Event Notification Service Manual

Sun™ ONE Messaging and Collaboration

Sun ONE Calendar Server 5.1.1; iPlanet™ Messaging Server 5.2

August 2002
816-6418-10

Contents

About This Manual

This manual describes the Event Notification Service (ENS) architecture and APIs for iPlanet™ Messaging Server and Sun™ ONE Calendar Server. It gives detailed instructions on the ENS APIs that you can use to customize your server installation. Note that Sun ONE is the brand formerly known as iPlanet. Both Sun ONE and iPlanet are brands of Sun Microsystems, Inc. During a transition period both brands appear in this manual.

This preface contains the following sections:

- Who Should Read This Manual
- What You Need to Know
- How This Manual is Organized
- Conventions Used in This Manual
- Where to Find Related Information

Who Should Read This Manual

This manual is for programmers who want to customize applications in order to implement iPlanet Messaging Server and Sun ONE Calendar Server.

What You Need to Know

This book assumes that you are a programmer with a knowledge of C/C++ and Java Messaging Service, and that you have a general understanding of the following:

- The Internet and the World Wide Web
- Messaging and calendaring concepts

How This Manual is Organized

This manual contains the following chapters and appendix:

- About This Manual (this chapter)
- Chapter 1, “Introduction to Event Notification Service”

  This chapter describes the Event Notification Service (ENS) components, architecture, and Application Programming Interfaces (APIs).
- Chapter 2, “Event Notification Service C API Reference”

  This chapter describes the ENS C API.
- Chapter 3, “Event Notification Service Java (JMS) API Reference”

  This chapter describes the ENS Java API and provides sample code.
- Chapter 4, “Sun ONE Calendar Server Specific Information”

  This chapter describes the Sun ONE Calendar Server event notifications and provides sample Sun ONE Calendar Server code.
- Chapter 5, “iPlanet Messaging Server Specific Information”

  This chapter describes the iPlanet Messaging Server event references and provides sample iPlanet Messaging Server code.
- “Glossary”

Conventions Used in This Manual

`Monospaced font` - This typeface is used for any text that appears on the computer screen. It is also used for filenames, distinguished names, functions, and examples.

Italicized font - This is used to represent text that you enter using information that is unique to your installation (for example, variables). It is used for server paths and names and account IDs.

All paths specified in this manual are in UNIX format. If you are using a Windows NT-based iPlanet Messaging Server or Sun ONE Calendar Server, you should assume the Windows NT equivalent file paths whenever UNIX file paths are shown in this book.

Where to Find Related Information

In addition to this manual, other documentation can be found at the following locations:

- iPlanet Messaging Server Documentation

  `http://docs.sun.com/prod/s1msgsrv`

- Sun ONE Calendar Server Documentation

  `http://docs.sun.com/prod/s1.s1cals`

Chapter 1

Introduction to Event Notification Service

This chapter provides an overview of the Event Notification Service (ENS) components, architecture, and Application Programming Interfaces (APIs).

This chapter contains these sections:

- Event Notification Service Overview
- Event Notification Service Architecture
- Event Notification Service API Overview

Event Notification Service Overview

The Event Notification Service (ENS) is Sun ONE's underlying publish-and-subscribe service available in the following Sun ONE products:

- iPlanet Calendar Server, Release 5.0 and 5.1, and Sun ONE Calendar Server 5.1.1
- iPlanet Messaging Server, Release 5.1 and later (integrated but not enabled)

NOTE See Appendix C in the *iPlanet Messaging Server 5.2 Administrator's Guide* for instructions on enabling and administering ENS in iPlanet Messaging Server.

ENS acts as a dispatcher used by Sun ONE applications as a central point of collection for certain types of *events* that are of interest to them. Events are changes to the value of one or more properties of a resource. In this structure, a URI (Uniform Resource Identifier) represents an event. Any application that wants to know when these types of events occur registers with ENS, which identifies events in order and matches notifications with subscriptions. Event examples include:

- Arrival of new mail to a user's inbox
- User's mailbox has exceeded its quota
- Calendar reminders

Specifically, ENS accepts reports of events that can be categorized, and notifies other applications that have registered an interest in certain categories of events.

Event Notification Service provides a server and APIs for publishers and subscribers. A publisher makes an event available to the notification service; and a subscriber tells the notification service that it wants to receive notifications of a specific event. See "Event Notification Service API Overview," on page 24 for more information on the ENS APIs.

ENS in Sun ONE Calendar Server

By default, ENS is enabled in Sun ONE Calendar Server. For Sun ONE Calendar Server you do not need to do anything else to use ENS.

A user who wants to subscribe to notifications other than the alarms generated by Sun ONE Calendar Server needs to write a subscriber.

There is sample ENS C publisher and subscriber code bundled with Sun ONE Calendar Server. See "Sun ONE Calendar Server Sample Code," on page 64 for that code.

Sample Sun ONE Calendar Server code is provided with the product in the following directory:

```
/opt/SUNWics5/cal/csapi/samples/ens
```

ENS in iPlanet Messaging Server

ENS and iBiff (the ENS publisher for iPlanet Messaging Server, also referred to as the notification plug-in to iPlanet Messaging Server) are bundled in iPlanet Messaging Server. However, by default, they are not enabled.

To subscribe to notifications in iPlanet Messaging Server, you need to first perform the following two items on the iPlanet Messaging Server host:

- Load the iBiff notification plug-in
- Stop and restart the messaging server

See Appendix C in the *iPlanet Messaging Server 5.2 Administrator's Guide* for the instructions to enable ENS on iPlanet Messaging Server.

A user who wants to subscribe to iPlanet Messaging Server notifications needs to write a subscriber to the ENS API. To do so, the subscriber needs to know what the various iPlanet Messaging Server notifications are. See Chapter 5, "iPlanet Messaging Server Specific Information" for that information.

iPlanet Messaging Server comes bundled with sample ENS C publisher and subscriber code. See "iPlanet Messaging Server Sample Code," on page 83 for more information.

Sample iPlanet Messaging Server code is provided with the product in the following directory:

server-root`/bin/msg/enssdk/examples`

Event References

Event references identify an event handled by ENS. Event references use the following URI syntax (as specified by RFC 2396):

scheme`://`*authority resource*`/[?`*param1*`=`*value*`1&`*param2*`=`*value2*`&`*param3*`=`*value3*`]`

where:

- *scheme* is the access method, such as `http`, `imap`, `ftp`, or `wcap`.

  For Sun ONE Calendar Server and iPlanet Messaging Server, the ENS scheme is `enp`.

- *authority* is the DNS domain or hostname that controls access to the resource.
- *resource* is the path leading to the resource in the context of the authority. It can be composed of several path components separated by a slash ("`/`").
- *param* is the name of a parameter describing the state of a resource.
- *value* is its value. There can be zero or more parameter/value pairs.

In general, all Sun ONE Calendar Server events start with the following:

```
enp:///ics
```

The iPlanet Messaging Server notification plug-in iBiff uses the following scheme and resource by default:

```
enp://127.0.0.1/store
```

NOTE Although the event reference has a URI syntax, the scheme, authority, and resource have no special significance. They are merely used as strings with no further interpretation in ENS.

Sun ONE Calendar Server Event Reference Example

The following is an example event reference URI to subscribe to all event alarms with a calendar ID of `jac`:

```
enp:///ics/alarm?calid=jac
```

NOTE This is not meant to be used by end users.

iPlanet Messaging Server Event Reference Example

The following is an example event reference that requests a subscription to all `NewMsg` events for a user whose user ID is `blim`:

```
enp://127.0.0.1/store?evtType=NewMsg&mailboxName=blim
```

When using ENS with iPlanet Messaging Server, the user id you specify is case sensitive.

NOTE This is not meant to be used by end users.

ENS Connection Pooling

The connection pooling feature of ENS enables a pool of subscribers to receive notifications from a single event reference. For every event, ENS chooses one subscriber from the pool to send the notification to. Thus, only one subscriber in the pool receives the notification. The ENS server balances sending of notifications among the subscribers. This enables the client to have a pool of subscribers that work together to receive all notifications from a single event reference.

For example, if notifications are being published to the event reference `enp://127.0.0.1/store`, a subscriber will normally subscribe to this event reference to receive notifications. To have a pool of subscribers receive all the notifications to this event reference, each subscriber in the pool only needs to subscribe to the event reference `enp+pool://127.0.0.1/store` instead. The ENS server chooses one subscriber from the pool to send the notification to.

NOTE The publisher still sends notifications to the simple event reference, in the example above `enp://127.0.0.1/store`, that is, the publisher has no knowledge of the subscriber pool.

Multiple Pool Extension

Connection pooling can support multiple pools of subscribers. That is, you can have two pools of subscribers, each pool receiving all the notifications from the event reference. The syntax of the event reference for the subscriber is:

`enp+pool[.`*poolid*`]://domain/event`

where *poolid* is a string using only base64 alphabet. (See RFC1521, Table 1, for what the base64 alphabet contains.) So, for example, to have two pools of subscribers to the event reference `enp://127.0.0.1/store`, each pool could subscribe to the following event references:

`enp+pool.1://127.0.0.1/store` --> for first pool of subscribers
`enp+pool.2://127.0.0.1/store` --> for second pool of subscribers

Event Notification Service Architecture

On the Solaris platform, ENS runs as a daemon, `enpd`, along with other daemons in various calendar or messaging server configurations, to collect and dispatch events that occur to properties of resources. On the Windows NT platform, ENS runs as a service, `enpd.exe`.

For ENS, an event is a change that happens to a resource, while a resource is an entity such as a calendar or inbox. For example, adding an entry to a calendar (the resource) generates an event, which is stored by ENS. This event can then be subscribed to, and a notification would then be sent to the subscriber.

The ENS architecture enables the following three things to occur:

- Notification - This is a message that describes an event occurrence. Sent by the event publisher, it contains a reference to the event, as well as any additional parameter/value pairs added to the URI, and optional data (the payload) used by the event consumers, but opaque to the notification service. Whoever is interested in the event can subscribe to it.
- Subscription - This is a message sent to subscribe to an event. It contains an event reference, a client-side request identifier, and optional parameter/value pairs added to the URI. The subscription applies to upcoming events (that is, a subscriber asks to be notified of upcoming events).
- Unsubscription - This message cancels (unsubscribes) an existing subscription. An event subscriber tells ENS to stop relaying notifications for the specified event.

Notify

ENS notifies its subscribers of an event by sending a notification. Notify is also referred to as "publish." A notification can contain the following items:

- An event reference (which, optionally, can contain parameter/value pairs)
- Optional application-specific data ("opaque" for ENS, but the publisher and subscriber agree apriori to the format of the data)

The optional application-specific data is referred to as the "payload."

There are two kinds of notifications:

- **Unreliable notification** - Notification sent from an event publisher to a notification server. If the publisher does not know nor care about whether there are any consumers, or whether they get the notification, this request does not absolutely need to be acknowledged. However, a publisher and a subscriber, who are mutually aware of each other, can agree to set up a reliable event notification link (RENL) between themselves. In this case, once the subscriber has processed the publisher's notification, it sends an acknowledgment notification back to the publisher.
- **Reliable notification** - Notification sent from a server to a subscriber as a result of a subscription. This type of notification should be acknowledged. A reliable notification contains the same attributes as an unreliable notification.

See "Publisher API," on page 33 for more information.

Subscribe

ENS receives a request to be notified of events. The request sent by the event subscriber is a subscription. The subscription is valid during the life of the session, or until it is cancelled (unsubscribed).

A subscription can contain the following items:

- An event reference (which, optionally, can contain parameter/value pairs)
- A request identifier

See "Subscriber API," on page 40 for more information.

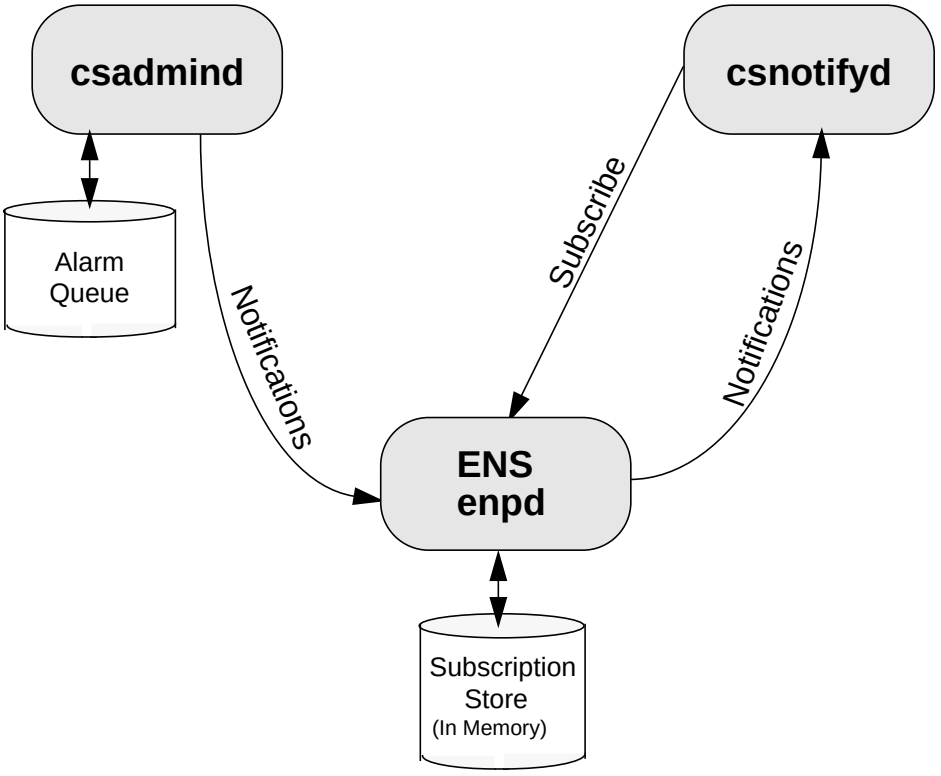
Unsubscribe

ENS receives a request to cancel an existing subscription. See "Subscriber API," on page 40 for more information.

How Sun ONE Calendar Server Interacts with ENS

Figure 1-1 on page 18 shows how ENS interacts with Sun ONE Calendar Server through the alarm queue and two daemons, `csadmind` and `csnotifyd`.

Figure 1-1 ENS in Sun ONE Calendar Server Overview



Sun ONE Calendar Server Alarm Queue

ENS is an alarm dispatcher. This decouples alarm delivery from alarm generation. It also enables the use of multiple delivery methods, such as email and wireless communication. The `csadmind` daemon detects events by sensing changes in the state of the alarm queue. The alarm queue's state changes every time an alarm is placed in the queue. An alarm is queued when a calendar event generates an alarm. The following URIs represent these kind of events:

for events:

`enp:///ics/eventalarm?calid=`*calid*`&uid=`*uid*`&rid=`*rid*`&aid=`*aid*

for todos (tasks):

`enp:///ics/todoalarm?calid=`*calid*`&uid=`*uid*`&rid=`*rid*`&aid=`*aid*

where:

- *calid* is the calendar ID.
- *uid* is the event/todo (task) ID within the calendar.
- *rid* is the recurrence id for a recurring event/todo (task).
- *aid* is the alarm ID within the event/todo (task). In case there are multiple alarms, the *aid* identifies the correct alarm.

The publisher `csadmind` dequeues the alarms and sends notifications to `enpd`. The `enpd` daemon then checks to see if anyone is subscribed to this kind of event and sends notifications to the subscriber, `csnotifyd`, for any subscriptions it finds. Other subscribers to alarm notifications (reminders) can be created and deployed within an Sun ONE Calendar Server installation. These three daemons interacting together implement event notification for Sun ONE Calendar Server.

Sun ONE Calendar Server Daemons

Sun ONE Calendar Server includes two daemons that communicate to the ENS daemon, `enpd`:

- `csadmind`

  The `csadmind` daemon contains a publisher that submits notifications to the notification service by sending alarm events to ENS. It manages the Sun ONE Calendar Server alarm queue. It implements a scheduler, which lets it know when an alarm has to be generated. At such a point, `csadmind` publishes an event. ENS receives and dispatches the event notification.

  To ensure alarm transfer reliability, `csadmind` requires acknowledgment for certain events or event types. (See “Alarm Transfer Reliability,” on page 20.) The `csadmind` daemon uses Reliable Event Notification Links (RENLs) to accomplish acknowledgment.

- `csnotifyd`

  The `csnotifyd` daemon is the subscriber that expresses interest in particular events (subscribes), and receives notifications about these subscribed-to events from ENS, and sends notice of these events and todos (tasks) to its clients by email.

Though the ability to unsubscribe is part of the ENS architecture, `csnotifyd` does not bother to unsubscribe to events for the following two reasons: there is no need to unsubscribe or resubscribe during normal runtime; and due to the temporary nature of the subscriptions store (it is held in memory), all subscriptions are implicitly unsubscribed when the connection to ENS is shutdown.

The `csnotifyd` daemon subscribes to `enp:///ics/alarm/`. The todo (task) or event is specified in a parameter.

Alarm Transfer Reliability

To ensure that no alarm ever gets lost, `csadmind` and `csnotifyd` use the RENL feature of ENS for certain types of alarms. For these alarms, `csadmind` requests an end-to-end acknowledgment for each notification it sends, while `csnotifyd`, after successfully processing it, generates a notification acknowledgment for each RENL alarm notifications it receives.

For these RENL alarms, should the network, the ENS daemon, or `csnotifyd` fail to handle a notification, `csadmind` will not receive any acknowledgment, and will not remove the alarm from the alarm queue. The alarm will, therefore, be published again after a timeout.

Sun ONE Calendar Server Example

A typical ENS publish and subscribe cycle for Sun ONE Calendar Server resembles the following:

1. The event subscriber, `csnotifyd`, expresses interest in an event (subscribes).
2. The event publisher, `csadmind`, detects events and sends notification (publishes).
3. ENS publishes the event to the subscriber.
4. The event subscriber cancels interest in the event (unsubscribes). This step happens implicitly when the connection to ENS is shutdown.

Figure 1-2 on page 21 illustrates this cycle and Table 1-1 on page 21 provides the narrative for the figure.

Figure 1-2 Example Event Notification Service Publish and Subscribe Cycle for Sun ONE Calendar Server

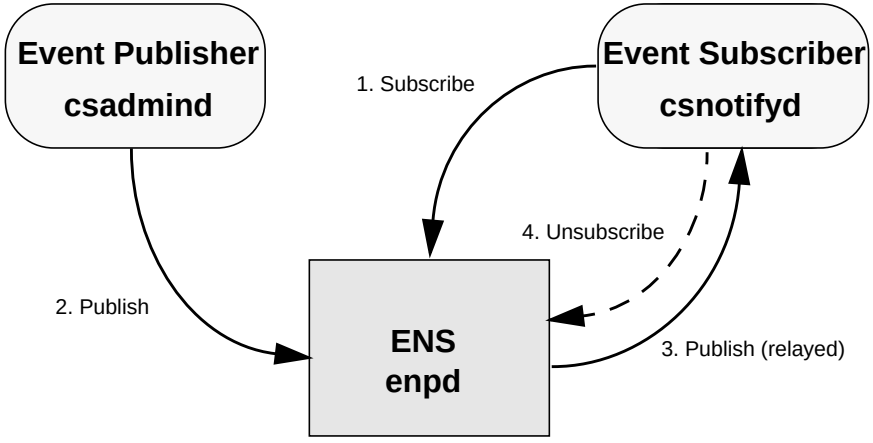


Table 1-1 Example Event Notification Service Publish and Subscribe Cycle

Action	ENS Response
1. The `csnotifyd` daemon sends a subscription request to ENS.	ENS stores the subscription in the subscriptions database.
2. The `csadmind` daemon sends a notification request to ENS.	ENS queries the subscriptions database for subscriptions matching the notification.
3. The `csnotifyd` daemon receives a notification from ENS.	When ENS receives a notification from a publisher, it looks up its internal subscription table to find subscriptions matching the event reference of the notification. Then for each subscription, it relays a copy of the notification to the subscriber who owns this subscription.
4. Currently, `csnotifyd` does not bother sending cancellation requests to ENS.	Because the subscriptions store is in memory only (not in a database), all subscriptions are implicitly unsubscribed when the connection to ENS is shutdown.

How iPlanet Messaging Server Interacts with ENS

Figure 1-3 on page 23 shows how ENS interacts with iPlanet Messaging Server. In this figure, each oval represents a process, and each rectangle represents a host computer running the enclosed processes.

The ENS server delivers notifications from the iPlanet Messaging Server notification plug-in to ENS clients (that is, iBiff subscribers). There is no guarantee of the order of notification prior to the ENS server because the events are coming from different processes (`MTA`, `stored`, and `imapd`).

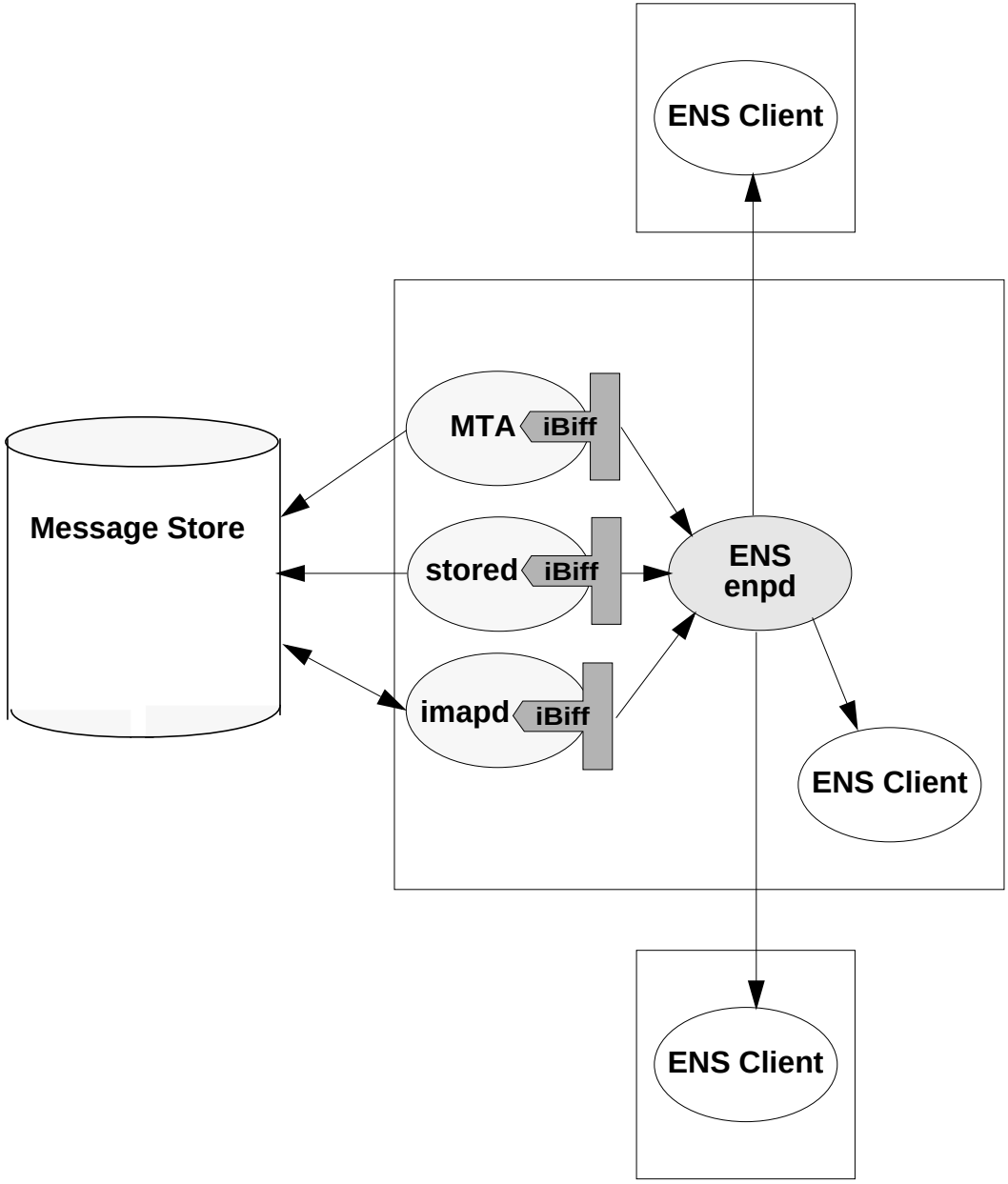
Notifications flow from the iBiff plug-in in the `MTA`, `stored`, and `imap` processes to ENS `enpd`. The ENS client subscribes to the ENS, and receives notifications. When iBiff is enabled, iPlanet Messaging Server publishes the notifications with the iBiff plug-in, but no iPlanet Messaging Server services subscribe to these notifications. A customer-provided ENS subscriber or client should be written to consume the notifications and do whatever is necessary. That is, iPlanet Messaging Server itself does not depend on or use the notifications for its functions, and this is why ENS and iBiff are not enabled by default when you install iPlanet Messaging Server.

The iPlanet Messaging Server architecture enforces that a given set of mailboxes is served by a given host computer. A given mailbox is not served by multiple host computers. There are several processes manipulating a given mailbox but only one computer host serving a given mailbox. Thus, to receive notifications, end-users only need to subscribe to the ENS daemon that serves the mailbox they are interested in.

iPlanet Messaging Server enables you to have either one ENS server for all mailboxes—that is, one ENS server for all the computer hosts servicing the message store—or multiple ENS servers, perhaps one ENS server per computer host. The second scenario is more scalable. Also, in this scenario, end users must subscribe to multiple ENS servers to get the events for mailboxes they are interested in.

Thus, the architecture requires an ENS server per computer host. The ENS servers and the client processes do not have to be co-located with each other or with messaging servers.

Figure 1-3 ENS in iPlanet Messaging Server Overview



Event Notification Service API Overview

This section provides an overview of the two APIs for ENS, a C API and a Java API, which is a subset of the Java Messaging Service (JMS) API. Starting with iPlanet Messaging Server 5.2 and Sun ONE Calendar Server 5.1, a Java API to ENS has been added. The Java API conforms to the Java Message Service specification (JMS). Two sample Java subscribers are provided using the JMS API.

For detailed information on the ENS C API, see Chapter 2, "Event Notification Service C API Reference." For detailed information on the Java (JMS) API, see Chapter 3, "Event Notification Service Java (JMS) API Reference." For JMS documentation, use the following URL:

```
http://java.sun.com/products/jms/docs.html
```

ENS C API Overview

ENS implements the following three APIs:

- Publisher API

  A publisher sends notification of a subscribed-to event to ENS, which then distributes it to the subscribers. Optionally, in Sun ONE Calendar Server, the application can request acknowledgment of receipt of the notification. To do this, a Reliable Event Notification Link (RENL) is necessary. An RENL has a publisher, a subscriber, and a unique ID, which identify notifications that are subject to acknowledgment. The publisher informs the application of the receipt of an acknowledgment by invoking the `end2end_ack` callback passed to `publish_a`. Currently, only Sun ONE Calendar Server supports RENL.

- Subscriber API

  A subscriber is a client to the notification service which expresses interest in particular events. When the notification service receives a notification about one of these events from a publisher, it relays the notification to the subscriber.

  A subscriber may also unsubscribe, which cancels an active subscription.

  In Sun ONE Calendar Server, to enable an RENL, the subscriber declares its existence to ENS, which then transparently generates notification acknowledgment on behalf of the subscriber application. The subscriber can revoke the RENL at any time.

- Publish and Subscribe Dispatcher API

  When an asynchronous publisher is used, ENS needs to borrow threads from a thread pool in order to invoke callbacks. The application can either choose to create its own thread pool and pass it to ENS, or it can let ENS create and manage its own thread pool. In either case, ENS creates and uses a dispatcher object to instantiate the dispatcher used (`pas_dispatcher_t`).

  GDisp (`libasync`) is the dispatcher supported.

ENS Java API Overview

The Java API for ENS uses a subset of the standard JMS API, with the addition of two new proprietary methods:

- `com.iplanet.ens.jms.EnsTopicConnFactory`
- `com.iplanet.ens.jms.EnsTopic`

The following list of JMS object classes is used in the Java API for ENS:

- `javax.jms.TopicSubscriber`
- `javax.jms.TopicSession`
- `javax.jms.TopicPublisher`
- `javax.jms.TopicConnection`
- `javax.jms.TextMessage`
- `javax.jms.Session`
- `javax.jms.MessageProducer`
- `javax.jms.MessageConsumer`
- `javax.jms.Message`
- `javax.jms.ConnectionMetaData`
- `javax.jms.Connection`

NOTE The Java API for ENS does not implement all the JMS object classes. When customizing, use only the object classes found on this list.

Building and Running Custom Applications

To assist you in building your own custom publisher and subscriber applications, iPlanet Messaging Server and Sun ONE Calendar Server include sample code. This section tells you where to find the sample code, where the APIs' include (header) files are located, and where the libraries are that you need to build and run your custom programs.

NOTE This section applies to the C API only.

Location of Sample Code

Sun ONE Calendar Server

Sun ONE Calendar Server includes four simple sample programs to help you get started. The code for these samples resides in the following directory:

```
/opt/SUNWics5/cal/csapi/samples/ens
```

iPlanet Messaging Server

iPlanet Messaging Server 5.1 and higher contains sample programs to help you learn how to receive notifications. These sample programs are located in the *server-root*`/bin/msg/enssdk/examples` directory.

Location of Include Files

Sun ONE Calendar Server

The include (header) files for the publisher and subscriber APIs are: `publisher.h`, `suscriber.h`, and `pasdisp.h` (publish and subscribe dispatcher). They are located in the CSAPI `include` directory. The default `include` path is:

```
/opt/SUNWics5/cal/csapi/include
```

iPlanet Messaging Server

The default `include` path for iPlanet Messaging Server is:

server-root`/bin/msg/enssdk/include`

Dynamically Linked/Shared Libraries

Sun ONE Calendar Server

Your custom code must be linked with the dynamically linked library `libens`, which implements the publisher and subscriber APIs. On some platforms all the dependencies of `libens` must be provided as part of the link directive. These dependencies, in order, are:

1. `libgap`
2. `libcyrus`
3. `libyasr`
4. `libasync`
5. `libnspr3`
6. `libplsd4`
7. `libplc3`

Sun ONE Calendar Server uses these libraries; therefore, they are located in the server's `bin` directory. The default `libens` path is:

```
/opt/SUNWics5/cal/bin
```

NOTE For NT, in order to build publisher and subscriber applications, you also need the archive files (`.lib` files) corresponding to all the earlier mentioned libraries. These are located in the CSAPI library directory, `lib`. The default `lib` path is:

drive`:\Program Files\iPlanet\cal\csapi\lib`

iPlanet Messaging Server

The libraries for iPlanet Messaging Server are located in the following directory:

server-root`/bin/msg/lib`

Refer to *server-root*`/bin/msg/enssdk/examples/Makefile.sample` to help determine what libraries are needed.This makefile contains instructions on how to compile and run the `apub` and `asub` programs. This file also describes what libraries are needed, and what the LD_LIBRARY_PATH should be.

Figure 1-4 `Makefile.sample` File

```
#
# Sample makefile
#
# your C compiler
CC = gcc

# LIBS
# Your library path should include <server-root>/bin/msg/lib
LIBS = -lens -lgap -lxenp -lcyrus -lchartable -lyasr -lasync

all: apub asub

apub: apub.c
        $(CC) -o apub -I ../include apub.c $(LIBS)

asub: asub.c
        $(CC) -o asub -I ../include asub.c $(LIBS)

run:
        @echo 'run <server-root>/msg-<instance>/start-ens'
        @echo run asub localhost 7997
        @echo run apub localhost 7997
```

NOTE The Windows NT distribution includes the following additional files:

server-root\bin\msg\enssdk\examples

bin\msg\enssdk\examples\libens.lib

bin\msg\enssdk\examples\libgap.lib

bin\msg\enssdk\examples\libxenp.lib

bin\msg\enssdk\examples\libcyrus.lib

bin/msg\enssdk\examples\libchartable.lib

bin\msg\enssdk\examples\libyasr.lib

bin\msg\enssdk\examples\libasync.lib

bin\msg\enssdk\examples\asub.dsw

bin\msg\enssdk\examples\apub.dsp

bin\msg\enssdk\examples\asub.dsp

To build on Windows NT:

1. A sample VC++ workspace is provided in `asub.dsw`. It has two projects in it: `asub.dsp` and `apub.dsp`.

   The required `.lib` files to link is in the same directory as `asub.c` and `apub.c`.

2. To run, it requires that the following DLLs are in your path.

   ```
   libens.dll
   libgap.dll
   libxenp.dll
   libcyrus.dll
   libchartable.dll
   libyasr.dll
   libasync.dll
   ```

   The simplest way to accomplish this is to include *server-root*`\msg\lib` in your `PATH`.

Runtime Library Path Variable

Sun ONE Calendar Server

In order for your custom programs to find the necessary runtime libraries, which are located in the `/opt/SUNWics5/cal/bin` directory, make sure your environment's runtime library path variable includes this directory. The name of the variable is platform dependent:

- SunOS and Linux: `LD_LIBRARY_PATH`
- NT: `PATH`
- HPUX: `SHLIB_PATH`

iPlanet Messaging Server

For iPlanet Messaging Server, you need to set your `LD_LIBRARY_PATH` to *`server-root`*`/bin/msg/lib`.

Chapter 2

Event Notification Service C API Reference

This chapter details the ENS C API; it is divided into three main sections:

- Publisher API
- Subscriber API
- Publish and Subscribe Dispatcher API

Publisher API Functions List

This chapter includes a description of the following Publisher functions, listed in Table 2-1:

Table 2-1 ENS Publisher API Functions List

Definition/Function	Description
`publisher_t`	Definition for a publisher.
`publisher_cb_t`	Generic callback function acknowledging an asynchronous call.
`publisher_new_a`	Creates a new asynchronous publisher.
`publisher_new_s`	Creates a new synchronous publisher.
`publish_a`	Sends an asynchronous notification to the notification service.
`publish_s`	Sends a synchronous notification to the notification service.
`publisher_delete`	Terminates a publish session.
`publisher_get_subscriber`	Creates a subscriber using the publisher's credentials.
`renl_create_publisher`	Creates an RENL, which enables the invocation of `end2end_ack`.

Table 2-1 ENS Publisher API Functions List *(Continued)*

`renl_cancel_publisher`	Cancels an RENL.

Subscriber API Functions List

This chapter includes a description of following Subscriber functions, listed in Table 2-2:

Table 2-2 ENS Subscriber API Functions List

Definition/Function	Description
`subscriber_t`	Definition of a subscriber.
`subscription_t`	Definition of a subscription.
`subscriber_cb_t`	Generic callback function acknowledging an asynchronous call.
`subscriber_notify_cb_t`	Synchronous callback; called upon receipt of a notification.
`subscriber_new_a`	Creates a new asynchronous subscriber.
`subscriber_new_s`	Creates a new synchronous subscriber.
`subscribe_a`	Establishes an asynchronous subscription.
`unsubscribe_a`	Cancels an asynchronous subscription.
`subscriber_delete`	Terminates a subscriber.
`subscriber_get_publisher`	Creates a publisher using the subscriber's credentials.
`renl_create_subscriber`	Creates the subscription part of the RENL.
`renl_cancel_subscriber`	Cancels an RENL.

Publish and Subscribe Dispatcher Functions List

This chapter includes a description of the following Publish and Subscribe Dispatcher functions, listed in Table 2-3:

Table 2-3 ENS Publish and Subscribe Dispatcher Functions List

Definition/Function	Description
`pas_dispatcher_t`	Definition of a publish and subscribe dispatcher.

Table 2-3 ENS Publish and Subscribe Dispatcher Functions List *(Continued)*

`pas_dispatcher_new`	Creates a dispatcher.
`pas_dispatcher_delete`	Destroys a dispatcher created with `pas_dispatcher_new`.
`pas_dispatch`	Starts the dispatch loop of an event notification environment.
`pas_shutdown`	Stops the dispatch loop on an event notification environment started with `pas_dispatch`.

Publisher API

The Publisher API consists of one definition and nine functions:

- `publisher_t`
- `publisher_cb_t`
- `publisher_new_a`
- `publisher_new_s`
- `publish_a`
- `publish_s`
- `publisher_delete`
- `publisher_get_subscriber`
- `renl_create_publisher`
- `renl_cancel_publisher`

publisher_t

Purpose.
A publisher.

Syntax
```
typedef struct enc_struct publisher_t;
```

Parameters
None.

Returns
Nothing.

publisher_cb_t

Purpose.
Generic callback function invoked by ENS to acknowledge an asynchronous call.

Syntax

```
typedef void (*publisher_cb_t) (void *arg, int rc, void *data);
```

Parameters

`arg`	Context variable passed by the caller.
`rc`	The return code.
`data`	For an open, contains a newly created context.

Returns
Nothing.

publisher_new_a

Purpose
Creates a new asynchronous publisher.

Syntax

```
void publisher_new_a (pas_dispatcher_t *disp,
                      void *worker,
                      const char *host,
                      unsigned short port,
                      publisher_cb_t cbdone,
                      void *cbarg);
```

Parameters

`disp`	P&S thread pool context returned by `pas_dispatcher_new`.
`worker`	Application worker. If not `NULL`, grouped with existing workers created by ENS to service this publisher session. Used to prevent multiple threads from accessing the publisher data at the same time.
`host`	Notification server host name.
`port`	Notification server port.

`cbdone`	The callback invoked when the publisher has been successfully created, or could not be created. There are three Parameters to `cbdone`: • `cbarg` The first argument. • A status code. If non-zero, the publisher could not be created; value specifies cause of the failure. • The new active publisher.
`cbarg`	First argument of `cbdone`.

Returns

Nothing. It passes the new active publisher as third argument of `cbdone` callback.

publisher_new_s

Purpose

Creates a new synchronous publisher.

Syntax

```
publisher_t *publisher_new_s (pas_dispatcher_t *disp,
                              void *worker,
                              const char *host,
                              unsigned short port);
```

Parameters

`disp`	P&S thread pool context returned by `pas_dispatcher_new.`
`worker`	Application worker. If not `NULL`, grouped with existing workers created by ENS to service this publisher session. Used to prevent multiple threads from accessing the publisher data at the same time.
`host`	Notification server host name.
`port`	Notification server port.

Returns

A new active publisher (`publisher_t`).

publish_a

Purpose

Sends an asynchronous notification to the notification service.

Syntax

```
void publish_a (publisher_t *publisher,
                const char *event_ref,
                const char *data,
                unsigned int datalen,
                publisher_cb_t cbdone,
                publisher_cb_t end2end_ack,
                void *cbarg,
                unsigned long timeout);
```

Parameters

`publisher_t`	The active publisher.
`event_ref`	The event reference. This is a URI identifying the modified resource.
`data`	The event data. The body of the notification message. It is opaque to the notification service, which merely relays it to the events' subscriber.
`datalen`	The length in bytes of the data.
`cbdone`	The callback invoked when the data has been accepted or deemed unacceptable by the notification service. What makes a notification acceptable depends on the protocol used. The protocol may choose to use the transport acknowledgment (TCP) or use its own acknowledgment response mechanism.
`end2end_ack`	The callback function invoked after acknowledgment from the consumer peer (in an RENL) has been received. Used only in the context of an RENL.
`cbarg`	The first argument of `cbdone` or `end2end_ack` when invoked.
`timeout`	The length of time to wait for an RENL to complete.

Returns

Nothing.

publish_s

Purpose

Sends a synchronous notification to the notification service.

Syntax

```
int publish_s (publisher_t *publisher,
               const char *event_ref,
               const char *data,
               unsigned int datalen);
```

Parameters

`publisher`	The active publisher.
`event_ref`	The event reference. This is a URI identifying the modified resource.
`data`	The event data. The body of the notification message. It is opaque to the notification service, which relays it to the events' subscriber.
`datalen`	The length in bytes of the data.

Returns

Zero if successful; a failure code if unsuccessful. If an RENL, the call does not return until the consumer has completely processed the notification and has successfully acknowledged it.

publisher_delete

Purpose

Terminates a publish session.

Syntax

```
void publisher_delete (publisher_t *publisher);
```

Parameters

`publisher`	The publisher to delete.

Returns

Nothing.

publisher_get_subscriber

Purpose

Creates a subscriber using the credentials of the publisher.

Syntax

```
struct subscriber_struct * publisher_get_subscriber(publisher_t
*publisher);
```

Parameters

publisher	The publisher whose credentials are used to create the subscriber.

Returns

The subscriber, or `NULL` if the creation failed. If the creation failed, use the subscriber_new to create the subscriber.

renl_create_publisher

Purpose

Declares an RENL, which enables the `end2end_ack` invocation. After this call returns, the `end2end_ack` argument is invoked when an acknowledgment notification matching the specified publisher and subscriber is received.

Syntax

```
void renl_create_publisher (publisher_t *publisher,
                            const char *renl_id,
                            const char *subscriber,
                            publisher_cb_t cbdone,
                            void *cbarg);
```

Parameters

`publisher`	The active publisher.
`renl_id`	The unique RENL identifier. This allows two peers to be able to set up multiple RENLs between them.
`subscriber`	The authenticated identity of the peer.
`cbdone`	The callback invoked when the RENL is established.
`cbarg`	The first argument of cbdone, when invoked.

Returns
Nothing.

renl_cancel_publisher

Purpose
This cancels an RENL. This does not prevent more notifications being sent, but should a client acknowledgment be received, the `end2end_ack` argument of publish will no longer be invoked. All RENLs are automatically destroyed when the publisher is deleted. Therefore, this function does not need to be called to free RENL-related memory before deleting a publisher.

Syntax
```
void renl_cancel_publisher (renl_t *renl);
```

Parameters

`renl`	The RENL to cancel.

Returns
Nothing.

Subscriber API

The Subscriber API includes two definitions and ten functions:

- `subscriber_t`
- `subscription_t`
- `subscriber_cb_t`
- `subscriber_notify_cb_t`
- `subscriber_new_a`
- `subscriber_new_s`
- `subscribe_a`
- `unsubscribe_a`
- `subscriber_delete`

- `subscriber_get_publisher`
- `renl_create_subscriber`
- `renl_cancel_subscriber`

subscriber_t

Purpose
A subscriber.

Syntax
```
typedef struct enc_struct subscriber_t;
```

Parameters
None.

Returns
Nothing.

subscription_t

Purpose
A subscription.

Syntax
```
typedef struct subscription_struct subscription_t;
```

Parameters
None.

Returns
Nothing.

subscriber_cb_t

Purpose
Generic callback function invoked by ENS to acknowledge an asynchronous call.

Syntax

```
typedef void (*subscriber_cb_t) (void *arg,
                                 int rc,
                                 void *data);
```

Parameters

`arg`	Context variable passed by the caller.
`rc`	The return code.
`data`	For an open, contains a newly created context.

Returns

Nothing

subscriber_notify_cb_t

Purpose

Subscriber callback; called upon receipt of a notification.

Syntax

```
typedef void (*subscriber_notify_cb_t) (void *arg,
                                        char *event,
                                        char *data,
                                        int datalen);
```

Parameters

`arg`	Context pointer passed to subscribe (`notify_arg`).
`event`	The event reference (URI). The notification event reference matches the subscription, but may contain additional information called event attributes, such as a `uid`.
`data`	The body of the notification. A MIME object.
`datalen`	Length of the data.

Returns

Zero if successful, non-zero otherwise.

subscriber_new_a

Purpose

Creates a new asynchronous subscriber.

Syntax

```
void subscriber_new_a (pas_dispatcher_t *disp,
                       void *worker,
                       const char *host,
                       unsigned short port,
                       subscriber_cb_t cbdone,
                       void *cbarg);
```

Parameters

`disp`	Thread dispatcher context returned by `pas_dispatcher_new`.
`worker`	Application worker. If not NULL, grouped with existing workers created by ENS to service this subscriber session. Used to prevent multiple threads from accessing the subscriber data at the same time. Only usable if the caller creates and dispatches the GDisp context.
`host`	Notification server host name or IP address.
`port`	Subscription service port number.
`cbdone`	The callback invoked when the subscriber session becomes active and subscriptions can be issued. There are three parameters to `cbdone`: • `cbarg` The first argument. • A status code. If non-zero, the subscriber could not be created; value specifies cause of the failure. • The new active subscriber (`subscriber_t`).
`cbarg`	First argument of `cbdone`.

Returns

Nothing. It passes the new active subscriber as third argument of `cbdone` callback.

subscriber_new_s

Purpose

Creates a new synchronous subscriber.

Syntax

```
subscriber_t *subscriber_new_s (pas_dispatcher_t *disp,
                                const char *host,
                                unsigned short port);
```

Parameters

`disp`	Publish and subscribe dispatcher returned by `pas_dispatcher_new`.
`worker`	Application worker. If not NULL, grouped with existing workers created by ENS to service this publisher session. Used to prevent multiple threads from accessing the publisher data at the same time. Only usable if the caller creates and dispatches the GDisp context.
`host`	Notification server host name or IP address.
`port`	Subscription service port number.

Returns

A new active subscriber (`subscriber_t`).

subscribe_a

Purpose

Establishes an asynchronous subscription.

Syntax

```
void subscribe_a (subscriber_t *subscriber,
                  const char *event_ref,
                  subscriber_notify_cb_t notify_cb,
                  void *notify_arg,
                  subscriber_cb_t cbdone,
                  void *cbarg):
```

Parameters

`subscriber`	The subscriber.

`event_ref`	The event reference. This is a URI identifying the event's source.
`notify_cb`	The callback invoked upon receipt of a notification matching this subscription.
`notify_arg`	The first argument of `notify_arg`. May be called at any time, by any thread, while the subscription is still active.
`cbdone`	Called when an unsubscribe completes. It has three Parameters: • `cbarg` (see below). • Status code. • A pointer to an opaque subscription object.
`cbarg`	The first argument of `cbdone`.

Returns

Nothing.

unsubscribe_a

Purpose

Cancels an asynchronous subscription.

Syntax

```
void unsubscribe_a (subscriber_t *subscriber,
                    subscription_t *subscription,
                    subscriber_cb_t cbdone,
                    void *cbarg);
```

Parameters

`subscriber`	The disappearing subscriber.
`subscription`	The subscription to cancel.
`cbdone`	Called when an unsubscribe completes. It has three parameters: • `cbarg` (see below). • Status code. • A pointer to an opaque subscription object.
`cbarg`	The first argument of `cbdone`.

Returns
Nothing.

subscriber_delete

Purpose
Terminates a subscriber.

Syntax
```
void subscriber_delete (subscriber_t *subscriber);
```

Parameters

`subscriber`	The subscriber to delete.

Returns.
Nothing

subscriber_get_publisher

Purpose
Creates a publisher, using the credentials of the subscriber.

Syntax
```
struct publisher_struct *subscriber_get_publisher (subscriber_t
*subscriber);
```

Parameters

`subscriber`	The subscriber whose credentials are used to create the publisher.

Returns
The publisher, or `NULL` if creation failed. In case the creation fails, use the `publisher_new`.

renl_create_subscriber

Purpose

Creates the subscription part of an RENL.

Syntax

```
renl_t *renl_create_subscriber (subscription_t *subscription,
                                const char *renl_id,
                                const char *publisher);
```

Parameters

subscription	The subscription.
renl_id	The unique RENL identifier. This allows two peers to be able to set up multiple RENLs between them.
publisher	The authenticated identity of the peer.

Returns

The opaque RENL object.

renl_cancel_subscriber

Purpose

This cancels an RENL. It does not cancel a subscription. It tells ENS not to acknowledge any more notifications received for this subscription. It destroys the RENL object, the application may no longer use this RENL. All RENLs are automatically destroyed when the subscription is canceled. Therefore, this function does not need to be called to free RENL-related memory before deleting a subscriber.

Syntax

```
void renl_cancel_subscriber (renl_t *renl);
```

Parameters

`renl`	The RENL to cancel.

Returns

Nothing.

Publish and Subscribe Dispatcher API

The Publish and Subscribe Dispatcher API includes one definition and four functions:

- pas_dispatcher_t
- pas_dispatcher_new
- pas_dispatcher_delete
- pas_dispatch
- pas_shutdown

NOTE The only thread dispatcher supported is GDisp (libasync).

pas_dispatcher_t

Purpose
A publish and subscribe dispatcher.

Syntax
```
typedef struct pas_dispatcher_struct pas_dispatcher_t;
```

Parameters
None.

Returns
Nothing.

pas_dispatcher_new

Purpose
Creates or advertises a dispatcher.

Syntax
```
pas_dispatcher_t *pas_dispatcher_new (void *disp);
```

Parameters

`dispcx`	The dispatcher context. If `NULL`, to start dispatching notifications, the application must call `pas_dispatch`. If not `NULL`, the dispatcher is a `libasync` dispatcher.

Returns

The dispatcher to use when creating publishers or subscribers (`pas_dispatcher_t`).

pas_dispatcher_delete

Purpose

Destroys a dispatcher created with `pas_dispatcher_new`.

Syntax

```
void pas_dispatcher_delete (pas_dispatcher_t *disp);
```

Parameters

`disp`	The event notification client environment.

Returns

Nothing.

pas_dispatch

Purpose

Starts the dispatch loop of an event notification environment. It has no effect if the application uses its own thread pool.

Syntax

```
void pas_dispatch (pas_dispatcher_t *disp);
```

Parameters

`disp`	The new dispatcher.

Returns
Nothing.

pas_shutdown

Purpose
Stops the dispatch loop of an event notification environment started with `pas_dispatch`. It has no effect if an application-provided dispatcher was passed to `pas_dispatcher_new`.

Syntax
```
void pas_shutdown (pas_dispatcher_t *disp);
```

Parameters

`disp`	The dispatcher context to shutdown.

Returns
Nothing.

Event Notification Service Java (JMS) API Reference

This chapter describes the implementation of the Java (JMS) API in ENS and the Java API itself.

This chapter contains these sections:

- Event Notification Service Java (JMS) API Implementation
- Java (JMS) API Overview
- Implementation Notes

Event Notification Service Java (JMS) API Implementation

The ENS Java API is included with iPlanet Messaging Server 5.2 and Sun ONE Calendar Server 5.1. The Java API conforms to the Java Message Service specification (JMS).

ENS acts as a provider to Java Message Service. Thus, it provides a Java API to ENS. The software consists of the base library plus a demo program.

Prerequisites to Use the Java API

To use the Java API, you need ENS enabled. For instructions on enabling ENS in iPlanet Messaging Server, see Appendix C in the *iPlanet Messaging Server 5.2 Administrator's Guide*. By default, ENS is already enabled in Sun ONE Calendar Server.

In addition, you need to install the following software, which is not provided with either iPlanet Messaging Server or Sun ONE Calendar Server:

- Java Development Kit (JDK) 1.2 or later
- Java Message Service 1.0.2a or later (tested with 1.0.2a)

You can download this software from `http://java.sun.com`.

Sample Java Programs

The iPlanet Messaging Server 5.2 sample programs, `JmsSample` and `JBiff`, are stored in the *`server-root`*`/bin/msg/enssdk/java/com/iplanet/ens/samples` directory. `JmsSample` is a generic ENS sample program. `JBiff` is iPlanet Messaging Server specific.

For `JBiff`, you will need the following additional items:

- Java Mail jar file (tested with JavaMail 1.2)
- Java Activation Framework (required by JavaMail, tested with JAF1.0.1)

You can download these items from `http://java.sun.com`.

Setting Up Your Environment

This section describes what to do to be able to compile and run the sample programs.

To Compile the JmsSample Program

1. Set your `CLASSPATH` to include the following:

   ens.jar file - `ens.jar`

   (For iPlanet Messaging Server, the `ens.jar` is located in the *`server-root`*`/java/jars/` directory.)

   Java Message Service - *`full-path`*`/jms1.0.2/jms.jar`

2. Change to the *`server-root`*`/bin/msg/enssdk/java` directory.
3. Run the following command:

   ```
   javac com/iplanet/ens/samples/JmsSample.java
   ```

To Compile the JBiff Program

1. Set your `CLASSPATH` to include the following:

   ens.jar file - `ens.jar`

   (For iPlanet Messaging Server, the `ens.jar` is located in the *`server-root`*`/java/jars/` directory.)

   Java Message Service - *`full-path`*`/jms1.0.2/jms.jar`

   JavaMail - *`full-path`*`/javamail-1.2/mail.jar`

   Java Activation Framework - *`full-path`*`/jaf-1.0.1/activation.jar`

2. Change to the *`server-root`*`/bin/msg/enssdk/java` directory.

3. Run the following command:

   ```
   javac com/iplanet/ens/samples/JBiff.java
   ```

To Run the JmsSample Program

1. Change to the *`server-root`*`/bin/msg/enssdk/java` directory.

2. Run the following command:

   ```
   java com.iplanet.ens.samples.JmsSample
   ```

3. You are prompted for three items:

   - ENS event reference (for example, for iPlanet Messaging Server: `enp://127.0.0.1/store`)
   - ENS hostname
   - ENS port (typically 7997)

4. Publish events.

   For iPlanet Messaging Server, the two ways to publish events are:

   - You can use the `apub` C sample program for ENS. See "iPlanet Messaging Server Sample Code," on page 83 for more information.
   - If you have enabled ENS, configure iBiff to publish iPlanet Messaging Server related events.

   For Sun ONE Calendar Server, events are published by the calendar server.

To Run the JBiff Demo Program

Prerequisite: To run the `JBiff` demo program, you need to enable ENS in iPlanet Messaging Server. See Appendix C in the *iPlanet Messaging Server 5.2 Administrator's Guide* for instructions on enabling ENS.

NOTE The demo is currently hardcoded to use the ENS event reference `enp://127.0.0.1/store`. This is the default event reference used by the iBiff notification plug-in.

1. Change to the *`server-root`*`/bin/msg/enssdk/java` directory.
2. Run the following:

   ```
   java com.iplanet.ens.samples.JBiff
   ```

3. The program prompts for your userid, hostname, and password.

   The code assumes that the ENS server and the IMAP server are running on *hostname*. The userid and password are the IMAP username and password to access the IMAP account.

The two test programs are ENS subscribers. You receive events from iBiff when email messages flow through iPlanet Messaging Server. Alternately you can use the `apub` C sample program to generate events. See "iPlanet Messaging Server Sample Code," on page 83 for more information.

Java (JMS) API Overview

The Java API for ENS uses a subset of the standard Java Messaging Service (JMS) API, with the addition of two new proprietary methods:

- `com.iplanet.ens.jms.EnsTopicConnFactory`
- `com.iplanet.ens.jms.EnsTopic`

JMS requires the creation of a `TopicConnectionFactory` and a `Topic`, which is provided by the two ENS proprietary classes.

For more information on the standard JMS classes and methods, see the JMS documentation at:

`http://java.sun.com/products/jms/docs.html`

New Proprietary Methods

The two proprietary method classes are `EnsTopicConnFactory` and `EnsTopic`.

com.iplanet.ens.jms.EnsTopicConnFactory

About the method

The method is a constructor that returns a `javax.jms.TopicConnectionFactory`. Instead of using a JNDI-style lookup to obtain the `TopicConnectionFactory` object, this method is provided.

Syntax

```
public EnsTopicConnFactory (String name,
                            String hostname,
                            int port,
                            OutputStream logStream)

                throws java.io.IOException
```

Arguments

Table 3-1 Arguments for EnsTopicConnFactory

Arguments	Type	Explanation
`name`	String	The client ID for the javax.jms.Connection
`hostname`	String	The hostname for the ENS server.
`port`	int	The TCP port for the ENS server.
`logStream`	OutputStream	Where messages are logged (cannot be null).

com.iplanet.ens.jms.EnsTopic

About this method

The method is a constructor that returns a `javax.jms.Topic`. Instead of using a JNDI-style lookup to obtain the `javax.jms.Topic`, this method is provided.

Syntax

```
public EnsTopic (String eventRef)
```

Arguments

Table 3-2 Arguments for EnsTopic

Arguments	Type	Explanation
`eventRef`	String	The ENS event reference.

Implementation Notes

This section describes items to be aware of when implementing the ENS Java API.

Shortcomings of the Current Implementation

The current implementation of the Java API does not supply an initial provider interface.

JMS Topic Connection Factory and ENS Destination are called out explicitly. These are `com.iplanet.ens.jms.EnsTopicConnFactory` and `com.iplanet.ens.jms.EnsTopic`. ENS does not use JNDI to get the `TopicConnectionFactory` and `Topic` objects.

Notification Delivery

The notification is delivered as a `javax.jms.TextMessage`. The parameter/values of the ENS event reference are provided as property names to the TextMessage. The payload is provided as the data of the TextMessage.

JMS Headers

- JMSDeliveryMode is always set to `NON_PERSISTENT` (that is, no storing of message for future delivery).
- JMSRedelivered is always set to false.
- JMSMessageID is set to an internal id. Specifically it is not set to the SMTP MessageID in the header of the email message for iPlanet Messaging Server.
- The payload is always a `javax.jms.TextMessage`. It corresponds to the ENS payload.

- JMSDestination is set to the full event reference (that is, it includes the parameter/values specific to this notification).
- JMSCorrelationID - Set to an internal sequence number.
- JMSTimestamp - Set to the time the message was sent.
  - For iPlanet Messaging Server and iBiff, this corresponds to the `timestamp` parameter.
  - This is unused in Sun ONE Calendar Server.
- JMSType - The type of notification.
  - For iPlanet Messaging Server and iBiff, this corresponds to the `evtType` parameter.
  - This is unused in Sun ONE Calendar Server.
- Additional properties:
  - Each parameter/value in the even reference becomes a property in the header. All property values are of type String.
- Unused headers are: JMSExpiration, JMSpriority, JMSReplyTo.

Miscellaneous

- MessageSelectors are not implemented.
- JMS uses the concept of durable and non-durable subscribers. A durable subscriber is a feature where notifications are guaranteed to be sent to subscribers even when they are offline, or if something catastrophic occurs, such as the ENS server going down after receiving the notification from the publisher but before delivering it to the subscriber.
  - Non-durable subscribers are implemented.
  - You can also use durable subscribers, however, the full functionality of being a durable subscriber is not implemented.
  - This aspect of being a durable subscriber is implemented: the publisher is acknowledged only after the subscriber receives a message.
  - This aspect of being a durable subscriber is not implemented: the message is not persistent, and delivery is not made to offline subscribers (after they come back online). In particular, JMSRedelivered is always set to false.

Sun ONE Calendar Server Specific Information

This chapter describes the Sun™ ONE Calendar Server specific items you need to use the ENS APIs.

This chapter contains these sections:

- Sun ONE Calendar Server Notifications
  - Alarm Notifications
  - Calendar Update Notifications
  - Advanced Topics
  - WCAP appid parameter and X-Tokens
- Sun ONE Calendar Server Sample Code

Sun ONE Calendar Server Notifications

There are two parts to the format of an Sun ONE Calendar Server notification:

- The event reference - A URL identifying the event.
- The payload - The data describing the event. Three different payload formats are supported: binary, text/calendar, and text/XML.

There are two types of calendar notifications: alarm notifications, which relay reminders; and calendar update notifications, which distribute changes to the calendar database.

Alarm Notifications

Alarm notifications notifications relay reminders. They are published by the `csadmind` daemon whenever it wants to send a reminder. The default subscriber for these alarms in Sun ONE is the `csnotifyd` daemon. Notifications consumed by `csnotifyd` have a binary payload and are acknowledged (reliable).

Additionally, the server can be configured to generate one additional notification for each reminder, which can be consumed by a third party notification infrastructure.

Table 4-1 has information on how the two different alarm notifications are enabled, their base event URLs, and the event payload format for each. (See "Format of Calendar Notifications," on page 63.)

Table 4-1 Alarm Notifications

Type	Enabled by	Base Event URL	Event Payload Format
Default alarm notification	Default	`enp:///ics/alarms`	Binary
Optional alarm notification	In `ics.conf`, the existence of a non-null value for `caldb.serveralarms.contenttype` enables the optional alarm notification.	Set the `ics.conf` file parameter `caldb.serveralarms.url` to this value: `enp:///ics/alarms`	Set the `ics.conf` file parameter `caldb.serveralarms.contenttype` to one of these values: `text/calendar` or `text/xml`

Event URL parameters are the same for either one:

- `calid` - Calendar ID
- `uid` - Component, either `event` or `todo` (task) ID
- `rid` - Recurrence ID
- `aid` - Alarm ID
- `comptype` - An event or a todo (task)

Calendar Update Notifications

Calendar update notifications distribute changes to the calendar database. They are published by the `cshttpd` or `csdwpd` daemons whenever a change is made to the database (if the notification is enabled for this type of change).

Table 4-2 lists each type of calendar update notification, and the `ics.conf` setting and base even URL for each of them.

Table 4-2 Calendar Update Notifications

Types	Enabling `ics.conf` Parameters (all parameters default to "yes" unless otherwise noted)	Base Event URL and Value or Event Payload
Attendee refesh actions	`caldb.berkeleydb.ensmsg.refreshevent`	`caldb.berkeleydb.ensmsg.refreshevent.url`
	`default=no`	default value: `enp:///ics/caleventrefresh`
Attendee refresh action	`caldb.berkeleydb.ensmsg.refreshevent.contenttype`	Event payload default value: `text/xml`
Attendee reply action	`caldb.berkeleydb.ensmsg.replyevent`	`caldb.berkeleydb.ensmsg.replyevent.url`
	`default=no`	default value: `enp:///ics/caleventreply`
Attendee reply action	`caldb.berkeleydb.ensmsg replyevent.contenttype`	Event payload default value: `text/xml`
Calendar creation	`caldb.berkeleydb.ensmsg.createcal`	`caldb.berkeleydb.ensmsg.createcal.url`
		default value: `enp:///ics/calendarcreate`
Calendar deletion	`caldb.berkeleydb.ensmsg.deletecal`	`caldb.berkeleydb.ensmsg.deletecal.url`
		default value: `enp:///ics/calendardelete`
Calendar modification	`caldb.berkeleydb.ensmsg.modifycal`	`caldb.berkeleydb.ensmsg.modifycal.url`
		default value: `enp:///ics/calendarmodify`

Table 4-2 Calendar Update Notifications *(Continued)*

Types	Enabling `ics.conf` Parameters (all parameters default to "yes" unless otherwise noted)	Base Event URL and Value or Event Payload

Event creation	`caldb.berkeleydb.ensmsg.createeevent`	`caldb.berkeleydb.ensmsg.createeevent.url`
		default value:
		`enp:///ics/caleventcreate`

Event modification	`caldb.berkeleydb.ensmsg.modifyevent`	`caldb.berkeleydb.ensmsg.modifyevent.url`
		default value:
		`enp:///ics/caleventmodify`

Event deletion	`caldb.berkeleydb.ensmsg.deleteevent`	`caldb.berkeleydb.ensmsg.deleteevent.url`
		default value:
		`enp:///ics/caleventdelete`

Todo (task) creation	`caldb.berkeleydb.ensmsg.createtodo`	`caldb.berkeleydb.ensmsg.createtodo.url`
		default value:
		`enp:///ics/caltodocreate`

Todo (task) modification	`caldb.berkeleydb.ensmsg.modifytodo`	`caldb.berkeleydb.ensmsg.modifytodo`
		default value:
		`enp:///ics/caltodomodify`

Todo (task) deletion	`caldb.berkeleydb.ensmsg.deletetodo`	`caldb.berkeleydb.ensmsg.deletetodo.url`
		default value:
		`enp:///ics/caltododelete`

Event URL parameters include:

- `calid` - Calendar ID
- `uid` - Component, either `event` or `todo` (`task`) ID
- `rid` - Recurrence ID

Format of Calendar Notifications

There are two parts to a notification:

- Event reference - URL identifying the event.
- Payload - Data describing the event. Three data formats are supported: binary, text/calendar, text/XML.

Advanced Topics

Normally, ENS notifications for attendee replies and organizer refreshes are published to the `caldb.berkeleydb.ensmsg.modifyevent` topic along with other modifications. By setting the `ics.conf` variable `caldb.berkeleydb.ensmsg.advancedtopics` to "yes" (the default is "no"), the ENS notifications can be published to separate modify, reply and refresh topics. This allows the consumer of the notification to understand more precisely what type of transaction triggered the notification.

Table 4-3 shows the topics ENS publishes notifications to depending on the setting of the `ics.conf` variable `caldb.berkeleydb.ensmsg.advancedtopics`.

Table 4-3 Advanced Topics Variable

Value of ics.conf Variable caldb.berkeleydb.ensmsg. advancedtopics	Topics to Which ENS Publishes Attendee Notifications
yes	caldb.berkeleydb.ensmsg.modifyevent caldb.berkeleydb.ensmsg.refreshevent caldb.berkeleydb.ensmsg.replyevent
no	caldb.berkeleydb.ensmsg.modifyevent

WCAP appid parameter and X-Tokens

When ENS sends out notifications of modifications made to existing events, it returns two X-Tokens with the notification, `X-NSCP-COMPONENT-SOURCE` and `X-NSCP-TRIGGERED-BY`.

The contents of the `X-NSCP-COMPONENT-SOURCE X-Token` varies depending on who originated the event and the absence or presence of the `appid` parameter in the original WCAP command that requested the event.

If the appid parameter is present in the original WCAP command, ENS returns its value in the X-NSCP-COMPONENT-SOURCE X-Token.(Only certain commands take the appid parameter. See the *Sun ONE Calendar Server Programmer's Manual* for further information on the appid parameter.) Using this mechanism, applications can "tag" ENS notifications in order to detect which ones it originated. The value of the appid command is a character string of the application's choosing. If the appid parameter is missing, standard values are assigned to the X-Token depending on the origin, see Table 4-4 that follows for the standard values).

The X-Token, X-NSCP-TRIGGERED-BY holds the name (uid) of the organizer or attendee that triggered the notification regardless of the absence or presence of the appid parameter.

Table 4-4 shows the effect of the presence of the appid parameter in WCAP commands on the value of the X-Token X-NSCP-COMPONENT-SOURCE.

Table 4-4 Presence of appid and Value of X-Token X-NSCP-COMPONENT-SOURCE

appid Present?	Value of X-Token X-NSCP-COMPONENT-SOURCE (with Request Origin)
no	Standard Values: WCAP (default) CALENDAR EXPRESS (from UI) ADMIN (from Admin tools)
yes	Value of appid

Sun ONE Calendar Server Sample Code

Sun ONE Calendar Server ships with a complete ENS implementation. If you wish to customize it, you may use the ENS APIs to do so. The following four code samples, a simple publisher and subscriber pair, and a reliable publisher and subscriber pair, illustrate how to use the ENS API. The sample code is provided with the product in the following directory:

```
/opt/SUNWics5/cal/csapi/samples/ens
```

Sample Publisher and Subscriber

This sample code pair establishes a simple interactive asynchronous publisher and subscriber.

Publisher Code Sample

```
/*
 * Copyright 2000 by Sun Microsystems, Inc.
 * All rights reserved
 *
 * apub : simple interactive asynchronous publisher using
 *
 * Syntax:
 *   apub host port
 */

#include <stdlib.h>
#include <stdio.h>

#include "pasdisp.h"
#include "publisher.h"

static pas_dispatcher_t *disp = NULL;
static publisher_t *_publisher = NULL;
static int _shutdown = 0;

static void _read_stdin();

static void _exit_usage()
{
   printf("\nUsage:\napub host port\n");
   exit(5);
}

static void _exit_error(const char *msg)
{
   printf("%s\n", msg);
   exit(1);
}

static void _call_shutdown()
{
   _shutdown = 1;
   pas_shutdown(disp);
}

static void _open_ack(void *arg, int rc, void *enc)
{
   _publisher = (publisher_t *)enc;
   (void *)arg;
```

```
    if (!_publisher)
    {
        printf("Failed to create publisher with status %d\n", rc);
        _call_shutdown();
        return;
    }

    _read_stdin();

    return;
}

static void _publish_ack(void *arg, int rc, void *ignored)
{
    (void *)ignored;

    free(arg);

    if (rc != 0)
    {

        printf("Publish failed with status %d\n", rc);

        _call_shutdown();

        return;
    }

    _read_stdin();

    return;
}

static void _read_stdin()
{

    static char input[1024];

    printf("apub> ");

    fflush(stdout);

    while (!_shutdown)
    {

        if ( !fgets(input, sizeof(input), stdin) )
        {
            continue;

        } else {
            char *message;
            unsigned int message_len;

            input[strlen(input) - 1] = 0; /* Strip off the \n */
```

```
            if (*input == '.' && input[1] == 0)
            {
                publisher_delete(_publisher);
                _call_shutdown();
                break;
            }

            message = strdup(input);
            message_len = strlen(message);
            publish(_publisher, "enp://yoyo.com/xyz",message,
                    message_len,
                    _publish_ack, NULL, (void *)message, 0);
            return;
        }
    }
    return;
}

main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];

    if (argc < 2) _exit_usage();

    if (*(argv[1]) == '0')
    {
        strcpy(host, "127.0.0.1");
    } else {
        strcpy(host, argv[1]);
    }

    if (argc > 2)
    {
        port = (unsigned short)atoi(argv[2]);
    }

    disp = pas_dispatcher_new(NULL);
    if (disp == NULL) _exit_error("Can't create publisher");

    publisher_new_a(disp, NULL, host, port, _open_ack, disp);

    pas_dispatch(disp);

    _shutdown = 1;

    pas_dispatcher_delete(disp);

    exit(0);
}
```

Subscriber Code Sample

```
/*
 * Copyright 1997 by Sun Microsystems, Inc.
 * All rights reserved
 *
 * asub : example asynchronous subscriber
 *
 * Syntax:
 *   asub host port
 */

#include <stdlib.h>
#include <stdio.h>

#include "pasdisp.h"
#include "subscriber.h"

static pas_dispatcher_t *disp = NULL;
static subscriber_t *_subscriber = NULL;
static subscription_t *_subscription = NULL;
static renl_t *_renl = NULL;

static void _exit_usage()
{
   printf("\nUsage:\nasub host port\n");
   exit(5);
}

static void _exit_error(const char *msg)
{
   printf("%s\n", msg);
   exit(1);
}

static void _subscribe_ack(void *arg, int rc, void *subscription)
{
   (void)arg;

   if (!rc)
   {
      _subscription = subscription;
      printf("Subscription successful\n");
   } else {
      printf("Subscription failed - status %d\n", rc);
      pas_shutdown(disp);
   }
}
```

```
static void _unsubscribe_ack(void *arg, int rc, void *ignored)
{
   (void *)ignored;
   (void *)arg;

   if (rc != 0)
   {
      printf("Unsubscribe failed - status %d\n", rc);
   }

   subscriber_delete(_subscriber);
   pas_shutdown(disp);
}

static int _handle_notify(void *arg, char *url, char *str, int len)
{
   (void *)arg;
   printf("[%s] %.*s\n", url, len, (str) ? str : "(null)");
   return 0;
}

static void _open_ack(void *arg, int rc, void *enc)
{
   _subscriber = (subscriber_t *)enc;

   (void *)arg;
   if (rc)
   {
      printf("Failed to create subscriber with status %d\n", rc);
      pas_shutdown(disp);
      return;
   }

   subscribe(_subscriber, "enp://yoyo.com/xyz",
             _handle_notify, NULL,
             _subscribe_ack, NULL);

   return;
}

static void _unsubscribe(int sig)
{
   (int)sig;
   unsubscribe(_subscriber, _subscription, _unsubscribe_ack, NULL);
}

main(int argc, char **argv)
{
   unsigned short port = 7997;
   char host[256];
```

```
    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0')
    {
       strcpy(host, "127.0.0.1");
    } else {
       strcpy(host, argv[1]);
    }
    if (argc > 2)
    {
       port = (unsigned short)atoi(argv[2]);
    }

    disp = pas_dispatcher_new(NULL);
    if (disp == NULL) _exit_error("Can't create publisher");

    subscriber_new_a(disp, NULL, host, port, _open_ack, NULL);

    pas_dispatch(disp);

    pas_dispatcher_delete(disp);

    exit(0);
}
```

Reliable Publisher and Subscriber

This sample code pair establishes a reliable asynchronous publisher and subscriber.

Reliable Publisher Sample

```
/*
 * Copyright 2000 by Sun Microsystems, Inc.
 * All rights reserved
 *
 * rpub : simple *reliable* interactive asynchronous publisher.
 *  It is designed to be used in combination with rsub,
 *  the reliable subscriber.
 *
 * Syntax:
 *   rpub host port
 */

#include <stdlib.h>
#include <stdio.h>

#include "pasdisp.h"
#include "publisher.h"
```

```
static pas_dispatcher_t *disp = NULL;
static publisher_t *_publisher = NULL;
static int _shutdown = 0;
static renl_t *_renl;

static void _read_stdin();

static void _exit_usage()
{

   printf("\nUsage:\nrpub host port\n");

   exit(5);
}

static void _exit_error(const char *msg)
{
   printf("%s\n", msg);
   exit(1);
}

static void _call_shutdown()
{
   _shutdown = 1;
   pas_shutdown(disp);
}

static void _renl_create_cb(void *arg, int rc, void *ignored)
{
   (void *)arg;
   (void *)ignored;

   if (!_publisher)
   {
      printf("Failed to create RENL - status %d\n", rc);
      _call_shutdown();
      return;
   }

   _read_stdin();

   return;
}

static void _publisher_new_cb(void *arg, int rc, void *enc)
{
   _publisher = (publisher_t *)enc;

   (void *)arg;
```

```
    if (!_publisher)
    {
        printf("Failed to create publisher - status %d\n", rc);
        _call_shutdown();
        return;
    }

    renl_create_publisher(_publisher, "renl_id", NULL,
                          _renl_create_cb,NULL);

    return;
}

static void _recv_ack(void *arg, int rc, void *ignored)
{
    (void *)ignored;

    if (rc < 0)
    {
        printf("Acknowledgment Timeout\n");
    } else if ( rc == 0) {
        printf("Acknowledgment Received\n");
    }
    fflush (stdout);

    _read_stdin();

    free(arg);

    return;
}

static void _read_stdin()
{
    static char input[1024];

    printf("rpub> ");
    fflush(stdout);
    while (!_shutdown)
    {
        if ( !fgets(input, sizeof(input), stdin) )
        {
            continue;
        } else {
            char *message;
            unsigned int message_len;

            input[strlen(input) - 1] = 0; /* Strip off the \n */
```

```
            if (*input == '.' && input[1] == 0)
            {
                publisher_delete(_publisher);
                _call_shutdown();
                break;
            }

            message = strdup(input);
            message_len = strlen(message);

            /* five seconds timeout */
            publish(_publisher, "enp://yoyo.com/xyz",
                    message, message_len,
                    NULL, _recv_ack, message, 5000);

            return;
        }
    }
    return;
}

main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];

    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0')
    {
        strcpy(host, "127.0.0.1");
    } else {
        strcpy(host, argv[1]);
    }
    if (argc > 2)
    {
        port = (unsigned short)atoi(argv[2]);
    }

    disp = pas_dispatcher_new(NULL);
    if (disp == NULL) _exit_error("Can't create publisher");

    publisher_new_a(disp, NULL, host, port, _publisher_new_cb,
                    NULL);

    pas_dispatch(disp);

    _shutdown = 1;

    pas_dispatcher_delete(disp);

    exit(0);
}
```

Reliable Subscriber Sample

```
/*
 * Copyright 1997 by Sun Microsystems, Inc.
 * All rights reserved
 *
 * asub : example asynchronous subscriber
 *
 * Syntax:
 *   asub host port
 */

#include <stdlib.h>
#include <stdio.h>

#include "pasdisp.h"
#include "subscriber.h"

static pas_dispatcher_t *disp = NULL;
static subscriber_t *_subscriber = NULL;
static subscription_t *_subscription = NULL;
static renl_t *_renl = NULL;

static void _exit_usage()
{
   printf("\nUsage:\nasub host port\n");
   exit(5);
}

static void _exit_error(const char *msg)
{
   printf("%s\n", msg);
   exit(1);
}

static void _subscribe_ack(void *arg, int rc, void *subscription)
{
   (void)arg;

   if (!rc)
   {
      _subscription = subscription;
      printf("Subscription successful\n");
      _renl = renl_create_subscriber(_subscription, "renl_id",
NULL);
   } else {
      printf("Subscription failed - status %d\n", rc)
      pas_shutdown(disp);
   }
}
```

```
static void _unsubscribe_ack(void *arg, int rc, void *ignored)
{
   (void *)ignored;
   (void *)arg;

   if (rc != 0)
   {
      printf("Unsubscribe failed - status %d\n", rc);
   }

   subscriber_delete(_subscriber);
   pas_shutdown(disp);
}

static int _handle_notify(void *arg, char *url, char *str, int len)
{
   (void *)arg;
   printf("[%s] %.*s\n", url, len, (str) ? str : "(null)");
   return 0;
}

static void _open_ack(void *arg, int rc, void *enc)
{
   _subscriber = (subscriber_t *)enc;

   (void *)arg;
   if (rc)
   {
      printf("Failed to create subscriber with status %d\n", rc);
      pas_shutdown(disp);
      return;
   }

   subscribe(_subscriber, "enp://yoyo.com/xyz",_handle_notify,
             NULL,_subscribe_ack, NULL);

   return;
}

static void _unsubscribe(int sig)
{
   (int)sig;
   unsubscribe(_subscriber, _subscription, _unsubscribe_ack, NULL);
}

main(int argc, char **argv)
{
   unsigned short port = 7997;
   char host[256];
```

```
    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0')
    {
       strcpy(host, "127.0.0.1");
    } else {
       strcpy(host, argv[1]);
    }
    if (argc > 2)
    {
       port = (unsigned short)atoi(argv[2]);
    }
    disp = pas_dispatcher_new(NULL);
    if (disp == NULL) _exit_error("Can't create publisher");
    subscriber_new_a(disp, NULL, host, port, _open_ack, NULL);
    pas_dispatch(disp);
    pas_dispatcher_delete(disp);
    exit(0);
}
```

iPlanet Messaging Server Specific Information

This chapter describes the iPlanet™ Messaging Server specific items you need to use the ENS APIs.

This chapter contains these sections:

- iPlanet Messaging Server Events and Parameters
- iPlanet Messaging Server Sample Code

iPlanet Messaging Server Events and Parameters

For iPlanet Messaging Server, there is only one event reference, which can be composed of several parameters. Each parameter has a value and a payload.

iPlanet Messaging Server supports the following types of events:

- NewMsg - New message was received by the system into the user's mailbox.
- DeleteMsg - User deleted a message (in the IMAP protocol, expunged) from the mailbox.
- UpdateMsg - Message was appended to the mailbox (other than by NewMsg). for example, the user copied an email message to the mailbox.
- ReadMsg - Message in the mailbox was read (in the IMAP protocol, the message was marked Seen).
- PurgeMsg - Message was purged (in the IMAP protocol, expunged) from the mailbox by the system.

The following applies to the above supported events:

- All events relate only to the INBOX.
- The NewMsg notification is issued only after the message is deposited in the user mailbox (as opposed to “after it was accepted by the server and queued in the message queue”).
- Both the `DeleteMsg` and the `PurgeMsg` events correspond to when a message is deleted from the user’s mailbox (in the IMAP protocol, the message is expunged). It is not when a message is marked for deletion in the IMAP protocol. The only difference between the two events is who deleted the message. `DeleteMsg` indicates that the user deleted the message, while `PurgeMsg` indicates that iPlanet Messaging Server deleted the message (for example, if the message has expired).
- The notification will carry several pieces of information depending on the event type, for example, `NewMsg` indicates the IMAP uid of the new message.
- Events are not generated for POP3 client access.

Parameters

iBiff will use the following format for the ENS event reference:

`enp://127.0.0.1/store`*?param=value&param1=value1&param2=value2*

The event key `enp://127.0.0.1/store` has no significance other than its uniqueness as a string. For example, the hostname portion of the event key has no significance as a hostname. It is simply a string that is part of the URI. However, the event key is user configurable. The list of iBiff configuration parameters is listed in a separate section below.

The second part of the event reference consists of parameter/value pairs. This part of the event reference is separated from the event key by a question mark (?). The parameter and value are separated by an equals sign (=). The parameter/value pairs are separated by an ampersand (&). Note that there can be empty values, for which the value simply does not exist.

Table 5-1 on page 79 describes the mandatory configuration parameters that need to be included in every notification.

Table 5-1 Mandatory Configuration Parameters

Parameter	Data Type	Description
`evtType`	string	Specifies the event type. One of NewMsg, UpdateMsg, ReadMsg, DeleteMsg, or PurgeMsg.
`mailboxName`	string	Specifies the mailbox name in the message store. The mailboxName has the format uid@domain, where uid is the userid, and domain is the domain the user belongs to. The @domain portion is added only when the user does not belong to the default domain (i.e. the user is in a hosted domain).
`timestamp`	64-bit integer	Specifies the number of milliseconds since the epoch (midnight GMT, January 1, 1970).
`process`	string	Specifies the name of the process that generated the event. If the process name is unknown, the process id will be used (an integer).
`hostname`	string	The hostname of the machine that generated the event.

Table 5-2 describes the optional configuration parameters, depending on the event type.

Table 5-2 Optional Configuration Parameters

Parameter	Data Type	Description
`numMsgs`	unsigned 32-bit integer	Specifies the number of existing messages.
`size`	unsigned 32-bit integer	Specifies the size of the message. Note that this may not be the size of payload, since the payload is typically a truncated version of the message.
`uidValidity`	unsigned 32-bit integer	Specifies the IMAP uid validity parameter.
`imapUid`	unsigned 32-bit integer	Specifies the IMAP uid parameter.
`uidSeqSeen`	string	Specifies the list of uids that are marked seen in IMAP syntax, such as “1:6.”
`lastUid`	unsigned 32-bit integer	Specifies the last IMAP uid value that was used.

Table 5-2 Optional Configuration Parameters *(Continued)*

Parameter	Data Type	Description
`hdrLen`	unsigned 32-bit integer	Specifies the size of the message header. Note that this might not be the size of the header in the payload, because it might have been truncated.
`qUsed`	signed 32-bit integer	Specifies the disk space used in quota in kilobytes.
`qMax`	signed 32-bit integer	Specifies the disk space quota in kilobytes. The value is set to -1 to indicate no quotas.
`qMsgUsed`	signed 32-bit integer	Specifies the number of messages used in quota. Should be the same value as numMsgs.
`qMsgMax`	signed 32-bit integer	Specifies the quota for max number of messages. The value is set to -1 to indicate no quotas.

NOTE Subscribers should allow for undocumented parameters when parsing the event reference. This allows for future compatibility when new parameters are added.

Payload

Depending on the event, there may be the following data in the payload portion of the ENS notification:

- The headers of the message - (string) - The length will be limited to a certain (configurable) size. See configuration parameters in a separate section below.
- The first few bytes of the body of the message - (string). The actual number of bytes will be configurable. See configuration parameters in a separate section below.

Table 5-3 shows the parameters that are available for each event type.

Table 5-3 Available Parameters for Each Event Type

Field Name	NewMsg, UpdateMsg	ReadMsg	DeleteMsg, PurgeMsg
`numMsgs`	Yes	No	Yes
`size`	Yes	No	No
`uidValidity`	Yes	Yes	Yes

Table 5-3 Available Parameters for Each Event Type *(Continued)*

Field Name	NewMsg, UpdateMsg	ReadMsg	DeleteMsg, PurgeMsg
imapUid	Yes	No	Yes
uidSeqSeen	No	Yes	No
uidSeqDel	No	Yes	No
lastUid	No	No	Yes
hdrLen	Yes	No	No
qUsed	Yes	No	Yes
qMax	Yes	No	Yes
qMsgUsed	Yes	No	Yes
qMsgMax	Yes	No	Yes
payload (headers/body)	Yes	No	No

Examples

The following example shows a NewMsg event reference (it is actually a single line that is broken up to several lines for readability):

```
enp://127.0.0.1/store?evtType=NewMsg&mailboxName=ketu310&timestamp=972423964000
&process=16233&hostname=ketu&numMsgs=1&size=3339&uidValidity=972423964&
imapUid=1&hdrLen=810
```

This is the associated payload, note that the body portion has been truncated:

```
Return-path: <>
Received: from process-daemon.ketu.siroe.com by ketu.siroe.com
(iPlanet Messaging Server 5.0  (built Oct 17 2000))
id <0G2Y00C01F4SIY@ketu.siroe.com> for ketu310@ims-ms-daemon
(ORCPT ketu310@siroe.com); Tue, 24 Oct 2000 14:46:04 -0700 (PDT)
Received: from ketu.siroe.com
(iPlanet Messaging Server 5.0  (built Oct 17 2000))
id <0G2Y00C01F4RIX@ketu.siroe.com>; Tue, 24 Oct 2000 14:46:04 -0700 (PDT)
Date: Tue, 24 Oct 2000 14:46:04 -0700 (PDT)
From: Internet Mail Delivery
Subject: Delivery Notification: Delivery has failed
To: ketu310@siroe.com
Message-id: <0G2Y00C05F4SIX@ketu.siroe.com>
MIME-version: 1.0
Content-type: multipart/report; report-type=delivery-status;
boundary="Boundary_(ID_VlTrnuIgC5ferJnL2SCzhQ)"


--Boundary_(ID_VlTrnuIgC5ferJnL2SCzhQ)
Content-type: text/plain; charset=us-ascii
Content-language
```

This is another example, this time for the DeleteMsg event (again it is a single line that is broken up for readability). Note that this example shows a mailboxName for the userid `blim` in the hosted domain `symult.com`.

```
enp://127.0.0.1/store?evtType=DeleteMsg&mailboxName=blim@symult.com&
timestamp=972423953000&process=15354&hostname=ketu&numMsgs=0&
uidValidity=972423928&imapUid=2&lastUid=2
```

And a third example showing a ReadMsg event (again the line is broken up for readability). Note that this example shows an empty value for the uidSeqSeen parameter. It also shares the same userid as the previous example, however this corresponds to a different user, a user in the default domain.

```
enp://127.0.0.1/store?evtType=ReadMsg&mailboxName=blim&timestamp=972423952000&
process=15354&hostname=ketu&uidValidity=972423928&uidSeqSeen=&uidSeqDel=1
```

iPlanet Messaging Server Sample Code

iPlanet Messaging Server ships with a complete ENS implementation but by default it is not enabled. To enable ENS in iPlanet Messaging Server, see Appendix C in the *iPlanet Messaging Server 5.2 Administrator's Guide*.

The following two code samples illustrate how to use the ENS API. The sample code is provided with the product in the following directory:

server-root/bin/msg/enssdk/examples

Sample Publisher

This sample code provides a simple interactive asynchronous publisher.

```
/*
 * Copyright 2000 by Sun Microsystems, Inc.
 * All rights reserved
 */
/*
 *
 *                              apub
 *                               --
 *         a simple interactive asynchronous publisher
 *                               --
 *
 * This simplistic program publishes events using the hard-coded
 * event reference
 *      enp://127.0.0.1/store
 * and the data entered at the prompt as notification payload.
 * Enter "." to end the program.
*
 * If you happen to run the corresponding subscriber, asub, on the
 * same notification server, you will notice the sent data printed
 * out in the asub window.
 * Syntax:
 *      $ apub <host> <port>
 * where
 *      <host> is the notification server hostname
 *      <port> is the notification server IP port number
 */

#include <stdlib.h>
#include <stdio.h>#include "pasdisp.h"
#include "publisher.h"
```

```
static pas_dispatcher_t *disp = NULL;
static publisher_t *_publisher = NULL;
static int _shutdown = 0;
static void _read_stdin();

static void _exit_usage()
{
   printf("\nUsage:\napub host port\n");
   exit(5);
}

static void _exit_error(const char *msg)
{
   printf("%s\n", msg);
   exit(1);
}

static void _call_shutdown()
{
   _shutdown = 1;
   pas_shutdown(disp);
}

static void _open_ack(void *arg, int rc, void *enc)
{
   _publisher = (publisher_t *)enc;
   (void *)arg;

if (!_publisher) {
       printf("Failed to create publisher with status %d\n", rc);
       _call_shutdown();
       return;
   }
   _read_stdin();
   return;
}

static void _publish_ack(void *arg, int rc, void *ignored)
{
   (void *)ignored;
   free(arg)
   if (rc != 0) {
       printf("Publish failed with status %d\n", rc);
       _call_shutdown();
       return;
   }

   _read_stdin();
   return;
}
```

```
static void _read_stdin()
{
    static char input[1024];
    printf("apub> ");
    fflush(stdout);
    while (!_shutdown) {
        if ( !fgets(input, sizeof(input), stdin) ) {
            continue;
         } else {
            char *message;
            unsigned int message_len;
            input[strlen(input) - 1] = 0; /* Strip off the \n */
           if (*input == '.' && input[1] == 0) {
               publisher_delete(_publisher);
               _call_shutdown();
               break;
            }
           message = strdup(input);
           message_len = strlen(message);
           publish(_publisher, "enp://127.0.0.1/store",
                   message, message_len,
                   _publish_ack, NULL, (void *)message, 0);
            return;
        }
    }
     return;
}

main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];
    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0') {
        strcpy(host, "127.0.0.1");
    }else {
        strcpy(host, argv[1]);
    }
    if (argc > 2) {
        port = (unsigned short)atoi(argv[2]);
    }
    disp = pas_dispatcher_new(NULL);
    if (disp == NULL) _exit_error("Can't create publisher");
    publisher_new_a(disp, NULL, host, port, _open_ack, disp);
    pas_dispatch(disp);
```

```
    _shutdown = 1;
    pas_dispatcher_delete(disp);
    exit(0);
}
```

Sample Subscriber

This sample code provides a simple subscriber.

```
/*
 * Copyright 1997 by Sun Microsystems, Inc.
 * All rights reserved
 *
 */

/*
 *
 *                              asub
 *                               --
 *                      a simple subscriber
 *                               --
 *
 * This simplistic program subscribes to events matching the
 * hard-coded event reference:
 *      enp://127.0.0.1/store
 * It subsequently received messages emitted by the apub processes
 * if any are being used, and prints the payload of each received
 * notification to stdout.
 *
 * Syntax
 *      $ asub <host> <port>
 * where
 *      <host> is the notification server hostname
 *      <port> is the notification server IP port number
 */

#include <stdlib.h>
#include <stdio.h>

#include "pasdisp.h"
#include "subscriber.h"

static pas_dispatcher_t *disp = NULL;
static subscriber_t *_subscriber = NULL;
static subscription_t *_subscription = NULL;
static renl_t *_renl = NULL;
```

```
static void _exit_usage()
{
   printf("\nUsage:\nasub host port\n");
   exit(5);
}

static void _exit_error(const char *msg)
{
   printf("%s\n", msg);
   exit(1);
}

static void _subscribe_ack(void *arg, int rc, void *subscription)
{
   (void)arg;
   if (!rc) {
      _subscription = subscription;
      printf("Subscription successful\n");
      subscriber_keepalive(_subscriber, 30000);
   }else {
      printf("Subscription failed - status %d\n", rc);
      pas_shutdown(disp);
   }
}

static void _unsubscribe_ack(void *arg, int rc, void *ignored)
{
(void *)ignored;
   (void *)arg;
   if (rc != 0) {
      printf("Unsubscribe failed - status %d\n", rc);
   }
   subscriber_delete(_subscriber);
   pas_shutdown(disp);
}

static int _handle_notify(void *arg, char *url, char *str, int len)
{
   (void *)arg;
   printf("[%s] %.*s\n", url, len, (str) ? str : "(null)");
   return 0;
}

static void _open_ack(void *arg, int rc, void *enc)
{
   _subscriber = (subscriber_t *)enc;
   (void *)arg;
   if (rc) {
```

```
        printf("Failed to create subscriber with status %d\n", rc);
        pas_shutdown(disp);
        return;
    }
    subscribe(_subscriber, "enp://127.0.0.1/store",
              _handle_notify, NULL,
              _subscribe_ack, NULL);
    return;
}
static void _unsubscribe(int sig)
{
    (int)sig;
     unsubscribe(_subscriber, _subscription, _unsubscribe_ack, NULL);
}
main(int argc, char **argv)
{
    unsigned short port = 7997;
    char host[256];
    if (argc < 2) _exit_usage();
    if (*(argv[1]) == '0') {
        strcpy(host, "127.0.0.1");
     }else {
        strcpy(host, argv[1]);
    }
    if (argc > 2) {
        port = (unsigned short)atoi(argv[2]);
    }  disp = pas_dispatcher_new(NULL);
    if (disp == NULL) _exit_error("Can't create publisher");
    subscriber_new_a(disp, NULL, host, port, _open_ack, NULL);
    pas_dispatch(disp);
    pas_dispatcher_delete(disp);
exit(0);
}
```

Implementation Notes

The current implementation does not provide security on events that can be subscribed to. Thus, a user could register for all events, and portions of all other users' mail. Because of this it is strongly recommended that the ENS subscriber be on the "safe" side of the firewall at the very least.

Glossary

consumed Notifications received and processed by a service are said to be consumed by the process.

event Generation of data for an event reference. For Sun ONE Calendar Server, this occurs when there is a change in a resource (calendar). For iPlanet Messaging Server, there is a list of events that occur (`NewMsg`, `DeleteMsg`, and so on).

event consumer Synonym for event subscriber.

Event Notification Service Application framework relaying notifications sent to subscribers by publishers.

event producer Synonym for event publisher.

event publisher An application that makes events known to other applications.

event reference Identifies an event handled by ENS. It complies with URI syntax defined by RFC 2396.

event subscriber An application that consumes events.

iBiff The name given to the plug-in that publishes message store notifications in iPlanet Messaging Server. It includes the specification of how to subscribe to the notifications.

notification Message describing an event occurrence. Sent by the event publisher, it contains a reference to the event as well as optional data used by the event consumers, but opaque to the notification service.

notification service Receives subscriptions and notifications from other servers. Relays notifications to subscribers.

notification server A notification service is made up of one or more server instances, each running on a separate host.

notify A synonym for publish.

payload The data describing an event. Three different payload formats are supported: binary, text/calendar, and text/XML.

publish Send a notification. An event publisher makes an event available to the notification service.

reliable event notification link (RENL) An RENL has a publisher, a subscriber, and a unique ID, which identify notifications that are subject to acknowledgment.

resource A piece of data accessed from the IP network. For example, a calendar is a resource.

resource state The value of attributes that describe a resource. For example, a meeting time.

subscribe Send a subscription. An event subscriber tells the notification service that it wants to receive notifications of a specific event.

subscription Message sent by the event subscriber. Contains an event reference, a client-side request identifier, and optional access control rules.

task In Calendar Express on the client side, a component of a calendar that specifies something to be done. On the server side, a task is also called a todo.

todo In Sun ONE Calendar Server, on the server side, a a component of a calendar that specifies something to be done. In Calendar Express on the client side, a todo is called a task.

unsubscribe Cancels a subscription. An event subscriber tells the notification service to stop relaying notifications for the specified event.

unsubscription This message cancels (unsubscribes) an existing subscription. An event subscriber tells the notification service to stop relaying notifications for the specified event.

Index

I

N

P

R

S

U