  Is there a valid certificate present for authentication purposes?
- security.username
  A username for authentication purposes.
- security.password
  A password for authentication purposes.
- security.user_pass_present
  Is there any valid username / password evidence for authentication present on this message.
- security.auth_failed
  Authorisation failed when an executeServiceDirective was called, this means that there was no role / serviceName mapping in iPlanet Trustbase Transaction Manager. This means that the message is not authorised to be sent to the requested service.

NOTE The attribute names for security specific attributes are declared as constants in uk.co.jcp.tbaseimpl.authenticator.SecurityContext Also see uk.co.jcp.tbase.environment.attribute.

Identrus Message Attributes

In the case of the Identrus message processing, the following additional message attributes are used:

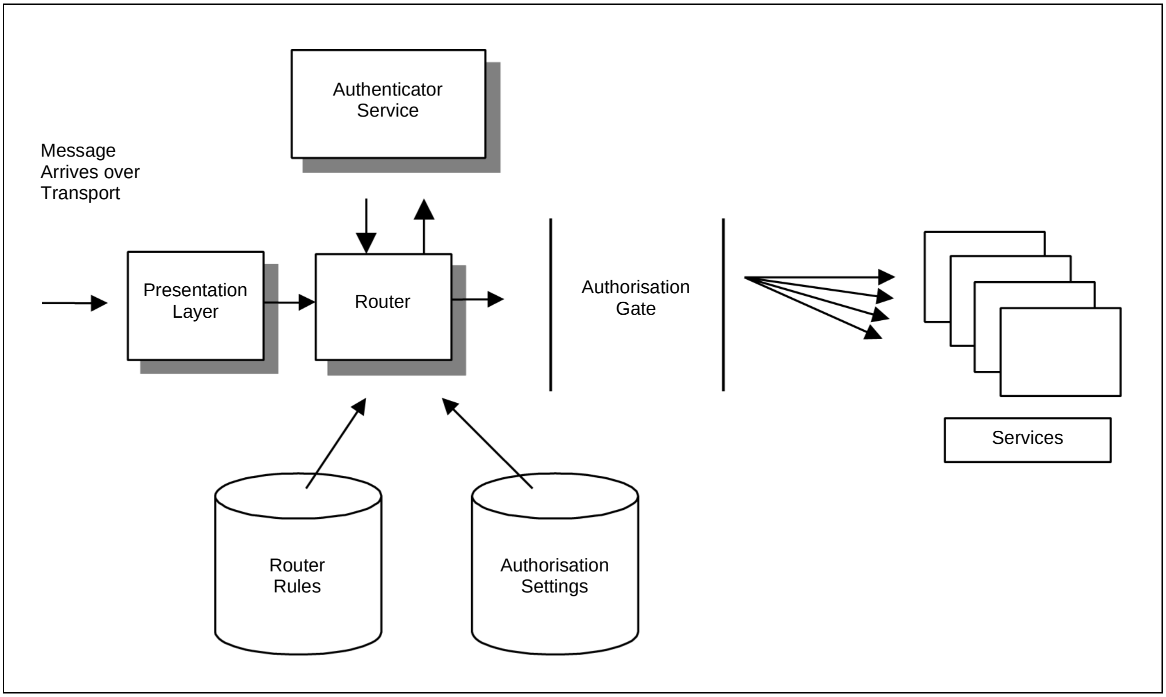
- DocType
  The DOCTYPE of the XML message that will identify the type of message. e.g. CSCRequest or PingRequest
- SigningCertPurposeId
  This optional attribute is only interpreted by the default Identrus Message Writer, if present it specifies the Certificate Store Attribute to be used to extract the signing certificate and private key. If this is not present then the message writer assumes that the Level 1 Inter-participant Signing Certificate is to be used.
- XMLSystemId
  This contains the 'system identifier' attribute from the incoming XML message - it is used by the Identrus Message Writer to fill in the XML System identifier on the corresponding response message.

NOTE The attribute names for Identrus specific attributes are declared as constants in com.iplanet.trustbase.identrus.IdentrusConstants.

Router Architecture

The Router performs the core rule based message routing which is central to the flexibility offered by iPlanet Trustbase Transaction Manager, but it is also responsible for gating access to services on the basis of authentication information and authorisation settings. Figure 3-1 shows the basic elements involved with getting a message through iPlanet Trustbase Transaction Manager and into the service that can process it.

Figure 3-1 Router Architecture



- The message arrives at iPlanet Trustbase Transaction Manager and makes its way through the presentation layers (protocol handlers and message readers), during which time the messageType is identified and set and a number of other standard attributes are also set. These attributes form the basis of the authentication evidence for the message.

- The message then arrives at the router. All messages are initially sent to the service identified by the name "Authenticator". There is a default 'Authenticator' service that comes with iPlanet Trustbase Transaction Manager, but this may be replaced by an alternative implementation if required.
- The message is sent to the default Authenticator Service that examines the attributes on the message to assess the authentication level of the message. The function of this service is defined in more detail below.
- The message arrives back at the router with a security context attribute (security.role) set, this context attribute determines the Role into which the message has been authenticated.
- The target service is located by processing the message through the routing rules and then the message is sent to the service. The detailed definition of these rules is given later on in this section.
- Before the message arrives at the service it must pass through an authorisation gate, this authorisation gate checks that there is a mapping between the role attribute assigned to the message and the target service. If such a mapping exists then the message is passed to the service, if no such mapping exists then the message is sent to an authorisation error service.

The stages described above are the basic stages which all iPlanet Trustbase Transaction Manager messages go through. The processing pipeline has been streamlined from the developers point of view when working with Identrus XML messages and is defined in the “Default routing” section in this chapter.

Authentication and Authorisation

The authentication mechanisms of iPlanet Trustbase Transaction Manager are not a separate component. Authentication data is gathered by the default iPlanet Trustbase Transaction Manager framework, this can be added by domain specific services. These services must map an identity onto a particular set of attributes, and iPlanet Trustbase Transaction Manager then uses these attributes to impose access control on other services.

Authentication

In order to ensure that the authentication and authorisation mechanisms are suitable for the widest variety of organisation, the iPlanet Trustbase Transaction Manager framework does not perform the authentication operations itself, but expects a service to be capable of verifying identity and mapping this to a particular role or roles. The system comes supplied with a basic identity that uses a digitally signed data item and an accompanying X509 digital certificate. The service is capable of mapping this data into a role or set of roles The basic authentication information may be presented in one of two ways:

- Implicitly - within the Transport protocol e.g. SSL
- Explicitly - within the message content e.g. signed message block of an Identrus XML message

The architecture is designed to reduce both of these to explicit authentication by removing the certificate used in the SSL handshake and using it to decorate the message passed to the authentication service. This allows the target system to impose different levels of authentication that would be difficult to achieve with an implicit 'black box' approach to transport level authentication.

The default service also provides a means of mapping a username and password onto a role for use by the router when authorising requests. This mechanism is commonly used when presenting configuration and administration information using the HTMl templating mechanism. It is recommended that the closed community mechanism be replaced by an enterprise wide username and password mechanism if the platform is to be used for large scale HTML based applications.

Authorisation

Given that we can identify the originator of a message, either through the digital certificate or an alternative means, we now require a mechanism of:

- Identifying the appropriate level of authorisation for the user
- Enforcing access to each service on the basis of authorisation

The identification of the proper authorisation level (or role) is performed by a service named 'Authenticator' in iPlanet Trustbase Transaction Manager. This service has a default implementation that uses the username/password tables and certificate tables in the authorisation database to determine a mapping for the security attributes on the message. The mapping of this information is described in the iPlanet Trustbase Transaction Manager Configuration & Installation Guide. Enforcing the access to each service is done automatically by the router. If a routing rule is matched and the body contains an 'executeServiceDirective' then before that directive is executed the router establishes if there is a mapping between the role assigned to the message and the required service. These mappings are held in the authorisation table and this table's use is described in the Trustbase Configuration & Installation Guide. If the mapping exists then the message is passed to the service. If no mapping exists then a new attribute is added to the message "security.auth_failed" and the message is sent back to the router. The routing rules should always detect an authorisation failure and execute an appropriate service to send an 'unauthorised' response to the client. Services may be executed unconditionally by using the 'unauthorisedExecuteServiceDirective' that does not attempt check for authorisation.

Default routing

In order to provide a simple development path for Identrus Messaging services, iPlanet Trustbase Transaction Manager provides a default routing mechanism for Identrus Messages. This default routing is applied to all Identrus Services developed and deployed as described in Chapter 8, “Building Identrus solutions” of this document.

It is anticipated that this default routing will be sufficient for most Identrus Services in the near term. Where the default routing is inadequate, the developer must write new routing rules.

Router Rules

The updates made to the Router for version 2.2 of the iPlanet Trustbase Transaction Manager allow a very simple approach to basic rules handling to be taken. In previous versions of the system, all services had to define a set of rules that explicitly mapped the transport of a message to and from the service. Now that rules can be defined dynamically by the system, this basic behaviour can be automatically defined. We now discuss how this is achieved.

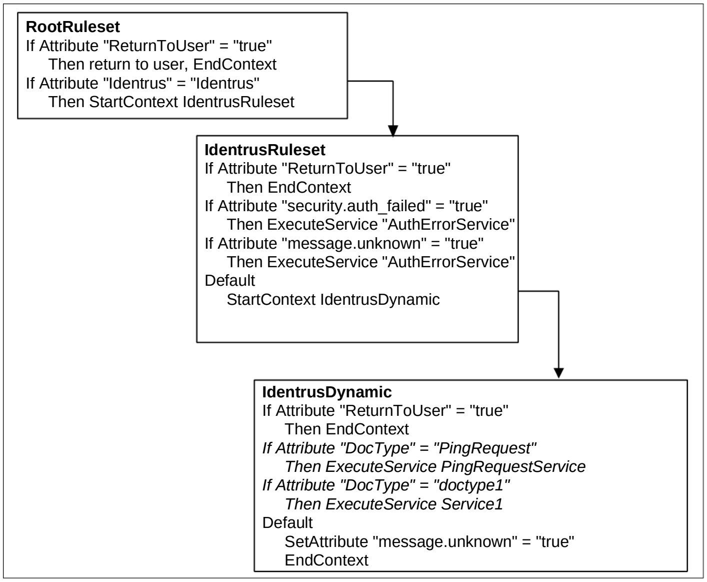
Routing to service

The deployment descriptor contained in the service JAR file will name the service that it contains. At system startup, the rules handler will determine which services are to be loaded, and for each service in the list will dynamically create a routing rule that forwards messages with the specified doctype to the service. If a service has a user-defined ruleset, the dynamically created rule will pass the message to this ruleset instead of directly to the service. This user defined ruleset lives in the JAR file with the service implementation and will normally not be required for services with simple message flows.

Return path

In order that the routing be kept as simple as possible, a message will be returned to a user if a specified attribute has been set on the message. This "ReturnToUser" attribute should be set by the service once processing is complete. This also applies to error messages produced at the service level.

Figure 3-2 Default routing rules



```
RootRuleset
If Attribute "ReturnToUser" = "true"
    Then return to user, EndContext
If Attribute "Identrus" = "Identrus"
    Then StartContext IdentrusRuleset

IdentrusRuleset
If Attribute "ReturnToUser" = "true"
    Then EndContext
If Attribute "security.auth_failed" = "true"
    Then ExecuteService "AuthErrorService"
If Attribute "message.unknown" = "true"
    Then ExecuteService "AuthErrorService"
Default
    StartContext IdentrusDynamic

IdentrusDynamic
If Attribute "ReturnToUser" = "true"
    Then EndContext
If Attribute "DocType" = "PingRequest"
    Then ExecuteService PingRequestService
If Attribute "DocType" = "doctype1"
    Then ExecuteService Service1
Default
    SetAttribute "message.unknown" = "true"
    EndContext
```

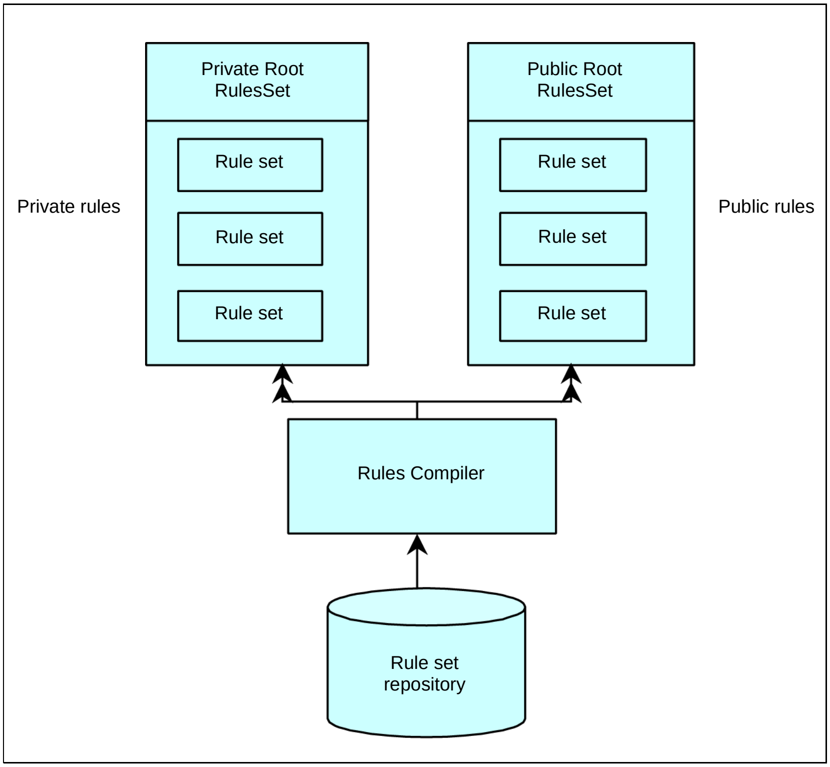
Advanced Routing

This section describes routing rules in detail, for simple Identrus services the developer should not have to write any routing rules in order to get messages to their service. If the default routing is inadequate or non-Identrus message processing is required then developers must resort to writing their own rules.

Routing Rulesets

The routing tables have two logical areas, one private and one public. Public rule sets may be added or configured through the management interface, private rule sets are used by iPlanet Trustbase Transaction Manager to provide basic services and may not be configured by the developer or systems administrator. At the top of this structure is the Root Rule Set which governs the way in which the private rules are executed and if none of the private rules match the parameters then the Public Root Rule Set is executed. System developers may then link their own task Rule Sets from this Rule Set (See Rule set definition in the next section)

Figure 3-3 Rule Sets



Each area contains a number of rule sets. Each public rule set will have a name, usually indicating the business task performed by the set of rules. The public rule sets are loaded from disk each time the platform starts up, and may be loaded explicitly by name as the result of a rule execution in the root rule set.

When a message is passed to the router, it will have already been associated with a context. Each context has an association with a rule set. This implies:

- The message is operating in its root context, and has the root rule set associated with it.
- The message is operating in a sub-context, and has a rule set from the public rules area associated with it.
- The exception to this is if an iPlanet Trustbase Transaction Manager message is being processed in which case a private rule set will be associated.

Rule Sets are processed sequentially, and the first rule to match the current conditions is used. The router takes each rule from the rule set in turn, starting with the first, and attempts to match the message type with a rule pre-condition. If a match is found then the action is executed. If no match is found then an error condition is returned. It is important to bear in mind that because rules are processed sequentially it is necessary to keep the more explicit rules at the beginning of the rule set with more general rules occurring later. Otherwise it is entirely possible that a general rule could preclude other rules from ever being executed.

It is considered good practice to keep the root rule set as small as possible. To achieve this, a rule in the root rule set should only match the first message in a business task i.e. an unsolicited message. The action should always be to start a new context, indicating the name of a rule set (and potentially the rule name to start at), that the message should be executed in. In doing this, the router will create a new context and associate the named rule set with it. The message will now be processed in the same manner.

A rule set is a single complete XML document that conforms to the DTD outlined in the next section. When the router starts up it looks in the Router section of the iPlanet Trustbase Transaction Manager initialisation file for the directory to find the rules that have been written by hand. It then looks for .xml files, and compiles these into the public rule set area. Each Rule Set contains zero or more rules as shown in the DTD fragment shown below:

```
<!ELEMENT ruleSet (rule*)>
<!ATTLIST ruleSet name CDATA #REQUIRED>
```

Each rule in the ruleset has a structure that is defined in the next section.

Router Rule Syntax

This section works through the Rule set DTD describing the purpose of each component. Each rule has three components:

- An attribute which names the rule. This is optional; the name may be used when starting new contexts if a specific rule should be evaluated.
- A set of pre-conditions that must all be met before the execution of the directives in the body.

- The body of the rule, this is the list of actions to be carried out if the pre-conditions are all met.

A typical rule name is illustrated below.

```
<!ELEMENT rule ( preconditions, body )>
<!ATTLIST rule name CDATA #IMPLIED>
<!ELEMENT preconditions (attributeCondition*)>
```

The preconditions are made up of zero or more Attribute conditions. Each attributeCondition must be satisfied for the body to be executed.

```
<!ELEMENT attributeCondition EMPTY>
<!ATTLIST attributeCondition name CDATA #REQUIRED>
<!ATTLIST attributeCondition operator ( greaterThan |
greaterThanEqualTo | lessThan | lessThanEqualTo | notEqual |
startsWith | contains | endsWith | equals ) "equals">
<!ATTLIST attributeCondition valueType ( any | null | string ) "string">
<!ATTLIST attributeCondition value CDATA #IMPLIED>
```

Each AttributeCondtion is evaluated by iPlanet Trustbase Transaction Manager as though it were in the form:

```
IF ( name operator value ) then pre-condition met
```

The valueType is normally defaults to 'string' and is not required in rules that have a value field. Where there is a requirement to test for the presence of an attribute, or if it needs to be tested against null, then the valueType field is used and the value field is ignored / not required.

```
<!ELEMENT body ((setAttribute | executeServiceDirective |
unauthorisedExecuteServiceDirective )*, (startContextDirective |
endContextDirective | returnToUserDirective )?)>
```

Once the pre-conditions have been met the body of the rule can be executed. The body of the rule may contain multiple actions that are carried out sequentially in the order that they appear. There are three actions, that are considered to be 'terminal' actions, which may only appear once in a rule and they must appear as the final action. Table 3-1 shows each of the allowable body directives.

Table 3-1 DTD Rule Body

Directive	Description
SetAttribute	Sets a iPlanet Trustbase Transaction Manager attribute on a message
executeServiceDirective	Transfers control from the router to the named service, ensuring that the mandatory authorisation checks are passed
unauthorisedExecuteServiceDire ctive	Transfers control from the router to the named service, without performing any of the mandatory authorisation checks.
ForkContextDirective	Duplicates an execution environment
EndProcessDirective	Discards the current execution environment, and returns control to the user
StartContextDirective	Transfers router control from one rule set to another
EndContextDirective	Ends the current context, and starts rule search again
ReturnToUserDirective	Returns control from the router to the system user

```
<!ELEMENT setAttribute EMPTY>
<!ATTLIST setAttribute name  CDATA #REQUIRED>
<!ATTLIST setAttribute value  CDATA #REQUIRED>
<!ATTLIST setAttribute location (MESSAGE | CONTEXT) "CONTEXT">
<!ATTLIST setAttribute type  (ASCEND_TRANSITIVE | NONE |
INHERIT_TRANSITIVE | INHERIT) "ASCEND_TRANSITIVE">
```

The setAttribute action allows the router to add attributes to either the message or the context. The name, value, location and type fields are all the same as the settings for creating an attribute via the API.

NOTE See API documentation for uk.co.jcp.tbase.environment.attribute.Attribute for more details.

```
<!ELEMENT executeServiceDirective EMPTY>
<!ATTLIST executeServiceDirective name CDATA #REQUIRED>
```

An executeServiceDirective must specify the name of the service to invoke. This form of executing a service will always perform an authorisation check before passing the message to the requested service.

The name of the service relates to the name given to the service in the ServiceRegistry section of the tbase.properties file where each new service.description entry consists of a service name and a classpath for that particular service.

```
<!ELEMENT unauthorisedExecuteServiceDirective EMPTY>
<!ATTLIST unauthorisedExecuteServiceDirective name CDATA #REQUIRED>
```

An unauthorisedExecuteServiceDirective must specify the name of the service to invoke. This form of executing a service will not perform any authorisation check before passing the message on to the service.

The name of the service relates to the name given to the service in the ServiceRegistry section of the tbase.properties file where each new service.description entry consists of a service name and a classpath for that particular service.

```
<!ELEMENT startContextDirective EMPTY>
<!ATTLIST startContextDirective ruleset CDATA #REQUIRED>
<!ATTLIST startContextDirective rule CDATA #IMPLIED>
```

In executing a startContextDirective the router discards the current rule set it has been searching and loads the rule set named in the directive. If an individual rule is also named in the directive then the router attempts to find that rule in the new rule set, otherwise the router begins at the start of the rule set and searches for the first rule with fulfilled preconditions.

A start context directive must contain a name for the rule set to be used. It may optionally contain a rule name to be executed in the specified rule set.

```
<!ELEMENT endContextDirective EMPTY>
```

An end context directive has no attributes or children.

```
<!ELEMENT returnToUserDirective EMPTY>
<!ATTLIST returnToUserDirective endContext ( true | false ) #REQUIRED>
```

A return to user directive tells the router whether to end the current context or not.

In some cases the context serves no further purpose once a service has been executed on behalf of the user and may therefore be discarded. However it is possible to envisage scenarios e.g. whenever authentication is required for a user to run services, where one would wish to maintain the 'Authorised' context between user interactions, thus allowing the user to repeatedly use services without being required to be authorised every time.

<!ELEMENT forkContextDirective EMPTY>

```
<!ELEMENT forkContextDirective EMPTY>
```

A ForkContextDirective produces a new environment identical to the current context.The message returned to the user will have the contextID set to the original context id, but will also contain a property FORKED_CONTEXT, with the new context id. This allows the client to subsequently interact with both environments.

```
<!ELEMENT endProcessDirective EMPTY>
```

EndProcessDirective terminates the current environment. It is designed to take some of the burden off the garbage collection by explicitly terminating an environment and removing it from persistent storage.The EndProcessDirective is also a ReturnToUserDirective.

NOTE Example rules and rulesets for core Trustbase services that comply with these data structures can be found in <install_dir>/TTM/current/Config/Rules/Public.

Complete Router Rule DTD

The complete DTD that specifies all of the allowable routing rules within iPlanet Trustbase Transaction Manager is shown below.

```
<!-- A rule set is zero or more rules -->
<!ELEMENT ruleSet (rule*)>
<!ATTLIST ruleSet name CDATA #REQUIRED>
<!-- A rule optionally has a name attribute and precodonditions and body elements -->
<!ELEMENT rule ( preconditions, body )>
<!ATTLIST rule name CDATA #IMPLIED>
<!-- preconditions element is a set of zero or more attribute condition elements -->
<!ELEMENT preconditions (attributeCondition*)>
<!-- an attributed condition is an empty tage with name, valueType and value attributes -->
<!-- the name is mandatory. The valueType may be any, null or string. if string is selected -->
<!-- then value must be present. If any is selected then the value matches any value and if -->
<!-- null is selected then the value is disregarded and assumed to be null -->
<!ELEMENT attributeCondition EMPTY>
<!ATTLIST attributeCondition name CDATA #REQUIRED>
<!ATTLIST sttributeCondition operator ( greaterThan | greaterThanEqualTo | lessThan | lessThanEqualTo |
notEqual | startsWith | contains | endsWith | equals ) \equals\>
<!ATTLIST attributeCondition valueType ( any | null | string ) \string\>
<!ATTLIST attributeCondition value CDATA #IMPLIED>
<!-- The body contains one of four directives -->
<!ELEMENT body ((setAttribute | executeServiceDirective | unauthorisedExecuteServiceDirective )*,
(startContextDirective | endContextDirective  | returnToUserDirective )?)>
<!-- A setAttribute directive -->
<!ELEMENT setAttribute EMPTY>
<!ATTLIST setAttribute name  CDATA #REQUIRED>
<!ATTLIST setAttribute value  CDATA #REQUIRED>
<!ATTLIST setAttribute location (MESSAGE | CONTEXT) \CONTEXT\>
<!ATTLIST setAttribute type  (ASCEND_TRANSITIVE | NONE | INHERIT_TRANSITIVE | INHERIT)
\ASCEND_TRANSITIVE\>
<!-- A start context directive must contain a name for the rule set to be used -->
<!-- it may optionally contain a rule name to be executed in the specified rule set -->
<!ELEMENT startContextDirective EMPTY>
<!ATTLIST startContextDirective ruleset CDATA #REQUIRED>
<!ATTLIST startContextDirective rule CDATA #IMPLIED>
<!-- An end context directive has no attributes or children -->
<!ELEMENT endContextDirective EMPTY>
<!-- An execute service directive must specify a name for the service to be executed -->
<!ELEMENT executeServiceDirective EMPTY>
<!ATTLIST executeServiceDirective name CDATA #REQUIRED>
<!-- An unauthorisedExecute service directive must specify a name for the service to be executed -->
<!ELEMENT unauthorisedExecuteServiceDirective EMPTY>
<!ATTLIST unauthorisedExecuteServiceDirective name CDATA #REQUIRED>
<!-- A return to user directive has one attribute specifying whether to end the current context -->
<!ELEMENT returnToUserDirective EMPTY>
<!ATTLIST returnToUserDirective endContext ( true | false ) #REQUIRED>
<!-- A fork context directive has no attributes or children -->
<!ELEMENT forkContextDirective EMPTY>
<!-- An end process directive has no attributes or children -->
<!ELEMENT endProcessDirective EMPTY>
```

Chapter 4

Configuration Management

There are two elements to the Configuration Manager sub-system, the Configuration Manager itself and its Configuration Store. The Configuration Manager is the central access point for all configuration requests and is responsible for maintaining Configuration Objects in a store. The Configuration Manager co-ordinates access to these objects for both read and update purposes.

Configuration Objects

Configuration Objects hold persistent data for Services and their Configuration Services. A given Service may reference any number of Configuration Objects, and any number of Services or Configuration Services may reference a given Configuration Object. Note that iPlanet Trustbase Transaction Manager components themselves use Configuration Objects. A Configurable Entity is any Service or component that uses Configuration Objects and the Configuration Manager. A Configuration Service is a Service that implements a read-write interface to the Configuration Object, which usually involves some graphical or HTML based interface allowing a user to change the values stored in the configuration object.

All configuration objects must implement the ConfigurationObject interface to ensure that the ConfigurationManager can manipulate them correctly. Configuration Objects are indexed in the store by ConfigUID objects.

NOTE Further details of the interface that a ConfigurationObject must present can be found in the API documentation for uk.co.jcp.tbase.config.ConfigurationObject.

Configuration Manager

A Service uses the SingletonConfigManager class to gain a reference to the Configuration Manager.

The Configuration Manager allows callers to get readable or writeable versions of Configuration Objects identified by ConfigUID objects.

There are two methods for getting a Configuration Object:

- The first returns a readable deep copy of the Configuration Object that can be used to set up a Service or for populating Configuration Service views. Configuration Objects gained by this method cannot have their changes applied back to the Configuration Store - this is because the returned Configuration Object does not contain a valid ConfigurationLock object. This form of Configuration Object will always be provided, regardless of whether there is an outstanding lock on the requested object, i.e. requests for read-only copies of ConfigurationObjects are never refused by the ConfigurationManager.
- The second method provides a deep copy of the Configuration Object that contains a configuration lock valid for a fixed period of time. This lock period will, itself, be configurable. After this period the lock will become invalid and any attempt to apply a change to the Configuration Store will be denied. The lock expires to provide a simple solution to deadlocking issues caused by developers not releasing allocated locks.

Updates and deletions of Configuration Objects can only be performed with a Configuration Object that contains a valid lock object.

NOTE A Service can only register a Configuration Object with a given ConfigUID if one does not already exist with that ConfigUID.

Configurable Entities may register for notification when a Configuration Object changes. When a change occurs, registered entities receive an event object that contains a read-only copy of the changed Configuration Object.

NOTE Further details of the methods that the ConfigurationManager. provides to manipulate ConfigurationObjects can be found in the API documentation for
uk.co.jcp.tbase.config.SingletonConfigManager
uk.co.jcp.tbase.config.ConfigManager.

Configuration Store

The Configuration Store is responsible for maintaining persistent storage of Configuration Objects. The version supplied with iPlanet Trustbase Transaction Manager uses an underlying JDBC implementation to store data. Alternative Configuration Stores may be implemented to store Configuration Objects in any given format.

The Configuration Store implements only primitive read, write and delete methods to access Configuration Objects via their ConfigUID key - the Configuration Manager handles higher level semantics such as locking.

The default JDBC store implementation cannot be replaced by developers and its interface is not public.

Configuration Services

A Configuration Service is the element that knows of the configurable attributes of a ConfigurationObject. It provides the functionality to present these attributes to an administrator via user interface, iPlanet Trustbase Transaction Manager itself uses HTML forms to present configuration data.

The Configuration Services supplied with iPlanet Trustbase Transaction Manager use the HTTP Reader and ScriptWriter classes to generate HTML pages/forms that allow information to be changed by the administrator. The Configuration Service is capable of configuring all the individual iPlanet Trustbase Transaction Manager components, including protocol analysers, message analysers, readers and writers, router and services. Upon receipt of updated attributes the Configuration Service is responsible for registering the changes with the Configuration Manager that will in turn notify other interested parties.

At least one Configuration Service will exist for each Configurable Entity registered in the platform. Note that there will only be a single runtime instance of any given Configuration Service per deployment, no matter how many instances of the actual entities that it configures are in existence.

NOTE There are no API classes associated with a Configuration Service, it is a normal iPlanet Trustbase Transaction Manager service and therefore must implement the uk.co.jcp.tbase.service.Service interface.

Chapter 5

Standard Services

Services provide actions in decision making process. Each service is a plug in component to the iPlanet Trustbase Transaction Manager's framework providing an implementation of the iPlanet Trustbase Transaction Manager's Service interface (See Installation and Configuration Guide for more details.

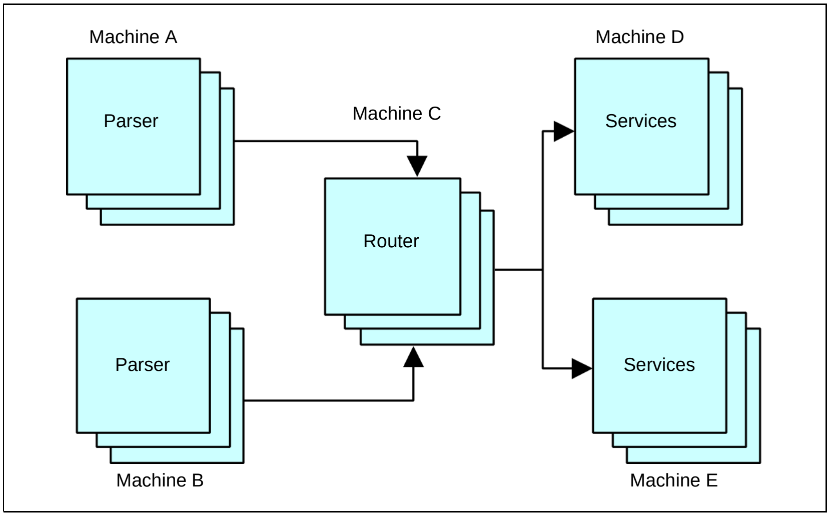
iPlanet Trustbase Transaction Manager is designed to run on a variety of platforms, in particular application servers. These provide a solution with the ability to service a large number of users at any single point in time. This is achieved by replicating the components i.e. running classes over a number of Java Virtual or physical machines. The iPlanet Trustbase Transaction Manager's Framework has been designed to fulfil all of these requirements. Each of the major sub-systems is capable of being replicated a number of times over a number of physical machines, but to achieve high volume processing the iPlanet Trustbase Transaction Manager's Framework imposes a number of constraints on a service.

Overview

A service should be:

- Stateless - Each component should be capable of processing a message irrespective of the context of the message
- Granular - Each component must perform a limited number of tasks, reducing the risk of a single point of failure

Figure 5-1 Component replication



A service is directly related to the processing of a message and must be stateless to allow replication.

These services usually fall into one of three major categories:

- Authentication and Authorisation
- Business processing including integration into existing systems
- User interaction

In dividing the function into discreet services that service one of these areas a variety of applications may be built over the same common infrastructure.

In particular, a common set of authentication and authorisation services will be used across the enterprise so devising services that only perform one or both of these tasks reduces the work required for each application.

Chapter 6

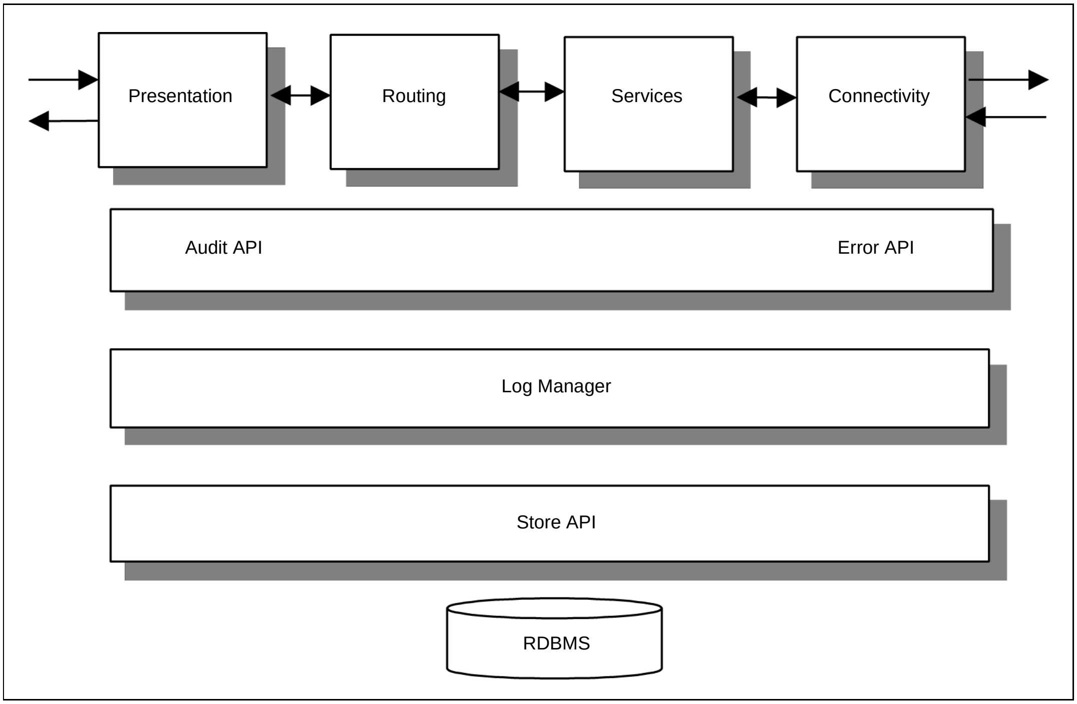
iTTM Logging: Error, Audit and Raw

iPlanet Trustbase Transaction Manager provides a set of libraries for use with both error and audit logging. The default implementation of these libraries allow the framework and business components to store information within a relational database for use by external browsers and error management consoles.

Both error and audit logs are controlled by a log manager. The log manager stores the data in an appropriate location depending upon its current configuration. This design provides a location independent mechanism for recording error and audit information, and allows components to be re-used between different iPlanet Trustbase Transaction Manager implementations. This log manager is activated each time iPlanet Trustbase Transaction Manager boots. Initially the configuration of the log manager is read from an initialisation file, but in subsequent boot sequences this is read from the persistent configuration object generated by the manager.

Overview

Figure 6-1 iPlanet Trustbase Transaction Manager log manager



The data to be logged by the log manager is supplied by the requesting component in the form of an error or audit log object. The log manager will check the type but not the content of the object being logged. The type checking involved ensures that the object passed is a sub-class of the iPlanet Trustbase Transaction Manager defined error log and audit log classes. This allows the developer to define new error and audit log types for use specifically by the solution e.g. Billing logs.

The Log manager is abstracted from the storage of the data by a Store API. This allows different implementations of a Log e.g. Billing information logs to be stored in different physical repositories, or particular log types e.g. Errors to generate events that are passed to Third Party monitoring systems.

Audit logs

Audit logs are generated at particular points in the lifetime of the iPlanet Trustbase Transaction Manager platform, specifically:

- Service start up events
- Service shutdown events, both normal and abnormal
- All configuration changes
- All security domain changes

These standard iPlanet Trustbase Transaction Manager audit log types are defined in the uk.co.jcp.tbaseimpl.log.audit.type package.

Audit Logging an Event

To cause an entry to be made in the iPlanet Trustbase Transaction Manager Audit Log, create a new instance of the AuditObject type that you wish to log (which must be a sub-type of uk.co.jcp.tbaseimpl.log.audit.AuditObject) and pass the instance to the static method AuditLog.log(...).

An example of an Audit being logged is shown below

```
package com.iplanet.trustbase.app;
public class DummyService
{
......
private static final String SERVICE_STARTED = "START";

// Audit the occurrence of this event in the Audit Log
AuditLog.log(new OperationBeginAudit( this.getClass(),
SERVICE_STARTED, new String[]{ serviceName.toString() });
......
}
```

The three parameters supplied to construct all AuditObjects are as follows:

- this.getClass()
  The first parameter is a class object, whose name will be used as the first part of the internationalisation bundle key for the message. This is normally the class that is logging the audit, to make it easy to relate audit messages to packages. But this may be any class object.

- SERVICE_STARTED
  This is any string which when concatenated with the class name provides a unique bundle key to locate the text of the message for audit purposes. See below for a description of the AuditBundle keys and values.
- new String[] { serviceName.toString() }
  This is an array of strings which are parameters that are combined with the AuditBundle message using the standard java.text.MessageFormat class to provide the final audit message.

As mentioned above, there needs to be an AuditBundle file on the class path and registered with the bundle manager (see “Defining New Audit Types” section) so that the actual audit message can be looked up. For the example above there would be a file AuditBundle.properties created in …./com/iplanet/trustbase/app which contained the following line:

```
com.iplanet.trustbase.app.DummyService_START=The service {0} has begun.
```

This is the fully qualified class name provided by the 'this.getClass()' parameter followed by a '_' character, followed by the SERVICE_STARTED string.

The message that would be associated with this audit would be 'The service {0} has begun.' Where the java.text.MessageFormat class will replace '{0}' with entry 0 in the String array passed to the audit constructor.

NOTE The API classes associated with Audit Logging are uk.co.jcp.tbaseimpl.log.audit.AuditLog, uk.co.jcp.tbaseimpl.log.audit.AuditObject and a number of standard concrete Audit Classes in uk.co.jcp.tbaseimpl.log.audit.type.

Defining New Audit Types

The developer may define new audit log types to allow application or domain specific messages. These must be a sub-class of the uk.co.jcp.tbaseimpl.log.audit.AuditObject class.

The most common use of these extended audit types is to log information about messages processed by the iPlanet Trustbase Transaction Manager Service.

Having implemented the new Audit type, the following steps must be completed in order to get the audit type to appear in the standard iPlanet Trustbase Transaction Manager Audit log.

- Create the new Audit class and put it into the JAR file for the service that will utilise the Audit. More details of deploying an iPlanet Trustbase Transaction Manager service can be found in later chapters.
- Create the relevant resource bundles for the audit messages. In order to allow simple localisation of messages into various languages, all audit type strings that appear in the iPlanet Trustbase Transaction Manager audit log are defined in resource bundles. These bundles are located in the same package (i.e. same directory) as the new AuditType class. The format of the name of this resource bundle must be the same as that defined by the standard Java java.util.ResourceBundle class. The format of the bundle is a series of name/value pairs. Where the name is constructed as fully-qualified-audit-type-classname_bundleKey, and the value is the String which should be placed in the AuditType field of the Audit Log.
- Create an entry in tbase.properties to register the new Audit Resource Bundle. To do this, add an entry in the section [`BundleProviderManager/Audit`] that references the fully qualified name of the new Audit Resource Bundle.
- Finally enable the new Audit Type in tbase.properties by adding an entry in the following section [`uk.co.jcp.tbaseimpl.log.present.audit.AuditLogPresentationConfigService`] for each of the new Audit Type classes defined by the service.
- Restart iPlanet Trustbase Transaction Manager to activate the new Audit Types, this is required to enable a new service and is therefore not an action normally associated with new Audit Types.

An example of an AuditObject implementation is shown below:

```
package uk.co.jcp.tbaseimpl.log.audit.type;
import uk.co.jcp.tbaseimpl.log.audit.*;

/** This class represents an audit object for recording that an
operation has begun
 */
public class OperationBeginAudit extends AuditObject
{
    public OperationBeginAudit ( Class auditClass , String
bundleKey , String [] params )
    {
        super ( auditClass , bundleKey , params );
    }
    public static String getAuditTypeString()
    {
        // The bundleKey is added to this class name to determine
the
        // actual string to be writteninto the AuditType field of
the log
        String bundleKey = "OPERATION_BEGIN";
        return
uk.co.jcp.tbaseimpl.log.audit.AuditObject.getAuditTypeString (
uk.co.jcp.tbaseimpl.log.audit.type.OperationBeginAudit.class,
bundleKey );
    }
}
```

The following file defines the Resource bundle and is located in the package hierarchy in the same place as the OperationBeginAudit class and will be called TheNewAuditBundle_en.properties.

```
uk.co.jcp.tbaseimpl.log.audit.type.OperationBeginAudit_OPERATION_BEGIN
= OPERATION_BEGIN
```

The new bundle is registered in tbase.properties with the following entry, note how the locale extension '_en' is not used in the bundle registration and the .properties extension is not required - this is in line with the standard Java java.util.ResourceBundle class:

```
[BundleProviderManager/Audit]
bundle=uk.co.jcp.tbaseimpl.log.audit.type.TheNewAuditBundle
```

The new Audit type is enabled by adding the following entry into tbase.properties.

```
[uk.co.jcp.tbaseimpl.log.present.audit.AuditLogPresentationConfigS
ervice]
auditlog.type.enabled=uk.co.jcp.tbaseimpl.log.audit.type.Operation
BeginAudit
```

Error handling and logging

Errors that occur in the iPlanet Trustbase Transaction Manager platform fall into two categories:

- Logic errors - Something about the data or processing is incorrect and the processing needs to reject a request gracefully.
- Exceptions - An unexpected condition has occurred, that breaks the processing logic

In both cases, iPlanet Trustbase Transaction Manager will use the Log manager to record that an error has occurred so that operational staff may perform appropriate analysis and corrective action.

Error Logging

iPlanet Trustbase Transaction Manager uses a single error log class that takes a severity, the class of object defining the error, and a programmer defined message, plus a set or zero or more string arguments which may be substituted into the message. The default error logging implementation, uk.co.jcp.tbaseimpl.log.error.ErrorLog, defines four constants that indicate the various severity levels:

Table 6-1 Error Severity Types

Constant	**Description**
INFORMATION	This constant is to be used to log informational events, such as unlikely sections of code being executed that are not necessarily errors - this should be used sparingly. This has a value of 0.
WARNING	This constant is to be used for error conditions that are expected and handled, but require logging for behaviour analysis. This has a value of 1.
ERROR	This constant is to be used for serious errors which indicate that something is inherently incorrect with the system or the information it contains, but that allow processing to continue, or be retried. This has a value of 2.
FATAL	This constant is to be used for fatal errors from which processing cannot recover, these errors would result in the abandoning of processing. This has a value of 3.

NOTE The API classes associated with Error Logging are uk.co.jcp.tbaseimpl.log.error.ErrorLog, uk.co.jcp.tbaseimpl.log.error.ErrorObject.

Logging an error from within iPlanet Trustbase Transaction Manager is as simple as calling ErrorLog.log(....) with an instance of an ErrorObject. There are many constructors for ErrorObject, but all of them have at least a string parameter that identifies the unique error code for this error.

The iPlanet Trustbase Transaction Manager error logging mechanism requires that every different occurrence of an error be given a code which unique throughout iPlanet Trustbase Transaction Manager. All of the error information for the Error Logging sub-system in iPlanet Trustbase Transaction Manager is contained in the following database tables:

All unique error codes have their details stored in Table 6-2:

Table 6-2 Error_codes

error_codes table	
errorcode	This is the unique errorcode string that identifies the error; this must be 7 characters exactly. The normal for an error code is XXXnnnn. Where XXX is a three-letter code for the service or subsystem and nnnn is a unique number. e.g. IPH0009 is an error in the Identrus Protocol Handler.
classname	The class from which this error is logged. This places a constraint that each error code may only be used from one place.
severity	This is the severity level of the error, described previously. Constants for each of the error severities can be located in the uk.co.jcp.tbaseimpl.log.error.ErrorLog class.
message	This is the localised version of the error message that will appear in the error log. Parameters may be used in this message as described by the standard Java class java.text.MessageFormat. The values to be placed in these parameters are passed in an array of strings that one of the ErrorObject constructors allows.

The actual error log table is described below, this table is not normally viewed by the administrator directly, instead there is an Oracle view called errorview that provides a resolved view of the errors that have been logged.

Table 6-3 Error

Error table	
errorid	This is a unique id for the error log entry; it is generated from a monotonic sequence that means that this field may be used to accurately order error messages in the order that they were logged.
errorcode	This is the errorcode of the error being logged, see the “error_codes table” description.
message	The final message string that is generated from the message string in the error_codes table combined with the variable parameters from the runtime system substituted in.
timestamp	This is an ORACLE DateTime field that identifies when the error was logged.
severity	This is the severity integer that is taken from the error_codes entry for this error.
classname	This is the classname that logged the error.
machineid	This is a string representing the IP address of the iPlanet Trustbase Transaction Manager that logged the error - this may be different in a multi-node IAS installation.
contextid	This context id field is for future expansion.

When an error is logged it is often accompanied by some free form string data which helps to store the context in which the error occurred to aid diagnosis. The most common example of such data is exception stack traces.

Table 6-4 Error Support

error_support table	
errorid	This links this entry to an entry in the “Error table”

error_support table	
datatype	This datatype is an arbitrary string identifier that categorises the data in the data field. The only value for this field defined by iPlanet Trustbase Transaction Manager is "STACKTRACE" which identifies the contents of the data field to be a Java Exception Stack Trace.
data	Free form string data

The tables described above encapsulate the whole data driven error logging mechanism that iPlanet Trustbase Transaction Manager supports.

Defining a New Error

Defining a new error is very simple; it involves making a new entry in the error_codes table. There are currently no tools to support this operation, but it may be accomplished through the use of basic SQL. The only consideration is that the error_code field must be unique - this is enforced by an ORACLE level table constraint, so if the new code is inserted without error then the code is unique.

A sample error code is inserted using the SQL defined below:

```
INSERT INTO error_codes values
(
'TST0001',
'com.iplanet.trustbase.sample.service',
'1',
'This is the only exception in the test service'
);
```

This error may then be logged using the following piece of code:

```
.....
}
catch (Exception exc)
{
    ErrorLog.log( new ErrorObject( "TST0001", exc );
}
.....
```

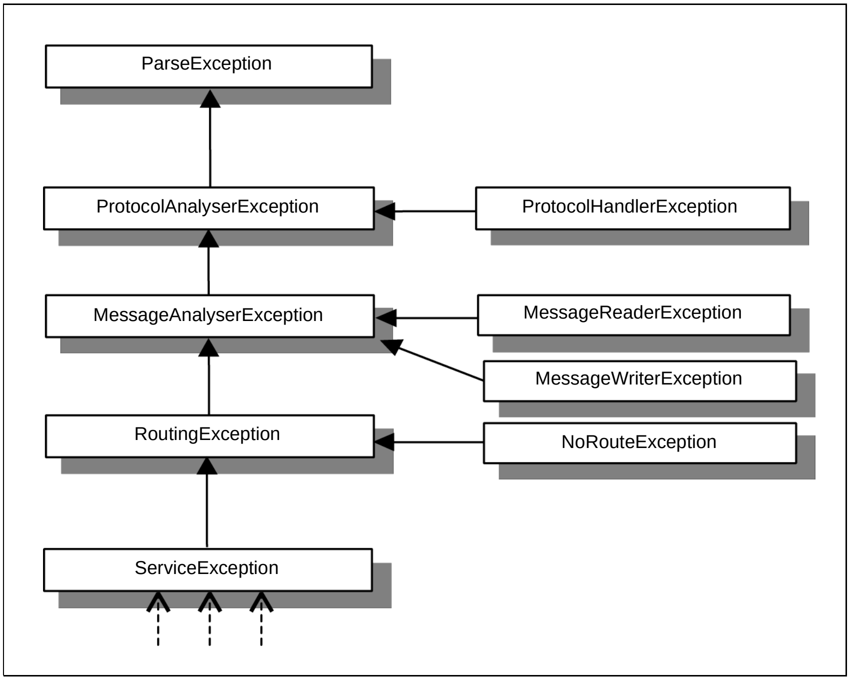
This will result in an entry being made in the error table as well as the related stack trace being logged in the error_support table.

Exception Handling

Exception handling, as with the logging of logical errors, should also log information before the appropriate exception is thrown, in order to enable users and developers of the system to analyse the situation that caused them to arise.

The iPlanet Trustbase Transaction Manager exception hierarchy is designed to allow the various components to throw exceptions back to their calling component without the calling component having to explicitly include handling code. This is achieved by having each successive 'level' of exceptions (as related to each successive 'level' of the code) derive from the previous level. For instance consider the example of a service throwing an exception derived from ServiceException. If the service does not explicitly handle the exception it will be passed back to the Router (as the calling component), and because ServiceException is derived from RoutingException we can expect the router to be able to handle the exception gracefully.

Figure 6-2 iPlanet Trustbase Transaction Manager Exception Hierarchy



It is imperative that developers of extension to the system maintain this hierarchical approach to exception inheritance in order that the handling code built into the current iPlanet Trustbase Transaction Manager system will function correctly.

The hierarchical nature of the iPlanet Trustbase Transaction Manager exceptions allows the exception handling to be done differently depending upon where the exception occurs. The components and the associated actions performed are shown in Table 6-5:

Table 6-5 Exceptions

Component	Base Exception	Actions
Servlet	TbaseRuntimeException	Catches TbaseRuntimeExceptions and generates the appropriate HTTP error
Protocol Analyser	ProtocolAnalyserException	may throw ParseException which will result in an HTTP error in the servlet
Message Analyser	MessageAnalyserException	Generated if a RemoteException has occurred, will result in an HTTP error in the servlet.
Message Readers	MessageReaderException	Is caught in the MessageAnalyser if it contains an embedded Message, then that Message is processed, otherwise the exception is propagated through to the servlet.
Message Writers	MessageWriterException	Is propagated back through the MessageAnalyser to the servlet that returns an HTTP error.
Router	RoutingException	Is propagated back through the MessageAnalyser to the servlet that returns an HTTP error
Services	ServiceException	May or may not contain a message, and is propagated back through the router to the MessageAnalyser where it is handled in the same fashion as the MessageReaderException

Raw Logging

The iPlanet Trustbase Transaction Manager provides a facility for logging the raw data comprising an applications inbound and outbound messages. Separate logs are maintained for each application that requires raw logging, and these logs are currently held in database tables. Logs may [optionally] be digitally signed, to make tampering with the logged data after-the-fact difficult.

The iPlanet Trustbase Transaction Manager is configured, by default, with a single raw log for Identrus data. In combination with the Identrus log, this meets the requirements for logging in an Identrus Transaction Coordinator.

An application which requires raw logging facilities must use it’s own raw log, which is created with the AddLoggerWizard. The AddLoggerWizard creates the appropriate database tables and associates them in iPlanetTrustbase Transaction Manager configuration with a name : the raw log store name. An application can thereafter store data in the new raw log by referring to it by that name, as this simple example shows:

```
/*

* MY_MIME_TYPE - is the mime type of your message. This is

* present only for historical purposes.

* rawxml - the XML String comprising raw message data

* storename - is the name of raw log store you wish to log to.

*/

MessageLogger mlg = SingletonMessageLoggerManager.getMessageLogger (
MY_MIME_TYPE );

MessageElements me = new MessageElements ( );

   me.put( MessageElements.RAW_DATA, rawxml );

/** the returned Vector of  Strings will only contain a single
String, which

* is the recordid field of the logged record, and can be used as a
foreign key for

* logging further application specific data which refers to the raw
data */

Vector ids = mlg.log ( storeName , rawxml );
```

The result of the log operation is a String identifier [contained in a Vector for historical reasons], which is the value of the recordid field in the database record created. This identifier can be used as a foreign key in an application specific log table, permitting the application specific log to refer to the raw log.

The raw log inserts a row into a relational database table for each log operation. The structure of the database table is described here. All raw log tables have the same structure, although each raw log uses a different table, whose name is determined when the raw log is created with the AddLoggerWizard.

The raw logging facility records raw incoming and outbound message data.

Table 6-6 Raw log

raw_data table	
Sessionid	The id of the raw log session that wrote this record
Logconnectionid	The id of the connection within the session
Recordid	The id of the record within the connection
recordmarker	A unique monotonically increasing identifier
timestamp	An integer which represents the UNIX time at which the record was logged.
rawdata	The Identrus Message XML, without the CertBundle fields. The certificates from the bundle are logged separately in the “cert_data_table”
digestofrecord	A SHA-1 digest of this record.
signeddigestofcalculation	An RSA signature of this record and data from the previous record.

Chapter 7

Identrus logging

The Identrus Transaction Coordinator specifications identify two specific logging actions, these being:

- Logging of all messages sent and received by the Transaction Coordinator (Raw logging)
- Generation of data for billing purposes

Overview

The iPlanet Trustbase Transaction Manager fulfils both of these requirements as a default action of processing an Identrus message. The data is stored within the RDBMS specified at installation time, and the tables are available for developers via standard JDBC to provide services that use this information.The following sections define the tables stored in the RDBMS and identify the relationships between each table. The iPlanet Trustbase Transaction Manager will utilise all of the tables described below for all Identrus messages; there should be no requirement for a developer to write to any of these tables.

Data definitions

Connection information

The SSL proxy and the SMTP mail listener both log data about the connections made through them. Table 7-1 provides the column definitions for the SSL Proxy:

Table 7-1 SSL Connection

ssl_connection table	
ConnectionId	Unique connection identifier
ClientCertIssuerDN	The connecting clients certificate Issuer DN
ClientCertSerialNumber	The connecting clients certificate serial number
CipherSuite	The cipher suite used for the SSL session
ConnectTime	The time at which the connection was made, this is an ORACLE DateTime field
TimeStampType	The type of timestamp
ConnectIPAddr	The connecting client's IP address
ConnectionFailed	Integer value indicating if the connection failed - a value of 1 indicates a failure.
ConnectionFailedReason	If a failure occurred, what was the SSL error code

The tables below provide the column definitions for the SMTP / SMIME connection logs: Data in Table 7-2 is extracted from the SMIME v2 signature body part on the message.

Table 7-2 SMIME Transport

smime_transport table	
ConnectionId	Provides a link back to the “smtp_message table”
peer_issuer_dn	The issuer_dn of the certificate that was used to verify the message

smime_transport table	
peer_cert_serial_number	The serial number of the certificate used to verify the message.
message_protection	The type of protection used to secure the message
time_stamp_type	The type of timestamp LOCAL or NETWORK
time_stamp	The time at which the entry was made

Table 7-3 SMTP Connection

smtp_connection table	
stream_id	Provides a link back to the “smtp_message table”
peer_ip_addr	The ip address of the submitting SMTP agent
timestamptype	The type of timestamp LOCAL or NETWORK
timestamp	The time at which the entry was made

Table 7-4 SMTP Message

smtp_message table	
stream_id	A unique id for the smime_transport
connection_id	A unique id for the smtp connection
recipients	The recipients of this message
sender	The sender of this message
timestamptype	The type of timestamp LOCAL or NETWORK
message_valid	Is the message valid? 1 indicates it is valid
message_invalid_reason	The reason for the invalidity of the message
timestamp	The date and time at which the entry was made

The ssl_connection and smtp_message tables both have connection_id fields that are passed to the iPlanet Trustbase Transaction Manager running in the application server. This connection_id is stored within the Identrus Log table allowing queries that link the originator information with the actual requests made.

Table 7-5 OCSP

ocsp_data table	
ocspid	A unique identifier for the record
type	OCSPREQUEST or OCSPRESPONSE
message	A text summary of the contents of the request or response
machine	The URL to which the request was submitted to or the response was received from
timestamp	The date and time that the entry was made
data	Base64 encoding of the request or response

Identrus log tables

The default presentation handlers for Identrus messages record the following data for each message that is sent or received.

In order to reduce the volume of data logged with each Identrus message the certificates contained with the message header are stripped out and stored in a certificate table. If the iPlanet Trustbase Transaction Manager has already logged a particular certificate in the table it will not be logged again. The information stored within the table is:

Table 7-6 Certdata

cert_data table	
IssuerDN	The issuer distinguished name of the certificate, RFC 2253 format string.
SerialNumber	The serial number of the certificate
CertData	The Base64 certificate data.
SubjectDN	The subject distinguished name from the certifcate, in RFC2253 format

This data is designed to be tamper evident, and services should under no circumstances modify data within the Identrus Log or Tamper tables. The tamper checking is achieved by producing a continuous hash that is stored with each record, and the current hash is stored within a signed record within a separate tamper table. The Tamper table fields are not described here, see the Installation and Configuration Guide for information on how to check the tamper status of records in the raw log.

The Identrus data table records identrus specific message data, which can be related to the raw log records in the raw_data table, using the rawrecordid foreign key [see the chapter Logging: Error, Audit and Raw for a description of raw logging]

identrus_data	
rawrecordid	the id of the associated raw log record
msggrpid	the Identrus MsgGrpId from the NIB of the message
doctype	the DOCTYPE of the message. e.g.CSCRequest, PingRequest etc..
connectionid	The connection id to link this record to the SSL or SMIME connection logs
protocoltype	The protocol over which the messge arrived e.g. HTTP or SMTP
input	Was this message inbound to the iPlanet Trustbase Transaction Manager or outbound ? A value of 1 indicates it was incoming

Billing records

Billing records are a sub-set of the information within the raw message log that provides sufficient information to determine who made each transaction. These tables are designed for used by third party tools that generate the actual Bill for the customer. The definitions for the bill table columns are as follows:

Table 7-7 Bill data

bill_data table	
RawRecordId	This will be the RawRecordId of the associated raw log table record.
SubjectDN	This will be the originator distinguished name extracted from the mandatory Identrus level 1 message signature. This will determine who should be billed.
IssuerDN	This will be the issuer distinguished name extracted from the mandatory Identrus level 1 message signature. This is to enable the identification of the exact key used to sign this message - in conjunction with the serial number field below.
SerialNumber	This will be the originator certificate serial number that may be used to identify the exact key used to sign the message - in conjunction with the issuer distinguished name.

Chapter 8

Building Identrus solutions

The iPlanet Trustbase Transaction Manager platform and associated development tools provide a means of developing Identrus applications starting from the base DTD through to installing the service for use in a run-time environment.

The following sections walk through the generation of an Identrus application called Ping. This application is designed to highlight the following concepts:

- Overall development process
- Use of iPlanet Trustbase Transaction Manager developer tools
- The class generation tool JAXHIT
- Using configuration objects
- Using the log manager

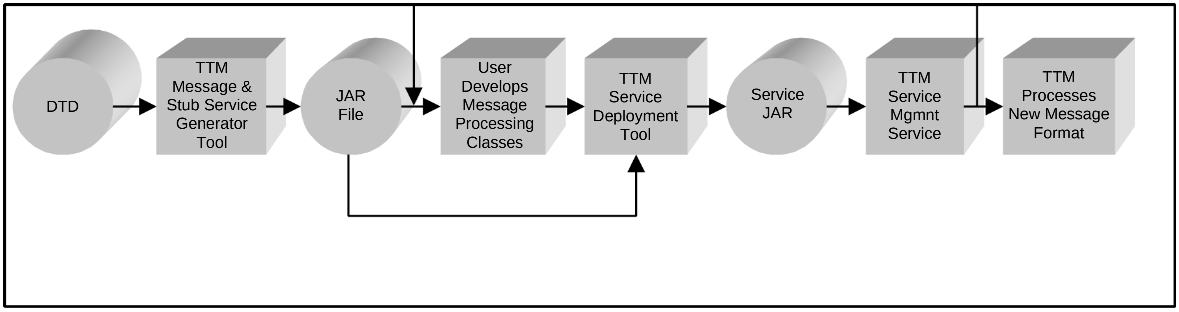
Methodology

Development process

The development of an application requires a base input of an Identrus compliant DTD. This is passed through a class generator that produces Java Classes for all of the elements in the DTD along with a set of stub rules and a service definition within a JAR file. The developer is then at liberty to extend this base data by implementing the appropriate business logic in the form of a set of Java classes called by the service stub that perform the appropriate actions.

The set of generated files and associated developer files are then packaged into a single JAR and deployed into an iPlanet Trustbase Transaction Manager directory structure. The iPlanet Trustbase Transaction Manager (when re-started) will locate a package descriptor within the JAR file and offer this new service for activation within the running environment. The overall process is outlined in Figure 8-1.

Figure 8-1 Development process


DTD
TTM Message & Stub Service Generator Tool
JAR File
User Develops Message Processing Classes
TTM Service Deployment Tool
Service JAR
TTM Service Mgmnt Service
TTM Processes New Message Format

Class generation

The iPlanet Trustbase Transaction Manager class generation tool is designed to produce a set of classes that may be used at two points in the message processing pipeline. We now provide a brief overview. (more details can be found in "The JAXHIT Class Generation Tool," on page 1 of the appendix at the end of this document). These are:

Presentation **-** The iPlanet Trustbase Transaction Manager XML parser uses the classes to validate incoming XML against the DTD.

Services **-** Business logic that accesses elements of the message

In order to access the class generation tool, a JDK 1.2.x or greater must be installed; a JRE does not contain the correct compilers to build an iPlanet Trustbase Transaction Manager service. Also the classpath must be correctly set, executing the following on your iPlanet Trustbase Transaction Manager installation can do this:

```
cd ……./Trustbase/TTM/Scripts
. ./jaxhit
```

This sets the CLASSPATH environment variable to the correct value for the iPlanet Trustbase Transaction Manager tools.

The class generation tool is available as a command line option and is invoked using the following:

```
jaxhit [-options] [<public Id> <DTD file>]
```

Where options include:

-help	displays help
-config <file>	Specify the name of the configuration file. (See “Message Handler Example,” on page 117 for more options on this
-cp <classpath>	Specify the classpath to use for compilation
-y	Do not confirm before overwriting existing files
-stub	Generate a stub service
-quiet	quiet output
-v	verbose output
-debug	debug output

And PublicID is the name of the service stub that will be generated. The Public Id is a mandatory requirement for all Identrus DTD's, and is in the standard form shown below:

```
-//IDENTRUS//service_name//EN
```

Where service name represents the Identrus service declared by the DTD. The Public ID is case sensitive and should be carefully entered as it forms part of the package name for the generated classes.

The tool will only generate classes for DTD elements that do not already have classes available on the classpath. Therefore, the core Identrus classes are not regenerated with each new Identrus service because they are already available in Trustbase jars.

A '-root' entry *must* be made for each of the elements in the DTD that can be the root XML element in a message. This is necessary to allow the automatic processing of the Identrus Message by the default presentation layer in the iPlanet Trustbase Transaction Manager.

A '-stub' option must be used if the tool is required to generate a stub iPlanet Trustbase Transaction Manager service for the new message types. The Java source file for this stub class is placed in the current directory when the tool completes. Do NOT re-package the stub class once generation has completed - the fully qualified name of the stub class is recorded in the tbasesvc.properties file in the generated jar file.

The Class generator will attempt to recursively load the referenced DTD(s). The references may be:

- Unqualified - The DTD must be located in the directory in which the class generator is run in
- Local - The file:/// qualifier is used
- Fully qualified - The HTTP://www qualifier is used

An invocation of the JAXHIT tool will produce the following output:

- A .zip containing
    - A set of generated classes, one for each entity in the DTD
    - A set of DTD's that represent the input actually used by the XML parser for generating the classes
    - A tbasesvc.properties file used by iPlanet Trustbase Transaction Manager to register the service
- A service stub in the current directory - only if the '-stub' option is used

Each of the generate classes implements the TbaseElement interface. This interface provides two specific roles:

- Parser - Allows the iPlanet Trustbase Transaction Manager parsing mechanisms to validate incoming XML messages
- Service - Provides a standard means of getting data from incoming messages and constructing valid responses without explicitly producing the XML.

There is a sub-class of TbaseElement called TbaseIdentifiedElement that represents elements that have an attribute named 'id' with a type of 'ID'. This allows a type-safe method of checking ID compliance when generating or validating XML DSIG signatures.

The service role of the TbaseElement provides the following function:

- Constructing the XML document hierarchy
- Reading and writing the element attributes by name
- Validating the object representation of the XML document
- Generating the String representation of the XML document described by the object hierarchy

All elements that are identified by a '-root' option at class generation time, will end up extending the base class com.iplanet.trustbase.identrus.message.IdentrusMessage. This enforces that the message has a valid structure (as defined by Identrus Core Network Messaging Definition) and the framework can carry out that mandatory processing of the message. An example of using this interface is shown in the example section at the end of this document.

NOTE The API classes for com.iplanet.trustbase.identrus.message.IdentrusMessage, com.iplanet.trustbase.xml.message.TbaseElement and com.iplanet.trustbase.xml.message.TbaseIdentifiedElement provide more information on their function.

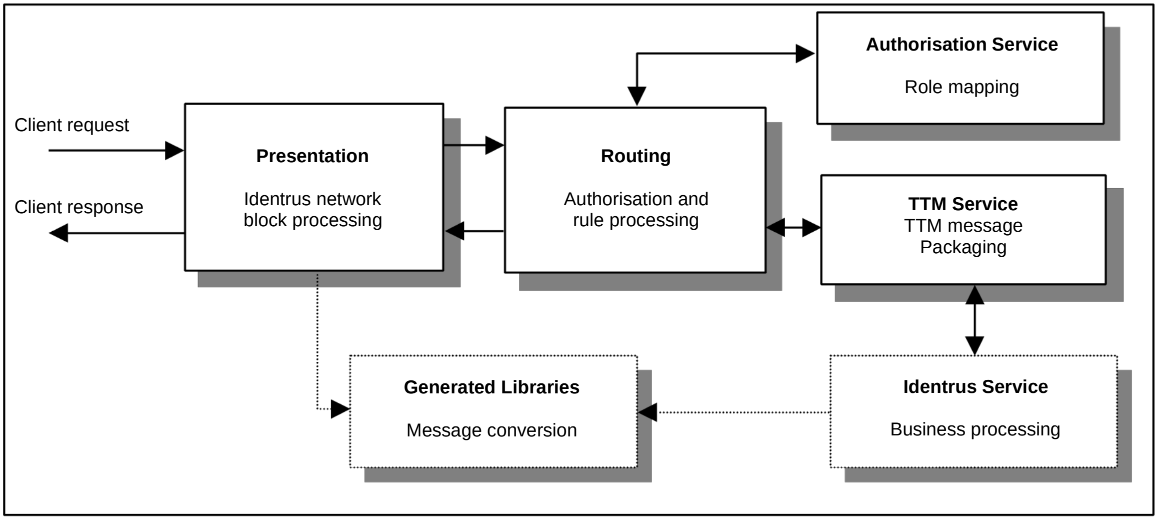
Service development

Development of a service is the act of completing the service stub into a functional business component.

Every service must be stateless so that it may be replicated by the platform. This means that a service must be capable of processing messages from different transactions and different users without holding information about a previous Session State Transaction. If a service is required to hold information about a session or transaction state, this must be persisted in a database, as the subsequent messages in a transaction are not guaranteed to be passed to the same host machine within a cluster.

All Identrus network message processing is performed by the iPlanet Trustbase Transaction Manager platform prior to and after the invocation of the ProcessIdentrusMessage method in the service stub class generated by the Class Generation tool. Figure 8-2 shows the message processing path with a Developer written Identrus Service using the generated message libraries and proxied by the iPlanet Trustbase Transaction Manager message process stub.

Figure 8-2 Message path processing



This means that for most Identrus services, the developer only needs to modify the generated stub class to implement the business processing of the incoming message and generate the core content of the outbound message.

The business processing does not need to do any of the mandatory message processing such as mandatory signature verification or generation on the message. These core processing tasks are completed by the iPlanet Trustbase Transaction Manager presentation layers.

To develop the service, include the generated jar file on the classpath during development, along with the core iPlanet Trustbase Transaction Manager libraries.

Service Building

Having successfully generated the messaging classes and developed the message processing logic extensions to the stub service, the next task is to build a iPlanet Trustbase Transaction Manager service jar.

Building the iPlanet Trustbase Transaction Manager service jar is a simple matter of the following steps:

- Using the standard JDK 'jar' tool, unpack the generated jar file into a directory.

```
mkdir DumpDirectory
cd DumpDirectory
jar -xvf ../jarfile.jar
cd ..
```

- Beneath the 'DumpDirectory' copy in all of the classes which are required for the processing of the message. Ensure that the correct package structure is maintained for all these classes. n.b. Remember that the service stub class *must not* have been re-packaged since the class generation created it.
- Using the standard JDK 'jar' tool, repack the archive

```
jar -cvf jarfile.jar -C DumpDirectory DumpDirectory/*
```

- The jar file is now ready for deployment.

Service Deployment

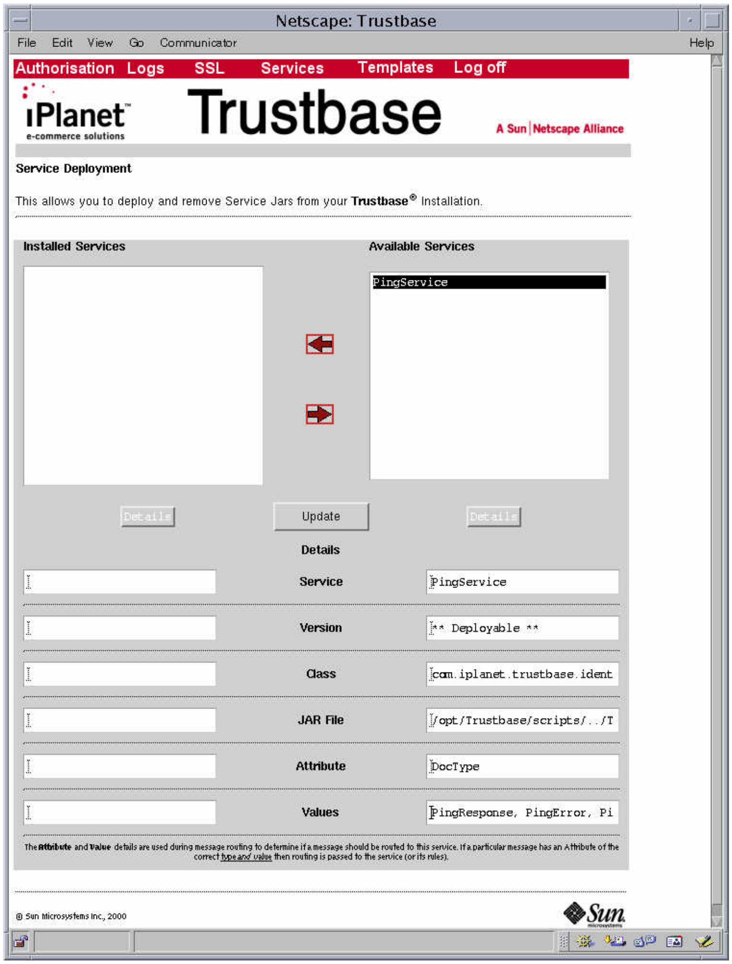
Once the service jar file has been built it can be deployed into iPlanet Trustbase Transaction Manager. This is done in three stages:

- Copy the jar file into the directory: …../Trustbase/TTM/current/deploy
- Use the administrator console to deploy the service into the running iPlanet Trustbase Transaction Manager
- Re-start iPlanet Trustbase Transaction Manager to activate the service

To deploy the service, logon to iPlanet Trustbase Transaction Manager as 'administrator' and select the 'Deployment' option from the services menu. If you have copied the service jar into the 'deploy' directory then the new service should appear on the left hand side of the deployment screen.

- To deploy the service select ⬅ followed by Update

Figure 8-3 Deploying PingService within iPlanet Trustbase Transaction Manager


Netscape: Trustbase
File Edit View Go Communicator Help
Authorisation Logs SSL Services Templates Log off
iPlanet™ e-commerce solutions
Trustbase
A Sun | Netscape Alliance
Installed Services
Available Services
Details
Update
Details
Sun microsystems

Service Deployment

This allows you to deploy and remove Service Jars from your **Trustbase**® Installation.

Available Services: PingService

Details

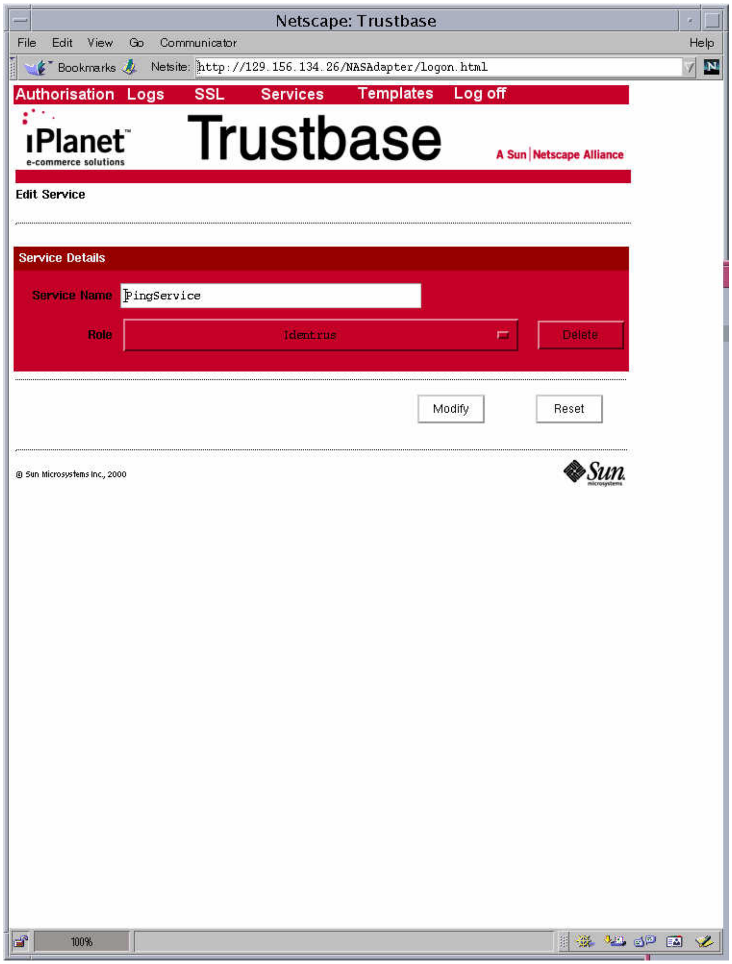
	Service	PingService
	Version	** Deployable **
	Class	com.iplanet.trustbase.ident
	JAR File	/opt/Trustbase/scripts/../T
	Attribute	DocType
	Values	PingResponse, PingError, Pi

The **Attribute** and **Value** details are used during message routing to determine if a message should be routed to this service. If a particular message has an Attribute of the correct type *and* value then routing is passed to the service (or its rules).

© Sun Microsystems Inc., 2000

Services may also require authorisation. Select <Authorisation><Add Service>. The service itself will need to be assigned a role, as illustrated below:

Figure 8-4 Assigning a role to a service



Netscape: Trustbase
File Edit View Go Communicator Help
Bookmarks Netsite: http://129.156.134.26/NASAdapter/logon.html
Authorisation Logs SSL Services Templates Log off
iPlanet e-commerce solutions
Trustbase
A Sun | Netscape Alliance
Modify
Reset
Delete
© Sun Microsystems Inc., 2000
Sun microsystems
100%

Edit Service

Service Details

Service Name: PingService

Role: Identrus

NOTE If no role is available you may have to define a role. Consult the Configuration and Installation Guide for more details about this.

Chapter 9

Ping Example

The following sections use the Ping DTD as defined in the Identrus messaging specification to walk through the steps in the development cycle as described in the previous chapter. Developing a service prior to deploying it involves a number of steps.

1. Create your DTD definitions that specify the syntactic structure of the messages you wish to send round the system.
2. Use JAXHIT to generate your java classes from your DTD definitions.
3. Write the Java code for the service using the Identrus API that assists the Identrus processing and validating of messages, certificates, keys and digital signatures.
4. Use the JDK JAR tool to build the final iPlanet Trustbase Transaction Manager service jar file.
5. Deploy the service within iPlanet Trustbase Transaction Manager by selecting the relevant configuration options as described below in this guide.
6. Finally, once it has been deployed within iPlanet Trustbase Transaction Manager itself you can run your service.

Create DTD Definitions

- This example uses the 'Ping' DTD defined in the Identrus Messaging specification and included below for completeness. Thus, create your DTD definitions which can be found in ping.dtd. (See IT-TCMPD, the Identrus TC Messaging Specification for the structure and definition of ping.dtd)

```
<!--PUBLIC ID for this DTD is: "-//IDENTRUS//PING DTD//EN"-->

<!ENTITY % CoreNetwork.dtd PUBLIC "-//IDENTRUS//CORE NETWORK
INFRASTRUCTURE DTD//EN" "corenetworkinfrastructure.dtd">
%CoreNetwork.dtd;

<!ELEMENT PingRequest (NIB, Signature, CertBundle, PingData)>
<!ATTLIST PingRequest
    id ID #REQUIRED
>
<!ELEMENT PingResponse (NIB, Signature, CertBundle, PingData)>
<!ATTLIST PingResponse
    id ID #REQUIRED
>
<!ELEMENT PingError (NIB, Signature, CertBundle, ErrorInfo)>
<!ATTLIST PingError
    id ID #REQUIRED
>
<!ELEMENT PingData (#PCDATA)>
<!ATTLIST PingData
    id ID #REQUIRED
>

<!ELEMENT ErrorInfo (VendorData*)>
<!ATTLIST ErrorInfo
    id ID #REQUIRED
    errorCode NMTOKEN #REQUIRED
>
<!ELEMENT VendorData (#PCDATA)>
<!ATTLIST VendorData
    id ID #REQUIRED
    dataType CDATA #REQUIRED
>
```

- These link to other dtd files that are supplied with iPlanet Trustbase Transaction Manager and are listed below:

```
XMLDSIG.dtd
corenetworkinfrastucture.dtd
Foundation.dtd
```

NOTE All dtd's can be found in <install-dir>/Trustbase/samples/ping/resources

API

Before writing your Java code you'll need to know some of the core API that is supplied with iPlanet Trustbase Transaction Manager for handling Identrus Messages:

```
com.iplanet.trustbase.identrus
```

This package allows access to Identrus message Specification

```
com.iplanet.trustbase.identrus.message
```

This package provides the necessary routines to allow Identrus message processing

```
com.iplanet.trustbase.identrus.security
```

This package processes certificates and keys

```
com.iplanet.trustbase.util.tree
```

Searches tree

```
com.iplanet.trustbase.xml.dsig
```

Generates and validates XML Digital signatures, this package is not required if the only signatures on the messages are the mandatory level 1 signatures. These mandatory signatures are checked / generated automatically by iPlanet Trustbase Transaction Manager.

```
uk.co.jcp.tbase.config
```

This package interfaces with configuration objects

PingService Source Code

PingService.java - the generated service stub is an implementation of the abstract IdentrusService in which the ProcessIdentrusMessage method parameter is the message received by the TTM platform.

By the time the message arrives at this ProcessIdentrusMessage method, it has already passed mandatory signature validation checks and been raw logged.

The method must create a response message to the incoming request. Using the supplied Identrus API's this is simple.

```
package com.iplanet.trustbase.sample;
import com.iplanet.trustbase.identrus.message.IdentrusMessage;
import uk.co.jcp.tbase.service.ServiceException;
import com.iplanet.trustbase.generated.IDENTRUS.PING_DTD.*;
import
com.iplanet.trustbase.generated.IDENTRUS.CORE_NETWORK_INFRASTRUCTURE_DTD
.*;
import com.iplanet.trustbase.identrus.IdentrusService;
import com.iplanet.trustbase.identrus.message.*;
import uk.co.jcp.tbaseimpl.log.error.*;
import uk.co.jcp.tbase.service.*;
/*** Stub Identrus service implementation. */
public class PingService extends IdentrusService
{
    public IdentrusMessage processIdentrusMessage( IdentrusMessage
message )
    {
        if (message instanceof PingRequest)
        {
            // Handle PingRequest
            PingResponse pr = new PingResponse();
            PingData pd = new PingData();
            pd.setPCDATA( "The ping was successful" );
            pr.setPingData( pd );
            try
            {
            NIBAccessor niba = NIBAccessor.getInstance(
message.getNetworkInfoBlk(), "2" );
            pr.setNetworkInfoBlk( niba );
            }
            catch (NIBAccessorException nie)
            {
            ErrorLog.log( new ErrorObject( "IDT0052", nie ));
            }
            return pr;
        }
        if (message instanceof PingResponse)
        {
            // Handle PingResponse
        }
        if (message instanceof PingError)
        {
            // Handle PingError
        }
            return message;
    }
    }
```

Having written this simple service it may be compiled ready to be built into a Service JAR.

Creating the Identrus Service JAR

The following instructions assume that you are using a Bourne Shell or derivative, and that you have a JDK (at least v1.2.2) installed.

1. From the Trustbase installation directory change into the Scripts sub-directory:

```
cd Scripts
```

2. Set the shell CLASSPATH environment variable to include all of the iTTM libraries

```
. ./setcp
```

3. Change directory into the Ping Sample resource directory

```
cd ../samples/ping/resources
```

4. Generate the classes from the ping.dtd file, using the JAXHIT tool

```
../../../Scripts/jaxhit -config IdentrusPingMessaging.xml -y
```

5. Change directory to the parent directory and compile the generated classes

```
cd ..
```

```
javac -d build
src/com/iplanet/trustbase/generated/IDENTRUS/PING_DTD/*.java
```

6. Compile the ping service classes

```
javac -d build -classpath $CLASSPATH:build
src/com/iplanet/trustbase/identrus/ping/*.java
```

7. Compile the ping client classes

```
javac -d build -classpath $CLASSPATH:build
src/com/iplanet/trustbase/identrus/tools/*.java
```

8. Build all of the classes and the service descriptor file (build/config/tbasesvc.properties) into a JAR file which can be deployed into iTTM.

```
jar -cvf pingsample.jar -C build com -C build config
```

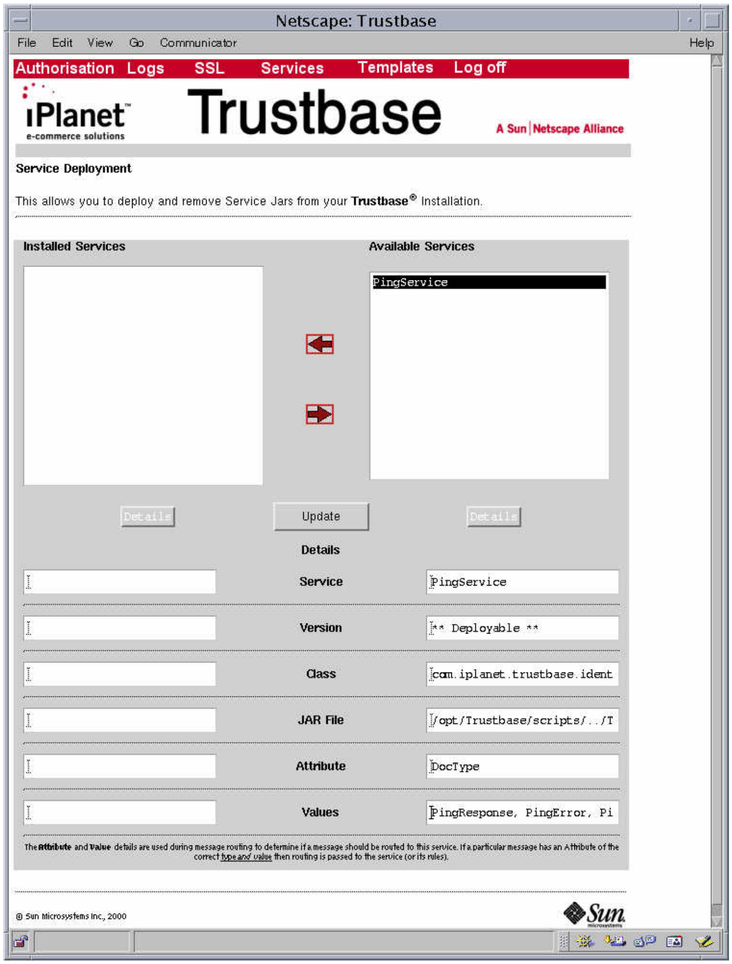
9. Make the new service JAR file available for deployment in iTTM

```
cp pingsample.jar ../../current/deploy
```

Deploying pingsample.jar within iPlanet Trustbase Transaction Manager

- Logon to iPlanet Trustbase Transaction Manager as Administrator
- Deploy PingService within iPlanet Trustbase Transaction Manager. Select <Services> followed by <Deployment>.
- Since Ping.jar has been placed in the deploy directory automatically by the builder, iPlanet Trustbase Transaction Manager picks up all the relevant information and places this in the screen headed "Available services" as illustrated below.
- To deploy the service select ⬅ followed by Update

Figure 9-1 Deploying PingService within iPlanet Trustbase Transaction Manager


Netscape: Trustbase
File Edit View Go Communicator Help
Authorisation Logs SSL Services Templates Log off
iPlanet e-commerce solutions
Trustbase
A Sun | Netscape Alliance
Details
Update
Details

Service Deployment

This allows you to deploy and remove Service Jars from your **Trustbase**® Installation.

Installed Services

Available Services

PingService

Details

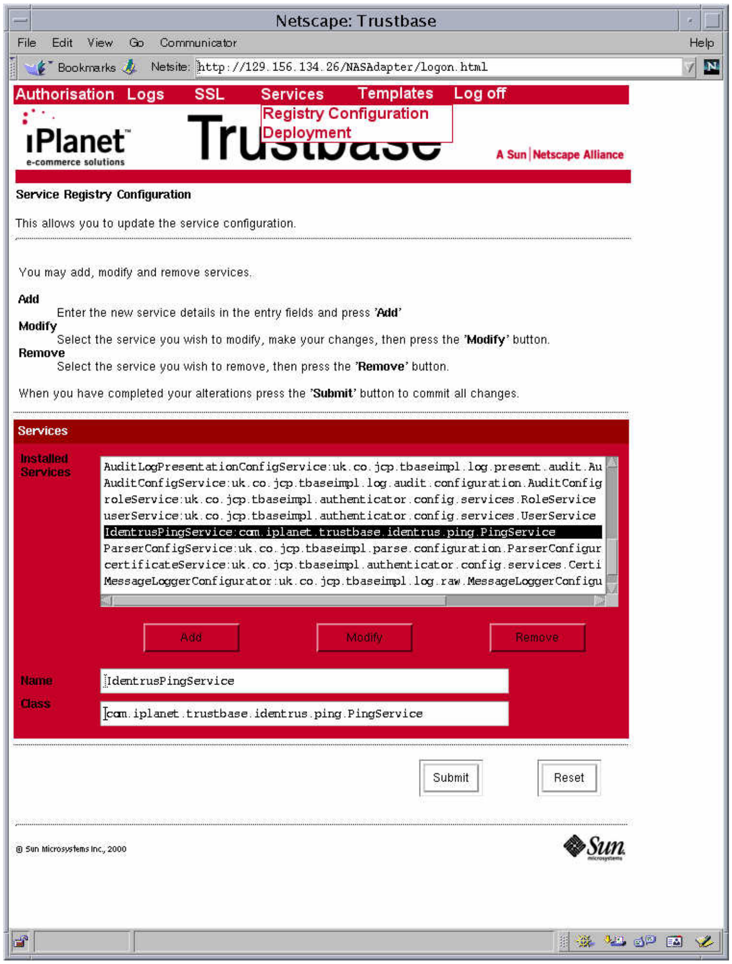
	Service	PingService
	Version	** Deployable **
	Class	com.iplanet.trustbase.ident
	JAR File	/opt/Trustbase/scripts/../T
	Attribute	DocType
	Values	PingResponse, PingError, Pi

The **Attribute** and **Value** details are used during message routing to determine if a message should be routed to this service. If a particular message has an Attribute of the correct type *and* value then routing is passed to the service (or its rules).

© Sun Microsystems Inc., 2000

- Services require registration. This is done automatically when the jar file is built from the builder tool. PingService can in fact be seen from the Configuration Console by selecting <Services> <Registry Configuration> as illustrated below:

Figure 9-2 Service Registry Configuration



Netscape: Trustbase
File Edit View Go Communicator Help
Bookmarks Netsite: http://129.156.134.26/NASAdapter/logon.html
Authorisation Logs SSL Services Templates Log off
Registry Configuration
Deployment
iPlanet™ e-commerce solutions
Trustbase
A Sun | Netscape Alliance

Service Registry Configuration

This allows you to update the service configuration.

You may add, modify and remove services.

Add
Enter the new service details in the entry fields and press '**Add**'

Modify
Select the service you wish to modify, make your changes, then press the '**Modify**' button.

Remove
Select the service you wish to remove, then press the '**Remove**' button.

When you have completed your alterations press the '**Submit**' button to commit all changes.

Services

Installed Services

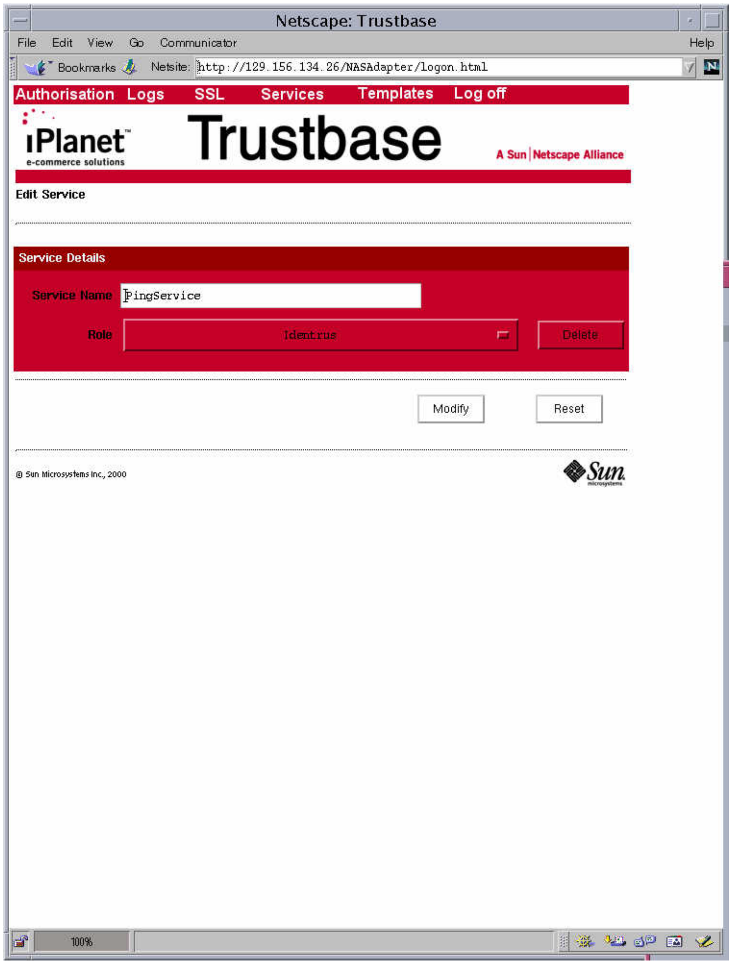
```
AuditLogPresentationConfigService:uk.co.jcp.tbaseimpl.log.present.audit.Au
AuditConfigService:uk.co.jcp.tbaseimpl.log.audit.configuration.AuditConfig
roleService:uk.co.jcp.tbaseimpl.authenticator.config.services.RoleService
userService:uk.co.jcp.tbaseimpl.authenticator.config.services.UserService
IdentrusPingService:com.iplanet.trustbase.identrus.ping.PingService
ParserConfigService:uk.co.jcp.tbaseimpl.parse.configuration.ParserConfigur
certificateService:uk.co.jcp.tbaseimpl.authenticator.config.services.Certi
MessageLoggerConfigurator:uk.co.jcp.tbaseimpl.log.raw.MessageLoggerConfigu
```

Name IdentrusPingService

Class com.iplanet.trustbase.identrus.ping.PingService

- Services may also require authorisation. Select <Authorisation><Add Service>. The service itself will need to be assigned a role, as illustrated below:

Figure 9-3 Assigning a role to a service



Netscape: Trustbase
File Edit View Go Communicator Help
Bookmarks Netsite: http://129.156.134.26/NASAdapter/logon.html
Authorisation Logs SSL Services Templates Log off
iPlanet™ e-commerce solutions
Trustbase
A Sun|Netscape Alliance
Edit Service
Service Details
Service Name
Role
Delete
Modify
Reset
© Sun Microsystems Inc., 2000
Sun microsystems
100%

Service Name: PingService

Role: Identrus

NOTE If no role is available you may have to define a role. Consult the Configuration and Installation Guide for more details about this.

Chapter 10

Message Handler Example

Introduction

To explain the scope and function of the Standard Message Path features of iTTM, the following sections define all of the requirements and restrictions placed upon users of the Standard Message Path functionality. A number of common questions are answered below.

What is the iTTM Standard Message Path ?

The Standard Message Path is the name given to a class of XML application messaging protocols which conform to the restrictions laid out below. The iTTM product implements a default set of iTTM processing pipeline components which provide a basic level of service to all compliant XML messages. A tool for generating XML message parsing classes and an outline message processing service for deployment in the iTTM platform.

What needs to be in place before the Standard Message Path may be used ?

The most important components which must be in place before starting to produce a Standard Message Path service are the XML messaging protocol and its associated DTD. It is important to have both the message structure definitions (i.e. DTD) and the messaging protocol (i.e. what actions and response messages are provoked by a given request message) defined before trying to create a Standard Message Path service to process the messaging protocol.

What restrictions does the Standard Message Path place on messaging protocols ?

The messaging is defined by DTD’s, the Standard Message Path messaging does not prescribe a specific DTD or XML Schema for messages, instead it makes a number of features mandatory for third parties when they are defining their own message structures. Taking this approach allows the consumers of Standard Message Path maximum flexibility in the design of their messaging protocols. The full restrictions are defined below, but can be summarised as:

- The messages must contain a transaction identifier’ which relates all requests and responses in a messaging exchange.
- There must exist an XML DSIG signature as an immediate child of the document rood element. This signature must sign at least one of the messaging elements.
- The use of the XML ‘ID ‘ attribute type is mandatory for all elements which may need to be inculded in the creation of a signature. The values of these ID attributes take a prescribed format.
- All DTD’s must be assigned a public identifier, this identifier is unique for each version of the messaging. i.e. revision 2 of a given DTD must have a different identifier to revision 1. This allows multiple versions of messaging to be simultaneously supported.
- The mime type of all Standard Message Path compliant messages must adhere to the structure defined below.

What basic functions does the default implementation of the Standard Message Path provide ?

The default iTTM message pipeline components that implement the default Standard Message Path processing behaviour provide the following services to all compliant messages:

XML Parsing and structural check

XML message validation against the DTD which defines them, ensuring grammatical correctness

Logging into a database of the ‘Raw’ XML bytes

Message signature checking of the mandatory message signature

Use of the certificate which validates the mandatory signature to establish an authentication level which is then used by the iTTM authorisation component to gate access to the processing service to legitimate requestors only.

The idea of the default components are that they provide enough basic service and message processing semantics that the user need only concentrate on the implementation of the message ‘payload’ processing, leaving the administrative processing to iTTM Standard Message Path components.

- Which tools are required to create a Standard Message Path service ?

There are a minimum number of tools required to create a Standard Message Path service, these a listed below:

- ❍ JDK v1.2.x or greater, this contains a Java compiler and the JAR tool for creating the final deliverable.
- ❍ JAXHIT class generation tool, which is provided by the iTTM installation
- ❍ An editor for writing Java source code

- Can the default implementation of the Standard Message Path be changed ?

The default implementation is designed to produce useful basic set of functions, but sometimes these will need to be augmented for specific application needs. Each of the iTTM Standard Message Path components has been designed to allow for sub-classing which allows specific areas of function to be added, augmented or removed. This includes the basic error handling framework provided by each of the components.

Development Life Cycle

This section defines the development lifecycle of Standard Message Path service, this lifecycle utilises tools that allow basic services to be deployed and tested very quickly. The basic lifecycle is shown below and each stage is further detailed in the following sections.

Design Messaging ◊ Generate Classes and Service descriptor with JAXHIT ◊ Write service Java code ◊ Compile ◊ Make JAR file ◊ Deploy into iTTM ◊ Test service

Design Messaging Structures and Protocols

The most important input to the lifecycle is the design of the messaging structures and protocols. The messaging protocol defines which requests will result in which actions and responses etc. The messaging protocol will be implemented by the 'service' Java code written by the developer. The message structures are defined in DTD's which are used as input to the next stage of the lifecycle. This section concentrates on the design of the messaging structures, the protocols are not restrained by the Standard Message Path.

The iTTM Standard Message Path does not prescribe a specific DTD or XML Schema for messages, instead it makes a number of features mandatory for third parties when they are defining their own message structures. Taking this approach allows the consumers of iTTM the maximum flexibility in the design of their messaging protocols, including extending or wrapping existing XML messages in a Standard Message Path compatible form.

The following terms are used in the definition of the messaging rules:

- Level 1 element.
  This is used to refer to the top level, or root, XML tag defined for the message.
- Level *n* element.
  This is used to refer to an XML tag that is the child of a level *n-1* XML tag, where level 1 represents the top level element.

The following points are all mandatory for any XML message to be deemed iTTM Standard Message Path compliant:

- Somewhere in the message structure there must exist a message identifier whose life is as long as the transaction and is globally unique. This message identifier is used by iTTM to relate all the messages in a transaction. If this identifier is not present then the Standard Message Path components will allocate and identifier whose life will be a single request/response message pair. i.e. multi-request transactions will not resume their messaging context.
- There must exist an XML DSIG <Signature> element as a level 2 element, this Signature element must comply with the following:
- All of the certificates required to verify the signature must be included as children of the signature in the standard DSIG manner. This includes the entire certificate chain from child to root.
- The signature must sign at least one element.

- The signature must only refer to elements within the current message. e.g. ‘detached’ external document signatures are not allowed.
- Must be defined by the DTD included with the iTTM distribution, it has the revision URL of http://www.w3.org/2000/09/xmldsig#. For the purposes of entity inclusion in the application messaging DTD’s its public identifier must be: http://www.w3.org/2000/09/xmldsig#. E.g. to include the XML DSIG DTD inside a messaging DTD add the following lines to the top of the messaging DTD and place the XML DSIG DTD supplied in the iTTM distribution in the same directory as the messaging DTD(s):

```
<!ENTITY % XMLDSIG.dtd PUBLIC "http://www.w3.org/2000/09/xmldsig#"
"./xmldsig.dtd">

%XMLDSIG.dtd;
```

- Whitespace processing for the purposes of signature generation/validation must be defined by canonicalisation algorithm used in the DSIG signature. Where this algorithm is not prescriptive enough (e.g. DOMHASH), in order to remove ambiguity created by whitespace characters contained between nodes, all non-significant whitespace should be removed prior to signature generation or validation.
- Where a messaging protocol contains base 64 encoded elements, the line breaking in the base 64 representation is not prescribed – but for the purposes of signature calculation all whitespace in the encoding is considered significant.
- The use of XML ID attributes is mandatory for any element which needs to be signed, the generation of these ID values is critical because the following must be true:
    - Each ID is unique within a single document
    - Once a block is signed, the ID may not be changed without invalidating the signature. This is significant if blocks are to be copied from one document to another along with their associated signature.
- In order to allow all iTTM messages to interoperate and to remove the burden of ID generation from the developer, the following scheme will be used for the generation of ID values when signing a message. This scheme is not mandatory, but any elements that do not have an ID value already assigned before an attempt to generate a signature is made will be completed using this

scheme. It would be advisable for all users of iTTM Standard Message Path messaging to adopt this scheme. Each ID value will be constructed from the 3 following elements, in the following order, separated by underscore ('_') characters.

1. Element Name
2. A 20 byte securely generated random number, represented as a hexadecimal string, where the most significant byte of the number is the first character, all letters in the number will be uppercase.
3. To guarantee uniqueness of the ID within the message, a further integer is added as the final component of the ID. The first occurrence of an ElementName will have a value of 1, all subsequent occurrences will be incremented by one. The integer is not mandatory if there is only one occurrence of an element name within a message. If the integer is not present then the final underscore separator is also omitted.

An example for the PingRequest element:

```
PingRequest_0102030405060708090A0B0C0D0E0F10111213_1 or
PingRequest_0102030405060708090A0B0C0D0E0F10111213
```

- All DTD's must be assigned a public identifier, this identifier must identify a particular version of a message definition. i.e. each version of each DTD must be assigned a different public identifier. This ensures that iTTM can easily distinguish different message versions and support multiple message versions simultaneously. The structure for a public identifier is defined to be:
  -//organisation//description//language
  The public identifier is case sensitive and is typically defined all in upper case. There must always be a system identifier defined for a DTD, this must be a valid URL from which the DTD can be fetched. As a further constraint, when a request/response message protocol is defined, the DTD that defines the top-level request element must be the same as the DTD that defines the appropriate response element. Entity inclusion may be used to 'import' other DTDs that define sub-ordinate elements.

- When sending a message to iTTM, it must have the following mime-type "application/ittm-*doctype*" where *doctype* is the XML DOCTYPE of the message in lowercase. That is the element name of the level 1 element in the message. e.g. application/ittm-pingrequest. This mime-type will ensure that the default protocol handler, message reader and message writer process the message along the iTTM Standard Message Path pipeline. If a different mime-type is required then customers will need to override each of the processing pipeline elements to implement their specific mime-type.

Generate Classes and Service descriptor with JAXHIT

The use of this tool is defined in “The JAXHIT Class Generation Tool,” on page 1 of the appendix at the end of this document, the tool can be started with a script called jaxhit which is found in the …../Trustbase/Scripts directory of the distribution. An example configuration file is described in the Appendix and another example is used in the sample Credit Check application supplied with the distributions and described in a later section of this document.

When using the jaxhit tool, all Standard Message Path messages must have a standard class as their <ElementBase> and must also define a <ServiceConfig>. The example JAXHIT configuration file for the sample CreditCheck application is described below:

```
<Config
  srcDir="src"
  basePackage="com.iplanet.trustbase.generated"
  build="false">
```

The first lines define the location of the generated classes and their base Java package root, the build flag is set to flase to stop the JAXHIT tool from trying to compile the generated classes.

```
  <DTDFile file="creditcheck.dtd" publicId="-//IPLANET//ITTM CREDIT
CHECK//EN"/>
```

This next line defines the base DTD files to have classes generated, if a DTD is entity included by a DTD named in a <DTDFile> element, then it will have its classes generated as the tool recurses through all included DTD files. The classpath that is defined when the JAXHIT tool is run is very important, before a class is generated, the classpath is checked to see if a class already exists. This feature allows for library DTD’s such as XMLDSIG to be generated just once and then used by many application DTD’s. This behaviour can be overridden by using the ‘force=”true”’ attribute on the <Config> element. See “The JAXHIT Class Generation Tool,” on page 1 of the appendix at the end of this document.

```
<DefaultElementBase/>
```

```
<ElementBase name="CreditCheckRequest">

    <ExtendsClass
name="com.iplanet.trustbase.standardpath.message.ITTMMessage"/>

  </ElementBase>

<ElementBase name="CreditCheckResponse">

    <ExtendsClass
name="com.iplanet.trustbase.standardpath.message.ITTMMessage"/>

  </ElementBase>

<ElementBase name="CreditCheckError">

    <ExtendsClass
name="com.iplanet.trustbase.standardpath.message.ITTMMessage"/>

  </ElementBase>
```

The lines above define the three allowable Document Root Elements, it tells JAXHIT that each of the Java classes should extend the existing "com.iplanet.trustbase.standardpath.message.ITTMMessage" class which is delivered as part of the iTTM distribution. If the behaviour of the ITTMMessage class has been overridden by a sub-class (see the 'Changing Standard Message Path Default Implementation' section later in this document) then the sub-class should be used in the ExtendsClass attribute. ONLY sub-classes of ITTMMessage are valid in Standard Message Path elements.

```
<ServiceConfig name="CreditCheckService"

className="com.iplanet.trustbase.standardpath.sample.creditcheck.Cr
editCheckService">

      <RootElement name="CreditCheckRequest"/>

      <RootElement name="CreditCheckResponse"/>

      <RootElement name="CreditCheckError"/>

  </ServiceConfig>

</Config>
```

The final lines define the contents of the Standard Message Path service descriptor file, The name of the service is used for authorisation once the service is deployed into iTTM, the class name defines the class that will be executed by iTTM to process the message 'payload'. This must implement the 'uk.co.jcp.tbase.service.Service' interface (see the next section). Each of the acceptable document root elements is enumerated as a <RootElement>, each of these entries will result in an iTTM routing rule which routes messages with that document type to the named service.

The result of using the jaxhit tool are the following items:

- Java Classes
  These classes represent the XML elements defined by the messaging DTD, and are capable of parsing the appropriate message elements (by being called by a SAX parser) and are also capable of validating the parsed elements against their DTD definition.

- Service Descriptor
  A properties file that defines the name of the service, the main class that implements the service and the list of XML document types that the service will process.

Write service Java code

This is the Java code which acts upon the incoming message, processes the content and generates the ‘payload’ for the response message. The service class must implement the ‘uk.co.jcp.tbase.service.Service’ interface. A skeleton service class is shown below and a small but complete service is provided with the credit check sample in the distribution.

```
package com.test.package;
import uk.co.jcp.tbase.service.Service;
import uk.co.jcp.tbase.service.ServiceException;
import uk.co.jcp.tbase.environment.Message;
import com.iplanet.trustbase.standardpath.message.ITTMMessage;
import com.iplanet.trustbase.standardpath.util.ITTMUtil;
import com.iplanet.trustbase.standardpath.util.ITTMConstants;
import uk.co.jcp.tbase.environment.attribute.Attribute;
import uk.co.jcp.tbase.environment.attribute.AttributeList;
import uk.co.jcp.tbase.environment.attribute.AttributeConstants;
import com.iplanet.trustbase.xml.message.TbaseIdentifiedElement;
import com.iplanet.trustbase.xml.message.TbaseElement;
import com.iplanet.trustbase.xml.message.TbaseElementException;
import uk.co.jcp.tbaseimpl.log.error.ErrorLog;
import uk.co.jcp.tbaseimpl.log.error.ErrorObject;
import java.rmi.RemoteException;
import java.io.IOException;
public class A_Service implements Service
{
   /** See parent class
        */
       public Message process( String serviceName, Message message )
             throws ServiceException, RemoteException
        {
      // Get the content and de-serialize it
```

```
        ITTMMessage request = null;
                   ITTMMessage response = null;
        try
        { request = (ITTMMessage)ITTMUtil.deserializeMessageContent(
message );}
        catch ( IOException iox )
        {
ErrorObject error = new ErrorObject( "AAA0001",iox );
            ErrorLog.log( error );
        }
        catch ( ClassNotFoundException cnfx )
        { ErrorObject error = new ErrorObject ( "AAA0002", cnfx );
            ErrorLog.log( error ); }
        catch (ClassCastException ccx)
        { ErrorObject error = new ErrorObject ( "AAA0003", ccx);
            ErrorLog.log( error ); }
        response = processITTMMessage(request);
// Set the ReturnToUser attribute to signal to the router that the
// message is to be sent back to the user
message.getAttributes().addAttribute(new Attribute("ReturnToUser",
"true", null));
// Remove he Request message DOC_TYPE attribute and replace it with
// the correct Response message DOC_TYPE
                message.getAttributes().removeAttributes(
ITTMConstants.DOC_TYPE );
                message.getAttributes().addAttribute(new Attribute(
ITTMConstants.DOC_TYPE,( (TbaseElement) response ).elementName(),
null ) );

                // Serialize the content into the TBASE message
                ITTMUtil.serializeMessageContent( message, response
); return message;}
};
```

The above skeleton service implementation provides for a call to ‘processITTMMessage(….)’ which takes the Java object tree that represents the request and processes it to produce a response message. Reading the Java API documentation for ITTMMessage and its associated classes will provide an insight into what sort of operations can be performed on the request. It must also be remembered that ‘request’ can be cast down into the actual message type class e.g. CreditCheckRequest, there are a number of different abstractions that may apply to a given generated class. Generating Java API documentation for the generated classes will help to provide a better understanding of the make up of each message class.

Compile

Any Java compiler is acceptable for the generation of Standard Message Path classes, it is recommended that the chosen compiler be 1.2.x compliant, all examples in this document assume that the JDK from JavaSoft is being used.

Make JAR file

Any Java compliant JAR tool is acceptable, all examples in this document assume that the JDK from JavaSoft is being used.

Deploy into iTTM

Deploying a the new service into iTTM is as simple as copying the final JAR file into the ..../Trustbase/current/deploy directory and then logging on to iTTM as administrator and use the Services --> Deployment menu option. If the service already exists then uninstall the old version before installing the new version. If the new service is not visible, then the service JAR file is probably incorrectly constructed.

Changing the Standard Message Path Default Implementation

The basic ITTMProtocolHandler, ITTMMessageReader and ITTMMessageWriter provide a default implementation that will be adequate for most purposes, but each of these classes has been designed to allow partial or total rewrite of their functionality via sub-classing. The following sections define the control flows and methods which may be overridden to change the behaviour of the standard pipeline for specific message types.

Base Message Class ITTMMessage

The base com.iplanet.trustbase.standardpath.message.ITTMMessage class must be the base class of all root document elements in a messaging DTD, it implements all of the generic message signature creation and validation function. A message may be signed and verified as an opaque entity (i.e. without a semantic understanding of its ‘payload’). If this class is sub-classed and the sub-class is used as the base class for root document elements then the basic signature and verification behaviour may be affected.

1. Overridable Methods

- getElementsToSign()
  This method returns the elements of the XML message which must be signed in the signature.

- getMessageRootElementName()
  Gets the name of the XML root element of this iTTM message

- getSigningCertificate()
  Get the child certificate which validates the message signature

- getSigningChains()
  Get all of the signing chains that validate the message signature - normally this will only be a single chain, but in a cross-signed PKI it may be multiple chains

- getValidatingTokenKeyStore()
  Establish which TokenKeyStore a.k.a. Trust Domain

- getXmlSignature()
  Get the mandatory level 2 DSIG signature

- setXmlSignature(...)
  Set the signature on the message, this will overwrite any signature that may currently be associated with the message
- signMessage(...)
  Create the mandatory level 2 signature and set it into the message.
- traverseTreeAndSetIds(...)
  Traverse the XML message tree and set all ID attributes in elements to a value if they are not already set.
- validateSignature(java.util.Date atDate)
  Validate the mandatory level 2 signature, using the appropriate trust domain.

Protocol Handler

The protocol handler (com.iplanet.trustbase.standardpath.protocol.ITTMProtocolHandler) is the first component in the incoming messaging pipeline and it can be sub-classed to change its behaviour. The control flow sections defines which methods are called in which order, allowing scope for changing the specific methods whilst leaving the control flow untouched.

Control Flow

The basic ITTM message processing pipeline provides an implementation of a protocol handler that accepts wild card mime types beginning with *application/ittm-*. This default handler performs the following functions in the following order, method names for each function are specified:

- readRawXML(...) - Read the Raw XML from the stream into an array.
- checkInputLength(...) - Check input length read in against the content specified length.
- preProcessXML(...) - Call a raw bytes pre-processing function. The default implementation performs no function here, it is an override point.
- getXMLEncodingIdentifier(...) - Extract the XML character encoding identifier from the raw bytes. The default implementation performs no function here, it is an override point.

- parseXmlMessage(...) - Use the class generator message factory to XML parse the message and create the internal object tree representation of the message which is made up of classes created by the Class Generator tool. Apart from structural XML encoding, no validation of the message occurs at this point.
- validateXmlMessage(...) - Validate the message, this validation checks that the message is structurally compliant with the DTD that describes it.
- rawLog(...) - Raw log the complete message string, this will result in a unique raw log identifier being returned to the protocol handler. This unique identifier is added as a iTTM attribute ITTMConstants.RAW_LOG_ID to the message, this enables other stages of message processing to relate their storage back to the original raw log record for the message.
- setContextId(...) - Extract a unique identifier from the message for use as an TTM external context identifier this identifier is very important in multiple request message protocols, it allows a context to be stored in TTM which lives longer than just the first request/response pair. It is a requirement of a compliant iTTM message that this identifier is present in all messages, although its form and location are not specified. The default implementation of this will generate a random identifier whose life will be one request/response pair, thus precluding any multiple message protocols.
- setDocType(...) - Set the ITTMConstants.DOC_TYPE attribute in the iTTM message with the DOCTYPE of the XML message.
- Set the iTTM message type to be IITMConstants.ITTM_MESSAGE_TYPE.

Overridable Methods

- checkInputLength(...)
  Check that the input length of the raw XML message matches the expected message content length
- getContentTypes()
  Get content types recognized by the ProtocolHandler.
- getRawLogStoreName()
  This returns the name of the raw log that will be used to store the message, by default this is the 'identrus' raw log
- getXMLBytes(...)
  To allow i18n, convert a Unicode string into raw XML bytes in a known character encoding
- getXMLEncodingIdentifier(...)
  Returns a java character encoding string for the message

- getXMLString(...)
  To allow i18n, convert raw XML bytes in a known character encoding into a Unicode string
- handle(...)
  Extract the Message type, determine response content type, and create an execution environment for the Message
- parseXmlMessage(...)
  Parse the XML message into the Java object tree and checks that the message is XML compliant structurally.
- preProcessXML(...)
  Before the XML is parsed this method is called to allow any pre-processing of the raw message bytes.
- processParsingException(...)
  An exception occurred whilst reading/parsing the XML, this does not include DTD validation
- processProcessingException(...)
  An exception has ocurred in the general processing of the XML message, e.g.
- processRawLoggingException(...)
  An exception occurred whilst trying to Raw log the XML
- processUnspecifiedException(...)
  An general exception has occurred
- processValidationException(...)
  An exception occurred whilst validating the incoming XML message against its DTD
- rawLog(...)
  Log the content of the TbaseElement into the database.
- readRawXML(...)
  Read raw XML message from the InputStream into the byte array.
- setContextId(...)
  This method sets the context id of the message.
- setDoctype(...)
  Set the DOCTYPE attribute in the iTTM message based on the DOCTYPE of the Xml message
- validateXmlMessage(...)
  Validate the message against its original DTD.

Error Handling

All errors in the main control flow are caught, error logged and then one of the process….(…) methods is called, to allow a point where error handling behaviour may be changed by sub-classing. There is an example of this sub-classing in the Credit Check Sample that is supplied with the distribution.

The protocol handler error processing methods need to return a ProtocolHandlerException, if this exception has been constructed with a response message bytes and response mime-type, then the response message is returned to the user without further processing. If no response message bytes are provided then an HTTP level error is returned on the connection e.g. FileNotFound.

Message Reader

The message reader (com.iplanet.trustbase.standardpath.message.ITTMMessageReader) is the final component in the incoming messaging pipeline and it can be sub-classed to change its behaviour. The control flow sections defines which methods are called in which order, allowing scope for changing the specific methods whilst leaving the control flow untouched.

Control Flow

A MessageReader parses the content of a Message from an InputStream. MessageReader may be a part of an application, and have specific knowledge of Message types, or they may be general purpose and have general knowledge of Message formats.

The basic ITTM message processing pipeline provides an implementation of a message reader that accepts wild card mime types beginning with *application/ittm-* and a message type of *ITTMConstants.ITTM_MESSAGE_TYPE*. This default handler performs the following functions in the following order, method names for each function are specified:

- Default implementation accepts wild card mime types beginning with 'application/ittm-' and a message type of ITTM_MESSAGE (defined by ITTMConstants.ITTM_MESSAGE_TYPE)

- Deserialise the XML message structure from the iTTM message
- Set the 'security.role' attribute to be 'unset'
- checkXMLSignature(...) - Check the mandatory iTTM message signature applied to the message, this includes certificate chain checks at the current date and that the root of the chain is trusted in a single domain.
- processSignature(...) - This is a hook to allow further signature processing to be added in a child class. The default implementation does nothing in this method.
- setTrustDomain(...) - This sets the ITTMConstants.PKI_IDENTIFIER attribute to be the name of the trust domain into which the message validated. This sets a trust domain context for all further crytographic operations on this message, including the signing of the response.
- setSecurityContext(...) - This method sets the message attributes which allow the authorisation component of iTTM to determine the authentication level of the message. This adds a number of attributes to the message, two for each certificate in the validating certificate chain. These are an attribute named SecurityContext.CERTIFICATE_DN+"."+counter, whose value is set to the issuer dn of the certificate at position 'counter' in the chain. 'counter' is 1 based, child certificate first. For each of these attributes there is also an attribute named SecurityContext.CERTIFICATE_SN+"."+counter, whose value is the serial number of the certificate at position 'counter' in the chain.
- doBilling(...) - This method provides a hook point for child classes to implement a billing stage in the message processing. The default method implementation does nothing.
- Serialise the XML message structure into the iTTM message ready to be routed to a service.

Overridable Methods

- checkXMLSignature(...)
  This method checks the mandatory level 2 XML signature on the message, including checking the certificate chain is valid at todays date and that its root certificate exists in one and only one trust domain.
- doBilling(...)
  Provide access the billing system to perfrom billing for the processed message.
- getTypeAndFormats()
  get a list of the Message types and formats supported by this MessageReader.

- processIncompleteCertChain(...)
  The certificate chain for validating the message signature was not supplied and could not be re-created from the certificates in the TokenKeyStore
- processInvalidCertChain(...)
  The certificate chain used to validate the message is invalid, signatures maybe corrupt or certificates may be out of the valid date range
- processInvalidSignature(...)
  Called if the mandatory level 2 signature check fails, this indicates that the message or signature has been tampered with.
- processProcessingException(...)
  An exception occurred during the processing of the mandatory level 2 signature
- processSignature(...)
  Allow further processing of the message signature such as the OCSP check, the default implementation of this method does nothing.
- processSignatureTrustFailure(...)
  Called if the root certificate in the mandatory level 2 signature certificate chain doesn't exist in any trust domain or if it exists in more than one domain.
- processUnspecifiedException(...) A general exception occurred during the processing of the message
- read(...)
  parse the content of a Message from the InputStream
- setSecurityContext(...)
  Set certificate chain attributes on the iTTM message, in preparation for the authorisation stage to check the authentication level of the message.
- setTrustDomain(...)
  Set the private key identifier to the ITTMConstants.PKI_IDENTIFIER message attribute.

Error Handling

All errors in the main control flow are caught, error logged and then one of the process….(…) methods is called, to allow a point where error handling behaviour may be changed by sub-classing. There is an example of this sub-classing in the Credit Check Sample that is supplied with the distribution.

The message reader error processing methods need to return a MessageReaderException, if this exception has been constructed with a response message, then the response message is returned to the user via the appropriate (based on the value of the message type and mime-type) message writer. If no response message is provided then an HTTP level error is returned on the connection e.g. FileNotFound.

Message Writer

The message writer (com.iplanet.trustbase.standardpath.message.ITTMMessageWriter) is the only component in the outbound messaging pipeline and it can be sub-classed to change its behaviour. The control flow sections defines which methods are called in which order, allowing scope for changing the specific methods whilst leaving the control flow untouched.

The ITTMMessageWriter component implements the DirectionalMessageWriter interface which is derived from the more generic MessageWriter interface. The ITTMMessageWriter component handles messages with mime types beginning with 'application/ittm-' ie ITTMMessages. It specifically handles messages whose type is either: ITTMConstants.ITTM_MESSAGE_TYPE or ITTMConstants.ITTM_AUTH_ERROR_MESSAGE_TYPE. The ITTMMessageWriter signs the outbound XML message using the XML DSIG profile specified for ITTMMessage, it then does an XML validation of the message, logs it and writes it to the client stream.

- Default implementation accepts wild card mime types beginning with 'application/ittm-'
- Check to see if the message is an authorisation failure, i.e. if its message type is set to ITTMConstants.ITTM_AUTH_ERROR_MESSAGE_TYPE - if so then call processAuthorisationError(...)
- Deserialise the XML message structure from the iTTM message
- checkDoNotSignAttribute(...) - Check for the existence of the ITTMConstants.ITTM_DO_NOT_SIGN message attribute, if this is present then don't sign the message. It must be noted that if the attribute is present but no signature element is on the message then it will fail XML validation before it is sent out.

- getSigningChain(...) & getSigningKey(...) - Create a message signature structure and add it to the message, always overwrite any existing structure in the default implementation. The XML DSIG profile used by the default implementation will use reference objects for all level 2 elements except the signature - these references will be bare-name x-pointers. Select a private key which has an attribute of ITTMConstants.ITTM_SIGNING_KEY associated with, or if the iTTM message attribute ITTM_SIGNING_KEY_ATTRIBUTE is present then select a private key with the named attribute. Use the iTTM message attribute ITTMConstants.PKI_IDENTIFIER to retrieve the trust domain name for the signing key to be used.
- Call an XML validation on the outbound message - check that it complies with its DTD.
- getXMLEncodingIdentifier(...) - Get the XML encoding identifier
- addDocType(...) - Set the DOCTYPE of the message, along with its system identifier (which can be extracted from XML_SYSID iTTM message attribute).
- rawLog(...) - Raw log the response message – passing in the original request raw log identifier (iTTM Message attribute RAW_LOG_IDENTIFIER). Using the message attribute will allow multiple message protocols to have all of their message related back to the original request – except in the case where no implementation has been provided for extracting an external context identifier.
- Set the mime-type of the response message.
- getXMLBytes(...) - Get the XML bytes and write to client stream.

Overridable Methods

- addDocType(...)
  Add tag to the message, identifying the public and system identifiers of the document.
- checkDoNotSignAttribute(...)
  Checks for existence of ITTMConstants.ITTM_DO_NOT_SIGN message attribute
- getContentType(...)
  Gets the actual mime-type to assign to data to a Message to be translated to an OutputStream by a call to write().
- getRawLogStoreName()
  This returns the name of the raw log that will be used to store the message, by default this is the 'identrus' raw log

- getSigningChain(...)
  Retrieve the chain to be used for validation of the XML signature which will be added to the document.
- getSigningKey(...)
  Retrieve the private key to be used for signing the XML document
- getTypeAndFormats()
  Gets a list of the Message types and formats supported by ITTMMessageWriter.
- getXMLBytes(...)
  To allow i18n, convert a Unicode string into raw XML bytes in a known character encoding
- getXMLEncodingIdentifier(...)
  Returns a java character encoding string for the message
- getXMLString(...)
  To allow i18n, convert raw XML bytes in a known character encoding into a Unicode string
- processAuthorisationFailureException(...)
  Called if there was an authorisation failure in the router whilst attempting to process the original request.
- processKeyStoreException(...)
  Called if there is a problem recovering the signing key or certificate chains from the stores
- processProcessingException(...)
  Called if there is a problem in the general processing of the response message
- processRawLoggingException(...)
  Called if there is a problem writing the response message to the raw log
- processSignatureCreateException(...)
  Called if there is a problem generating the mandatory level 2 signature structure for the response message
- processUnspecifiedException(...)
  A general exception occurred during the processing of the message
- rawLog(...)
  Raw Logs the response XML bytes into the Database
- write(...)
  Translates a Message to an external format, and write to an OutputStream

Error Handling

All errors in the main control flow are caught, error logged and then one of the process….(…) methods is called, to allow a point where error handling behaviour may be changed by sub-classing. There is an example of this sub-classing in the Credit Check Sample that is supplied with the distribution.

The message writer error processing methods need to set the content of the iTTM message to be the message to be returned. This message content is then written directly to the return stream by the error processing method. If no response message is written to the output stream and instead a MessageWriterException is thrown, then an HTTP level error is returned on the connection e.g. FileNotFound

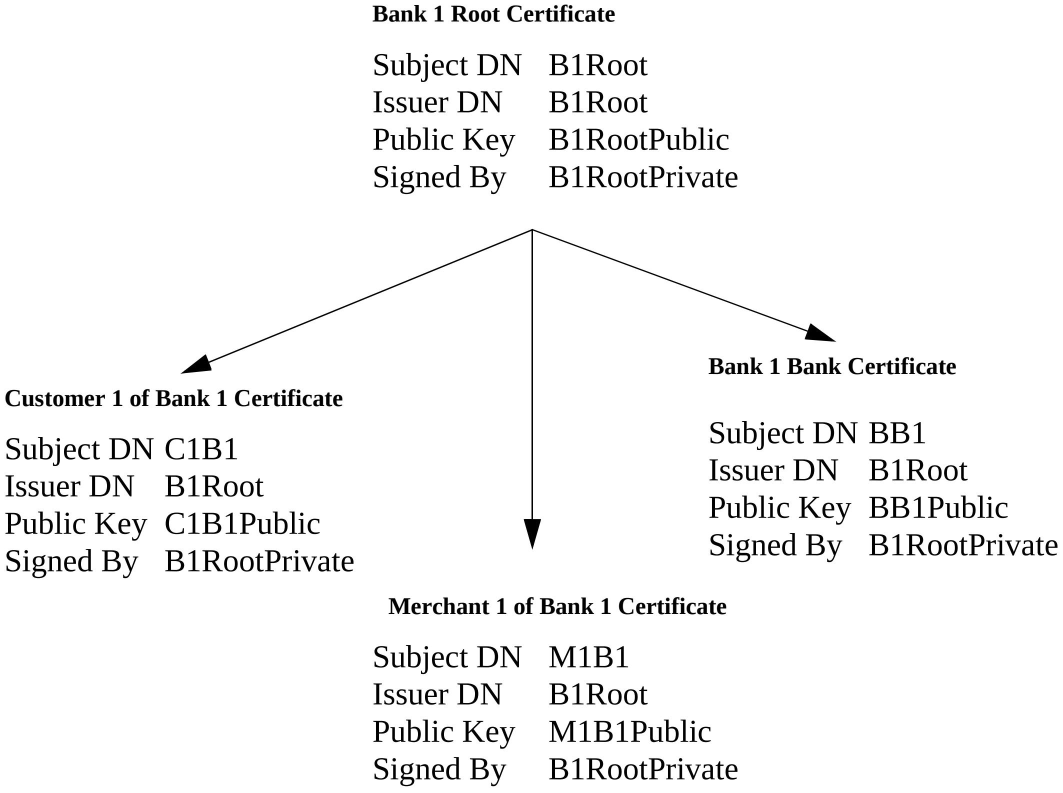
Example Application

The Example Credit Check Messaging Protocol

The example messaging protocol will implement a very simple credit check function between a merchant and his bank, regarding the customer. The messaging will be supported by a very simple PKI that can be easily extended to interoperate with another bank.

Public Key Infrastructure

The standard 3 party transaction need only have the following very simple PKI:



Bank 1 Root Certificate

Subject DN	B1Root
Issuer DN	B1Root
Public Key	B1RootPublic
Signed By	B1RootPrivate

Customer 1 of Bank 1 Certificate

Subject DN	C1B1
Issuer DN	B1Root
Public Key	C1B1Public
Signed By	B1RootPrivate

Bank 1 Bank Certificate

Subject DN	BB1
Issuer DN	B1Root
Public Key	BB1Public
Signed By	B1RootPrivate

Merchant 1 of Bank 1 Certificate

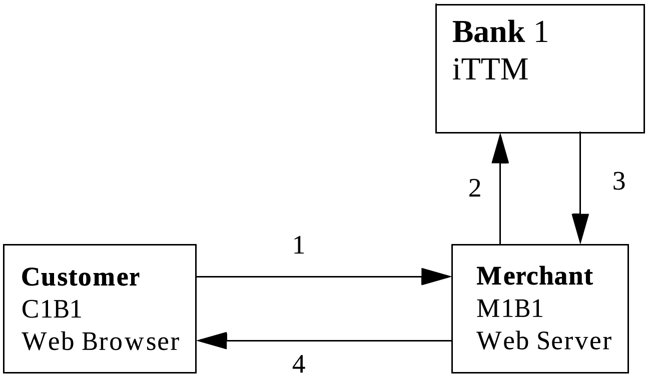
Subject DN	M1B1
Issuer DN	B1Root
Public Key	M1B1Public
Signed By	B1RootPrivate

This PKI has certificates for the following specific uses:

1. Bank 1 Root is only for issuing other certificates, this certificate is available to all customers and merchants.
2. Customer 1 of Bank 1 certificate is used to verify all signatures made by that customer, the private key remains in the possession of the customer, who signs requests and presents this certificate for the recipient to validate the requests.
3. Merchant 1 of Bank 1 certificate is used to verify all signatures made by that merchant. The merchant only ever sends requests to his associated bank, all of these requests must be signed by the merchant's private key.
4. Bank 1 Bank certificate is used when bank 1 contacts any other bank or any of its merchants.

Message Protocol

The basic 3 party model is described below



Three Party Variant

The three party variant of the transaction is defined by the following sequence of messages and occurs when the customer in the transaction is a customer of Bank 1 (i.e. it has a certificate issued by Bank 1).

1. This message would normally be a set of request parameters together with a PKCS#7 signature, generated by a smartcard signing plug-in. The request parameters are signed using the C1B1 private key and the C1B1 certificate and B1Root certificate are included in the PKCS#7 and the whole lot is sent to the merchant web-server.

2. This message would be generated by the merchant once the merchant has examined the request parameters and verified the PKCS#7 signature. The merchant would then generate a CreditCheckRequest message to be sent to Bank 1, asking for a credit rating for customer C1B1. This message contains the C1B1 certificate and is signed by the M1B1 private key.

3. This message would be generated by the bank once the bank has done the following:

   a. Verify the signature of M1B1 on the CreditCheckRequest. If the signature check fails then a CreditCheckError message is returned with the status of ‘signature_fail’.

   b. Verify the certificate status of M1B1 certificate with its local OCSP responder. If the certificate is revoked or its status is unknown then a CreditCheckError message is returned with a status of ‘certificate_invalid’.

   c. Check that M1B1 is authorised to make CreditCheckRequests. If the merchant is not authorised to make a CreditCheckRequest then a CreditCheckError message is returned with the status of ‘unauthorised’

   d. The Customer Certificate contained in the CreditCheckRequest is extracted and examined. If this certificate was issued by this bank then a local OCSP responder check is made to establish if the certificate is still valid. If the certificate status is revoked or unknown then a CreditCheckResponse message is returned with a CreditRating of ‘Unknown’.

   e. Once this has passed then a call to an external credit reference database is made to establish the rating of the customer. The credit rating is returned in a CreditCheckResponse Message containing the appropriate CreditRating.

   f. The returned message (CreditCheckError or CreditCheckResponse) is always signed with the BB1 private key.

4. This message is merchant specific and is based on the credit rating received by the merchant from his bank. This message is only sent once the merchant has verified the bank signature on the response message and is satisfied that the message can be traced back to the B1Root certificate that the merchant implicitly trusts.

Message Definition

This section contains the DTD's that describe the credit check messaging. These DTD's may be used by the iTTM Class Generator to create the message parsing classes for the implementation.

The construction of the messages should be self-explanatory if the DTD is read in conjunction with message protocol definition, but a few extra notes about message construction:

- The txid attribute in the TxBlock of the message should be the same for all messages associated with a given transaction. This txid would be generated by the merchant and would remain with all messages. This txid can also be used by iTTM as an external context identifier. Although, this message protocol does not require the use of contexts within iTTM that last longer than a single request/response pair.
- The ClientCertificate is optional is the CreditCheckResponse, it may be useful to return this so that credit ratings can be easily cached against certificates. It may also be useful if a CreditCheckResponse is being contained within another protocol that may require access to the certificate that was used to get the credit rating.
- The ClientCertficate element contains a base 64 encoded X.509 certificate.
- The construction of the signature element can be done in any number of ways providing it adheres to the basic requirements of iTTM standard messaging.
- The vendorcode attribute of the ErrorInfo element is there to allow a vendor to attach a specific reason code to a failure.

The Following is the full DTD for this example

```
<!--PUBLIC ID for this DTD is: "-//IPLANET//ITTM CREDIT
CHECK//EN"-->

<!ENTITY % XMLDSIG.dtd PUBLIC "http://www.w3.org/2000/09/xmldsig#"
"./xmldsig.dtd">

%XMLDSIG.dtd;

<!ELEMENT CreditCheckRequest ( TxBlock, Signature, ClientCertificate
)>

<!ATTLIST CreditCheckRequest

   id ID #REQUIRED>

<!ELEMENT CreditCheckResponse ( TxBlock, Signature, CreditRating,
ClientCertificate? )>

<!ATTLIST CreditCheckResponse

   id ID #REQUIRED>

<!ELEMENT CreditCheckError (TxBlock, Signature, ErrorInfo )>

<!ATTLIST CreditCheckError

   id ID #REQUIRED>

<!ELEMENT TxBlock EMPTY>

<!ATTLIST TxBlock

   id ID #REQUIRED

   txid CDATA #REQUIRED

   txtime CDATA #REQUIRED>

<!ELEMENT ClientCertificate (#PCDATA)>

<!ELEMENT CreditRating EMPTY>

<!ATTLIST CreditRating

   id ID #REQUIRED

   rating ( AAA | AA | A | B | C | D | Unknown ) #REQUIRED>

<!ELEMENT ErrorInfo EMPTY>

<!ATTLIST ErrorInfo id ID #REQUIRED

   errorCode ( unparsable_content | unknown_dtd | signature_fail |
certificate_invalid | unauthorized | unspecified_error ) #REQUIRED

   vendorcode CDATA #IMPLIED>
```

Building & Installing the Example

The following instructions assume that you are using a Bourne Shell or derivative, and that you have a JDK (at least v1.2.2) installed.

1. From the Trustbase installation directory change into the Scripts sub-directory:

```
cd Scripts
```

2. Set the shell CLASSPATH environment variable to include all of the iTTM libraries

```
. ./setcp
```

3. Change directory into the Credit Check Sample resource directory

```
cd ../samples/creditcheck/resources
```

4. Generate the classes from the creditcheck.dtd file, using the JAXHIT tool

```
../../../Scripts/jaxhit -config CreditCheckExample.xml -y
```

5. Change directory to the parent directory and compile the generated classes

```
cd ..
```

```
javac -d build
src/com/iplanet/trustbase/generated/IPLANET/ITTM_CREDIT_CHECK/*.jav
a
```

6. Compile the credit check sample classes

```
javac -d build -classpath $CLASSPATH:build
src/com/iplanet/trustbase/standardpath/sample/creditcheck/*.java
```

7. Build all of the classes and the service descriptor file (build/config/tbasesvc.properties) into a JAR file which can be deployed into iTTM.

```
jar -cvf creditchecksample.jar -C build com -C build config
```

8. Make the new service JAR file available for deployment in iTTM

```
cp creditchecksample.jar ../../current/deploy
```

9. Install the additional error codes that the sample uses into the iTTM error database. This is done by executing the SQL script resources/credit_check_error_codes.sql against the database and user that iTTM uses.

10. Log on to iTTM as administrator and use the service deployment tool to deploy the service, using the Services --> Deployment menu option.

11. Change directory to the directory which contains the iTTM scripts, redefine the CLASSPATH environment variable to include the newly added creditcheck JAR using the setcp script.

```
cd ../../Scripts
```

```
. ./setcp
```

12. Stop iTTM

```
./stopias
```

13. Use the token key tool to mark one of the existing private keys as suitable for use by the Standard Message Path components to use to sign outbound messages.

```
./runtokenkeytool
```

```
> listkeys
```

```
> addalias -newalias ITTM_SIGNING_KEY -issuer "…issuer_dn…" -serial
serial_number
```

```
> quit
```

14. Start iTTM

```
./startias
```

15. Execute the test client which will submit a valid credit check request message to iTTM, the command line below assumes that you are running on a machine called foo.bar.com and that you have a key store password of ‘a_password’. The java code for this client is available in the ../samples/creditcheck/src tree. This will result in an error, because the certificate used to sign the message is not one of those authorised to use the new Credit Check Service – no authorisation entries have been made yet.

```
java
com.iplanet.trustbase.standardpath.sample.creditcheck.CreditCheckCl
ient -url http://foo.bar.com/NASApp/NASAdapter/TbaseNASAdapter
-domainuri file:../foo -password a_password -trustdomain identrus
```

16. The result of the previous step is an HTTP level error (FileNotFound), this is the behaviour of the default Standard Message Path Components. This HTTP level error is not normally acceptable for a messaging protocol, most protocols will define an error message which provides the client with a more acceptable response. As part of the CreditCheck sample messaging, a response message type of CreditCheckError is supposed to be returned under error conditions. In order to meet this requirement a new protocol handler, message reader and message writer will need to be installed to override the error handling behaviour of the default standard message path equivalents. Implementations

of these can be found in ../samples/creditcheck/src, they are installed by logging on to iTTM as administrator and selecting the Parser ◊ Configuration menu option. Add the following entries in the three sub sections of the configuration screen:

a. Message Readers, add an entry in Name: CreditCheckMessageReader, add entry in class: com.iplanet.trustbase.standardpath.sample.creditcheck.CreditCheckMessageReader, press the Add button.

b. Message Writers, add an entry in Name: CreditCheckMessageWriter, add entry in class: com.iplanet.trustbase.standardpath.sample.creditcheck.CreditCheckMessageWriter, press the Add button.

c. Protocol Handlers, add an entry in Name: CreditCheckProtocolHandler, add entry in class: com.iplanet.trustbase.standardpath.sample.creditcheck.CreditCheckProtocolHandler, press the Add button.

Press the Submit button to commit all of the changes to iTTM.

17. Restart iTTM to apply the configuration changes

```
./stopias
```

```
./startias
```

18. Re-run the client tool as described in step 15. This time a properly formed, XML DSIG signed CreditCheckError message is returned and displayed on the screen.

19. The final stage is to add an authorisation for the message, so that it can be legitimately routed to the CreditCheckService that has been written. In order to authorise the message, three configurations need to be added. This configuration is done by logging on to iTTM as administrator.

A role into which credit check requests may be authenticated, select Authorisation ◊ Add Role menu option and add a new role, with a description, leaving the Active checkbox ticked.

A service authorisation entry which maps access to the CreditCheckService to any client that authenticates into the newly defined role, select Authorisation ◊ Add Service menu option. complete the form, it is important to make sure that the service name is exactly right, the service name can be discovered by looking at the ‘Service’ entry shown on the Services ◊ Deployment screen, for the example this is CreditCheckService (case is important).

A certificate authentication entry, which maps any client whose XML DSIG signature can be verified by the named certificate into the newly defined role, select Authorisation ◊ Add Certificate. Entries made on this form describe a certificate in the PKI from which this authentication will occur, this may be an individual child certificate or at the other end of the scale it may be a root certificate. The Issuer DN is an RFC2253 formatted name string and the serial number is a decimal version of the certificate serial number. These two pieces of information can be displayed about any certificate in the PKI using the tokenkeytool (see step 13). The maximum depth indicates the number of child levels below the described certificate that will be authenticated by this entry, this allows a root certificate to be entered and then restrict the depth of allowable child certificates. For the example, an entry for the root certificate can be made with the correct depth to include the certificate which signs the request message (The one marked with the ITTM_SIGNING_KEY alias in step 13). The role must be set to the newly defined role, and the access checkbox should be left checked. Press the Add button and then press the Submit button.

20. Restart iTTM to apply the configuration changes

```
./stopias
```

```
./startias
```

21. Re-run the client tool as described in step 15. This time a properly formed, XML DSIG signed CreditCheckResponse message is returned and displayed on the screen.

Glossary and References

The objectives of this chapter are to cover

- Software Platform
- Protocols
    - Transport Protocols
    - Security Related Protocols
    - Trading Protocols
    - Message Protocols
- Glossary
    - Security related terms
    - Java Related terms
    - Server Definitions

Software Platform

Solaris 8 and JDK

`http://www.sun.com/software/solaris/cover/sol8.html`

Java

`http://www.javasoft.com`

iPlanet Application Server 4.1

`http://www.iplanet.com/products/infrastructure/app_servers/index.html`

iPlanet Web Server 6.0

`http://www.iplanet.com/products/infrastructure/web_servers/index.html`

Oracle 8i

`http://www.oracle.com`

Hardware Security nCipher KeySafe 1.0 and CAFast

`http://www.ncipher.com`

Transport Protocols

HTTP

HTTP/1.0 or 1.1 protocol:

`http://www.w3.org/Protocols/rfc1945/rfc1945.txt`

`http://www.ietf.org/rfc/rfc1945.txt`

SMTP RFC821

`ftp://ftp.isi.edu/in-notes/rfc821.txt http://www.imc.org/ietf-smtp/`

Security Related Protocols

S/MIME Version 2 Message Specification

`ftp://ftp.isi.edu/in-notes/rfc2311.txt`

`http://www.imc.org/ietf-smime`

`http://www.ietf.org/rfc/rfc2311.txt`

DOMHASH

`http://www.ietf.org/rfc/rfc2803.txt`

OCSP

`http://www.ietf.org/rfc/rfc2560.txt`

Certificate requests and responses

PKCS10 requests RFC2314 can be found in

`http://www.ietf.org/rfc.html`

PKCS7 responses RFC2315 can be found in

`http://www.ietf.org/rfc.html`

Trading Protocols

Identrus

`http://www.identrus.com`

Transaction Coordinator requirements (IT-TCFUNC)

Core messaging specification (IT-TCMPD)

Certificate Status Check Messaging specification (IT-TCCSC)

Message Protocols

DOM

`http://www.w3.org/TR/REC-DOM-Level-1/`

DTD

`http://www.w3.org/XML/1998/06/xmlspec-v20.dtd`

XML

`http://www.w3.org/TR/REC-xml`

XML Syntax Processing specification

`http://www.w3.org/TR/xmldsig-core`

HTML

HTML 3.2 as specified in

`http://www.w3.org/TR/REC-html32.html`

Security related terms

3DES Similar to DES.

Application protocol An application protocol is a protocol that normally layers directly on top of the transport layer (e.g., TCP/IP). Examples include HTTP, TELNET, FTP, and SMTP.

Asymmetric cipher See “Public key cryptography”.

ASN.1 Abstract Syntax Notation One.

Authentication Authentication is the ability of one entity to determine the identity of another entity.

base64 A representation of characters in digital format using a 65 character subset of U.S. ASCII.

BBS A random number generating algorithm.

BER Basic encoding Rules used with X509.

Block cipher A block cipher is an algorithm that operates on plaintext in groups of bits, called blocks. 64 bits is a typical block size.

Bulk cipher A symmetric encryption algorithm used to encrypt large quantities of data.

Cipher Block Chaining Mode (CBC) CBC is a mode in which every plaintext block encrypted with the block cipher is first eXclusive-OR-ed with the previous ciphertext block (or, in the case of the first block, with the initialization vector).

Certificate As part of the X.509 protocol (a.k.a. ISO Authentication framework), certificates are assigned by a trusted Certificate Authority and provide verification of a party's identity and may also supply its public key.

Client The application entity that initiates a connection to a server.

Client write key The key used to encrypt data written by the client.

Client write MAC secret The secret data used to authenticate data written by the client.

Connection A connection is a transport (in the OSI layering model definition) that provides a suitable type of service. For SSL, such connections are peer to peer relationships. The connections are transient. Every connection is associated with one session.

CRL Certificate Revocation List A certificate that is not valid but still within its expiry date.

Data Encryption Standard (DES) DES is a very widely used symmetric encryption algorithm. DES is a block cipher.

DER Distinguished Encoding rules used in X509.

DH A public-key cryptographic algorithm for encrypting and decrypting data.

Digital Signature Standard (DSS) A standard for digital signing, including the Digital Signing Algorithm, approved by the National Institute of Standards and Technology, defined in NIST FIPS PUB 186, "Digital Signature Standard," published May, 1994 by the U.S. Dept. of Commerce.

Digital signatures Digital signatures utilise public key cryptography and one-way hash functions to produce a signature of the data that can be authenticated, and is difficult to forge or repudiate.

DSA Digital Signature Algorithm.

Handshake An initial negotiation between client and server that establishes the parameters of their transactions.

Initialization Vector (IV) When a block cipher is used in CBC mode, the initialisation vector is eXclusive-OR-ed with the first plaintext block prior to encryption.

IDEA A 64-bit block cipher designed by Xuejia Lai and James Massey.

Message Authentication Code (MAC) A Message Authentication Code is a one-way hash computed from a message and some secret data. Its purpose is to detect if the message has been altered.

Master secret Secure secret data used for generating encryption keys, MAC secrets, and IVs.

MD5 MD5 is a secure hashing function that converts an arbitrarily long data stream into a digest of fixed size.

MIME MultiPURPOSE Internet Mail Extension

Message digest A digest algorithm converts data of any size, via a one-way hashing function, into a small fixed size unique representation. Message digests are used extensively in the generation of digital signatures and integrity checking of data.

PBE Password based encryption

PEM Privacy enhanced mail

Public Key Infrastructure (PKI) Defines protocols to support online interaction.

Public key cryptography A class of cryptographic techniques employing two-key ciphers. Messages encrypted with the public key can only be decrypted with the associated private key. Conversely, messages signed with the private key can be verified with the public key.

One-way hash function A one-way transformation that converts an arbitrary amount of data into a fixed-length hash. It is computationally hard to reverse the transformation or to find collisions. MD5 and SHA are examples of one-way hash functions.

OSI Open Systems Inter-Connection.

RC2, RC4 Proprietary bulk ciphers from RSA Data Security, Inc. RC2 is block cipher and RC4 is a stream cipher.

RFC A series of authoritative discussion documents. Requests for Comments.

RSA A very widely used public-key algorithm that can be used for either encryption or digital signing.

Salt Non-secret random data used to make export encryption keys resist pre-computation attacks.

Server The server is the application entity that responds to requests for connections from clients. The server is passive, waiting for requests from clients.

Server write key The key used to encrypt data written by the server.

Server write MAC secret The secret data used to authenticate data written by the server.

Session A SSL session is an association between a client and a server. Sessions are created by the handshake protocol. Sessions define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.

Session identifier A session identifier is a value generated by a server that identifies a particular session.

SHA The Secure Hash Algorithm is defined in FIPS PUB 180-1. It produces a 20-byte output

SSL Secure sockets layer

Stream cipher An encryption algorithm that converts a key into a cryptographically-strong keystream, which is then eXclusive-OR-ed with the plaintext.

Symmetric cipher See “Bulk cipher”.

TSL Transport security layer

X690 The ASN.1 specification

X509 An authentication framework based on ASN.1 BER and DER and base64.

Java Related terms

Abstract class A class that contains one or more abstract methods, and therefore can never be instantiated. Abstract classes are defined so that other classes can extend them and make them concrete by implementing the abstract methods.

Abstract method A method that has no implementation.

API Application Programming Interface The specification of how a programmer writing an application accesses the behaviour and state of classes and objects.

Applet A program written in Java to run within a Java-compatible web browser, such as HotJava™ or Netscape Navigator™.

Atomic Refers to an operation that is never interrupted or left in an incomplete state under any circumstance.

Bean A reusable software component. Beans can be combined to create an application.

Class In Java, a type that defines the implementation of a particular kind of object. A class definition defines instance and class variables and methods, as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass will implicitly be Object.

Classpath A classpath is an environmental variable that tells the Java Virtual Machine and other Java applications (for example, the Java tools located in the JDK1.1.X\bin directory) where to find the class libraries, including user-defined class libraries.

Codebase Works together with the code attribute in the <APPLET> tag to give a complete specification of where to find the main applet class file: code specifies the name of the file, and codebase specifies the URL of the directory containing the file.

Core class A public class (or interface) that is a standard member of the Java Platform. The intent is that the Java core classes, at minimum, are available on all operating systems where the Java Platform runs. A 100%-pure Java program relies only on core classes, meaning it can run anywhere.

Critical section A segment of code in which a thread uses resources (such as certain instance variables) that can be used by other threads, but that must not be used by them at the same time.

Deprecation Refers to a class, interface, constructor, method or field that is no longer recommended, and may cease to exist in a future version.

Derived from Class X is "derived from" class Y if class X extends class Y. See also “Extends”.

Exception An event, during program execution, that prevents the program from continuing normally; generally, an error. Java supports exceptions with the try, catch, and throw keywords. See also “Exception handler”.

Exception handler A block of code that reacts to a specific type of exception. If the exception is for an error that the program can recover from, the program can resume executing after the exception handler has executed.

Extends Class X extends class Y to add functionality, either by adding fields or methods to class Y, or by overriding methods of class Y. An interface extends another interface by adding methods. Class X is said to be a subclass of class Y. See also “Derived from”.

GUI Graphical User Interface. Refers to the techniques involved in using graphics, along with a keyboard and a mouse, to provide an easy-to-use interface to some program.

HotJava™ Browser An easily customisable Web browser developed by Sun Microsystems that is written in Java.

HTML HyperText Markup Language This is a file format, based on SGML, for hypertext documents on the Internet. It is very simple and allows for the embedding of images, sounds, video streams, form fields and simple text formatting. References to other objects are embedded using URLs.

HTTP Hypertext Transfer Protocol The Internet protocol, based on TCP/IP, used to fetch hypertext objects from remote hosts. See also “TCP/IP”.

IDL Java Interface Definition Language Java API's that provide standards-based interoperability and connectivity with CORBA (Common Object Request Broker Architecture).

Instance An object of a particular class. In Java programs, an instance of a class is created using the new operator followed by the class name.

Interface In Java, a group of methods that can be implemented by several classes, regardless of where the classes are in the class hierarchy.

IP Internet Protocol The basic protocol of the Internet. It enables the unreliable delivery of individual packets from one host to another. It makes no guarantees about whether or not the packet will be delivered, how long it will take, or if multiple packets will arrive in the order they were sent. Protocols built on top of this add the notions of connection and reliability. See also “TCP/IP”.

JAR file format JAR (Java Archive) is a platform-independent file format that aggregates many files into one. Multiple Java applets and their requisite components (.class files, images, sounds and other resource files) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction. It also supports file compression and digital signatures.

JavaBeans™ A portable, platform-independent reusable component model.

Java Database Connectivity (JDBC™) An industry standard for database-independent connectivity between Java and a wide range of databases. The JDBC™ provides a call-level API for SQL-based database access.

Java™ Development Kit (JDK™) A software development environment for writing applets and application in Java.

Java™ Foundation Class (JFC) An extension that adds graphical user interface class libraries to the Abstract Windowing Toolkit (AWT).

Java Platform The Java™ Virtual Machine and the Java core classes make up the Java Platform. The Java Platform provides a uniform programming interface to a 100% Pure Java program regardless of the underlying operating system.

Java Remote Method Invocation (RMI) A distributed object model for Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.

Java Runtime Environment (JRE) A subset of the Java™ Development Kit for end-users and developers who want to redistribute the JRE. The JRE consists of the Java Virtual Machine, the Java Core Classes, and supporting files.

JavaScript™ A Web scripting language that is used in both browsers and Web servers. It's only loosely related to Java and the name causes unnecessary confusion. Like any scripting language, it's used mostly to tie other components together or to accept user input.

Java™ Virtual Machine (JVM) The part of the Java Runtime Environment responsible for interpreting Java bytecodes.

JDK™ Java™ Development Kit A software development environment for writing applets and application in Java.

JFC Java™ Foundation Class An extension that adds graphical user interface class libraries to the Abstract Windowing Toolkit (AWT).

JRE Java Runtime Environment A subset of the Java Developer Kit for end-users and developers who want to redistribute the JRE. The JRE consists of the Java Virtual Machine, the Java Core Classes, and supporting files.

Just-in-time (JIT) Compiler A compiler that converts all of the bytecode into native machine code just as a Java program is run. This results in run-time speed improvements over code that is interpreted by a Java Virtual Machine.

JVM Java Virtual Machine The part of the Java Runtime Environment responsible for interpreting Java bytecodes.

Multithreaded Describes a program that is designed to have parts of its code execute concurrently. See also "Thread".

NCSA National Center for Supercomputer Applications.

Package A group of types. Packages are declared with the package keyword.

Process A virtual address space containing one or more threads.

RPC Remote Procedure Call. Executing what looks like a normal procedure call (or method invocation) by sending network packets to some remote host.

Sandbox Comprises a number of co-operating system components, ranging from security managers that execute as part of the application, to security measures designed into the Java Virtual Machine and the language itself. The sandbox ensures that a distrusted, and possibly malicious, application can't gain access to system resources.

Secure Socket Layer (SSL) A protocol that allows communication between a Web browser and a server to be encrypted for privacy. It can also provide communication between other entities.

Synchronized A Java keyword that, when applied to a method or code block, guarantees that at most one thread at a time executes that code.

TCP/IP Transmission Control Protocol based on IP. This is an Internet protocol that provides for the reliable delivery of streams of data from one host to another. See also "IP Internet Protocol".

Thin Client A system that runs a very light operating system with no local system administration and executes Java applications delivered over the network.

Thread The basic unit of program execution. A process can have several threads running concurrently, each performing a different job, such as waiting for events or performing a time-consuming job that the program doesn't need to complete before going on. When a thread has finished its job, the thread is suspended or destroyed. See also “Process”.

Unicode A 16-bit character set defined by ISO 10646. All Java source is written in Unicode.

URL Uniform Resource Locator. A standard for writing a text reference to an arbitrary piece of data in the WWW. A URL looks like "protocol://host/localinfo" where protocol specifies a protocol to use to fetch the object (like HTTP or FTP), host specifies the Internet name of the host on which to find it, and localinfo is a string (often a file name) passed to the protocol handler on the remote host.

Virtual machine An abstract specification for a computing device that can be implemented in different ways, in software or hardware. You compile to the instruction set of a virtual machine much like you'd compile to the instruction set of a microprocessor. The Java Virtual Machine consists of a bytecode instruction set, a set of registers, a stack, a garbage-collected heap, and an area for storing methods.

Wrapper An object that encapsulates and delegates to another object to alter its interface or behaviour in some way.

Server Definitions

API Application Programming Interface,

ASP Active Server Pages

Attribute An attribute is a string value that may be used in conjunction with a set of rules by the router to determine the next action to perform. Attributes are used to populate contexts with information about a message.

Business Logic Business logic is the 'user' code in the system. Business logic executes tasks such as 'debit account', 'retrieve balance' etc.

Configurable Entity Is any Service or component that uses Configuration Objects and the Configuration Manager.

Configuration Object Configuration Objects hold persistent configuration data for services.

Configuration Service Is a Service that implements a read-write interface to the Configuration Object.

Connection Manager Describes the process with which iPlanet Trustbase Transaction Manager communicates with external entities. It utilises the following objects to accomplish this task... Protocol Maps, Protocol Analysers, Handlers, Message Readers and Writers.

Connector The Connector is the main Connection Manager interface. It makes requests external to iPlanet Trustbase Transaction Manager. It takes a iPlanet Trustbase Transaction Manager Message containing the request, and a Destination Object describing the endpoint for the request.

Context Keeps a record of the current state of a given transaction.

Context Directive The action components that make up a ruleset.

CORBA Common Object Request Broker Architecture.

CSS Cascading Stylesheet.

Destination Represents the destination of an external request, made by the Connector. An application specifies an implementation of Destination, and a ProtocolMap that can transform the destination into a ProtocolDescriptor for the Connector, which can then make and manage the actual connection.

Directive The 'action' part of a rule that is executed when the preconditions are true.

DMZ De-militarised Zone.

DOM Domain Object Model.

DTD Data Type Definition or Document Type Definition.

EJB Enterprise Java Bean.

Environment A set of contexts that are associated with a particular message.

Host Environment Adaptor The environment adapter forms the interface between a host such as a web server or application server and iPlanet Trustbase Transaction Manager itself.

HSM Hardware Security Module.

HTML HyperText Markup Language.

IDL Interface Definition Language.

JDBC Java Database Connectivity.

JWS Java Web Server.

lastService An attribute containing the nameof the most recently executed service.

MessageType An attribute contained within a message which holds the type of a given message. Message types are externally defined by the user.

Message An internal representation of a request from the user or a response from the server. Messages are routed within the system.

Message Analyser Provides the logic to identify which message reader or writer to use for a particular message based on the transport and the external format of the content.

Message Log Manager Instantiates and allows access to Message Loggers. The message loggers are accessed according to which mime type they have registered interest in.

Message Logger Logs incoming and outgoing messages in their raw unprocessed form. The log can then be later queried and manipulated through the logManager or directly through the back end database engine.

Message Reader A Message Reader parses the remaining content of a Message from the InputStream, into the Message's content field. Message Readers may be a part of an application, and have specific knowledge of Message types, or they may be general purpose and have general knowledge of Message formats

Message Registry A section of the tbase.properties file that provides a mapping between a message type and the message readers and writers used to process it.

Message Writer A Message Writer translates processed Message objects into the clients required presentation protocol, and write the results onto an Output Stream, which is provided by the Protocol Analyser.

NAS Netscape Application Server

NSK Non Stop Kernel

OAS Oracle Application Server.

PKI Public Key Infrastructure.

Precondition A precondition is a boolean expression which must be true for its corresponding directive to be executed. Preconditions are expressed in terms of attributes and their values. There are two types of precondition; an assertion that an attribute with a particular name exists and an assertion that the attribute with a given name not only exists but has a specific value.

Private Rule Set Repository A collection of rule sets that ships with the iPlanet Trustbase Transaction Manager and allows built in services (such as the configuration services) to function**.**

Protocol Analyser Provides the logic to identify which protocol handler to use for a particular message type.

Protocol Descriptor Holds a description of the endpoint, transport and presentation protocols for a connection, in the form of a URL, and the format of the message to be sent, and as a mime type Implements Destination. It can be used with the SimpleProtocolMap for direct Destination addressing.

Protocol Handler The protocol Handler Component extracts the message type and context ID from the header of a message. There is usually one protocol handler for a particular message class e.g. iPlanet Trustbase Transaction Manager messaging, OFX etc. The protocol handler then routes appropriate protocol to the Message Analyser**.**

Protocol Map An application specifies ProtocolMap implementations to map it's Destination implementations to URLs and mime types that the Connector can use to make an actual connection.

Protocol Map Manager Manages a set of ProtocolMap implementations, selecting an appropriate ProtocolMap to translate a particular Destination implementation into a ProtocolDescriptor.

Public Rule Set Repository A user configurable collection of user-defined Rule Sets Rules in the private Rule Set Repository take precedence over rules in the public Rule Set Repository.

RMI Remote Method Invocation.

Role Role is not set of attributes, it is the name of a particular attribute which the system recognises. There are several such attributes including:

lastService - the nameof the most recently executed service
messageType - the type of a given message
role - a string representing the capacity in which the user is

using the system, e.g. role - operator, role =administrator etc.

Router The router provides a mechanism for imposing structure and ordering on the execution of services in a secure way which doesn't necessitate code changes.

Rule A Rule contains three components: a Rule Name, Precondition and Directive. If the precondition is true then the directive is executed.

Rule Name Every rule has a name. The rule is referred to by its name in the context of a ruleset.

Ruleset A collection of rules that route messages to one or more services to achieve a given task.

Service An object implementing business logic. Services are written by the user.

Service Registry A registry of services! Used to provide a lookup between service names and the classes that implement them.

Session A session is the container for all of the tasks a user is performing over a period of time.

Skeleton The server/remote portion of a distributed object under CORBA and RMI. The skeleton is invoked by the Stub. See also “Stub”.

SP Service Provider

State A collection of attributes associated with a task at a given instant in time.

Stub The client portion of a distributed object using mechanisms such as CORBA or RMI. The Stub is designed to hide the fact that the implementation of it's related object is not locally located. See also “Skeleton”.

Task A unit of work at the business level. A ruleset defines how a task will be executed.

TISS Transport Independent Stub Service

URL Uniform Resource Locator

X500 Set of Open Standards for directory services. See, for instance, Country code that is defined as an ISO standard `http://www.iso.ch` and X500 standard `http://www.itu.int/itudoc/itu-t/rec/x/x500up/x500.html`

XML Extensible Markup Language

XSL Extensible Stylesheet Language

Index

Appendix A

The JAXHIT Class Generation Tool

What is JAXHIT?

The purpose of the classgen tool is to create a Java language binding for an XML DTD. This means that it will generate Java classes representing the structures defined in the DTD, with a correspondence of one Java class per element in the DTD. The resulting classes can be used to parse existing XML into a Java representation, or to create a new XML document by constructing it programmatically.

JAXHIT Operation

The JAXHIT tool takes information from two sources; the DTD file and a configuration file. The configuration file specifies all the options and parameters used in the code generation.

Command Line

```
jaxhit [-options] [<public Id> <DTD file>]
```

where options include:

-help	displays help
-config <file>	Specify the name of the configuration file
-cp <classpath>	Specify the classpath to use for compilation
-y	Do not confirm before overwriting existing files
-stub	Generate a stub service
-quiet	quiet output
-v	verbose output
-debug	debug output

The public id and DTD file must be specified in the config file or on the command line.

Configuration File Format

The configuration file for the class generator is an XML document. The Config element is the root element of the configuration document. It holds global settings for the generation. It contains the following attributes:

Name	**Description**	**Default**	**Required**
srcDir	The directory in which to keep the generated source files	-	No
libDir	The directory in which to keep the compiled classes	-	No
outputJar	The name of the output Jar file to create	-	No
basePackage	The root package under which all generated classes will be located	com.iplanet.trustbase.generated	No
baseDir	A root location from which all other file references are taken to be relative	Current directory	No
recurse	Whether to attempt to generate classes for all DTD files referenced as ENTITY inclusions from the top level DTD	false	No
force	Whether to generate classes that are already present on the system classpath	false	No
build	Whether to attempt to compile the generated classes	true	No
createService	Whether to generate the stub service	false	No
classPath	The semicolon seperated class path to be used when compiling the generated classes. As the compiler cannot pick up the system class path when the class generator is invoked with the java –jar <jarfile> syntax, the classpath must be specified here instead.	-	No

Content

The Config element can contain the following child elements:

Element	**Occurrences**
DTDFile	0 or more
DefaultElementBase	0 or 1
ElementBase	0 or more
ServiceConfig	0 or 1

DTDFile Element

This element specifies the location and public ID for a DTD

Attributes

Name	**Description**	**Default**	**Required**
file	The path to the DTD	-	Yes
publicId	The Public Id of the DTD	-	Yes

Content

The DTDFile element has no content.

ElementBase Element

This element specifies element-specific options used in the generation process.

Attributes

Name	Description	Default	Required
name	The name of the element to process	-	Yes
inherit	Whether to inherit values from the Default-ElementBase (if present)	true	No
methodPrefix	A string to prepend to the generated get/set/add/remove methods	-	No

Content

The ElementBase element can contain the following child elements:

Element	Occurrences
ExtendsClass	0 or 1
ImplementsInterface	0 or more
AdvancedOptions	0 or 1
AttributeType	0 or more

DefaultElementBase Element

This element specifies the default options used in the generation process.

Attributes

Name	Description	Default	Required
methodPrefix	A string to prepend to the generated get/set/add/remove methods	-	No

Content

The DefaultElementBase element can contain the following child elements:

Element	Occurrences
ExtendsClass	0 or 1
ImplementsInterface	0 or more
AdvancedOptions	0 or 1
AttributeType	0 or more

AdvancedOptions Element

This element specifies advanced options used in the generation process.

Attributes

The AdvancedOptions element has no attributes

Content

The AdvancedOptions element can contain the following child elements:

Element	Occurrences
PreParseFragment	0 or 1
PostParseFragment	0 or 1
ClassDoc	0 or 1

PreParseFragment Element

This element specifies Java code to be included in the generated class, and called prior to the parse of sub elements. Because the code is called within the SAX parse process, the only exception that the code may throw is org.xml.sax.SAXException

Attributes

The PreParseFragment element has no attributes

Content

The textual content is the Java code to be inserted

PostParseFragment Element

This element specifies Java code to be included in the generated class, and called after parsing of the sub elements. Because the code is called within the SAX parse process, the only exception that the code may throw is org.xml.sax.SAXException

Attributes

The PostParseFragment element has no attributes

Content

The textual content is the Java code to be inserted

ClassDoc Element

This element specifies text to be included in the Class JavaDoc for a generated class. Note that as text specified using this mechanism is included in a Javadoc comment, usage of "*/" and "/*" character combinations is likely to cause problems.

Attributes

Name	Description	Default	Required
file	The file name of a file containing the text to be used	-	No

Content

The ClassDoc element can also contain textual content. If the file attribute *and* the textual content are present, then the text will be appended to the file text before inclusion.

ImplementsInterface Element

This element specifies an interface for a generated class to implement.

Attributes

Name	Description	Default	Required
name	The fully qualified name of an interface	-	Yes

Content

The ImplementsInterface element has no content.

ExtendsClass Element

This element specifies a base class for a generated class to extend.

Attributes

Name	Description	Default	Required
name	The fully qualified name of a class	-	Yes

Content

The ExtendsClass element has no content.

AttributeType Element

By default, the accessor methods for attributes take and return the Java String type. This element allows an attribute's accessor methods to operate on a different type. The primitive types supported are:

- int
- long

- float
- double
- boolean

It is also possible to specify the fully qualified name of a class that has the following properties:

- a public constructor taking a single String argument, and throwing no caught exceptions
- an orthogonal toString() implementation.

Attributes

Name	Description	Default	Required
name	The name of the attribute	-	Yes
type	The type to use	-	Yes

Content

The AttributeType element has no content.

ServiceConfig Element

This element specifies options specific to the generation of the service stub. For each root element that the service is expected to process, there should be a corresponding RootElement child element.

Attributes

Name	Description	Default	Required
name	The friendly name of the service	-	No
className	The fully qualified class name of the service	-	Yes
RoutingAttribute	The name of the message attribute to use for routing	DocType	No

Content

The ServiceConfig element can contain the following child elements:

Element	Occurrences
RootElement	1 or more

RootElement Element

This element identifies a root element (i.e. a top level element) in the messaging scheme.

Attributes

Name	Description	Default	Required
name	The name of the root element	-	Yes
RoutingValue	A value that the RoutingAttribute may take. When the RoutingAttribute takes this value the message will be routed to the service.	The name of the root element	No

Content

The RootElement element has no content.

Example Config File

```
<!—- This config file causes the generated source to be retained in
the "src" subdirectory of the current directory. The source is
compiled and the generated classes are retained in "output.jar". A
service stub is generated. -->

<Config

srcDir="src"

force="true"

outputJar="output.jar"

createService="true">

<-- Load the DTD "test.dtd" -->

<DTDFile file="jaxhit.dtd" publicId="-//classgen//generated"/>

<-- Specify that the class generated for the "TestMessage" element
defined in the DTD should implement the interface
"org.foo.test.Message" -->

<ElementBase name="TestMessage">

<ImplementsInterface name="org.foo.test.Message">

</ElementBase>

<-- Specify that the service stub should be "org.foo.Service". It
accepts only one message type, that being "RootMessage"-->

<ServiceConfig name="MyService" className="org.foo.Service">

<RootElement name="TestMessage"/>

</ServiceConfig>

</Config>
```

The Generated Interface

There is a one to one mapping between elements in the DTD and generated classes. Each generated class implements the TbaseElement interface from the package com.iplanet.trustbase.xml.message, and those classes mapping to elements that have an ID attribute implement TbaseIdentifiedElement, from the same package.

In addition to the methods specified in the TbaseElement interface, the following methods are generated.

Attributes

Each attribute defined in the ATTLIST declaration for the element will have a get method and a set method generated. The method signatures will be

public String getAttr();

public void setAttr(String attr);

where "attr" is replaced by the name of the attribute. Any characters in the attribute name that are not a legal Java identifier part will be replaced by the underscore character '_'.

If the attribute is mapped to a Java type with the AttributeType configuration options, then the generated method signatures will reflect the type specified.

Content

For the purposes of generation, the content model of an element is considered to consist of two types of content. Singly addressable elements are those that can not be specified more than once. Elements that can be specified more than once are considered to be part of a group.

For example, the content model (A, B, C*) consists of the singly addressable elements A and B, and the group containing the element C.

For each singly addressable element, a get / set pair is produced:

```
public A getA();

public void setA(A a);
```

For each member of a group, the following methods are produced:

```
public void addC(C c);

public void removeC(C c);

public C[] getC();
```

Other methods

- Each generated class has a convenience construction method that takes input from a stream and returns an instance of the class if one can be constructed from the stream.
  public static <type> fromXML(Reader reader, boolean validate) throws SAXException,
  where <type> is the type of the class in which the method is defined, reader is a reader for the XML stream and validate is whether to validate during parse.
- **public static <type> fromXML(Reader reader) throws SAXException**
  This is the same as above, except that it does validation by default
- A copy method is provided that does a deep copy of the hierarchy.
  public TbaseElement copy();

Constructors

The generated classes have up to 4 constructors.

The default constructor

A constructor taking all the attributes of the element (excluding **fixed** attributes), if there are any

A constructor taking all the attributes and all the content of the element. Whether this constructor is generated depends on whether there are any groups (* or +) in the content model. If noe (i.e. all the elements in the content are directly addressible), this constructor will be generated

The constructor used during the SAX parse process. This should not be used in user code.

Using the Generated Classes

Constructing instances of generated classes from XML

1. **If you have a complete document (complete with DOCTYPE declaration)**
   Use com.iplanet.trustbase.xml.message.MessageBuilder.buildMessage(), passing in a Reader for the XML. This will return an instance of TbaseElement which you can cast to the correct type.

2. **If you know in advance the type of the element you wish to read**
   Use the static fromXML() method provided on the generated class you wish to construct. This will return an instance of the generated class

3. **If you do not know the type of the element in advance, and the XML does not have a DOCTYPE header**
   Get an instance of MessageBuilder with one of the static getInstance() methods.
   Set the public Id by calling resolveEntity(publicId, null).
   Call the parse() method with a Reader for the XML. This will return an instance of a generated class which you can cast to the correct type.

4. **If you have no idea what you're trying to read (but it is well formed XML)**
   Use com.iplanet.trustbase.xml.message.TbaseAnyElement.fromXML(), passing in a Reader. This will return an instance of TbaseAnyElement. The structure created by this method will not be typed in any way, by will contain all the data present in the document. Structures created in this manner can be reparsed into generated classes with the reParse() method, which takes the public Id of the DTD in which the element was defined. This method returns a TbaseElement which can be cast appropriately.

Outputting XML from generated classes

Call the toXMLString() method on any generated class. This will return a string representation of the XML structure. Note that the resulting XML will not have a DOCTYPE declaration. If you want to use the XML string as an entire document, you should insert the DOCTYPE declaration with the following format:

```
<DOCTYPE elementName PUBLIC "publicId" "systemId">
```

Where elementName is the name of the root element in the message, publicId is the public Id of the DTD (which can be obtained from the generated class using the publicId() method) and systemId is a URI that locates the DTD. The system Id need only be a valid identifier if the message is to be read by a non-Trustbase system.

FAQ

1. **How do I keep the generated source code?**
   Specify the srcDir Attribute in the configuration file. All the generated source code will be retained in the specified directory

2. **Where is the source for my stub service?**
   In the directory specified by srcDir. If you did not specify srcDir, then you will not have a stub service class.

3. **Why is the stub service class not compiled along with the other source files?**
   The stub service is not useful by itself – you need to provide a meaningful implementation!

4. **Why do I need to supply a public ID?**
   The public ID is used to generate a package name for the generated code. This is useful because it allows common elements to be represented by a single codebase.

5. **What does JAXHIT actually stand for?**
   Java Architecture for XML Handling in Trustbase

6. **I want all but a few of the generated classes to implement a certain interface. Is there a way to override the settings in DefaultElementBase for these elements?**
   Yes. For each element that should not inherit the settings in DefaultElementBase, create an ElementBase with the "inherit" attribute set to "false". Only the settings specified explicitly will be applied to the generated class.

7. **The compilation step fails. Why?**
   There are a number of possible reasons for this. Most likely is a problem with the base interfaces and / or classes that you specified in the config file. If you are relying on the generated interface matching a certain pattern, make sure that this is in fact the case. Also, certain element or attribute names can cause conflicts with Object. For example, if you had an attribute called Class, the accessor method would be getClass() - clearly this will fail. This problem can be solved by using the methodPrefix attribute in the ElementBase config for the affected element, such that the generated method name becomes, for example, xGetClass().

8. **Attributes with enumerated values are supposed to have type-safe get / set methods. Why doesn't this appear to work?**
   Because the default parser used with JAXP (Crimson) has a bug. Use the Xerces parser instead, using the following command line:

```
java
-Djavax.xml.parsers.SAXParserFactory=org.apache.xerces.jaxp.SAXPars
erFactoryImpl -jar jaxhit.jar <options>
```

and make sure that the Xerces jar is in the same place as jaxhit.jar.