Oracle® Communications Network Charging and Control

Voucher and Wallet Server Technical Guide

Release 15.2

January 2026

Contents

About This Document

Scope

The scope of this document includes all the information required to install, configure and administer the Voucher and Wallet Server (VWS) application.

Audience

This guide was written primarily for system administrators and persons installing, configuring and administering the VWS application.   However, sections of the document may be useful to anyone requiring an introduction to the application.

Prerequisites

Although there are no prerequisites for using this guide, familiarity with the target platform would be an advantage.

A solid understanding of UNIX and a familiarity with IN concepts are an essential prerequisite for safely using the information contained in this technical guide. Attempting to install, remove, configure or otherwise alter the described system without the appropriate background skills, could cause damage to the system; including temporary or permanent incorrect operation and complete loss of service.

This manual describes system tasks that should only be carried out by suitably trained operators.

Related Documents

The following documents are related to this document:

- *Service Logic Execution Environment Technical Guide*
- *Service Management System Technical Guide*
- *Service Management System User's Guide*
- *Event Detail Record Reference Guide*

If this Voucher and Wallet Server is installed with Charging Control Services, these documents are also related to this document:

- *Charging Control Services Technical Guide*
- *Charging Control Services User's Guide*
- *Feature Nodes Reference Guide*

Document Conventions

Typographical Conventions

The following terms and typographical conventions are used in the Oracle Communications Network Charging and Control (NCC) documentation.

Formatting Convention	Type of Information
Special Bold	Items you must select, such as names of tabs. Names of database tables and fields.
Italics	Name of a document, chapter, topic or other publication. Emphasis within text.
Button	The name of a button to click or a key to press. **Example:** To close the window, either click **Close**, or press **Esc**.
Key+Key	Key combinations for which the user must press and hold down one key and then press another. Example: **Ctrl+P** or **Alt+F4**.
`Monospace`	Examples of code or standard output.
`Monospace Bold`	Text that you must enter.
variable	Used to indicate variables or text that should be replaced with an actual value.
menu option > menu option >	Used to indicate the cascading menu option to be selected. Example: **Operator Functions > Report Functions**
hypertext link	Used to indicate a hypertext link.

Specialized terms and acronyms are defined in the glossary at the end of this guide.

Chapter 1

System Overview

Overview

Introduction

This chapter provides a high-level overview of the application. It explains the basic functionality of the system and lists the main components.

It is not intended to advise on any specific Oracle Communications Network Charging and Control (NCC) network or service implications of the product.

In this Chapter

This chapter contains the following topics.

Introduction to VWS

Introduction

The Voucher and Wallet Server (VWS) provides high-performance, real-time charging and subscriber account management functions.

The VWS solution maintains voucher, wallet and reservation details in the E2BE database on the Voucher and Wallet Server. It enables call processing applications to bill mobile customers.

Billing information is kept logically separate from call processing on the SLCs, allowing it to be used by multiple clients.

Functions

The role of the VWS is to manage all the billing/charging information associated with call processing.

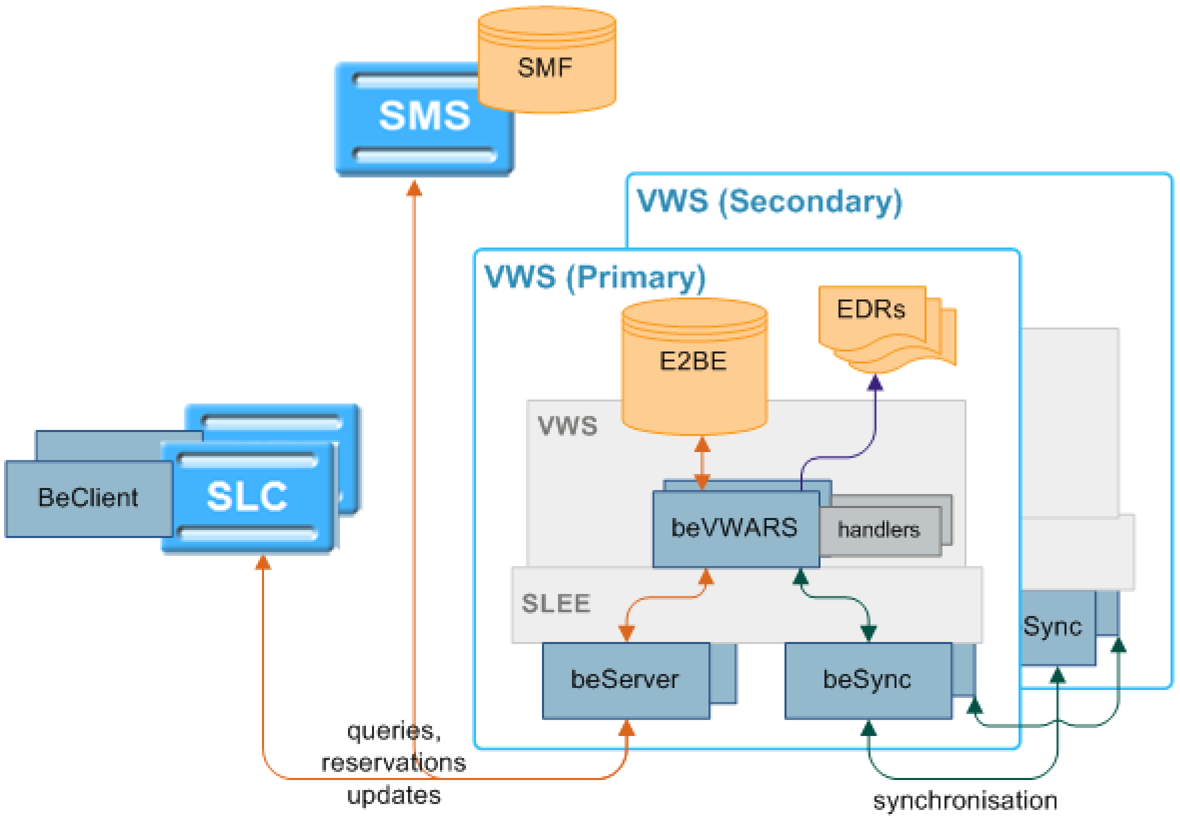
The VWS provides:

- Subscriber account management
- Management of multiple wallets and balance types
- Real-time rating of services
- Structure for tariffing on transactions
- Reservation, debit and credit requests

- Voucher management, including voucher query and redemption requests
- Failover and machine redundancy
- EDRs and EDR archiving

Main components diagram

This diagram shows the main components of VWS. They mostly run as separate processes on the SLEE, as illustrated below. Their operation needs to be coordinated, as the state of the entire VWS affects their behavior.



Main components

This table describes the main components in VWS.

Process	Role	Further information
BeClient	The BeClient is a SLEE interface that runs on the SLC and handles any process that uses the libBeClientIF library to connect to the beServer. The main BeClient is the BeClient provided by VWS for the SLC. (Other applications can provide other processes to handle other activities such as ccsBeOrb, which handles interaction between the SMS UI and the Voucher and Wallet Server nodes.)	*BeClient* (on page 50)
beServer	Handles connections from client processes (including BeClient processes) and controls routing to beVWARS processes. You can run more than one beServer process to improve performance on Sun CMT hardware.	*beServer* (on page 65)

Process	Role	Further information
beVWARS	beVWARS is the core of the VWS. More than one beVWARS will usually be running on a VWS. It: • Reads and caches wallet and voucher information from the E2BE database • Manages all queries, reservations and updates against wallets • Manages all queries, redemptions and state changes for vouchers • Writes files used to synchronize data • Writes EDRs.	*beVWARS* (on page 94)
beVWARS plug-in handlers	Perform business-case specific operations on wallets and vouchers. Some plug-in handlers are provided by VWS, but other applications can extend VWS logic by providing additional plug-ins. **Examples:** CCS beVWARS plug-in handlers manage monthly: • Spend accumulation and upgrade • Account activation	*beVWARS plugins* (on page 7)
beSync	Synchronizes data between the Voucher and Wallet Servers in a VWS pair. You can run more than one beSync process to improve performance on Sun CMT hardware.	*beSync* (on page 76)
E2BE database	The databases on the VWSs. They hold a subset of the data from the SMF database on SMS.	NA

Billing Interfaces

VWS supports external interfaces to bill for third-party services, such as:

- Calling card services
- Data charging services
- SMS charging services
- Universal Parlay Gateway

VWS Domains

Each VWS domain is made up of a pair of Voucher and Wallet Server. For more information about Voucher and Wallet Server pairs, see *Data redundancy* (on page 18).

Domains can provide a full set of functions, or can be configured to offer a specific set of functions.

Domains are configured in the Service Management screen in CCS. For more information about configuring domains, see *CCS User's Guide*.

About improving performance

If the VWS server will be processing high volumes of traffic, you can improve performance by configuring the SLEE to run multiple instances of the beServer and beSync processes. Running multiple instances of these processes enables the VWS server to load-share:

- beServer client connections over multiple process spaces

- beSync synchronization connections over multiple process spaces

For information on configuring beServer, see *beServer* (on page 65). For information on configuring beSync, see *beSync* (on page 76).

Note: In the diagrams in the following sections in this guide that show the beServer and beSync processes, only one instance of each is shown.

Wallets, Balances and Buckets

Wallets

Each subscriber account is linked to one or two wallets.

A wallet is a group of balances owned by the subscriber and available to pay for prepaid services offered by the platform.

Example: A subscriber could have a “General Cash” balance and a “Free Notification” balance in their wallet. Each balance has its own expiry date, which means that any value left in the balance after this date will be removed.

Wallet states

The wallet has a state which:

- Indicates what phase of the life cycle the associated subscriber account currently is in
- Determines whether the subscriber can use his or her services

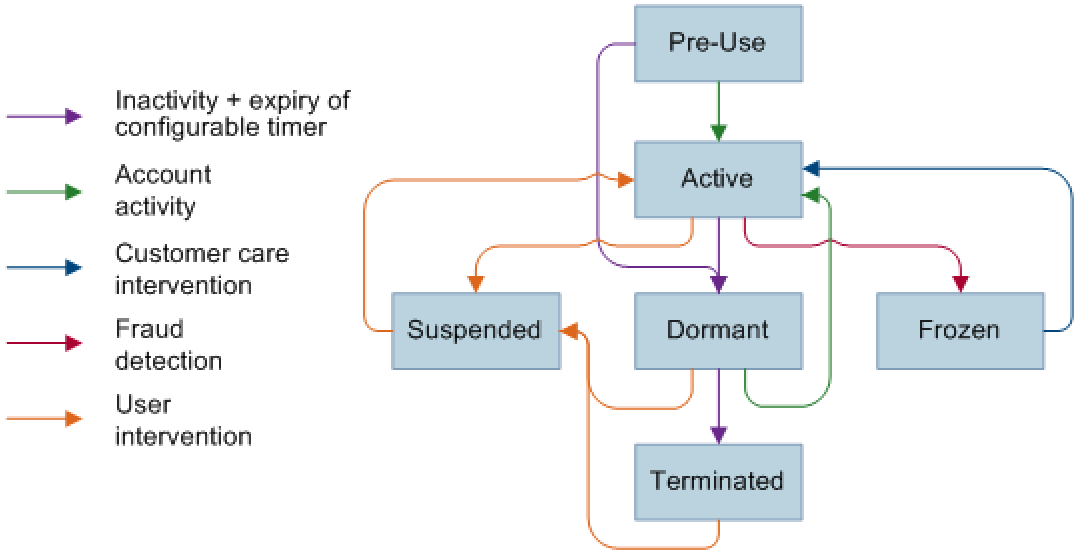
This table describes the available wallet states.

State	Description
Pre-use	The first state after the subscriber account/wallet is created.
Active	The state during which normal wallet operations are handled. All services can be used. When a subscriber uses their account for the first time (via any paid service), the wallet moves from the Pre-use state into the Active state. The initial expiry dates for the balances and the wallet are set when the wallet is activated.
Dormant	If the subscriber does not use any services for a configurable period of time, the account/wallet is put in the Dormant state. The Dormant state is useful for reporting inactive subscribers. All services are still accessible.
Frozen	If fraudulent activity is detected on the subscriber account, the subscriber account/wallet is changed into the Frozen state. All services are disabled until manual intervention by an operator.
Suspended	The suspended state can be used by the operator to temporarily disable a subscriber's account.
Terminated	When the wallet's expiry date is reached, the subscriber account/wallet is moved into the terminated state.

Note: Wallets may also be affected by expiry dates. For more information about wallet and bucket expiry, see *Wallet and Bucket Expiry* (on page 23).

Wallet lifecycle

This diagram shows the states in a standard life cycle of a wallet.



Wallet life cycle plans

A wallet life cycle plan comprises a set of wallet life cycle periods. The plan can be associated with a subscriber's wallet through the subscriber's product type.

The plan is used to extend the existing states of the wallet with customizable sub-states called periods. The beginning of each period can be configured as an offset of days before or after the wallet expiration date. A period ends when the next one, if defined, starts or when the wallet expires.

For each period in the wallet life cycle you can define the features that will be available. This includes:

- Session charges
- Available named event operations
- Enabled general charges and recharges

For information on configuring wallet life cycle plans, see the Wallet Management section in *CCS User's Guide*.

VWS associations

When a wallet is created, it is created on a specific VWS Voucher and Wallet Server pair. This Voucher and Wallet Server pair handles all the updates and information queries for that wallet.

Processes which send a wallet request will usually check to which VWS to send the request before the request is sent. If the wallet request is sent to the wrong VWS, the VWS will return an error.

Migrating wallets

If the Voucher and Wallet Server (VWS) pair is storing CCS wallets, the wallets can be migrated from one VWS to another using the **UBE Account Balancing** tab. For more information about this tab, see *CCS User's Guide*. For more information about how CCS migrates wallets between VWSs, see *CCS Technical Guide*.

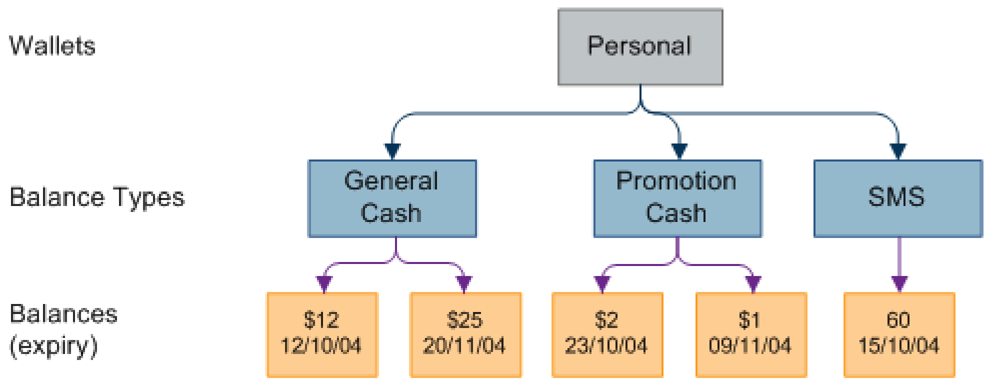
Balances

Balances record a value in a wallet by collecting buckets into a group. Each bucket records a specific value with an optional expiry date. There are two main types of balances:

- Chargeable balances which record value that can be used for services
- Internal balances which are used for internal values and triggers within the system

Wallets, balances and buckets relationship

This diagram shows how buckets are collected into a balance value, and balances are connected to a wallet.



Note: Internal balances have the same structure.

Wallet and bucket events

When beVWARS receives a request which involves a wallet, it will load the wallet and all the wallet's buckets. This includes loading a wallet for a query, update or charging operation. When the wallet is loaded, wallet load event plug-ins are triggered. These event plug-ins will take any required actions on the wallets and buckets as necessary.

Background processing

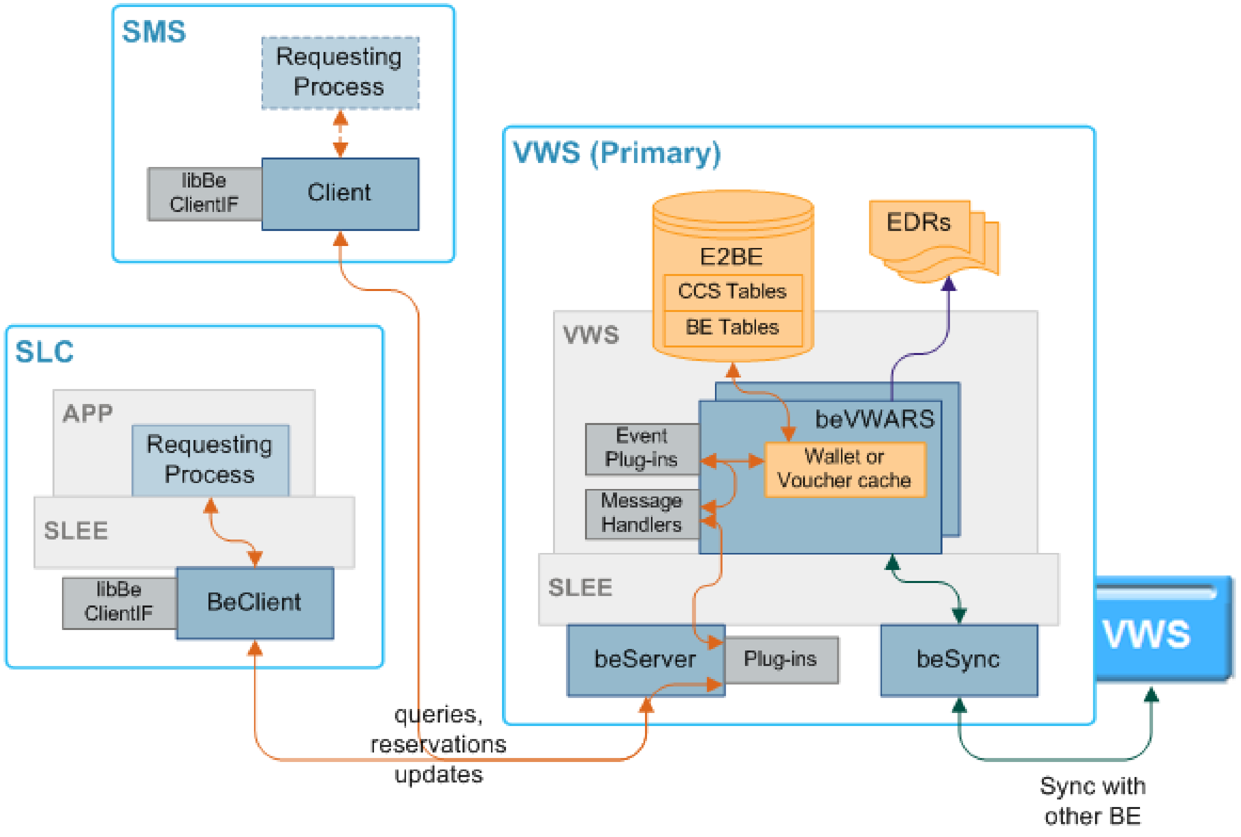
In normal processing, events are triggered only when a subscriber or customer care representative interacts with the wallet. Some events (such as expiries and periodic charges) should be triggered regardless of whether the wallet has been used by a subscriber or customer care representative. In order to process these events, beGroveller sends lists of wallet IDs to beVWARS for processing. This processing triggers any events which are due to occur in the same way a normal interaction would, except wallet events triggered from beGroveller lists do not trigger any message handlers.

For more information about how wallets and buckets are expired, see *Wallet and Bucket Expiry* (on page 23).

Request handling

Reservations and billing diagram

This diagram shows how VWS handles requests.



beVWARS plugins

beVWARS plug-ins can trigger on any event which requires reading data about a wallet or voucher (including maintenance through SMS UI, call charges, or named events).
Because the plug-ins run before the original request is processed, any action such as expiring a bucket or wallet is run before any charges are applied. This means expired wallets or buckets are never available, even if they still exist in the database.

Applications which are using the VWS for charging or voucher redemption usually provide beVWARS plug-ins to handle the specific application logic required by the application. VWS provides these plug-ins as standard:

- *beVWARSExpiry* (on page 117)
- *beVWARSMergeBuckets* (on page 121)
- *libbeEventFactory* (on page 130)

Request processing

This table describes how VWS handles requests from service applications.

Note: For information about how CCS handles charging for calls or SMS and the CCS plug-ins mentioned in the process, see *CCS Technical Guide*.

Stage	Description
1	Requesting process sends a request to the BeClient process (usually BeClient on the SLC, but also PIbeClient and ccsBeOrb on the SMS). **Note:** Other applications and specific functions can use other BeClient processes, but the ones mentioned are the most common.
2	BeClient process checks for plug-ins to handle this message. Plug-ins are specified in the `plugins` (on page 57) parameter in **eserv.config**. **Example:** *libBeClientIF* (on page 129) will usually be triggered in addition to any application-specific plug-ins such as libccsClientPlugins which applies CCS logic.
3	If the message has a BE ID of 0, libclientBcast will send the request to all VWS Voucher and Wallet Servers to locate the Voucher and Wallet Server which holds the details which are relevant to the request. If the message has a BE ID other than 0, BeClient will send the request to the VWS pair with that id. **Note:** The BE ID of a VWS is set by the `serverId` (on page 42) parameter. **Example:** If the request is a voucher redeem, the libclientBcast library will cause the BeClient to send a request to all Voucher and Wallet Server pairs to locate the Voucher and Wallet Server pair which holds the data for the voucher which is about to be redeemed.
4	beServer receives the request from the BeClient process via FOX over TCP/IP. It determines the message type and checks whether there are any handlers for this message. Handlers are configured in the `handlers` (on page 69) parameter in **eserv.config**.
5	beServer creates a context to store information for the request. The information includes: • The original request • The BeClient • Any message handler that is handling the request • The state the message handler is in • The beVWARS process which will handle the request
6	beServer checks for and runs routing plug-ins configured in the `messageRoutingPlugins` (on page 71) parameter in **eserv.config**. **Note:** If this message is part of a sequence (but not the first), beServer will send the message to the same beVWARS as the other parts of the sequence.
7	beServer forwards the message across the SLEE to the correct beVWARS.
8	beVWARS determines whether there is a message handler for this message type. Message handlers are configured in the `handlers` (on page 98) parameter in **eserv.config**. **Note:** If there is no message handler for this type, beVWARS will log an error.
9	When the initial message handler is triggered, it will query either the wallet or the voucher cache. If the cache does not contain the details or the details in the cache are stale, beVWARS reads the details from the database.
10	Depending on the message type, different event and message plug-ins will be triggered. Plug-ins (including event handlers) are configured in the `plugins` (on page 99) parameter in **eserv.config**.

Stage	Description
	Example: If a wallet is interacted with, beVWARSExpiry will check the expiry dates on all buckets in the wallet, and will expire any buckets which are overdue. For more information about expiry handling, see *Wallet and Bucket Expiry* (on page 23). **Note:** At least one plug-in must be provided to process requests from an application which is using VWS for charging. For more information about the specific plug-ins which are triggered, what order they are triggered in, and what they do, see the technical guide for that application.
11	For updates and reservations (but not queries), beVWARS starts the synchronization process by writing the change to the sync files. Synchronization enables the VWS to replay operations in the event of a failure and maintain redundancy in the event of a failure. For more information about how the synchronization process works, see Data synchronization. For updates only, beVWARS also flushes the data. It: • Updates the wallet cache from the database • Updates the database (this update will be held in a queue until the next flush of COMMITs to the E2BE database) • Writes the changes to the EDR file (these updates are also queued for bulk writing) **Note:** The EDRs will be written by the beVWARS which processed the update. This avoids duplicate EDRs being written in the event of a failover. For more information about how beVWARS writes data, see *Queuing and flushing updates* (on page 19). For more information about EDRs, see *EDR Processing* (on page 29).
12	beVWARS on the primary VWS sends the acknowledgment back to BeClient via beServer.
13	BeClient passes the acknowledgment back to the requesting process.

Wallet and voucher caches

beVWARS maintains a wallet cache and a voucher cache to store up to date information about the wallets and vouchers it maintains. beVWARS updates the cache record for a wallet or voucher whenever one of the following occurs:

- Wallet or voucher is queried
- Wallet reservation or update is received
- Voucher is redeemed

A wallet or voucher record is removed from the cache if the record for it expires before a new request for that record is retrieved.

Supported requests

This table describes the types of messages VWS supports.

Message Type	Code	Description
Initial Reservation	IR	Reserve a charge amount.
Subsequent Reservation	SR	Reserve another charge amount.
Commit Reservation	CR	Apply reserved charge.
Revoke Reservation	RR	Abandon a reservation.

Message Type	Code	Description
Named Event	NE	Attempt to charge a named event.
Named Event Rate	NER	Used by Named Event feature node.
Apply Tariffed Charge	ATC	Used by DUCR feature node.
Initial Named Event Reservation	INER	Attempt to reserve a named event.
Subsequent Named Event Reservation	SNER	Named event reservation which follows on from an Initial Events Reservation (INER).
Confirm Named Event Reservation	CNER	Apply reserved named event.
Revoke Named Event Reservation	RNER	Abandon a named event reservation.
Voucher Information	VI	Query a voucher.
Voucher Update	VU	Update voucher details.
Voucher Redeem	VR	Reserves a voucher.
Commit Voucher Redeem	CVR	Wallet changed successfully, redeem voucher.
Revoke Voucher Redeem	RVR	Abandon a voucher reservation.
Voucher Redeem Wallet	VRW	Tells BeClient to start a Voucher Redemption.
Voucher Type Recharge	VTR	Recharge a wallet using a voucher type name.
Voucher Type Recharge Confirm	VTRC	Perform product type swap.
Wallet General Recharge	WGR	Recharge wallet and buckets.
Wallet Update	WU	Update wallet details (not buckets/balances).
Wallet Create	WC	Create a new wallet.
Wallet Delete	WD	Delete an existing wallet.
Wallet Information	WI	Query a wallet and its buckets/balances.
Bad PIN	BPIN	Increase the Bad PIN balance.
Reload the MFile	LDMF	Reload an updated MFile.

walletDeleteBufferSize

Syntax:	`walletDeleteBufferSize = `*`num`*
Description:	The number of wallet deletes in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`walletDeleteBufferSize = 1000`

walletIds

Syntax:	`walletIds = `*`[ID1`*`, `*`ID2`*`, ...]`
Description:	List of subscriber wallet ids we want to trace.
Type:	Array, Integer
Optionality:	Optional if beClient parameter supplied, mandatory if beClient not supplied.
Allowed:	Any valid wallet ID.

Default:	None
Notes:	To obtain the wallet id(s) for a given CLI/subscriber use the showCLI.sh script on the BE where tracing is to occur.
Example:	`walletIds = [ 382, 385 ]`

```
walletIds = [
   382,
   385
]
```

`walletLowWaterMark`

Syntax:	`walletLowWaterMark = num`
Description:	The number of outstanding wallet IDs to grovel, before sending a request to beGroveller for another batch of wallet IDs to grovel.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	100
Notes:	
Example:	`walletLowWaterMark = 100`

Wallets

Each subscriber account is linked to one or two wallets.

A wallet is a group of balances owned by the subscriber and available to pay for prepaid services offered by the platform.

Example: A subscriber could have a “General Cash” balance and a “Free Notification” balance in their wallet. Each balance has its own expiry date, which means that any value left in the balance after this date will be removed.

`writerIfName`

Syntax:	`writerIfName = "name"`
Description:	The SLEE name for the VWS component - beWriter.
Type:	String
Optionality:	
Allowed:	
Default:	"beWriter"
Notes:	For more information about the SLEE, see *SLEE Technical Guide*.
Example:	`writerIfName = "beWriter"`

XmlTcap Parameters

beServiceTrigger/XmlTcap is configured by the following parameters from the `triggering` section in the **eserv.config** file on the VWS:

```
triggering = {
    Control_Plan = "cpname"
    Service_Handle = "handle"
    scps = [ "ip:port" ]
    }
    triggering = {
        Control_Plan = "Reward"
        Service_Handle = "CCS_BPL"
        CCSNamespace = "http://eng-prf-zone01-z1/wsdls/ON/CCSNotifications.wsdl"
        edr = false
        scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
        osd_scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
        failureRetryTime = 60
        storageInterface = beEventStorageIF
        triggerInterface = beServiceTrigger
        operationSet = CMX ON
        operation = Invoke OSD
        responseTag = Result
        maxRatePerUAS = 0
        throttleLife = 30
        timeBetweenThrottles = 10
        tcpTxMaxBuf = 262144
        tcpRxMaxBuf = 131072
}   # triggering
```

Example eserv.config configuration

This is an example of the `BE` section of an **eserv.config** file (comments have been removed). It is not intended to be used in a production environment, but only to illustrate the configurations available.

Details on the configurations in this file are located in various sections later in this guide.

```
BE = {
    serverId = 11
    amPrimary = true
    oracleUserAndPassword="/"
    beLocationPlugin = "libGetccsBeLocation.so"

    timerIfName = "Timer"

    enableGrovelling = true

    freeDiskSpaceWarningThres = 100
    freeDiskSpaceShutdownThres = 10
    freeDiskSpaceCheckInterval = 300
    lowDiskSpaceNotificationInterval = 30

    beServer = {
        clientSelectTime = 1000000
        quiesceLength = 100000
        serverPortOverride = 1500
        clientSocketBufferSize = 10240
        enableStatistics = true
        errorOnRecordStatistics = false
        maxDownstreamQueueLength = 1000
        downstreamOverloadSleepUSec = 100000
        dbConnCheckTime = 5
        recoveryReportInterval = 60
        shutdownDelayTime = 4
        startupRetryPeriodSeconds = 2

        notEndActions = [
            {type="IR  ", action="ACK "}
            {type="SR  ", action="ACK "}
            {type="SR  ", action="NACK"}
            {type="INER", action="ACK "}
            {type="SNER", action="ACK "}
            {type="SNER", action="NACK"}
        ]
    handlers = [
        "libbeServerPingPlugin.so"
    ]

    messageRoutingPlugins = [
        "libbeMsgRouterDefault.so"
    ]

    msgRouterDefault = {
        roundRobinTypes = [
            "VI  "
        ]
        routeOnVoucherNumber = true
    }

    purge = {
        purgeInterval = 300
        vwarsTimeout = 10
        expectedKeep = 60
        noExpectedKeep = 3600
    }

    routingVoucherNumberLength = 10

} # BE.beServer

beVWARS = {

    voucherReservationPeriodSeconds = 120
    useTimeFromClient = true
    maxTransactionsPerSet = 7
    maxOpenDialogTime = 5.0
    maxDownstreamQueueLength = 10000
    downstreamOverloadSleepUSec = 100000
    minResyncReservationLength = 5
    createBucketExpiryDays = 30
    pluginSkipTimeOnStartup = 30
    gapBeforeRestartingPluginSkip = 60
    clearEmptyBuckets = true

    walletCache = {
        maxSize = 10000
        maxLoopSize = 500
        checkBeforeFlush = false
    }

    voucherCache = {
        maxLoopSize = 500
        checkBeforeFlush = false
        flushPeriodSeconds = 60 # -1
        maxSize = 2
        voucherRevokeOnTimeout = true
            # when a voucher reservation is expired, revokes it if set to true
            # this takes precedence over voucherCommitOnTimeout
        voucherCommitOnTimeout = false
            # when a voucher reservation is expired, commits it if set to true
            # however voucherRevokeOnTimeout takes precedence if set
    }

    groveller = {
        periodMsec = 1200
        requestHighWaterMark = 1
```

```
            walletLowWaterMark = 100
            requestTimeout = 300
            peerDatabaseLogin = ""
            peerWalletCheckRetrySeconds = 60
        }

        duplicateDetection = {
            keepDirectSeconds = 60.0
            keepSyncSeconds = 60.0
            directMaxDelaySeconds = 1.0
            syncMaxDelaySeconds = 1.0
        }

        setLastActivationDateStates = [
            [PREU]
        ]

        plugins = [
            "beVWARSExpiry.so"
        ]

        handlers = [
            "beVWARSCCDRHandler.so"

        ]

        syncWriter = {
            maxRecordsPerFile = 100
            maxSecondsPerFile = 2
        }

        dbWriter = {
            flushPeriod = 10
            cdrOutputDirectory = "/IN/service_packages/E2BE/logs/CDR"

            balanceCreateBufferSize = 1000
            balanceUpdateBufferSize = 1000
            balanceDeleteBufferSize = 1000
            bucketCreateBufferSize = 1000
            bucketUpdateBufferSize = 1000
            bucketDeleteBufferSize = 1000
            walletCreateBufferSize = 1000
            walletUpdateBufferSize = 1000
            walletDeleteBufferSize = 1000
            voucherCreateBufferSize = 1000
            voucherUpdateBufferSize = 1000
            voucherDeleteBufferSize = 1000
        }
        tracing = {
            enabled = true
            debugLevel = "all"
            walletIds = [
                <walletid1>,
                <walletid2>
            ]
            beClients = [
                "<Beclient1>",
                "<Beclient2>"
            ]
        }

    } # BE.beVWARS

    beVWARSExpiry = {
        expireNegativeBuckets = false
        removeEmptyBuckets = false
        expireBucketsForExpiredWallets = false
        expireAtMidnightTZ = "Asia/Vladivostok"
    }

    beVWARSMergeBuckets = {
        maxBuckets = -1
        triggerPlugins = false
    }

    beCDRMover = {
        oracleService = ""
        oracleUser = "e2be_admin"
        oraclePassword = "password"
        outDirectory = "/IN/service_packages/E2BE/logs/CDR"
        destinationDirectory = "/IN/service_packages/E2BE/logs/CDR-out"
        timeout = 4
        numberOfRecordsToCommit = 10
        commitTimeSeconds = 10
    } # BE.beCDRMover

    cmnPushFiles = {
        CDR = [
            "-d", "/IN/service_packages/E2BE/logs/CDR-out"
            "-r", "/IN/service_packages/CCS/logs/CDR-in"
            "-h", "smp1hostname"
            "-F"
        ]
    }

    beSync = {
        shared = {
            noWorkSleepTime = 0.2

            spoolDirectory = "/IN/service_packages/E2BE/sync"
            spoolChunkSize = 16
            badFileDirectory = "/IN/service_packages/E2BE/tmp"

            maxDownstreamQueueLength = 10000
            downstreamOverloadSleepUSec = 100000
        }

        sink = {
            inSyncThresholdSeconds = 5
            inSyncReportingPeriodRecords = 10000
            maxSecsToWaitForRemoteOperations = 5
            retryConnectionDelaySeconds = 30
            maxRetriesBeforeStart = 5
            localUpdateChunkSize = 100
            heartbeatPeriodSeconds = 10
        }

        source = {
            recordSendingChunkSize = 50
            maxQueueLength = 50
        }
    }

    BeClient = {
        clientName = "scpClient"
        heartbeatPeriod = 10000000
        maxOutstandingMessages = 100
        connectionRetryTime = 2

        plugins = [
            {
                config="",
                library="libclientBcast.so",
                function="makeBroadcastPlugin"
            }
        ]

        notEndActions = [
                {type="IR  ", action="ACK "}
                {type="SR  ", action="ACK "}
                {type="SR  ", action="NACK"}
                {type="INER", action="ACK "}
                {type="SNER", action="ACK "}
                {type="SNER", action="NACK"}
        ]

        billingEngines = [
            {
                id = 1,
                primary   = { ip="123.123.123.123", port=1500 },
                secondary = { ip="123.123.123.124", port=1500 }
            }
        ]
    }

    beGroveller = {
        quorumHost = "produsms-cluster"
        maxIDsPerResponse = 160
        retrySeconds = 60
        processExpiredBuckets = true
        noProcessingTimes = [
            { startsAt = "06:00", endsAt = "09:30" }
            { startsAt = "11:30", endsAt = "14:00" }
            { startsAt = "16:00", endsAt = "21:00" }
        ]
        connectionRetryTime = 60
        heartbeatPeriod = 300000000
        filledBufferThreshold = 480
        ludProcessingTime = "14:04"
    }

    triggering = {
        Control_Plan = "Reward"
        Service_Handle = "CCS_BPL"
        CCSNamespace = "http://eng-prf-zone01-z1/wsdls/ON/CCSNotifications.wsdl"
        edr = false
        scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
        osd_scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
        failureRetryTime = 60
        storageInterface = beEventStorageIF
        triggerInterface = beServiceTrigger
        operationSet = CMX ON
        operation = Invoke OSD
        responseTag = Result
        maxRatePerUAS = 0
        throttleLife = 30
        timeBetweenThrottles 10
        tcpTxMaxBuf = 262144
        tcpRxMaxBuf = 131072
    } # triggering
}
```

Wallet life cycle period checks

Before processing a request, the request handler will check that the corresponding feature in the current wallet life cycle plan period is enabled. If the feature is disabled for the subscriber's wallet, then the request will fail.

This table lists the request handler and the feature it checks for when processing requests.

Request Handler	Wallet Life Cycle Period Features Checked
IR	Session Charge
IARR	Session Charge
NE	General Charge and First Named Event Class if the amount is greater than or equal to 0 (debit) General Recharge and First Named Event Class if the amount is less than 0 (credit)
INER	Session Charge and First Named Event Class
ATC	General Charge if the amount is greater than or equal to 0 (debit) General Recharge if the amount is less than 0 (credit)
DA	General Charge if the amount is greater than or equal to 0 (debit) General Recharge if the amount is less than 0 (credit)
WGR	General Recharge

Merging wallets processes

This table describes the main components involved in merging wallets.

Process	Role	Further information
beServer	Handles connections from client processes (including BeClient processes) and controls routing to beVWARS processes.	*beServer* (on page 65)
beVWARS	beVWARS is the core of the VWS. More than one beVWARS will usually be running on a VWS. It: • Reads and caches wallet and voucher information from the E2BE database • Manages all queries, reservations and updates against wallets • Manages all queries, redemptions and state changes for vouchers • Writes EDRs.	*beVWARS* (on page 94)
beVWARSMergeBuckets	This beVWARS plug-in merges buckets in the same balance when there are too many buckets in the wallet.	*beVWARSMergeBuckets* (on page 121) *beVWARS plugins* (on page 7)
E2BE database	The databases on the VWSs. They hold a subset of the data from the SMF.	NA

Data Management

Data redundancy

In a redundant configuration there are two Voucher and Wallet Servers in each VWS domain: a primary and a secondary. The primary is the node with:

- 'true' specified in the *amPrimary* (on page 41) parameter
- The highest node number

In normal conditions, the primary VWS performs all subscriber account, wallet and balance actions for the pair. The secondary VWS maintains a duplicate set of data.

If a single Voucher and Wallet Server in a pair is down, the system will work as normal. When the other peer comes back up:

1 It will resynchronize with the uninterrupted peer without prompting
2 Service will continue as if nothing happened

If the network link between BeClients and beServers, or between peer servers in a redundant pair, is disconnected, those BeClients that can see at least one member of a VWS pair should be able to keep running. When connectivity is restored, changes made to records held on the peers are resynchronized.

If the primary VWS fails, the secondary VWS performs the functions of a primary VWS until the primary VWS becomes available again.

For more information about failover and recovery, see *BE States* (on page 26).

BeClients and connection failure

If a BeClient process cannot connect to the beServer on the primary VWS, it will retransmit any outstanding messages to the secondary VWS for processing. Subsequent messages will go to the secondary VWS, until the primary VWS recovers. When the primary VWS recovers, BeClient sends new transactions to the primary VWS.

This prevents call crossover conditions, where the beginning of a call could start on one VWS and end on another.

Throttling

If one beVWARS process is throttling, the beServer will stop accepting any new requests on the client sockets.

Database update consistency

The E2BE database seldom reflects the complete state of the running system, because updates are almost always pending. To maintain a single consistent view of the state of records in the database, all wallet or voucher accesses are processed through beVWARS. beVWARS is responsible for all updates to resynchronized database fields.

Because beVWARS sends updates to beSync before confirmation of the update on the local VWS has happened, updates can be applied to the remote VWS and not the local VWS. If the local VWS fails before updates have been applied, then the updates are retrieved and applied during resynchronization.

beVWARS data updates

Each beVWARS performs the following data update tasks:

- Buffering database updates and EDRs (for performance reasons)
- Applying database updates to the database in the order they were produced
- Writing EDR data to flat files

- Writing updates to sync files
- Informing beSync there is a sync file to process

Queuing and flushing updates

Updates fall into one of these categories:

- Bucket creations, updates or deletions
- Balance creations, updates or deletions
- Wallet creations, updates or deletions
- Voucher creations, updates or deletions

A buffer is maintained for each of these categories to store updates for later binding to the database via a bulk bind operation. Each buffer has a configurable maximum size.

beVWARS also maintains an EDR cache to queue EDR details for later writing to disk.

Each time beVWARS processes a transaction, it checks the following criteria to determine if it should perform a flush:

- One of the buffers is full
- The configurable buffer/cache flush period has been reached
- The writer subsystem is told to flush and commit (on shutdown, for example)

If a flush is triggered, all buffers are written to the E2BE database and the EDR cache is written to the EDR files.

Flush process

This describes the stages involved in a flush operation.

Stage	Description
1	beVWARS writes all EDR records in the cache to a new file.
2	beVWARS records the EDR file name in the E2BE database (BE_CDR_FILE).
3	Update the BE_VWARS_SEQ_NUM table entry for the current beVWARS process. Sets: • 'last local sequence number' to the sequence number of the last update in the buffers, and • 'remote sequence number' to the last value sent from the remote beSync).
4	Bulk bind and run the database statement associated with each database buffer. The buffers are flushed in this order: • BE_WALLET, BE_BALANCE, BE_BUCKET, then BE_VOUCHER Inserts • BE_WALLET, BE_BALANCE, BE_BUCKET, then BE_VOUCHER Updates • BE_BUCKET, BE_BALANCE, BE_WALLET, then BE_VOUCHER Deletes
5	Commit the changes to the E2BE database.
6	Generate and send a COMMIT message to the beSync process containing the last local and remote sequence numbers.
7	Update the last committed local update sequence number.
8	Perform a wallet cache flush. This frees up space in the wallet cache by releasing any entries which were protected until a flush applied the transaction.

Changing number of beVWARS

Because of the method used to keep the VWS pairs synchronised, you must run the same number of beVWARS on both Voucher and Wallet Server nodes.

Follow these steps to change the number of beVWARS to run on the VWS.

Note: If you reduce the number of beVWARS on a pair, any transactional updates that are incomplete when the change is made will be lost.

Step	Action14.25
1	Set the number of beVWARS interface instances to run by editing the INTERFACE line for beVWARS in **SLEE.cfg** for each VWS in the pair.
2	Stop the SLEE on both VWSs in the pair. The VWS will move to the disabled state.
3	Start the SLEE on both VWSs in the pair. • The SLEE will start the number of beVWARS instances specified in the updated **SLEE.cfg** file. • The VWS will move to a running state.

For more information about configuring SLEE interfaces, stopping the SLEE, and starting the SLEE, see *SLEE Technical Guide*.

Synchronization

Data synchronization

Synchronization is used to ensure database updates and EDRs are not lost. This is achieved by beVWARS recording a stream of updates to flat files, so the updates can:

- Be replayed in the event of a failure (for example, a hardware fault, power failure, software failure)
- Provide a persistent stream of updates to be delivered to the secondary VWS for application to a remote E2BE database in the interests of redundancy and failover

It is possible for the remote beSync to drop the connection and later connect and request older updates at any point. The sync files are only removed when both VWSs have committed the updates.

The current position in the transaction stream for each VWS within a pair is recorded as a pair of sequence numbers. These numbers record the last locally sourced update and the last remotely sourced update received and written to the database.

Sequence numbers are managed for each beVWARS, so lost updates are visible as gaps in the sequence numbers for the beVWARS handling that traffic.

Synchronization between two VWSs in a VWS domain can be viewed as two continuous streams containing all reservations, updates, and deletions made on one VWS to the other. The two streams are connections from:

1 Primary beSync source to secondary beSync sink.
2 Secondary beSync sink to primary beSync source.

The two stream connections between the beSync processes on the primary and secondary VWS nodes are maintained using the internal port that you specify when you configure the VWS domain. If you are running more than one instance of beSync on the VWS nodes, then two stream connections will be maintained for each instance. The internal port number is incremented by 1 (one) for each additional instance of beSync.

For information on configuring VWS domains, see the section on Service Management in *CCS User's Guide*.

Example

If you are running two instances of beSync and the internal port number is 1500 then the connections between:

- beSync0 on VWS1 and beSync0 on VWS2 use port 1500

- beSync1 on VWS1 and beSync1 on VWS2 use port 1501

Sync files

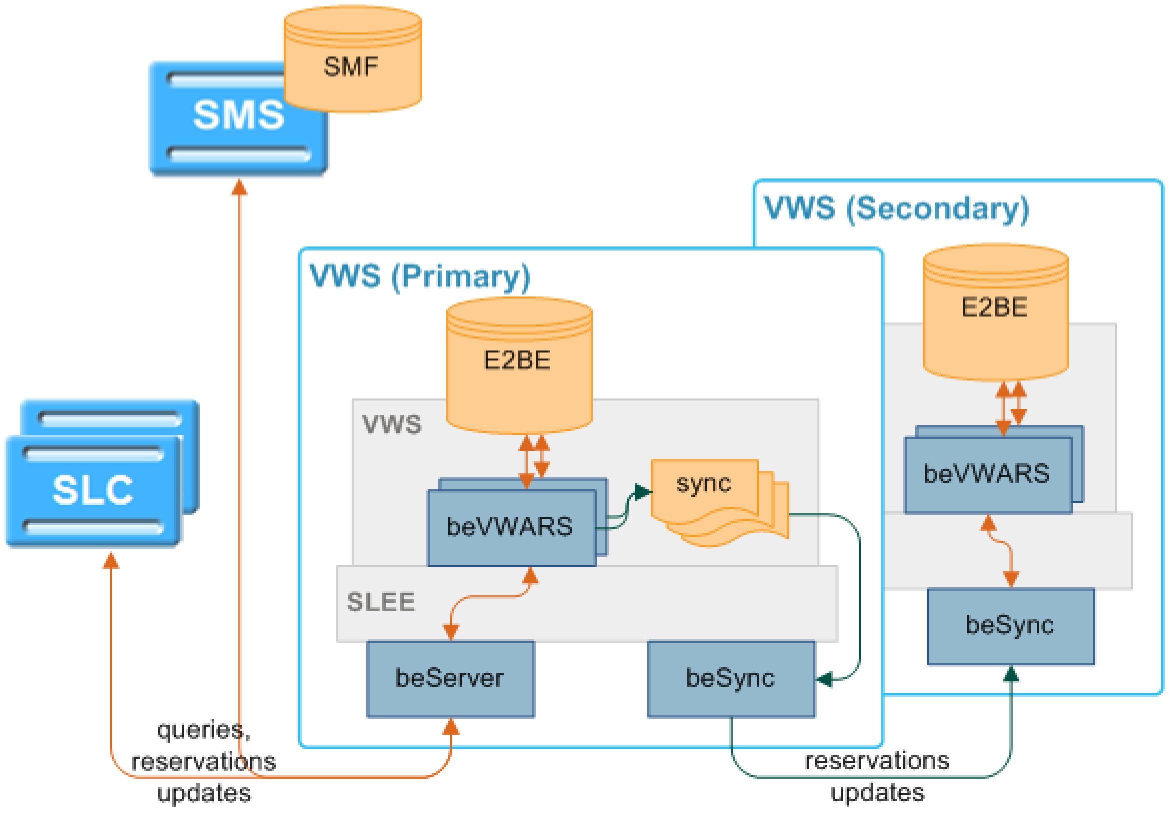
Files are named with the form "sync-*VWARS*-YYYYMMDDHHMMSS-*UUU*" where:

- UUU is a unique number used to differentiate when two or more files are produced in a second
- VWARS is the beVWARS number to distinguish between beVWARS instances.

The files are stored in numbered directories within the main sync file directory. The name of the subdirectories follows the beVWARS number which wrote the sync file. This is to aid the beSync during recovery of individual beVWARS instances.

Synchronization diagram

This diagram shows the processes and data involved in the synchronization.



Synchronization process

This table describes how the VWS keeps the Voucher and Wallet Servers in a pair in sync during an update or reservation.

Note: This process starts at the point the beVWARS has triggered all the configured event plug-ins and message handlers and is ready to start a reservation or update.

Stage	Description
1	For each reservation or update, beVWARS: • Updates the cache

Stage	Description
	• Writes the update or reservation to the sync file • Updates the sync file sequence number If there is no sync file, the beVWARS creates one.
2	If the request is an update, the beVWARS on the primary VWS writes the EDR to record the transaction.
3	beVWARS closes its sync file after a configurable period or a configurable number of entries, or force-closes it (if a commit message is received). It writes the current transaction set to the file and performs a file flush on the current file to ensure the last transaction set is written safely to disk. beVWARS sends a new file notification to beSync containing the name of the file.
4	beSync receives the notification and starts to read the new sync file. The notification contains the name of the sync file and the beVWARS number. The beSync process queues received sync file names, so it knows the order in which to process them without performing expensive directory searching operations.
5	The beSync sink function on the secondary VWS opens a TCP connection to the corresponding source function on the beSync on the primary VWS and requests updates.
6	The beSync source function on the primary VWS listens on a defined TCP port, for connections from the corresponding beSync sink. When it receives a request, it reads the updates and reservations from a transaction set in the sync file and sends them to beSync on the secondary VWS. A transaction set is a specific sequence number range from a single beVWARS's sync file. It then waits for another request. **Note:** If the number of commands exceeds a configured maximum, outstanding updates are queued until the number is reduced. This helps to prevent surges of activity that may stress the VWSs and delay the synchronization.
7	beSync on the secondary VWS sends the update or reservation to a local beVWARS process.
8	The beVWARS on the secondary VWS updates the: • Cache • Sync file sequence number The beVWARS on both the primary and secondary VWSs flush the reservations or updates. They: • Update the database • Send a COMMIT message to the local beSync.
9	The beSync on the secondary VWS sends the COMMIT to the beSync on the primary VWS.
10	The beSyncs on both VWSs verify the updates and reservations in their sync files against the updates and reservations in the COMMIT message(s) they have received.
11	When all the updates and reservations in a sync file have been verified by a COMMIT message, beSync deletes the file.

Resynchronizations

This table describes the stages involved in resynchronizations.

Note: The running VWS refers to the Voucher and Wallet Server which has been running as a primary. It also refers to the other VWS if the two Voucher and Wallet Servers have been running in isolation. The recovering VWS refers to the Voucher and Wallet Server which has been disabled.

Stage	Description
1	If a resynchronization is triggered, the beSync on the recovering VWS queries a local beVWARS for its last update numbers (both local and remote update numbers).
2	beSync on the recovering compares the sequence number from the local beVWARS with the last local update sequence number to check whether any local updates have been missed. If the numbers do not match, beSync streams all local uncommitted transactions to the recovering beVWARS.
3	When all local updates have been sent, beSync sends a “request all reservations” message to the beSync on the running VWS specifying which beVWARS instance to update.
4	The running VWS sends updated beVWARS context and reservations for the recovering beVWARS number from the beServer on the running VWS to the beServer on the recovering VWS via the beSync processes.
5	The beVWARS on the running VWS indicates all contexts have been sent correctly and the beSync on the recovering VWS requests the beSync on the recovering VWS to start streaming updates.
6	As remote updates are received by the beSync on the recovering VWS, they are delivered to the appropriate beVWARS instance for application to the database. When streamed updates are close enough to real-time relative to the running VWS, which is still actively processing traffic, the beSync process will notifies the recovering beVWARS to move into running state.
7	After the recovering beVWARS has been enabled, it sends a message to the beServer to move into running state.

For more information about the different failure scenarios which can trigger a resynchronization, see *Failure scenarios* (on page 145).

Wallet and Bucket Expiry

Introduction

Like most functions, wallet expiry and bucket expiry and removal are triggered when a wallet is loaded. VWS uses beVWARSExpiry to control when wallet expiry events are triggered. Additional wallet expiry processing can be done by plug-ins and processes provided by other applications. VWS provides basic bucket handling, though this functionality can be extended by plug-ins which are triggered on bucket expiry or bucket delete/removal events.

Note: Expiry handling is optional. If no expiry dates are configured for wallets and/or buckets, no expiry handling will be run.

Wallet management processes

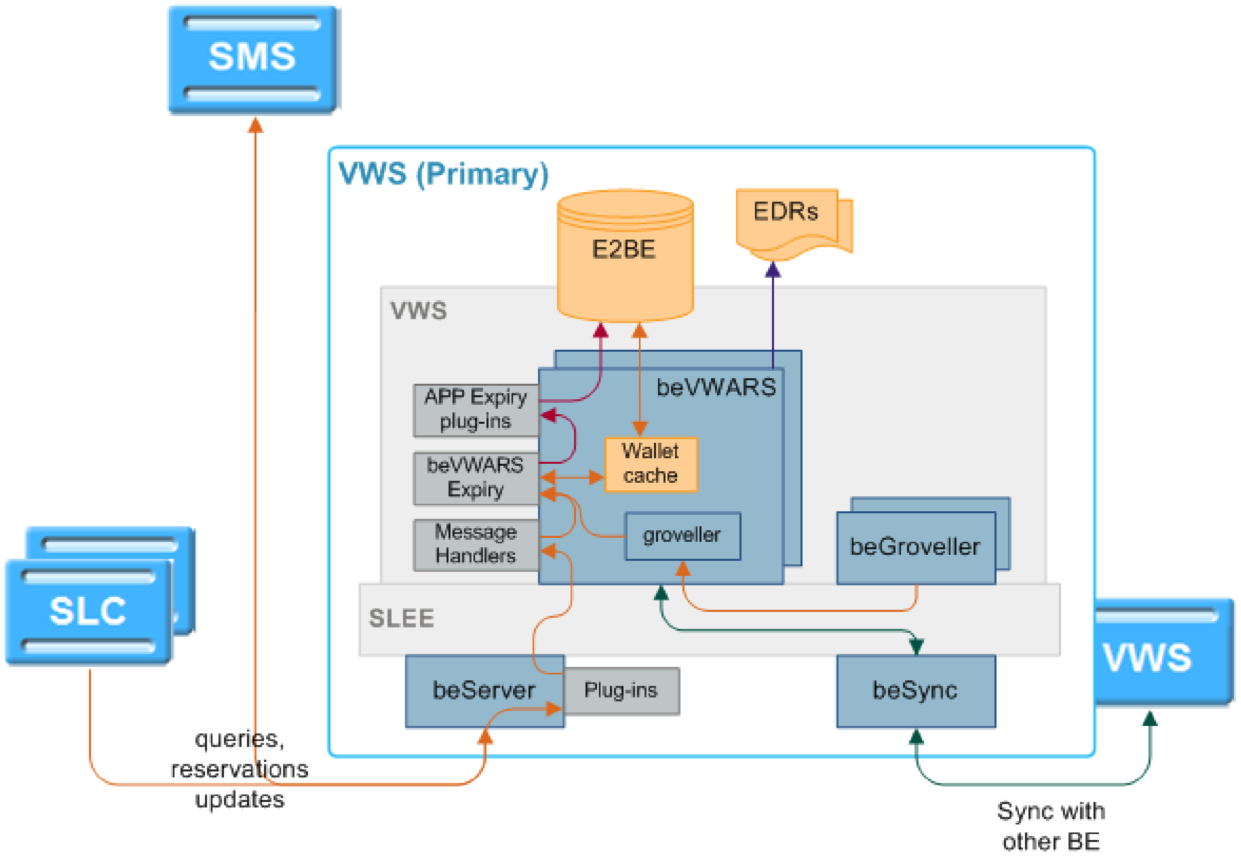
This table describes the main components in VWS.

Process	Role	Further information
beServer	Handles connections from client processes (including BeClient processes) and controls routing to beVWARS processes.	*beServer* (on page 65)

Process	Role	Further information
beGroveller	beGroveller triggers processing on wallets which have not been triggered by a subscriber action. This enables VWS to ensure required actions are taken against all wallets and buckets.	*beGroveller* (on page 59)
beVWARS	beVWARS is the core of the VWS. More than one beVWARS will usually be running on a VWS. It does the following: • Reads and caches wallet and voucher information from the E2BE database • Manages all queries, reservations and updates against wallets • Manages all queries, redemptions and state changes for vouchers • Writes sync files • Writes EDRs	*beVWARS* (on page 94)
beVWARSExpiry	beVWARSExpiry monitors wallets and buckets, checking for wallets and buckets which have passed their expiry date. If it finds a wallet or bucket which requires expiring, it processes the record as configured and triggers any Expiry plug-ins with a Wallet Event or Bucket Event.	*beVWARSExpiry* (on page 117) *beVWARS plugins* (on page 7)
Expiry plug-ins	beVWARSExpiry starts an expiry event when it finds an expired wallet or bucket. Each expiry event can trigger one or more expiry plug-ins. Each expiry plug-in will take its own action.	Technical guide for the application using VWS. For an example, see *CCS Technical Guide*.
E2BE database	The databases on the VWSs. They hold a subset of the data from the SMF.	NA

Expiry diagram

This diagram shows the basic processes, communication and relationships for expiring wallets and buckets.



Wallet and bucket expiry processing

This table describes how wallets and buckets are expired when they have passed their expiry date.

Stage	Description
1	*beVWARS* (on page 94) loads a wallet. Loading a wallet can be triggered by one of the following: • A query, request or reservation from a requesting process as part of normal processing • beVWARS processing a wallet from a list of wallet IDs to grovel from *beGroveller* (on page 59).
2	Loading the wallet triggers *beVWARSExpiry* (on page 117).
3	For wallets which are Active or Dormant, beVWARSExpiry checks whether the wallet has passed its expiry date. **Tip:** Wallet expiry date checking is defined by `expireAtMidnightTZ` (on page 118). If `expireAtMidnightTZ` is set, beVWARSExpiry will expire the wallet and buckets the next time they are loaded after the midnight in the specified time zone which follows the expiry date.

Stage	Description
	If the wallet has not expired, beVWARSExpiry checks whether any of the buckets in the wallet have passed their expiry date. For each bucket which has passed its expiry date, beVWARSExpiry deletes the bucket. If two buckets expire at exactly the same time, the buckets will be processed in bucket ID order. This triggers any beVWARS event plug-ins which are designed to handle Bucket Expiry events. **Note:** If a bucket is expired, it will be deleted unless an event plug-in provides specific logic which retains the bucket. **Example:** ccsVWARSPeriodicCharge processes expiring periodic charge buckets. It keeps the periodic charge bucket and sets the expiry date to a point in the future.
4	If the wallet has passed its expiry date, bevWARSExpiry sets the wallet's state to terminated and fires a Wallet Expiry event. This triggers any beVWARS event plug-ins which are designed to handle Wallet Expiry events. Expiry plug-ins which handle Wallet Expiry events can be provided as part of another application such as CCS. **Example:** When triggered by a Wallet Expiry event, ccsVWARSExpiry writes an EDR and a adds the wallet to a list of expired wallets which is used to update the HLR records.
5	beVWARSExpiry checks the configuration. If `expireBucketsForExpiredWallets` (on page 119) is set to true, it deletes all buckets with a positive or zero value. If `expireNegativeBuckets` (on page 119) is also set to true, beVWARSExpiry will also delete buckets with negative values. This triggers any beVWARS event plug-ins which are designed to handle Bucket Expiry events.
6	If `removeEmptyBuckets` (on page 120) is set to true, beVWARSExpiry deletes all buckets with a 0 balance from the E2BE. **Exception:** If the last bucket in a wallet which has not expired has a value of 0, that bucket will be left. If the parameter `clearEmptyBuckets` is set to false, `removeEmptyBuckets` flag will be disabled.

For more information about the expiry plug-ins provided with other applications, see the application's technical guide.

BE States

Introduction

In a VWS domain, there are two Voucher and Wallet Servers, in a redundant configuration.

If one VWS in a pair is down, the system will work as normal. When the other peer comes back up:

1 It will resynchronize with the uninterrupted peer without prompting
2 Service will continue as if nothing happened.

If the network link between BeClients and beServers (or between VWSs in a domain) is disconnected, the BeClients that can see at least one member of a VWS pair should be able to keep running. When connectivity is restored, changes made to records held on the VWSs are resynchronized.

BE states

The beVWARS is responsible for maintaining the current state of a Voucher and Wallet Server. There are three possible states for a VWS:

- Running

- Recovering
- Disabled

Running

This is the normal state of a VWS.

In this state:

- beSync is streaming updates and receiving streamed updates
- beServer is accepting connections from BeClients and processing requests

beSync listens on a defined TCP port, for connections from the beSync on the peer VWS. When this connection is open and streaming, all reservations and transactions are sent to the other VWS.

Disabled

This is the initial state of a VWS, and it can return to this state in a variety of failure scenarios.

In this state:

- beServer does not accept any connections from BeClients
- beSync does not accept any connections from the peer VWS
- The beGroveller does not run
- No internal processing is performed

beVWARS failure

When the SLEE watchdog notices a beVWARS process has failed, the beServer will:

- Cease to read new work from the client sockets
- Allow the remaining active beVWARS instances to quiesce
- Close all client connections (when all beVWARS instances are idle)

When all connections are closed, the BeClient processes will failover to the secondary Voucher and Wallet Server. The local beServer removes all existing context and beVWARS routes for the failed beVWARS. These are recovered during the beVWARS recovery, which delivers all context from the remote VWS.

Recovering

In this state:

- The beGroveller does not run
- Synchronization can be in any state
- beServer should not be accepting new connections from the BeClients.

beServer begins in a recovery state expecting to receive all of its contexts from the beServer on the peer, and getting the OK from all local beVWARS indicating they are in sync before accepting client connects and client requests.

On VWS recovery/startup, the local sync files are processed to ensure there are no lost local updates. Then a connection is made to the peer VWS, to request all updates since the last remote update received. For more information about this process, see *Resynchronizations* (on page 22).

If one VWS is disabled for an extended period of time, its peer will amass a significant number of updates in the sync directory specified by *spoolDirectory* (on page 79) (typically in the **/IN/service_packages/E2BE/sync/** directory). When the VWS is re-enabled, the updates will be requested and the VWSs will return to a synchronized state.

beVWARS recovery

Individual beVWARS processes recover independently. If a beVWARS process fails, the other beVWARS processes do not detect this, and continue to function (though, due to the beServer disabling connections, they will only be processing remote transactions from the now active secondary VWS).

The failed beVWARS independently goes through its recovery process along with the beSync process, until it is able to move back into running state. The beServer asks the remote beServer for all contexts for the beVWARS which failed.

State transitions

Here is what a user can expect to see, in the transition of a Voucher and Wallet Server from one state to another.

Note: The initial state of the VWS should be disabled.

Transition	Description
Disabled to Recovering	beServer should prepare to accept contexts. beVWARS should prepare to send contexts to the beServer, and should prepare to receive Operations. This should include a complete new Reservation load, so all existing reservations should be erased. beVWARS will not request work from beGroveller. beGroveller will not return wallet IDs for groveling to beVWARS. beSync should initiate recovery.
Disabled to Running	Not a possible transition.
Recovering to Running	beServer should start accepting connections. beVWARS can start to accept new requests and can start to send requests to beGroveller for lists of wallet IDs to grovel. beGroveller will determine if it should run. If it should, it will start to return wallet IDs to grovel to beVWARS. For more information about the beGroveller, see *beGroveller* (on page 59). beSync should proceed as it was (it usually leads the recovery process).
Recovering to Disabled	beServer should terminate all connections. beVWARS should disable the beGroveller. beVWARS should stop requesting work from beGroveller. beGroveller will stop accepting requests from beVWARS for wallet IDs to grovel. an inactive state.
Running to Recovering	beServer should terminate all connections. beVWARS should stop requesting work from beGroveller. beGroveller will stop accepting requests from beVWARS for wallet IDs to grovel. beSync should disconnect open connections and initiate recovery.
Running to Disabled	beServer should terminate all connections. beVWARS should stop requesting work from beGroveller. beGroveller will stop accepting requests from beVWARS for wallet IDs to grovel. beSync should close all open connections and return to an inactive state.

EDR Processing

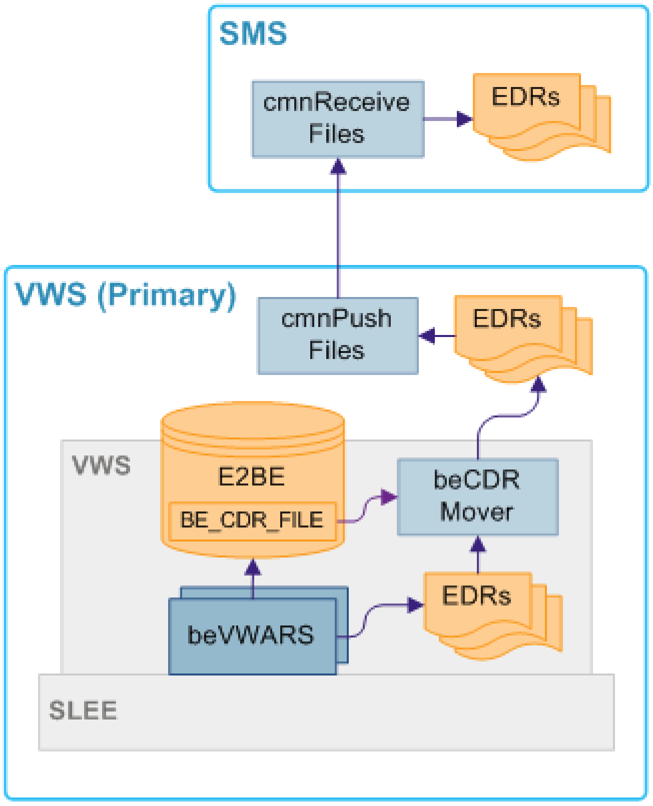
Introduction

Each Voucher and Wallet Server in a domain logs EDRs for all actions which are successfully completed on the local VWS.

EDR processing diagram

This diagram shows how EDRs are processed by VWS.

Note: EDRs can be post-processed on the SMS.



VWS EDR processing

This process describes how VWS processes EDRs.

Stage	Description
1	beVWARS receives an update request from the local beServer.
2	beVWARS updates the relevant cache and queues the EDR write until the next flush. For more information about queuing and flushing, see *Queuing and flushing updates* (on page 19).
3	When the next flush is triggered, beVWARS: • Writes all queued EDR records to a new EDR file • Records the EDR file name in the E2BE database (in the BE_CDR_FILE table).

Stage	Description
	Note: Entering the EDR file name in the BE_CDR_FILE table indicates that the EDRs in the file should be: • Accepted by the rest of the system • Transmitted to the SMS for consolidation into the SMF database.
4	beCDRMover moves completed EDR files from the working directory to the output directory.
5	cmnPushFiles transfers the EDR file to the SMS.

EDR triggers

EDRs are written on the Voucher and Wallet Servers when a wallet or voucher is modified. The following messages, among others, cause the beVWARS to write EDRs:

- Call End Notification
- Wallet Recharge Request
- Named Event

MFile Updates

Introduction

The MFile contains a subset of the Voucher and Wallet Server data, used to reduce network traffic on the system. Some of the information held within the VWS changes less frequently, such as Tariffs. It is this data which is copied to the MFile and held on the VWS. The system reads this MFile, enabling it to retrieve data quickly, thereby reducing network traffic to the Voucher and Wallet Servers.

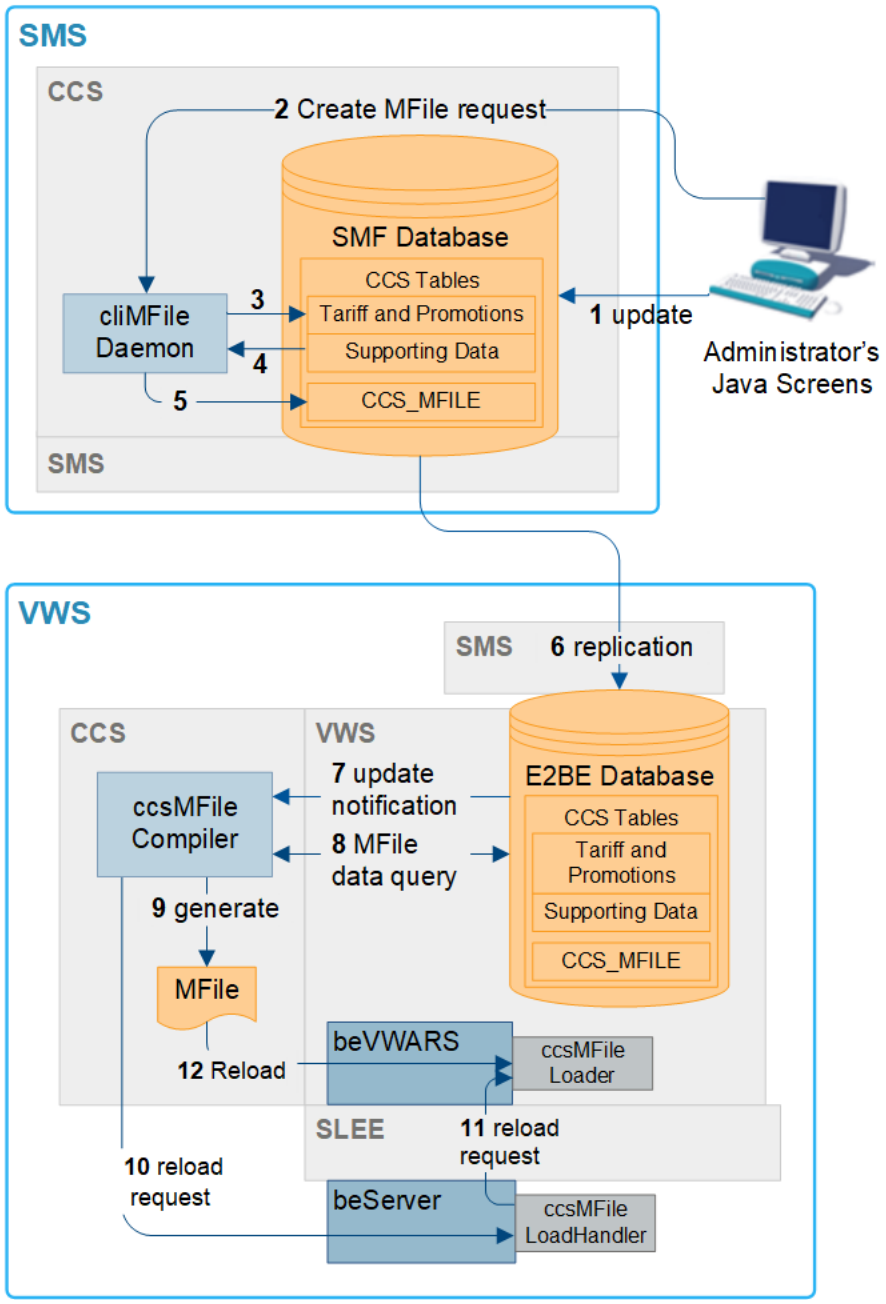
MFile data types

A MFile will need to be recompiled if any of the following data types are changed:

- Discount Period
- Discount Sets
- Geography Sets
- Billing Periods
- CLIxDN Mappings
- Tariff Plans
- Product Types
- Tariff Plans
- Currency

Update process diagram

Here is an example showing an update to an MFile.



Update process - mfile

This table describes the process through which MFiles are updated.

Stage	Description
1 - 4	Through the Prepaid Charging user interface (UI) on the SMS node, the system administrator updates the details contained in the MFile and clicks **Save**.
5	A new entry is added to the CCS_MFILE table in the SMF database.

Stage	Description
6	The relevant tables in the SMF are updated and the data is transferred to the VWS nodes using replication.
7	When the new CCS_MFILE entry arrives on the BE, VWS sends a notification to the ccsMFileCompiler.
8	The ccsMFileCompiler updates the MFile file name table (CCS_MFILE) in the BE database.
9	ccsMFileCompiler then generates a new MFile from the updated data in the E2BE database.
10	If ccsMFileCompiler has not already connected to the beServer, it uses the `beLocationPlugin` (on page 41) to extract the location of the beServer from the BE database. After establishing the connection, or if it is already connected, ccsMFileCompiler sends a request to the beServer to reload the MFile.
11	The ccsMFileLoadHandler message handler on the beServer forwards the reload request to the ccsMFileLoader message handler in beVWARS.
12	beVWARS uses ccsMFileLoader to reload the new MFile.

For more information about the ccsMFile processes, see *CCS Technical Guide*.

Statistics

Introduction

VWS statistics are generated by each VWS VWS, and then transferred at periodic intervals to the SMS for permanent storage and analysis.

VWS also records statistics for applications which use the VWS, such as CCS. For more information about these statistics, see the application's technical guide.

SMS statistics subsystem

The statistics system provided by SMS provides the functionality which collects the statistical events logged by VWS processes.

For more information about the SMS statistics subsystem, see *SMS Technical Guide*.

Collected statistics

This table describes the statistics produced by VWS processes.

Statistic	Description
NUM_TOTAL_REQ	Total number of requests sent to the VWS.

Note: All statistics are collected with a period of 1800 seconds.

For more information about the request messages these statistics measure, see *Supported requests* (on page 9).

Chapter 2

Configuration

Overview

Introduction

This chapter explains how to configure the Oracle Communications Network Charging and Control (NCC) application.

In this chapter

This chapter contains the following topics.

Configuration Overview

Introduction

This topic provides a high level overview of how the VWS application is configured.

Note: There are several configuration options that are not explained in this chapter. These options should not be changed by the user without first consulting Oracle for technical support.

Configuration process overview

This table describes the steps involved in configuring a VWS for the first time.

Stage	Description
1	The environment that VWS will run in must be configured correctly. This includes: • If the directory VWS was installed into was not the recommended directory (**/IN/service_packages/E2BE**), setting the root directory • Configuring the location of the EDR directories
2	The **eserv.config** file must be configured for the following machines: • SMS nodes • SLC nodes • VWS nodes The example file should be copied into the main **eserv.config**, and any mandatory parameters configured. The parameters which must be set are listed at the top of the **eserv.config** file. For more information, see **eserv.config** *Configuration* (on page 34).
3	The screen-based configuration tasks must be completed through the CCS User Interface (UI).
4	The **SLEE.cfg** file must contain references to the VWS SLEE applications and interfaces.

Configuration components

VWS is configured by the following components:

Component	Locations	Description	Further Information
eserv.config	all SMSs and VWSs	VWS is configured by the `BE` section of **eserv.config**.	*eserv.config Configuration* (on page 34)
SLEE.cfg	all VWSs	**SLEE.cfg** sets up SLEE interfaces and applications.	*SLEE.cfg* (on page 45)
Domains screen	SMF database	Domains must be set up which define the Voucher and Wallet Servers, and available services in the Domain screen in the CCS UI.	*User Interface-Based Configuration Tasks* (on page 44)

Configuring the Environment

Oracle environment variables

The VWS UNIX system account ebe_oper requires the standard ORACLE environment variables to be present.

eserv.config Configuration

Introduction

The **eserv.config** file is a shared configuration file, from which many Oracle applications read their configuration. Each Oracle machine (SMS, SLC and VWS) has its own version of the configuration file, containing configuration relevant to that machine. The configuration file contains many different parts or sections; each application reads the parts of the **eserv.config** file that contains data relevant to it.

It is located in the following directory:

/IN/service_packages/

The **eserv.config** file format allows hierarchical groupings, and most applications make use of this to divide up the options into logical groupings.

Configuration File Format

To organize the configuration data within the **eserv.config** file, some sections are nested within other sections. Configuration details are opened and closed using either { } or [ ].

- Groups of parameters are enclosed with curly brackets – { }
- An array of parameters is enclosed in square brackets – [ ]
- Comments are prefaced with a # at the beginning of the line

To list things within a group or an array, elements must be separated by at least one comma or at least one line break. Any of the following formats can be used, as in this example:

```
{ name="route6", id = 3, prefixes = [ "00000148", "0000473"] }
{ name="route7", id = 4, prefixes = [ "000001049" ] }
```

or

```
{ name="route6"
```

```
            id = 3
            prefixes = [
                "00000148"
                "0000473"
            ]
        }
        { name="route7"
            id = 4
            prefixes = [
                "000001049"
            ]
        }
```

or

```
        { name="route6"
            id = 3
            prefixes = [ "00000148", "0000473" ]
        }
        { name="route7", id = 4
            prefixes = [ "000001049" ]
        }
```

eserv.config Files Delivered

Most applications come with an example **eserv.config** configuration in a file called **eserv.config.example** in the root of the application directory, for example, **/IN/service_packages/eserv.config.example**.

Editing the File

Open the configuration file on your system using a standard text editor. Do not use text editors, such as Microsoft Word, that attach control characters. These can be, for example, Microsoft DOS or Windows line termination characters (for example, ^M), which are not visible to the user, at the end of each row. This causes file errors when the application tries to read the configuration file.

Always keep a backup of your file before making any changes to it. This ensures you have a working copy to which you can return.

Loading eserv.config Changes

If you change the configuration file, you must restart the appropriate parts of the service to enable the new options to take effect.

Example eserv.config configuration

This is an example of the `BE` section of an **eserv.config** file (comments have been removed). It is not intended to be used in a production environment, but only to illustrate the configurations available.

Details on the configurations in this file are located in various sections later in this guide.

```
BE = {
    serverId = 11
    amPrimary = true
    oracleUserAndPassword="/"
    beLocationPlugin = "libGetccsBeLocation.so"

    timerIfName = "Timer"

    enableGrovelling = true

    freeDiskSpaceWarningThres = 100
    freeDiskSpaceShutdownThres = 10
```

```
freeDiskSpaceCheckInterval = 300
lowDiskSpaceNotificationInterval = 30

beServer = {
    clientSelectTime = 1000000
    quiesceLength = 100000
    serverPortOverride = 1500
    clientSocketBufferSize = 10240
    enableStatistics = true
    errorOnRecordStatistics = false
    maxDownstreamQueueLength = 1000
    downstreamOverloadSleepUSec = 100000
    dbConnCheckTime = 5
    recoveryReportInterval = 60
    shutdownDelayTime = 4
    startupRetryPeriodSeconds = 2

    notEndActions = [
        {type="IR  ", action="ACK "}
        {type="SR  ", action="ACK "}
        {type="SR  ", action="NACK"}
        {type="INER", action="ACK "}
        {type="SNER", action="ACK "}
        {type="SNER", action="NACK"}
    ]

    handlers = [
        "libbeServerPingPlugin.so"
    ]

    messageRoutingPlugins = [
        "libbeMsgRouterDefault.so"
    ]

    msgRouterDefault = {
        roundRobinTypes = [
            "VI  "
        ]
        routeOnVoucherNumber = true
    }

    purge = {
        purgeInterval = 300
        vwarsTimeout = 10
        expectedKeep = 60
        noExpectedKeep = 3600
    }

    routingVoucherNumberLength = 10

} # BE.beServer

beVWARS = {

    voucherReservationPeriodSeconds = 120
    useTimeFromClient = true
    maxTransactionsPerSet = 7
    maxOpenDialogTime = 5.0
    maxDownstreamQueueLength = 10000
    downstreamOverloadSleepUSec = 100000
    minResyncReservationLength = 5
    createBucketExpiryDays = 30
    setLastUseDateOnActivation = true
    maxSendReservationsToSync = 1000
```

```
clearEmptyBuckets = true
pluginSkipTimeOnStartup = 30
gapBeforeRestartingPluginSkip = 60

walletCache = {
    maxSize = 10000
    maxLoopSize = 500
    checkBeforeFlush = false
}

voucherCache = {
    maxLoopSize = 500
    checkBeforeFlush = false
    flushPeriodSeconds = 60 # -1
    maxSize = 2
    voucherRevokeOnTimeout = false
        # when a voucher reservation is expired, revokes it if set to true
        # this takes precedence over voucherCommitOnTimeout
    voucherCommitOnTimeout = false
        # when a voucher reservation is expired, commits it if set to true
        # voucherRevokeOnTimeout takes precedence if set
}

groveller = {
    periodMsec = 1200
    requestHighWaterMark = 1
    walletLowWaterMark = 100
    requestTimeout = 300
    peerDatabaseLogin = ""
    peerWalletCheckRetrySeconds = 60
}

duplicateDetection = {
    keepDirectSeconds = 60.0
    keepSyncSeconds = 60.0
    directMaxDelaySeconds = 1.0
    syncMaxDelaySeconds = 1.0
}

setLastActivationDateStates = [
    [PREU]
]

plugins = [
    "beVWARSExpiry.so"
]

handlers = [
    "beVWARSCCDRHandler.so"

]

syncWriter = {
    maxRecordsPerFile = 100
    maxSecondsPerFile = 2
}

dbWriter = {
    flushPeriod = 10
    cdrOutputDirectory = "/IN/service_packages/E2BE/logs/CDR"

    balanceCreateBufferSize = 1000
    balanceUpdateBufferSize = 1000
```

```
            balanceDeleteBufferSize = 1000
            bucketCreateBufferSize = 1000
            bucketUpdateBufferSize = 1000
            bucketDeleteBufferSize = 1000
            walletCreateBufferSize = 1000
            walletUpdateBufferSize = 1000
            walletDeleteBufferSize = 1000
            voucherCreateBufferSize = 1000
            voucherUpdateBufferSize = 1000
            voucherDeleteBufferSize = 1000
        }
        tracing = {
            enabled = true
            debugLevel = "all"
            walletIds = [
                <walletid1>,
                <walletid2>
            ]
            beClients = [
                "<Beclient1>",
                "<Beclient2>"
            ]
        }

    } # BE.beVWARS

    beVWARSExpiry = {
        expireNegativeBuckets = false
        removeEmptyBuckets = false
        expireBucketsForExpiredWallets = false
        expireAtMidnightTZ = "Asia/Vladivostok"
    }

    beVWARSMergeBuckets = {
        maxBuckets = -1
        triggerPlugins = false
    }

    beCDRMover = {
        oracleService = ""
        oracleUser = "e2be_admin"
        oraclePassword = "password"
        outDirectory = "/IN/service_packages/E2BE/logs/CDR"
        destinationDirectory = "/IN/service_packages/E2BE/logs/CDR-out"
        timeout = 4
        numberOfRecordsToCommit = 10
        commitTimeSeconds = 10
    } # BE.beCDRMover

    cmnPushFiles = {
        CDR = [
            "-d", "/IN/service_packages/E2BE/logs/CDR-out"
            "-r", "/IN/service_packages/CCS/logs/CDR-in"
            "-h", "smp1hostname"
            "-F"
        ]
    }

    beSync = {
        shared = {
            noWorkSleepTime = 0.2

            spoolDirectory = "/IN/service_packages/E2BE/sync"
            spoolChunkSize = 16
        badFileDirectory = "/IN/service_packages/E2BE/tmp"

        maxDownstreamQueueLength = 10000
        downstreamOverloadSleepUSec = 100000
    }

    sink = {
        inSyncThresholdSeconds = 5
        inSyncReportingPeriodRecords = 10000
        maxSecsToWaitForRemoteOperations = 5
        retryConnectionDelaySeconds = 30
        maxRetriesBeforeStart = 5
        localUpdateChunkSize = 100
        heartbeatPeriodSeconds = 10
    }

    source = {
        recordSendingChunkSize = 50
        maxQueueLength = 50
    }
}

BeClient = {
    clientName = "scpClient"
    heartbeatPeriod = 10000000
    maxOutstandingMessages = 100
    connectionRetryTime = 2

    plugins = [
        {
            config="",
            library="libclientBcast.so",
            function="makeBroadcastPlugin"
        }
    ]

    notEndActions = [
            {type="IR  ", action="ACK "}
            {type="SR  ", action="ACK "}
            {type="SR  ", action="NACK"}
            {type="INER", action="ACK "}
            {type="SNER", action="ACK "}
            {type="SNER", action="NACK"}
    ]

    billingEngines = [
        {
            id = 1,
            primary   = { ip="123.123.123.123", port=1500 },
            secondary = { ip="123.123.123.124", port=1500 }
        }
    ]
}

beGroveller = {
    quorumHost = "produsms-cluster"
    maxIDsPerResponse = 160
    retrySeconds = 60
    processExpiredBuckets = true
    noProcessingTimes = [
        { startsAt = "06:00", endsAt = "09:30" }
        { startsAt = "11:30", endsAt = "14:00" }
        { startsAt = "16:00", endsAt = "21:00" }
        ]
        connectionRetryTime = 60
        heartbeatPeriod = 300000000
        filledBufferThreshold = 480
        ludProcessingTime = "14:04"
    }

    triggering = {
        Control_Plan = "Reward"
        Service_Handle = "CCS_BPL"
        CCSNamespace = "http://eng-prf-zone01-z1/wsdls/ON/CCSNotifications.wsdl"
        edr = false
        scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
        osd_scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
        failureRetryTime = 60
        storageInterface = beEventStorageIF
        triggerInterface = beServiceTrigger
        operationSet = CMX ON
        operation = Invoke OSD
        responseTag = Result
        maxRatePerUAS = 0
        throttleLife = 30
        timeBetweenThrottles 10
    } # triggering
}
```

BE Shared Parameters

Purpose

The BE section of the **eserv.config** file for the VWS contains parameters that are shared by various VWS background processes. These parameters define the settings that are common to the background processes.

Configuration

VWS accepts these parameters from **eserv.config**.

```
    serverId = int
    amPrimary = true|false
    oracleUserAndPassword="/"
    beLocationPlugin = "lib"

    timerIfName = "str"

    enableGrovelling = true|false

    freeDiskSpaceWarningThres = MB
    freeDiskSpaceShutdownThres = MB
    freeDiskSpaceCheckInterval = secs
    lowDiskSpaceNotificationInterval = secs
```

Example BE shared parameters configuration

The following section sets the shared BE configuration parameters.

Note: The comments have been removed.

```
BE = {
    serverId = 11
    amPrimary = true
    oracleUserAndPassword="/"
beLocationPlugin = "libGetccsBeLocation.so"

timerIfName = "Timer"

enableGrovelling = true

freeDiskSpaceWarningThres = 100
freeDiskSpaceShutdownThres = 10
freeDiskSpaceCheckInterval = 300
lowDiskSpaceNotificationInterval = 30
}
```

Parameters

Here are the available shared VWS parameters.

amPrimary

Syntax:	`amPrimary = true|false`
Description:	True if this is the primary VWS in the pair.
Type:	Boolean
Optionality:	Optional, default used if not set
Allowed:	
Default:	true
Notes:	
Example:	`amPrimary = false`

beLocationPlugin

Syntax:	`beLocationPlugin = "`*`lib`*`"`
Description:	The plug-in library that finds the Voucher and Wallet Server details of the Voucher and Wallet Servers to connect to.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	libGetccsBeLocation.so
Notes:	This library must be in the LD_LIBRARY_PATH.
Example:	`beLocationPlugin = "libGetccsBeLocation.so"`

enableGrovelling

Syntax:	`enableGrovelling = true|false`
Description:	Whether or not to process wallets when spare resources are available.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	• true – Use beGroveller to trigger events on wallets and balances that are not used often. • false – Do not do background triggering of events on wallets and balances.
Default:	true
Notes:	
Example:	`enableGrovelling = true`

`oracleUserAndPassword`

Syntax:	`oracleUserAndPassword = "`*`usr/pwd`*`"`
Description:	The Oracle user and password for the connections to the E2BE database for VWS processes.
Type:	String
Optionality:	Optional, default used if not set
Allowed:	
Default:	"/"
Notes:	The default sets no user and password.
Example:	`oracleUserAndPassword = "/"`

`serverId`

Syntax:	`serverId = `*`id`*
Description:	The ID of the VWS pair.
Type:	Integer
Optionality:	
Allowed:	
Default:	1
Notes:	Set to 1 if this is not a VWS
Example:	`serverId = 11`

`timerIfName`

Syntax:	`timerIfName = "`*`name`*`"`
Description:	The name for the SLEE Timer interface component.
Type:	String
Optionality:	Optional, default used if not set
Allowed:	
Default:	Timer
Notes:	Must match the handle of the timer interface in **SLEE.cfg**. For more information about the SLEE, see *SLEE Technical Guide*.
Example:	`timerIfName = "Timer"`

`freeDiskSpaceCheckInterval`

Syntax:	`freeDiskSpaceCheckInterval = `*`seconds`*
Description:	How often (in seconds) beServer checka whether there is more than *freeDiskSpaceWarningThres* (on page 43) space free on the disk. If there is less than *freeDiskSpaceShutdownThres* (on page 43), beServer closes its connections and stops accepting requests.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	• 0 – Disables the disk space check • Positive Integer – Checks the available disk space at the specified interval
Default:	300
Notes:	If a shutdown is triggered, beServer logs a critical-level error to the syslog.
Example:	`freeDiskSpaceCheckInterval = 300`

`freeDiskSpaceShutdownThres`

Syntax:	`freeDiskSpaceShutdownThres =` *`MB`*
Description:	When free disk space in MB is below this threshold, beServer closes its connections and stops taking new requests. Threshold applies to the partitions containing the directories set by the following parameters: • `spoolDirectory` (on page 79) • `cdrOutputDirectory` (on page 112)
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	• 0 – Disables the threshold shutdown • Positive Integer – The MB threshold
Default:	10
Notes:	When the disk space has come back above the threshold, beServer re-enables and returns to running state. If beServer stops taking requests, it will log a critical-level error to the syslog. Disk space is checked by beServer at the frequency set by `freeDiskSpaceCheckInterval` (on page 42).
Example:	`freeDiskSpaceShutdownThres = 10`

`freeDiskSpaceWarningThres`

Syntax:	`freeDiskSpaceWarningThres =` *`MB`*
Description:	Low disk space threshold, in Megabytes, for the partitions containing the directories set by the following parameters: • `spoolDirectory` (on page 79) • `cdrOutputDirectory` (on page 112)
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	• 0 – Disables the threshold warning • Positive Integer – The MB threshold
Default:	100
Notes:	If the threshold is reached, an error-level warning is logged to the syslog. Disk space is checked by beServer at the frequency set by `freeDiskSpaceCheckInterval` (on page 42).
Example:	`freeDiskSpaceWarningThres = 200`

`lowDiskSpaceNotificationInterval`

Syntax:	`lowDiskSpaceNotificationInterval =` *`seconds`*
Description:	The number of seconds between logging the error triggered by the low disk space warning or shutdown thresholds being triggered.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	30

Notes:	The threshold is set by `freeDiskSpaceWarningThres` (on page 43) and `freeDiskSpaceShutdownThres` (on page 43). The notification interval should be set to a value higher than `freeDiskSpaceCheckInterval` (on page 42), as it will only log an error if the check interval has recorded a low disk space condition since the last error was logged.
Example:	`lowDiskSpaceNotificationInterval = 30`

Deprecated SLEE Name Definitions

The parameters listed in this section have been deprecated and should not be used. You should delete them from the BE section of **eserv.config** if they are currently defined.

`grovellerIfNamePrefix`

Syntax:	`grovellerIfNamePrefix = "`*`name`*`"`
Default:	beGroveller
Example:	`grovellerIfNamePrefix = "beGroveller"`

`serverIfName`

Syntax:	`serverIfName = "`*`name`*`"`
Default:	beServer
Example:	`serverIfName = "beServer"`

`syncIfName`

Syntax:	`syncIfName = "`*`name`*`"`
Default:	beSync
Example:	`syncIfName = "beSync"`

`vwarsIfNamePrefix`

Syntax:	`vwarsIfNamePrefix = "`*`name`*`"`
Default:	beVWARS
Example:	`vwarsIfNamePrefix = "beVWARS"`

User Interface-Based Configuration Tasks

Introduction

These procedures are normally performed only once, after the installation and initial configuration of the system.

For more information about accessing the CCS screens, see *CCS User's Guide*.

Defining VWS locations

The system requires the location of VWS machines to be defined. These are defined using the New Domain or Edit Domain screens, accessed from the Service Management screen.

For more information about configuring domains, see *CCS User's Guide*.

SLEE.cfg

About Configuring VWS SLEE Interfaces

The VWS includes the beVWARS, beSync, and beServer SLEE interfaces that run on the VWS nodes. For these processes to run correctly, they must be configured in the **SLEE.cfg** file. The SLEE is automatically configured during installation to run one or more instances of each by the following lines in **SLEE.cfg:**

```
INTERFACE=beVWARS  beVWARSStartup.sh /IN/service_packages/E2BE/bin instance_count EVENT
INTERFACE=beSync   beSyncStartup.sh  /IN/service_packages/E2BE/bin instance_count EVENT
INTERFACE=beServer beServerStartup.sh /IN/service_packages/E2BE/bin instance_count EVENT
```

Where *instance_count* is the number of instances of the interface process to run.

Note: The actual startup script names can vary.

You should only update this configuration if you want to change the number of instances to run any of these processes. For example, if there is a high volume of traffic on the VWS, you can improve performance by running additional instances of these processes.

For more information about configuring SLEE interfaces, see the discussion about configuring the SLEE in *SLEE Technical Guide*.

About Configuring MAXEVENTS

The value of MAXEVENTS sets the maximum number of event objects that the system can hold in shared memory. If MAXEVENTS is exceeded when the system is running, no more events or calls will be accepted and alarm messages will be sent. This means that you should set MAXEVENTS to a value that is big enough to handle an overload situation. You can estimate this value by using the following formula:

(*num_beServers* * *max_beServer_queue*) + (*num_beVWARS* * *max_beVWARS_queue*) + (*num_beSyncs* * *max_beSync_queue*) + *contingency*

Where:

- *num_beServers* is the number of instances of the beServer interface defined in the **SLEE.cfg** configuration file.
- *max_beServer_queue* is the maximum number of beVWARS response events that can be queued up for the beServer. This is the value specified for the `BE.beVWARS.maxDownstreamQueueLength` parameter in the **eserv.config** configuration file.
- *num_beVWARS* is the number of instances of the beVWARS interface defined in the **SLEE.cfg** configuration file.
- *max_beVWARS_queue* is the value specified in the **eserv.config** configuration file for either `BE.beSync.maxDownstreamQueueLength` or `BE.beServer.maxDownstreamQueueLength`, whichever value is greater.
- *num_beSyncs* is the number of instances of the beSync interface defined in the **SLEE.cfg** configuration file.
- *max_beSync_queue* is the maximum number of beVWARS events that can be queued up for the beSync. This is the value specified for the `BE.beVWARS.maxDownstreamQueueLength` parameter in the **eserv.config** configuration file.
- *contingency* is an estimated value for any additional VWS events such as VWS control messages. A typical value for contingency would be 5000.

Example

This example shows how to calculate the value for MAXEVENTS for one beServer interface, six instances of the beVWARS interface, and two instances of the beSync interface, and where:

- `BE.beVWARS.maxDownStreamQueueLength = 10000`
- `BE.beSync.maxDownStreamQueueLength = 50000`
- `BE.beServer.maxDownStreamQueueLength = 50000`
- contingency = 5000

MAXEVENTS = (1 * 10000) + (6 * 5000) + (2 * 10000) + 5000 = 65000

You configure MAXEVENTS in the **SLEE.cfg** configuration file. For more information about configuring MAXEVENTS, see the discussion about configuring the SLEE in *SLEE Technical Guide*.

Loading SLEE.cfg changes

If you change the **SLEE.cfg** file, you must restart the SLEE to enable the new options to take effect.

For more information about restarting the SLEE, see *SLEE Technical Guide*.

Chapter 3

Background Processes

Overview

Introduction

This chapter explains the processes that are started automatically by Service Logic Execution Environment (SLEE).

Note: This chapter also includes some plug-ins to background processes which do not run independently.

In this chapter

This chapter contains the following topics.

beCDRMover

Purpose

beCDRMover moves completed EDR files from the working directory to a directory from which they are copied to the SMS. The inter-machine transfer is completed by cmnPushFiles.

Startup

This task is started by entry be_1 in the inittab, via the shell script:

```
/IN/service_packages/E2BE/bin/beCDRMoverStartup.sh
```

Note: The above is a default and may vary as per configuration.

Configuration

beCDRMover accepts the following parameters from **eserv.config**.

```
beCDRMover = {
    oracleService = "str"
    oracleUser = "name"
    oraclePassword = "str"
    outDirectory = "dir"
    destinationDirectory = "dir"
    timeout = int
    numberOfRecordsToCommit = num
    commitTimeSeconds = num
}
```

Parameters

Here are the available beCDRMover parameters.

`destinationDirectory`

Syntax:	`destinationDirectory = "`*`dir`*`"`
Description:	The destination directory into which EDRs are moved.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	/IN/service_packages/E2BE/logs/CDR
Notes:	Must be a valid directory
Example:	`destinationDirectory = "/var/edr/UBE/dest"`

`commitTimeSeconds`

Syntax:	`commitTimeSeconds = `*`num`*
Description:	The maximum amount of time, in seconds, to leave transactions uncommitted.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	10
Notes:	
Example:	`commitTimeSeconds = 5`

`numberOfRecordsToCommit`

Syntax:	`numberOfRecordsToCommit = `*`num`*
Description:	EDRs are moved in batches. This parameter defines the number of records in each batch.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	10
Notes:	
Example:	`numberOfRecordsToCommit = 10`

`oraclePassword`

Syntax:	`oraclePassword = "`*`str`*`"`
Description:	The Oracle password VWS processes to connect to the E2BE database.

Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	e2be_admin
Notes:	
Example:	`oraclePassword = "`*password*`"`

oracleService

Syntax:	`oracleService = "`*name*`"`
Description:	The Oracle service.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	""
Notes:	
Example:	`oracleService = ""`

oracleUser

Syntax:	`oracleUser = "`*name*`"`
Description:	The Oracle user that VWS uses to connect to the E2BE.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	"e2be_admin"
Notes:	
Example:	`oracleUser = "e2be_admin"`

outDirectory

Syntax:	`outDirectory = "`*dir*`"`
Description:	The directory from which EDRs are moved.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	IN/service_packages/E2BE/logs/CDR
Notes:	
Example:	`outDirectory = "/var/edr/UBE"`

timeout

Syntax:	`timeout = `*seconds*
Description:	Time (in seconds) before the EDR move is regarded as failed.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	

Default:	4
Notes:	The `timeout` value should be set to 4 seconds or less. If it is set to higher than 4 seconds, multiple "file cannot be deleted" messages will appear in the syslog.
Example:	`timeout = 4`

Example configuration

This is an example of the beCDRMover section of an **eserv.config** file on a VWS (comments have been removed).

```
beCDRMover = {
    oracleService = ""
    oracleUser = "e2be_admin"
    oraclePassword = "password"
    outDirectory = "/IN/service_packages/E2BE/logs/CDR"
    destinationDirectory = "/IN/service_packages/E2BE/logs/CDR-out"
    timeout = 4
    numberOfRecordsToCommit = 10
    commitTimeSeconds = 10
} # BE.beCDRMover
```

Failure

If beCDRMover fails, no EDR files will be moved from the input directory until it is restarted.

Output

The beCDRMover writes error messages to the system messages file, and also writes additional output to:

/IN/service_packages/E2BE/tmp/beCDRMover.log

Note: Above is default and can vary if configured differently to the default values.

BeClient

Purpose

BeClient is a SLEE interface that runs on the SLC. It communicates with the Voucher and Wallet Server using FOX over TCP/IP.

BeClient maintains connections to all primary nodes within all of the configured VWS domains. It switches from the primary to the secondary Voucher and Wallet Server if the TCP connection breaks or if a failure to detect Voucher and Wallet Server heartbeat occurs.

BeClient is designed to be mostly ignorant of the messages it routes. This enables it to be used with enhanced protocols without requiring upgrades.

Startup

The BeClient is a SLEE interface and is started during SLEE initialization by the following line in **SLEE.cfg**:

```
INTERFACE=BeClient BeClientStartup.sh /IN/service_packages/BE/bin/ instance_count
EVENT
```

Where *`instance_count`* is the number of instances to run of the BeCLient process.

Note: If you are running multiple BeClient instances, then each BeClient process will have the value of `instance_count` - 1 appended to its name. So the first BeClient process will be named BeClient0 and subsequent BeClient processes will be named BeClient1, BeClient2 and so on. Nothing will be appended to the process name when you configure only one beClient instance to run.

For more information about starting and stopping BeClient processes, see *SLEE Technical Guide*.

Configuration

In order to operate, BeClient plug-in reads the `BeClient` section of the **eserv.config** file. The `BeClient` section is listed below.

```
BeClient = {

    clientName = "str"

    heartbeatPeriod = microsecs
    messageTimeoutSeconds = seconds
    maxOutstandingMessages = int
    reportPeriodSeconds = seconds
    primaryFailbackIntreval = seconds
    connectionRetryTime = seconds

    plugins = [
        {
            config="confStr",
            library="lib",
            function="str"
        }
        [...]
    ]

    confStr = {
        plugin configuration
    }

    notEndActions = [
        {type="str", action="[ACK |NACK]"}
        [...]
]
    billingEngines = [
        {   id = int,
            primary   = { ip="ip", port=port },
            secondary = { ip="ip", port=port }
        }
        [...]
    ]
}
```

Parameters

BeClient has no command line parameters.

The BeClient supports the following parameters from the `BE` section of **eserv.config**.

`billingEngines`

Syntax:	`billingEngines = [`
	`{ id = int`
	`primary = { ip="ip", port=port },`
	`secondary = { ip="ip", port=port }`
	`}`
	`[...]`
	`]`

Description:	Overrides connection details that `beLocationPlugin` (on page 41) obtains from the database.
Type:	Parameter array.
Optionality:	Optional (`beLocationPlugin` finds connection details if not set).
Allowed:	
Default:	
Notes:	Identifies the Voucher and Wallet Servers and assigns their Internet connection details.

Example:	`billingEngines = [`
	`{ id = 1,`
	`primary = { ip="192.0.2.0", port=1500 },`
	`secondary = { ip="192.0.2.1", port=1500 }`
	`}`
	`]`

`id`

Syntax:	`id = int`
Description:	This unique identifier for this Voucher and Wallet Server configuration.
Type:	Integer
Optionality:	Required, if this section is used
Allowed:	
Default:	
Notes:	This parameter is part of the `billingEngines` parameter array.
Example:	`id = 1`

`primary`

Syntax:	`primary = { ip="ip", port=port }`
Description:	The `primary` parameter group defines the Internet Protocol (IP) address and associated port number of the primary Voucher and Wallet Server.
Type:	Parameter array
Optionality:	Required if this section is used
Allowed:	
Default:	
Notes:	This parameter is part of the `billingEngines` parameter array.

Examples:	`primary = { ip="192.0.2.0", port=1500 }`
	`primary = { ip = "2001:db8:0000:1050:0005:0600:300c:326b", port=1500 }`
	`primary = {ip = "2001:db8:0:0:0:500:300a:326f", port=1500 }`
	`primary = { ip = "2001:db8::c3", port=1500 }`

`secondary`

Syntax:	`secondary = { ip="`*`ip`*`", port=`*`port`*` }`
Description:	The `secondary` parameter group defines the Internet Protocol (IP) address and associated port number of the secondary Voucher and Wallet Server.
Type:	Array
Optionality:	Required, if this section is used
Allowed:	
Default:	
Notes:	This parameter is part of the `billingEngines` parameter array.

Examples:	`secondary = { ip="192.0.2.1", port=1500 }`
	`secondary = { ip = "2001:db8:0000:1050:0005:0600:300c:326b", port=1500 ]`
	`secondary = {ip = "2001:db8:0:0:0:500:300a:326f", port=1500 }`
	`secondary = { ip = "2001:db8::c3", port=1500 }`

`ip`

Syntax:	`ip = "`*`ip`*`"`
Description:	The Internet Protocol (IP) address of the Voucher and Wallet Server.
Type:	String
Optionality:	Required
Allowed:	IP version 4 (IPv4) addresses, IP version 6 (IPv6) addresses
Default:	None
Notes:	This parameter is part of either the primary, or the secondary parameter group of the `billingEngines` parameter array. You can use the industry standard for omitting zeros when specifying IPv6 addresses.

Examples:	`ip = "192.0.2.0"`
	`ip = "2001:db8:0000:1050:0005:0600:300c:326b"`
	`ip = "2001:db8:0:0:0:500:300a:326f"`
	`ip = "2001:db8::c3"`

`port`

Syntax:	`port = `*`port`*
Description:	The port number associated with the address of the Voucher and Wallet Server.
Type:	Integer
Optionality:	Required
Allowed:	
Default:	None
Notes:	This parameter is part of either the primary or secondary parameter group of the `billingEngines` parameter array.
Example:	`port = 1500`

broadcastOptions

Syntax:	`broadcastOptions = {`
	`    aggregateNAckCodes = [`*`config`*`]`
	`}`

Description:	Name of configuration section for the BeClient Broadcast plug-in libclientBcast.
Type:	Parameter array
Optionality:	
Allowed:	
Default:	
Notes:	libclientBcast is used by a range of processes which connect to the beServer, including: • BeClient • PIbeClient • ccsBeOrb For more information about libclientBcast, see `libclientBcast` (on page 130).

Example:	`broadcastOptions = {`
	`    aggregateNAckCodes = [ ]`
	`}`

aggregateNAckCodes

Syntax:	`aggregateNAckCodes = [`
	`    "NVOU"`
	`]`

Description:	When this parameter is set, the BeClient waits for a response from all the VWS pairs in use and filters the responses from the broadcast request using the configured NAck codes.
Type:	Parameter array
Optionality:	
Allowed:	NVOU
Default:	
Notes:	When a voucher recharge request is broadcast, this ensures that all the available VWS pairs are checked for the required voucher before a voucher not found message is returned to the requesting process.
Example:	

clientName

Syntax:	`clientName = "`*`name`*`"`
Description:	The unique client name of the process.
Type:	String
Optionality:	Required
Allowed:	Must be unique.
Default:	The host name of the local machine.
Notes:	The server generates clientId from a hash of *`str`*. If more than one client attempts to connect with the same name, then some connections will be lost. This parameter is used by libBeClientIF.
Example:	`clientName = "scpClient"`

`connectionRetryTime`

Syntax:	`connectionRetryTime = `*`seconds`*
Description:	The maximum number of seconds the client process will wait for a connection to succeed before attempting a new connection.
Type:	Integer
Optionality:	Required
Allowed:	
Default:	5
Notes:	This parameter is used by libBeClientIF.
Example:	`connectionRetryTime = 2`

`heartbeatPeriod`

Syntax:	`heartbeatPeriod = `*`microsecs`*	
Description:	The number of microseconds during which a Voucher and Wallet Server heartbeat message must be detected, or the BeClient process will switch to the other VWS in the pair.	
Type:	Integer	
Optionality:	Required	
Allowed:	0	Disable heartbeat detection.
	positive integer	Heartbeat period.
Default:	3000000	
Notes:	1 000 000 microseconds = 1 second. If no heartbeat message is detected during the specified time, client process switches to the other Voucher and Wallet Server in the pair. This parameter is used by libBeClientIF.	
Example:	`heartbeatPeriod = 10000000`	

`maxOutstandingMessages`

Syntax:	`maxOutstandingMessages = `*`num`*
Description:	The maximum number of messages allowed to be waiting for a response from the Voucher and Wallet Server.
Type:	Integer
Optionality:	Required
Allowed:	
Default:	If this parameter is not set, the maximum is unlimited.
Notes:	If more than this number of messages are waiting for a response from the Voucher and Wallet Server, the client process assumes the Voucher and Wallet Server is overloaded. In this event, the client process refuses to start new calls but continues to service existing calls. The messages are queued until the Voucher and Wallet Server has reduced its outstanding load. This parameter is used by libBeClientIF.
Example:	`maxOutstandingMessages = 100`

messageTimeoutSeconds

Syntax:	`messageTimeoutSeconds = `*`seconds`*
Description:	The time that the client process will wait for the server to respond to a request.
Type:	Integer
Units:	Seconds
Optionality:	Required
Allowed:	1-604800 Number of seconds to wait.
	0 Do not time out.
Default:	2
Notes:	After the specified number of seconds, the client process will generate an exception and discard the message associated with the request. This parameter is used by libBeClientIF.
Example:	`messageTimeoutSeconds = 2`

notEndActions

Syntax:

```
notEndActions = [
    {type="str", action="[ACK|NACK]"}
    [...]
]
```

Description:	The `notEndActions` parameter array is used to define the messages associated with dialogs that should not have their dialog closes, because the dialog is closed by default. This facilitates failover.
Type:	Parameter array.
Optionality:	Required
Allowed:	
Default:	
Notes:	If the incoming dialog for a call closes and the last response received was of the notEndActions type, the client process sends an ABRT message. The ABRT message allows the VWS to remove the reservation. An example of this situation would be where slee_acs has stopped working. This parameter is used by libBeClientIF. For more information about slee_acs, see *ACS Technical Guide*.

Example:

```
notEndActions = [
    {type="IR ", action="ACK "}
    {type="SR ", action="ACK "}
    {type="SR ", action="NACK"}
    {type="INER", action="ACK "}
    {type="SNER", action="ACK "}
    {type="SNER", action="NACK"}
]
```

plugins

Syntax:	`plugins = [ { config="" library="lib" function="str" } ... ]`

```
plugins = [
     {
         config=""
         library="lib"
         function="str"
     }
     ...
 ]
```

Description: Defines any client process plug-ins to run. Also defines the string which maps to their configuration section.

Type: Parameter array

Optionality: Optional (as plug-ins will not be loaded if they are not configured here, this parameter must include any plug-ins which are needed to supply application functions; for more information about which plug-ins to load, see the `BeClient` section for the application which provides the BeClient plug-ins).

Allowed:

Default: Empty (that is, do not load any plug-ins).

Notes: The libclientBcast plug-in must be placed last in the plug-ins configuration list.

For more information about the libclientBcast plug-in, see *libclientBcast* (on page 130).

This parameter is used by libBeClientIF.

Example:

```
plugins = [
     {
         config="broadcastOptions"
         library="libclientBcast.so"
         function="makeBroadcastPlugin"
     }
 ]
```

primaryFailbackInterval

Syntax: `primaryFailbackInterval = seconds`

Description: *seconds* defines the failback interval. If the number of seconds since the VWS sent the last request for a session running on the secondary BE is greater than the specified failback interval, then all subsequent requests for the session will be sent to the primary BE. During the failback interval, the secondary BE will synchronize requests to the primary BE.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

- 0 – For immediate failback
- -1 – To disable primary failback
- A positive integer

Default: -1

Notes: Setting this parameter will not affect failover behavior. A session will failover to the other BE if a communications error means that it cannot continue processing on the current BE.

Example: `primaryFailbackInterval = 10`

reportPeriodSeconds

Syntax:	`reportPeriodSeconds = `*`seconds`*
Description:	The number of seconds separating reports of failed messages.
Type:	Integer
Units:	Seconds
Optionality:	Required
Allowed:	
Default:	10
Notes:	BeClient issues a failed message report: • For timed-out messages • For unrequested responses • For new calls rejected because of congestion • For messages with invalid Voucher and Wallet Server identifiers • If new and subsequent requests fail because both Voucher and Wallet Servers have stopped working VWS heartbeat detection must be enabled for the parameter to work. Set `reportPeriodSeconds` to more than `heartbeatPeriod`. This parameter is used by libBeClientIF.
Example:	`reportPeriodSeconds = 10`

Example configuration

The following configuration is an example BeClient section of **eserv.config** on a Voucher and Wallet Server node. Comments have been removed.

```
BeClient = {
    clientName = "scpClient"
    heartbeatPeriod = 3000000
    messageTimeoutSeconds = 2
    maxOutstandingMessages = 100
    reportPeriodSeconds = 10
    primaryFailbackIntreval = 10
    connectionRetryTime = 2

    plugins = [
        {
            config="broadcastOptions",
            library="libclientBcast.so",
            function="makeBroadcastPlugin"
        }
    ]

    broadcastOptions = {
        aggregateNAckCodes = [
            "NVOU"
        ]
    }

    notEndActions = [
        {type="IR  ", action="ACK "}
        {type="SR  ", action="ACK "}
        {type="SR  ", action="NACK"}
        {type="INER", action="ACK "}
        {type="SNER", action="ACK "}
        {type="SNER", action="NACK"}
    ]

    billingEngines = [
        {    id = 1,
             primary   = { ip="192.0.2.0", port=1500 },
             secondary = { ip="192.0.2.1", port=1500 }
        }
    ]
}
```

Output

The BeClient writes error messages to the system messages file, and also writes additional output to:

/IN/service_packages/CCS/tmp/BeClient.log

Note: The above are defaults and can vary.

beGroveller

Purpose

The beGroveller processes wallets daily on the primary VWS (while the VWS is active) and performs wallet inquiries. This triggers all beVWARS plug-ins that are activated on wallet inquiry (for example, beVWARSExpiry). This activity catches up on due events for wallets that have not been accessed for some time. This keeps the E2BE database relatively up to date and means operations such as MSC deactivation for removed accounts always happen (although later than they are scheduled to occur).

Additionally, the beGroveller runs a night time run to process all the wallets that have not been accessed during the day.

The beGroveller is designed to run on the primary VWS, although it will failover to the secondary if necessary. For more information about which VWS the beGroveller runs on, see *beGroveller quorum* (on page 148).

Tuning the beGroveller is a balance between the need to keep the database running smoothly for business purposes, and the load imposed by the process. Configuring the beGroveller for less than 100 ms per wallet (= 10 wallets/second) is not recommended.

Process

The beGroveller maintains multiple asynchronous connections to the VWS database; a single connection for each beVWARS requesting grovel activity.

Here is the beGroveller process.

Stage	Description
1	A connection is assigned to the first beVWARS instance requiring grovel activity and a buffer is opened for it.
2	The wallets with currently expired buckets are retrieved and stored to the buffer in a collection set for the beVWARS. The buffer is then closed.
3	The beVWARS requests are then processed directly from the set instead of being continuously fetched from the database.
4	The beGroveller is responsible for maintaining the set of wallets in the beVWARS buffer. When it is empty, or it drops below a configured threshold, then the buffer is automatically reopened and more wallet details are collected.

Stage	Description
5	Successive beVWARS instances perform one of the following: • Use an existing connection that is not currently managing a buffer • Open a new connection if all the current connections are in operation
6	At the end of the day the beGroveller creates a list of all the wallets that have not been accessed during the day, and these are processed during the overnight run.

Startup

This task is started by the SLEE, by the following line in **SLEE.cfg**:

```
INTERFACE=beGroveller  beGroveller  /IN/service_packages/E2BE/bin instance_count
EVENT
```

Where `instance_count` is the number of instances to run of the beGroveller process.

Notes:

- To enable beGroveller to run, you must set the `enableGrovelling` (on page 41) parameter to true.
- If you configure the SLEE to run multiple instances of the beGroveller, then each beGroveller process will have the value of `instance_count` - 1 appended to the process name. So the master beGroveller process will be named beGroveller0 and subsequent slave beGroveller processes will be named beGroveller1, beGroveller2 and so on. If you configure only one instance of the beGroveller, then nothing will be appended to the process name.

For more information about configuring SLEE interfaces, see *SLEE Technical Guide*.

Configuration

The beGroveller uses parameters from these parameter groups in the **eserv.config** file on VWS nodes:

- beGroveller
- beVWARS *groveller parameters* (on page 105)

beGroveller also uses the `enableGrovelling` (on page 41) shared parameter from the BE section of **eserv.config**.

The `beGroveller` group contains parameters in the structure shown below.

```
beGroveller = {
    quorumHost = "host"
    maxIDsPerResponse = ids
    retrySeconds = seconds
    processExpiredBuckets = true|false
    consecutiveFetch = num
    noProcessingTimes = [
        { startsAt = "HH:MM", endsAt = "HH:MM" }
    ...
    ]
    connectionRetryTime = seconds
    heartbeatPeriod = microseconds
    filledBufferThreshold = num
    ludProcessingTime = "HH:MM"

}
```

Example configuration

This is an example of the beGroveller section of an **eserv.config** file on a VWS (comments have been removed).

```
beGroveller = {
    quorumHost = "produsms-cluster"
    maxIDsPerResponse = 160
    retrySeconds = 60
    processExpiredBuckets = true
    noProcessingTimes = [
        { startsAt = "06:00", endsAt = "09:30" }
        { startsAt = "11:30", endsAt = "14:00" }
        { startsAt = "16:00", endsAt = "21:00" }
    ]
    connectionRetryTime = 60
    heartbeatPeriod = 300000000
    filledBufferThreshold = 480
    ludProcessingTime = "14:04"
}
```

Parameters

Parameters of the beGroveller group are listed below.

connectionRetryTime

Syntax:	`connectionRetryTime = ` *`seconds`*
Description:	The number of seconds between attempts to establish a connection to the beServer on the local VWS and the remote VWS in this pair.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	60
Notes:	The connection to beServer establishes whether or not the local and remote VWSs are in the running state. If it fails to make a connection, beGroveller will log an error to the syslog. For more information about states, see *BE States* (on page 26).
Example:	`connectionRetryTime = 60`

consecutiveFetch

Syntax:	`consecutiveFetch = ` *`num`*
Description:	Maximum number of consecutive fetches between other priority checks.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	3000
Notes:	This number is hard coded in versions prior to 2.4.0.22.
Example:	`consecutiveFetch = 3000`

filledBufferThreshold

Syntax:	`filledBufferThreshold = `*`num`*
Description:	Threshold for the minimum number of wallet ID entries stored in the buffer. A refill is needed when the number of entries in the buffer falls below this number.
Type:	Integer
Optionality:	Required
Allowed:	
Default:	320
Notes:	A separate buffer will be used for each beGroveller client. The beGroveller will continue to fetch wallets until all the client buffers are full. It will then wait until the number of entries in one of the buffers falls below the minimum before fetching more wallets.
Example:	`filledBufferThreshold = 400`

heartbeatPeriod

Syntax:	`heartbeatPeriod = `*`microsecs`*
Description:	The heartbeat period for the beGroveller connection to the beServer through beClientIF.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	300000000 (300 seconds)
Notes:	
Example:	`heartbeatPeriod = 300000000`

ludProcessingTime

Syntax:	`ludProcessingTime = "`*`HH`*`:`*`MM`*`"`
Description:	Defines the hour of the day when the last used date logic will run. If the beGroveller starts at a later time in the day, then the last used date logic processing will be delayed until the next day. If this hour occurs in a no processing period, then the last used date logic processing will be delayed until the end of the no processing period.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	A valid time in the format *HH*:*MM*
Default:	00:00
Notes:	
Example:	`ludProcessingTime = "10:00"`

maxIDsPerResponse

Syntax:	`maxIDsPerResponse = `*`ids`*
Description:	The number of wallet IDs to send to a beVWARS process when it requests wallets to grovel.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	160

Notes:	beVWARS processes request IDs when they run out other work to do. Setting too low will make groveling slow. Setting it too high will make the response exceed the SLEE event size. 1k events will fit in about 160. 2k events will fit in ~330. Fit in as many as your event size will allow. For more information about SLEE event sizes, see *SLEE Technical Guide*.
Example:	`maxIDsPerResponse = 160`

noProcessingTimes

Syntax:

```
noProcessingTimes = [
     {startsAt = "HH:MM", endsAt = "HH:MM"}
     ...
]
```

Description:	The time periods during each day when beGroveller should not return any wallet IDs to beVWARS which are requesting wallet IDs to grovel.
Type:	Array
Optionality:	Optional (default used if not set)
Allowed:	
Default:	No time restrictions.
Notes:	

Example:

```
noProcessingTimes = [
     { startsAt = "06:00", endsAt = "09:30" }
     { startsAt = "11:30", endsAt = "14:00" }
     { startsAt = "16:00", endsAt = "21:00" }
]
```

startsAt

Syntax:	`startsAt = "HH:MM"`
Description:	The hour and minute to start a period of not sending wallets to be groveled beVWARS processes.
Type:	String
Optionality:	Rrequired if `noProcessingTimes` is set
Allowed:	
Default:	No default
Notes:	The period is finished by the `endsAt` (on page 63) parameter paired with this startsAt parameter in the {} set. This parameter is part of the `noProcessingTimes` (on page 63) parameter array.
Example:	`startsAt = "06:00"`

endsAt

Syntax:	`endsAt = "HH:MM"`
Description:	The hour and minute to finish a period of not sending wallets to be groveled beVWARS processes.
Type:	String
Optionality:	Required if `noProcessingTimes` is set
Allowed:	
Default:	No default

Notes:	The period is started by the `startsAt` (on page 63) parameter paired with this endsAt parameter in the {} set. This parameter is part of the `noProcessingTimes` (on page 63) parameter array.
Example:	`endsAt = "09:30"`

processExpiredBuckets

Syntax:	`processExpiredBuckets = `*`true`*`|`*`false`*	
Description:	Activates or deactivates expired bucket processing.	
Type:	Boolean	
Optionality:	Optional (default used if not set)	
Allowed:	true	Activate expired bucket processing. The beGroveller will fetch wallets as often as required and process buckets based on the bucket expiry date.
	false	Deactivate expired bucket processing. The beGroveller will fetch wallets for processing once per day at the time set by the `ludProcessingTime` parameter.
Default:	true	
Notes:		
Example:	`processExpiredBuckets = true`	

quorumHost

Syntax:	`quorumHost = "`*`host`*`"`
Description:	The host name or IP address of a machine on the same VWS subnet to use as a quorum device. The quorum machine is used to break a tie when trying to decide if beGroveller should allow the beVWARS processes to process wallets.
Type:	String
Optionality:	Required. If this is not set, or if the specified machine is not on the same VWS subnet, grovelling may not start (see Notes).
Allowed:	
Default:	Not specified
Notes:	This value is used when a beGroveller cannot see the other VWS in a pair (that is, it cannot ping the other VWS). In this case, beGroveller needs to decide which VWS is partitioned from the rest of the network. If this beGroveller can see `quorumHost` but not the other VWS, it will grovel. This means that `quorumHost` must be another device on the same subnet that the VWS nodes use for communication. A good value for `quorumHost` may be the logical address of a SMS cluster, or the IP address of a non-clustered SMS, but you should confirm this with the network administrator.
Example:	`quorumHost = "produsms-cluster"`

retrySeconds

Syntax:	`retrySeconds = `*`seconds`*
Description:	How many seconds to tell beVWARS to wait before sending another request for wallet IDs to process. Used when beGroveller cannot find any wallets which need groveling to send to a beVWARS which has requested wallets to grovel.

Type:	Integer	
Optionality:	Optional (default used if not set)	
Allowed:	0	beGroveller will attempt to calculate a useful delay to set. Either: • the time that a wallet for the requesting beVWARS will expire + 1 minute (up to a maximum of 1 hour), or • 300 (five minutes).
	positive integer	The number of seconds beVWARS should wait before asking for more wallet IDs to grovel.
Default:	0	
Notes:		
Example:	`retrySeconds = 60`	

beServer

Purpose

Handles connections from client processes (including BeClient processes) and controls routing to beVWARS processes.

It maintains a list of connected clients, and loads plug-ins to handle different request types.

The beServer is a finite state machine, handling one request at a time until either a response can be sent back, or more information is needed and a further request is sent to the beVWARS.

The beServer deals with:

- Multiple client connections (via be protocol)
- Pluggable message handlers (per message type x message version)
- Call context for call state for plug-ins
- SLEE event message passing
- Switchable accepting messages from client
- Resynchronizable call context.

Plug-ins

beServer can be extended by:

- Routing handlers specified in the `messageRoutingPlugins` (on page 71) parameter (such as `libbeMsgRouterDefault` (on page 129))
- Message handlers specified in the `handlers` (on page 69) parameter.

The beServer will attempt to process messages using its own handlers; if no handler is found the message will be sent to the beVWARS for processing. Message handlers are generally provided by other applications such as CCS to provide application-specific functions such as asking the beVWARS (through the SLEE) for account information, reservations, and billing.

For more information about the plug-ins provided by CCS, see *CCS Technical Guide*.

About running multiple beServer processes

You can run multiple instances of the beServer to improve performance. The first beServer process (beServer0) is the master beServer and all other instances of the beServer are its slave processes. The master beServer determines which slave beServer to use for each new VWS client connection. It checks the status of the slave beServers and load balances client connections across all slave beServers on the VWS (the master beServer also acts as a slave in this respect and will assign connections to itself as required by the connection loading). After a VWS client connection has been assigned to a slave beServer, it will remain attached to that slave beServer for the lifetime of the connection.

Note: You can configure the master beServer process to always handle specific VWS client interface connections itself. By default, this includes the beGroveller and ccsMfileCompiler connections. See *clientLoadWeightings* (on page 67) for more information.

Startup

This task is started by the SLEE, by the following line in **SLEE.cfg**:

```
INTERFACE=beServer beServerStartup.sh /IN/service_packages/E2BE/bin instance_count EVENT
```

Where `instance_count` is the number of instances to run of the beServer process.

Note: If you configure the SLEE to run multiple instances of the beServer, then each beServer process will have the value of `instance_count` - 1 appended to the process name. So the first beServer process will be named beServer0 and subsequent beServer processes will be named beServer1, beServer2 and so on. If you configure only one instance of the beServer, then nothing will be appended to the process name.

For more information about configuring SLEE interfaces, see *SLEE Technical Guide*

Configuration

The beServer is configured by the parameters in the following section of **eserv.config** file:

```
beServer = {
    clientSelectTime = microsecs
    quiesceLength = microsecs
    serverPortOverride = port
    clientSocketBufferSize = bytes
    enableStatistics = true
    errorOnRecordStatistics = false
    maxDownstreamQueueLength = int
    downstreamOverloadSleepUSec = microsecs
    dbConnCheckTime = seconds
    recoveryReportInterval = seconds

    notEndActions = [
        {type="str", action="[ACK|NACK]"}
        [...]
    ]

    handlers = [
        "lib"
        [...]
    ]

    messageRoutingPlugins = [
        "lib"
        ...
    ]

    msgRouterDefault = {
        roundRobinTypes = [
            "TYPE"
            ...
        ]
        routeOnVoucherNumber = true|false
    }

    purge = {
        purgeInterval = seconds
        vwarsTimeout = seconds
        expectedKeep = seconds
        noExpectedKeep = seconds
    }

    routingVoucherNumberLength = int
    slaveLocalSocketDirectory = "directory"

    clientLoadWeightings = [
        {name="client_name", weighting=value}
        {...}
    ]

}
```

Parameters

Parameters of the `beServer` group are listed below.

clientLoadWeightings

Syntax:	`clientLoadWeightings = [`
	`    {name="client_name", weighting=value}`
	`    {name="client_name", weighting=value}`
	`    ...`
	`    ]`

Description:	Defines the load weighting value to assign to each type of client connected to the beServer. This improves load sharing over multiple beServer interfaces. • *`client_name`* is a the name of a client interface configured in **SLEE.cfg**. • *`value`* is the load weighting value and indicates the expected traffic load from the specified client interface. A larger value indicates a greater expected load. The weighting value for the beGroveller and ccsMFileCompiler clients should be zero (0) and you should not change their value. Setting the weighting value to zero forces the master beServer to always handle the connection itself.
Type:	Array
Optionality:	Optional (default used if not set)
Allowed:	
Default:	100 except ccsMfileCompiler (default 0), and beGroveller (default 0)
Notes:	The clientLoadWeightings configuration is used when there are multiple instances of the beServer interface running. It is not used if only one beServer interface is running.

Example:

```
clientLoadWeightings = [
    {name="ccsBeClient", weighting=200}
    {name="ccsMFileCompiler", weighting=0}
    {name="ccsBeGroveller", weighting=0}
    {name="ccsBeOrb", weighting=10}
    {name="ccsBeResync", weighting=100}
    {name="ccsBatchCharge", weighting=10}
    {name="ccsDomainMigration", weighting=50}
    {name="ccsAccount", weighting=10}
    {name="ccsPeriodicCharge", weighting=100}
    {name="ccsChangeDaemon", weighting=50}
    {name="ccsSLEEChangeDaemon", weighting=50}
    {name="PIbeClient", weighting=10}
]
```

clientSelectTime

Syntax: `clientSelectTime = `*`microsecs`*

Description: The number of microseconds between each instant where beServer checks the SLEE for events.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 1000000

Notes:

- 1 000 000 microseconds = 1 second.
- If an event is waiting on the SLEE, beServer ignores this setting and makes the next check immediately afterwards. This allows a second event to be detected without delay.
- If the **eserv.config** file is reloaded, beServer will re-read the clientSelectTime parameter.

Example:

clientSocketBufferSize

Syntax: `clientSocketBufferSize = `*`bytes`*

Description: The maximum message size in bytes expected from the BeClients connected to the beServer.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

Default: 10240

Notes: A message larger than this value will not be constructed properly.

Example: `clientSocketBufferSize = 10240`

enableStatistics

Syntax: `enableStatistics = true|false`

Description: Enable statistics gathering for beServer for all received client message types.

Type: Boolean

Optionality: Optional

Allowed:

true	Record statistics.
false	Do not record statistics.

Default: false

Notes:	
Example:	`enableStatistics = false`

errorOnRecordStatistics

Syntax:	`errorOnRecordStatistics = true|false`
Description:	Enable statistics warnings/errors when there is a problem recording a statistic for known/unknown message type.
Type:	Boolean
Optionality:	Optional
Allowed:	true — Generate warning/error. false — Do not generate warning/error.
Default:	false
Notes:	
Example:	`errorOnRecordStatistics = false`

dbConnCheckTime

Syntax:	`dbConnCheckTime =` *`seconds`*
Description:	The number of seconds between each check that beServer is connected to, and logged on to, the Oracle database.
Type:	Integer
Optionality:	Optional
Allowed:	
Default:	1
Notes:	If the Oracle database is not available, the current Voucher and Wallet Server is disabled and BeClient routes calls to the other Voucher and Wallet Server.
Example:	`dbConnCheckTime = 1`

downstreamOverloadSleepUSec

Syntax:	`downstreamOverloadSleepUSec =` *`microsecs`*
Description:	If a downstream process like beVWARS is overloaded, this parameter sets the number of microseconds that beServer will wait before rechecking the process.
Type:	Integer
Optionality:	Required
Allowed:	
Default:	
Notes:	• 1 000 000 microseconds = 1 second. • This value must be shorter than the SLEE watchdog timeout period.
Example:	`downstreamOverloadSleepUSec = 100000`

handlers

Syntax:

```
handlers = [
    "lib"
    [...]
]
```

Description: The handlers parameter array contains plug-in library files that beServer must load.

Type:	Parameter array
Optionality:	Optional
Allowed:	
Default:	
Notes:	• Plug-in library files contain message handlers for requests from clients. A typical file might be libbeServerPingPlugin.so. • The order that plug-in files are listed in the array is important. A handler can be loaded twice, causing the last handler to be the one used. • If the **eserv.config** file is reloaded, beServer will re-read the plug-in library files in the handlers parameter array.
Example:	`handlers = [ "libbeServerPingPlugin.so" ]`

quiesceLength

Syntax:	`quiesceLength =` *`microsecs`*
Description:	The number of microseconds that the beServer will restrict traffic to only sending responses to outstanding requests from clients.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	100000 (1/10 second)
Notes:	beServer will not read any new work during a quiesced interval so the Voucher and Wallet Server has a chance to confirm the result of requests to the clients. This minimizes failing over requests to the other VWS in the pair that have been successfully processed on this VWS, but the confirmation has not been sent to the client. Client requests that have yet to be read will build up during this time, and when they exceed the maximum queue length, BFT will kick in on the client. The beServer will close the socket after quiesceLength has passed, and all traffic will be directed at the other VWS in this pair. This value should be set to the maximum time it takes to process all outstanding requests currently on the SLEE. Any longer and outstanding requests on the sockets will be delayed unnecessarily. For more information about VWS error states and recovery, see *Process Failure Recovery* (on page 147).
Example:	`quiesceLength = 100000`

maxDownStreamQueueLength

Syntax:	`maxDownStreamQueueLength =` *`len`*
Description:	The maximum number of pending events on any beVWARS.
Type:	Integer
Optionality:	Required
Allowed:	
Default:	
Notes:	If pending events exceed this number, beServer refers to the `downstreamOverloadSleepUSec` parameter.
Example:	`maxDownStreamQueueLength = 1000`

messageRoutingPlugins

Syntax:	`messageRoutingPlugins = [`
	 `    "`*`lib`*`"`
	 `]`

Description:	Which message routing plug-ins to load.
Type:	Array
Optionality:	Optional (default used if not set)
Allowed:	
Default:	
Notes:	These plug-ins tell the beServer which beVWARS to pass requests to. Requests based on a wallet or a voucher must continue to be serviced by the same beVWARS so it can keep the wallet or voucher cached. For more information, see *libbeMsgRouterDefault* (on page 129).

Example:	`messageRoutingPlugins = [`
	 `    "libbeMsgRouterDefault.so"`
	 `]`

msgRouterDefault

Syntax:	`msgRouterDefault = {`
	 `    roundRobinTypes = []`
	 `}`

Description:	Defaults for the message routing plug-ins loaded by `messageRoutingPlugins` (on page 71).
Type:	Array
Optionality:	
Allowed:	
Default:	
Notes:	Includes the `roundRobinTypes` (on page 71) parameter
Example:	

roundRobinTypes

Syntax:	`roundRobinTypes = [`
	 `    "`*`type`*`"`
	 `    ...`
	 `]`

Description:	Default routing for **libbeMsgRouterDefault.so**.
Type:	Array of four-character strings.
Optionality:	Optional (default used if not set)
Allowed:	
Default:	VI
Notes:	If a message does not have a [WALT] or [VNUM] tag and its message type is in this array, it will be round robined around beVWARS to share load. The CCS VI message may or may not have a [VNUM] field. You can also organize the elements in this array in one line, using a comma ',' to separate the types.

Example:	`roundRobinTypes = [`
	 `    "VI  "`
	 `]`

routeOnVoucherNumber

Syntax:	`routeOnVoucherNumber = true|false`
Description:	What method to use to determine which beVWARS process to route voucher redeem requests to.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	true — Use a hash of the Voucher Number to route to beVWARS. Compatible with CCS 3.1.4 and earlier.
	false — Use Voucher ID to route to beVWARS. Compatible with CCS 3.1.5 and later.
Default:	true
Notes:	This parameter is used by the libbeMsgRouterDefault library.
Example:	`routeOnVoucherNumber = false`

notEndActions

Syntax:

```
notEndActions = [
    {type="type", action="ACK|NACK"}
    [...]
]
```

Description: This parameter array identifies messages that will be followed by subsequent message.

Type: Parameter array

Optionality: Required

Allowed:

Default:

Notes:

Example:

```
notEndActions = [
    {type="IR  ", action="ACK "}
    {type="SR  ", action="ACK "}
    {type="SR  ", action="NACK"}
    {type="INER", action="ACK "}
    {type="SNER", action="ACK "}
    {type="SNER", action="NACK"}
]
```

purge

Syntax:

```
purge = {
    purgeInterval = seconds
    vwarsTimeout = seconds
    expectedKeep = seconds
    noExpectedKeep = seconds
}
```

Description: The purge parameter group contains parameters that control purges.

Type: Parameter group.

Optionality: Optional

Allowed:

Default:

Notes: Running purge stresses the system with high loads.

Example:

`expectedKeep`

Syntax:	`expectedKeep =` *`seconds`*
Description:	A plug-in can specify the number of seconds it will wait for a request for a context that it wants to keep. This parameter sets additional time, after the plug-in's time, that beServer keeps a context if (during this period) no request for the context is made.
Type:	Integer
Optionality:	Optional
Allowed:	
Default:	60
Notes:	• This parameter is part of the `purge` parameter group. • If the **eserv.config** file is reloaded, beServer will re-read the parameter.
Example:	`expectedKeep = 60`

`noExpectedKeep`

Syntax:	`noExpectedKeep =` *`seconds`*
Description:	If the plug-in does not specify a time it will wait for a request for a wanted context, this parameter defines the number of seconds that beServer will keep the context.
Type:	Integer
Optionality:	Optional
Allowed:	
Default:	3600
Notes:	• This parameter is part of the `purge` parameter group. • If the **eserv.config** file is reloaded, beServer will re-read the parameter. • This parameter should be set to the equivalent value in seconds as the CCS `volumeReservationLength` value. See *CCS Technical Guide*
Example:	`volumeReservationLength = 2` (days) `noExpectedKeep` = `172800` (number of seconds in 2 days)

`purgeInterval`

Syntax:	`purgeInterval =` *`seconds`*
Description:	The number of seconds between purges.
Type:	Integer
Optionality:	Optional
Allowed:	
Default:	300
Notes:	• This parameter is part of the `purge` parameter group. • If the **eserv.config** file is reloaded, beServer will re-read the parameter.
Example:	`purgeInterval = 300`

`vwarsTimeout`

Syntax:	`vwarsTimeout =` *`seconds`*
Description:	The number of seconds between the moment that beServer sends a request to the beVWARS and the moment that beServer fabricates an exception response.
Type:	Integer

Units:	Seconds
Optionality:	Optional
Allowed:	
Default:	10
Notes:	• This parameter is read during a purge. • This parameter is part of the `purge` parameter group. • If the **eserv.config** file is reloaded, beServer will re-read the parameter.
Example:	`vwarsTimeout = 10`

recoveryReportInterval

Syntax:	`recoveryReportInterval =` *`seconds`*
Description:	The number of seconds between logging each recovery report to the syslog while in recovery mode.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	60
Notes:	The recovery report records how many beVWARS processes beServer is waiting to go into running state before it will go into running state. For more information about the different states, see *Process Failure Recovery* (on page 147).
Example:	`recoveryReportInterval = 60`

routingVoucherNumberLength

Syntax:	`routingVoucherNumberLength =` *`len`*
Description:	The length of the prefix of the voucher number to use for routing voucher messages to beVWARS processes.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	10
Notes:	For CCS vouchers, this should match the length of the voucher number, not the length of the voucher signature. For more information about voucher numbers and voucher signatures, see *Voucher Manager User's Guide*.
Example:	`routingVoucherNumberLength = 10`

serverPortOverride

Syntax:	`serverPortOverride =` *`port`*	
Description:	The port number beServer uses as an alternative to the one defined by the `beLocationPlugin` (on page 41).	
Type:	Integer	
Optionality:	Optional	
Allowed:	-1	Do not override beLocationPlugin.
	any valid port	Port for beServer to use.
Default:	-1	
Notes:	This parameter is usually used for testing.	
Example:	`serverPortOverride = 1500`	

`slaveLocalSocketDirectory`

Syntax:	`slaveLocalSocketDirectory = "`*`directory_name`*`"`
Description:	Specifies the directory to use for files created by interprocess communication (IPC) objects, such as sockets, semaphores, and shared memory. The IPC objects enable communication between master and slave beServer interfaces.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	A valid directory location.
Default:	/tmp
Notes:	None
Example:	`slaveLocalSocketDirectory = "/tmp"`

Example configuration

This is an example of the beServer section of the **eserv.config** file on a VWS node (comments have been removed).

```
beServer = {
    clientSelectTime = 1000000
    quiesceLength = 100000
    serverPortOverride = 1500
    clientSocketBufferSize = 10240
    maxDownstreamQueueLength = 1000
    downstreamOverloadSleepUSec = 100000
    dbConnCheckTime = 5
    recoveryReportInterval = 60

    notEndActions = [
        {type="IR  ", action="ACK "}
        {type="SR  ", action="ACK "}
        {type="SR  ", action="NACK"}
        {type="INER", action="ACK "}
        {type="SNER", action="ACK "}
        {type="SNER", action="NACK"}
    ]

    handlers = [
        "libbeServerPingPlugin.so"
    ]

    messageRoutingPlugins = [
        "libbeMsgRouterDefault.so"
    ]

    msgRouterDefault = {
        roundRobinTypes = [
            "VI  "
        ]
        routeOnVoucherNumber = true
    }

    purge = {
        purgeInterval = 300
        vwarsTimeout = 10
        expectedKeep = 60
        noExpectedKeep = 3600
        }

        routingVoucherNumberLength = 10
        slaveLocalSocketDirectory = "/tmp"

        clientLoadWeightings = [
            {name="ccsBeClient", weighting=200}
            {name="ccsMFileCompiler", weighting=0}
            {name="ccsBeGroveller", weighting=0}
            {name="ccsBeOrb", weighting=10}
            {name="ccsBeResync", weighting=100}
            {name="osaChamScs", weighting=100}
            {name="ccsBatchCharge", weighting=10}
            {name="ccsDomainMigration", weighting=50}
            {name="ccsAccount", weighting=10}
            {name="ccsPeriodicCharge", weighting=100}
            {name="ccsChangeDaemon", weighting=50}
            {name="ccsSLEEChangeDaemon", weighting=50}
            {name="PIbeClient", weighting=10}
        ]

    } # BE.beServer
```

Output

The beServer writes error messages to the system messages file, and also writes additional output to the following location by default:

/IN/service_packages/E2BE/tmp/beServer.log

beSync

Purpose

Synchronizes data between the Voucher and Wallet Servers in a VWS pair.

beSync collects all updates and reservations being made, and writes them to disk. It then reads them from disk and sends them to the other VWS, as and when it can.

For more information on beSync and how it interacts with other VWS components, see *Synchronization* (on page 20).

Startup

This task is started by the SLEE, by the following line in **SLEE.cfg**:

```
INTERFACE=beSync   beSyncStartup.sh   /IN/service_packages/E2BE/bin instance_count
EVENT
```

Where *instance_count* is the number of instances to run of the beSync process.

Note: If you configure the SLEE to run multiple instances of the beSync process, then each beSync process will have the value of *instance_count* - 1 appended to the process name. So the first beSync process will be named beSync0 and subsequent beSync processes will be named beSync1, beSync2 and so on. If you configure only one instance of beSync then nothing is appended to the process name.

For more information about configuring SLEE interfaces, see *SLEE Technical Guide*.

Configuration

beSync accepts the following parameters from **eserv.config**.

```
beSync = {

    shared = {
        noWorkSleepTime = seconds

        spoolDirectory = "dir"
        spoolChunkSize = num
        badFileDirectory = "dir"

        maxDownstreamQueueLength = num
        downstreamOverloadSleepUSec = int
    }

    sink = {
        inSyncThresholdSeconds = seconds
        inSyncReportingPeriodRecords = records

        maxSecsToWaitForRemoteOperations = seconds

        remoteBEhostname="host"
        remoteBEport=port

        retryConnectionDelaySeconds = seconds
        maxRetriesBeforeStart = num

        localUpdateChunkSize = size
        heartbeatPeriodSeconds = seconds
    }

    source = {
        listenInterface="ip"
        listenPort = port

        recordSendingChunkSize = num
        maxQueueLength = num
    }
}
```

Parameters

Here are the parameters in the `beSync` section.

shared parameters

The `shared` sub-section of `beSync` defines the beSync shared items.

badFileDirectory

Syntax:	`badFileDirectory = "`*`dir`*`"`
Description:	Directory to move corrupted resync files to.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	Any directory path.
Default:	"/IN/service_packages/E2BE/tmp"
Notes:	Files in this directory will be called **file.bad**.
Example:	`badFileDirectory = "/IN/service_packages/E2BE/tmp"`

`downstreamOverloadSleepUSec`

Syntax:	`downstreamOverloadSleepUSec =` *`int`*
Description:	When a downstream process, a beVWARS is overloaded, sleep for this period before rechecking.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	100000
Notes:	This value must be shorter than the SLEE watchdog timeout period.
Example:	`downstreamOverloadSleepUSec = 100000`

`maxDownstreamQueueLength`

Syntax:	`maxDownstreamQueueLength =` *`num`*
Description:	The maximum number of pending events on any beVWARS. When more than this number of events are queued on any of the processes, beSync will sleep.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	10000
Notes:	See *maxQueueLength* (on page 82) for throttling based on the remote beSync. **Important:** Care should be taken when setting `BE.beSync.maxDownstreamQueueLength` as this is the value that slows a full resync by keeping each of the beVWARS processes busy. If you allow a full resync to run as fast as possible, it will use up all of the events.
Example:	`maxDownstreamQueueLength = 10000`

`noWorkSleepTime`

Syntax:	`noWorkSleepTime =` *`seconds`*
Description:	The sleep time in seconds.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	0.2
Notes:	Should be small (0.x) in production, and larger in test (2.0). Lower values will cause more CPU usage.
Example:	`noWorkSleepTime = 0.2`

`spoolChunkSize`

Syntax:	`spoolChunkSize =` *`num`*
Description:	The number of records to read and send in one cycle.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	16
Notes:	
Example:	`spoolChunkSize = 16`

spoolDirectory

Syntax:	`spoolDirectory = "`*`dir`*`"`
Description:	This is where all transactions are written to disk by beVWARS so they can be replayed to one of the following: • The remote VWS in the pair • If there has been a failure, the local VWS in the pair
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	**/IN/service_packages/E2BE/sync**
Notes:	Available space in the directory set by this parameter is checked by the beSync against the limits set in the Disk space parameters.
Example:	`spoolDirectory = "/var/logs/sync"`

sink parameters

The `sink` sub-section of `beSync` defines the sink parameters for beSync. This is the component that *receives* operations from the remote beSync.

heartbeatPeriodSeconds

Syntax:	`heartbeatPeriodSeconds = `*`seconds`*
Description:	How often in seconds heartbeat packets are sent on a connection.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	10
Notes:	
Example:	`heartbeatPeriodSeconds = 10`

inSyncThresholdSeconds

Syntax:	`inSyncThresholdSeconds = `*`seconds`*
Description:	How close (in seconds) to real-time before the beSync is enabled.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	5
Notes:	
Example:	`inSyncThresholdSeconds = 5`

inSyncReportingPeriodRecords

Syntax:	`inSyncReportingPeriodRecords = `*`records`*
Description:	The number of records between checks against real-time.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	

Default:	10000
Notes:	
Example:	`inSyncReportingPeriodRecords = 10000`

`localUpdateChunkSize`

Syntax:	`localUpdateChunkSize =` *`size`*
Description:	Tuning parameter.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	100
Notes:	
Example:	`localUpdateChunkSize = 100`

`maxRetriesBeforeSeconds`

Syntax:	`maxRetriesBeforeSeconds =` *`num`*
Description:	The number of attempts to contact the other VWS in the pair before we start regardless.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	5
Notes:	
Example:	`maxRetriesBeforeSeconds = 5`

`maxSecsToWaitForRemoteOperations`

Syntax:	`maxSecsToWaitForRemoteOperations =` *seconds*
Description:	During the synchronization process, the maximum number of seconds beSync waits for a remote operation message before enabling beVWARS to move to the Running state.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	0 – beSync does not wait before enabling beVWARS to move to the Running state. Positive integer – Specifies the number of seconds beSync waits for a remote operation message.
Default:	5
Notes:	
Example:	`maxSecsToWaitForRemoteOperations = 5`

`remoteBEhostname`

Syntax:	`remoteBEhostname = "`*`host`*`"`
Description:	Overrides the DB configuration of the remote VWS host.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	The name of the remote BE host.
Notes:	

Example:	

`remoteBEport`

Syntax:	`remoteBEport = `*`port`*
Description:	Overrides the DB configuration of the remote VWS port
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	Valid port number
Default:	2001
Notes:	
Example:	`remoteBEport = 2001`

`retryConnectionDelaySeconds`

Syntax:	`retryConnectionDelaySeconds = `*`seconds`*
Description:	The maximum number of seconds between connection attempts
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	30
Notes:	
Example:	`retryConnectionDelaySeconds = 30`

source parameters

The `source` sub-section `beSync` defines the source parameters for beSync. This is the component that *sends* operations to the remote beSync.

`listenInterface`

Syntax:	`listenInterface = "`*`ip`*`"`
Description:	Overrides the DB configuration for what we listen to.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	Internet Protocol version 4 (IPv4) addresses, IP version 6 (IPv6) addresses
Default:	0.0.0.0
Notes:	You can use the industry standard for omitting zeros when specifying IPv6 addresses.

Examples:	`listenInterface = "192.0.2.0"`
	`listenInterface = "2001:db8:0000:1050:0005:0600:300c:326b"`
	`listenInterface = "2001:db8:0:0:0:500:300a:326f"`
	`listenInterface = "2001:db8::c3"0"`

`listenPort`

Syntax:	`listenPort = `*`port`*
Description:	Overrides the DB configuration.
Type:	Integer
Optionality:	Optional (default used if not set)

Allowed:	
Default:	2001
Notes:	
Example:	`listenPort = 2001`

maxQueueLength

Syntax:	`maxQueueLength =` *num*
Description:	How many messages can queue on the socket before we stop sending and stop getting work from the beVWARS.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	50
Notes:	
Example:	`maxQueueLength = 50`

recordSendingChunkSize

Syntax:	`recordSendingChunkSize =` *num*
Description:	The number of records to send in one poll cycle.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	50
Notes:	
Example:	`recordSendingChunkSize = 50`

Example configuration

This is an example of the `beSync` section of the **eserv.config** file on a VWS (comments have been removed).

```
beSync = {
    shared = {
        noWorkSleepTime = 0.2

        spoolDirectory = "/IN/service_packages/E2BE/sync"
        spoolChunkSize = 16
        badFileDirectory = "/IN/service_packages/E2BE/tmp"

        maxDownstreamQueueLength = 10000
        downstreamOverloadSleepUSec = 100000
    }

    sink = {
        inSyncThresholdSeconds = 5
        inSyncReportingPeriodRecords = 10000
        maxSecsToWaitForRemoteOperations = 5
        retryConnectionDelaySeconds = 30
        maxRetriesBeforeStart = 5
        localUpdateChunkSize = 100
        heartbeatPeriodSeconds = 10
    }
```

```
        source = {
            recordSendingChunkSize = 50
            maxQueueLength = 50
        }
    }
```

Output

The beSync writes error messages to the system messages file, and also writes additional output to:

/IN/service_packages/E2BE/tmp/beSync.log

Note: The above are defaults and can vary.

beServiceTrigger

Purpose

beServiceTrigger sends BPL requests to instances of the xmlTcapIF and NCC Open Services Development (OSD) requests to the osdInterface running on separate SLC nodes within the same IN platform. It runs as a SLEE interface on the primary VWS only.

beServiceTrigger accepts beServiceTrigger events from other BE SLEE interfaces running on the same VWS. For each beServiceTrigger event received, it first checks whether an operationSetName is defined in the event. If an operationSetName is:

- Defined, it sends the related OSD operation to the OSD interface running on a separate SLC node
- Not defined, it creates and sends a new BPL request to an available instance of an XML TCAP interface running on a separate SLC node.

About the beServiceTrigger User

The beServiceTrigger user allows beServiceTrigger to access external systems, such as a client ASP that is accessed through the OSD component during event processing. beServiceTrigger retrieves the user credentials (username and password) from a secure credentials vault on the SMS node. The credentials vault is used for storing user names and passwords securely and for authorizing users.

You can set the beServiceTrigger user and password by using the beServiceTriggerUser utility. See *Setting the beServiceTrigger User and Password* (on page 144) for more information.

Example

An example of the use of the beServiceTrigger is the Rewards plug-in in the beVWARS. The Rewards plug-in sends a beServiceTrigger request to run a control plan to apply rewards to subscribers on non-VWS charging domains.

Characteristics

beServiceTrigger has the following characteristics:

- It only accepts beServiceTrigger events sent by other SLEE interfaces running on the same (primary) VWS
- When processing beServiceTrigger events, a new BPL or OSD request is issued to the next available XML TCAP or OSD interface without any acknowledgment to the requesting interface. The traffic between the beServiceTrigger and each particular xmlTcapIf / osdInterface is handled synchronously. No BPL / OSD request will be sent to the same xmlTcapIf/ osdInterface instance until the processing of the previous BPL / OSD request has finished.

- Apart from processing incoming beServiceTrigger events, the beServiceTrigger interface communicates with the beVWARS interfaces in order to produce EDRs as a result of processing BPL / OSD responses
- Overall, the processing of beServiceTrigger events and connections to different xmlTcapIf / osdInterface instance is done asynchronously. This allows events to be processed and requests to different xmlTcapIf / osdInterface instances to be handled in parallel.

Process

This section describes how beServiceTrigger processes a beServiceTrigger event from the Rewards plug-in of beVWARS.

Stage	Description
1	beServiceTrigger receives a beServiceTrigger event from the Rewards plug-in and immediately creates a new BPL request ready to be sent to the next available xmlTcapIf. **Note:** The Rewards plug-in is not notified about the events received or the BPL requests being sent.
2	When an xmlTcapIf becomes available, the enqueued BPL request is sent and the xmlTcapIf then becomes unavailable until a response is received or the request times out.
3	Incoming beServiceTrigger events and BPL responses are handled asynchronously to allow new BPL requests to be sent to available xmlTcapIf instances.
4	After a BPL response arrives, the corresponding xmlTcapIf becomes available to process further requests. A request to create a Control Plan Service Invoke EDR (type 7) is sent to the appropriate beVWARS interface.

Startup

This task is started by the following line in **SLEE.cfg**:

```
INTERFACE=beServiceTrigger  beSerTrigStartup.sh
/IN/service_packages/E2BE/bin EVENT
```

Note: Only one instance of the beServiceTrigger interface is allowed per VWS SLEE.

Valid interfaces

The beServiceTrigger requires and uses slightly different configuration depending on the interface used.

XmlTcap is the default interface unless the operation set is defined in the request, in which case the OSD interface is used.

XmlTcap parameters

- Control_Plan
- Service_Handle
- scps

OSD parameters

- CCSNamespace
- osd_scps
- operationSet
- operation

Parameters common to both intrefaces

- edr

- failureRetryTime
- storageInterface
- triggerInterface
- responseTag
- maxRatePerUAS
- throttleLife
- timeBetweenThrottles
- maxConnections

XmlTcap Parameters

beServiceTrigger/XmlTcap is configured by the following parameters from the `triggering` section in the **eserv.config** file on the VWS:

```
triggering = {
    Control_Plan = "cpname"
    Service_Handle = "handle"
    scps = [ "ip:port" ]
    }
    triggering = {
        Control_Plan = "Reward"
        Service_Handle = "CCS_BPL"
        CCSNamespace = "http://eng-prf-zone01-z1/wsdls/ON/CCSNotifications.wsdl"
        edr = false
        scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
        osd_scps = [ "cmxdevscp1:3072", "cmxdevscp2:3072" ]
        failureRetryTime = 60
        storageInterface = beEventStorageIF
        triggerInterface = beServiceTrigger
        operationSet = CMX ON
        operation = Invoke OSD
        responseTag = Result
        maxRatePerUAS = 0
        throttleLife = 30
        timeBetweenThrottles = 10
        tcpTxMaxBuf = 262144
        tcpRxMaxBuf = 131072
}   # triggering
```

Control_Plan

Syntax:	`Control_Plan = "`*`cpname`*`"`
Description:	The default control plan name that will be used in BPL requests if none is present in the SLEE event.
Type:	String
Optionality:	Optional
Allowed:	
Default:	Empty
Notes:	
Example:	`Control_Plan = "Reward"`

scps

Syntax:	`scps = [`
	`    "`*`ip`*`:`*`port`*`"`
	`    ...`
	`]`

Description:	Lists the host name or Internet Protocol (IP) address, and port of each xmlTcapInterface SLC to which beServiceTrigger connects. If you specify an IP version 6 (IPv6) address and port combination, then you must enclose the IPv6 address in square brackets [], see example for details.
Type:	Array
Optionality:	Required. In any row of the array, *`ip`* must be specified but *`port`* is optional.
Allowed:	• *ip* – An IP address or symbolic host name • *port* – Integer in the range 0 to 65535
Default:	*`port`* defaults to 3072
Notes:	An example of an IPv4 address is **`192.0.2.1`**. An example of an IPv6 address is **`2001:db8:`***`n`*`:`*`n`*`:`*`n`*`:`*`n`*`:`*`n`*`:`*`n`* where *`n`* is a group of 4 hexadecimal digits. The industry standard for omitting zeros is also allowed. An example of an address in symbolic name format is **`primary_smc`**.
Example:	see below

```
scps = [
    "198.51.100.1"
    "192.0.2.1:4000"
    "[2001:db8:0000:1050:0005:0600:300c:326b]:3004"
    "[2001:db8:0:0:0:500:300a:326f]:1234:SMF"
    "[2001:db8::c3]:1234:SMF"
    "2001:db8:1050:0:0:300a:0300:126c"
    "primary_smc"
    "secondary_smc:3006"
]
```

Service_Handle

Syntax:	`Service_Handle = "`*`handle`*`"`
Description:	The default service handle that will be used in BPL requests if none is present in the SLEE event.
Type:	String
Optionality:	Optional
Allowed:	
Default:	Empty
Notes:	
Example:	`Service_Handle = "CCS_BPL"`

OSD Parameters

beServiceTrigger/OSD is configured by the following parameters from the triggering section in the **eserv.config** file on the VWS:

```
triggering = {
    CCSNamespace = "URL"
    osd_scps = [ "ip:port" ]
    operationSet = "name"
    operation = "name"
}
```

CCSNamespace

Syntax:	`CCSNamespace = "`*`URL`*`"`
Description:	The default Namespace that will be put into OSD requests if none is present in the SLEE event.
Type:	String
Optionality:	Optional
Allowed:	
Default:	Empty
Notes:	
Example:	`CCSNamespace = "http://eng-prf-zone01-z1/wsdls/ON/CCSNotifications.wsdl"`

osd_scps

Syntax:

```
scps = [
    "ip:port"
    ...
]
```

Description:	Lists the host name or Internet Protocol (IP) address, and port of each xmlTcapInterface SLC to which beServiceTrigger connects. If you specify an IP version 6 (IPv6) address and port combination, then you must enclose the IPv6 address in square brackets [], see example for details.
Type:	Array
Optionality:	Required. In any row of the array, *`ip`* must be specified but *`port`* is optional.
Allowed:	• *ip* – An IP address or symbolic host name • *port* – Integer in the range 0 to 65535
Default:	*`port`* defaults to 3072
Notes:	An example of an Internet protocol address is **`192.0.2.1`**. An example of an IPv6 address is **`2001:db8:`***`n`*`:`*`n`*`:`*`n`*`:`*`n`*`:`*`n`*`:`*`n`* where n is a group of 4 hexadecimal digits An example of an address in symbolic name format is **`primary_smc`**.

Example:

```
osd_scps = [
    "192.0.2.2
    "192.0.2.1:4000"
    "[2001:db8:0000:1050:0005:0600:300c:326b]:3004"
    "[2001:db8:0:0:0:500:300a:326f]:1234:SMF"
    "[2001:db8::c3]:1234:SMF"
    "2001:db8:300c:0:600:300c:0:126b"
    "primary_smc"
    "secondary_smc:3006"
]
```

operation

Syntax:	`operation = "`*`name`*`"`
Description:	The name of the OSD operation to invoke the service when none is set in the SLEE event.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	Empty

Notes:	
Example:	`operation = "Notification"`

operationSet

Syntax:	`operationSet = "`*`name`*`"`
Description:	The name of the operation set that contains the template used to invoke the service when none is set in the SLEE event.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	Empty
Notes:	
Example:	`operationSet = "NotificationSet"`

Common parameters

beServiceTrigger common parameters are configured by the following from the triggering section in the **eserv.config** file on the VWS:

```
triggering = {
    edr = true | false
    failureRetryTime = seconds
    storageInterface = "name"
    triggerInterface = "name"
    responseTag = name
    maxRatePerUAS = num
    throttleLife = seconds
    timeBetweenThrottles = millisecs
    maxConnections = integer
}
```

edr

Syntax:	`edr = true|false`
Description:	Should an EDR be produced when a response is received.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	true, false
Default:	false
Notes:	
Example:	`edr = false`

failureRetryTime

Syntax:	`failureRetryTime = `*`seconds`*
Description:	The length of time in seconds between attempts to send the message to the SLC.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	60
Notes:	
Example:	`failureRetryTime = 60`

maxConnections

Syntax:	`maxConnections = `*`integer`*
Description:	The maximum number of connections from beServiceTrigger to interfaces on the SLC.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	0 or any positive integer. 0 indicates no maximum.
Default:	25
Notes:	Increase the value of *maxConnections* as the number of rows in the be_event_storage table increases.
Example:	`maxConnections = 25`

maxRatePerUAS

Syntax:	`maxRatePerUAS = `*`num`*
Description:	The maximum rate (messages/second) each SLC is able to handle before throttling.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	0
Notes:	0 means unlimited.
Example:	`maxRatePerUAS = 0`

responseTag

Syntax:	`responseTag = `*`name`*
Description:	The name of the tag in the response message to use to populate the EDR.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	Result
Notes:	
Example:	`responseTag = Result`

storageInterface

Syntax:	`storageInterface = "`*`name`*`"`
Description:	The name of the interface used to store events for sending later, either due to a failure, or a request for a delayed send.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	beEventStorageIF
Notes:	
Example:	`storageInterface = "beEventStorageIF"`

tcpTxMaxBuf

Syntax:	`tcpTxMaxBuf = `*`size`*
Description:	Maximum size of TCP send buffer.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	1048576
Notes:	
Example:	`tcpTxMaxBuf = 262144`

tcpRxMaxBuf

Syntax:	`tcpRxMaxBuf = `*`size`*
Description:	Maximum size of TCP receive buffer.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	1048576
Notes:	
Example:	`tcpRxMaxBuf = 131072`

throttleLife

Syntax:	`throttleLife = `*`seconds`*
Description:	The length of time in seconds a throttle will exist for before the attempts to back it off.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	30
Notes:	
Example:	`throttleLife = 30`

timeBetweenThrottles

Syntax:	`timeBetweenThrottles = `*`millisecs`*
Description:	The length of time in milliseconds between throttle messages being sent to the storage interface.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	
Default:	10
Notes:	
Example:	`timeBetweenThrottles 10`

triggerInterface

Syntax:	`triggerInterface = "`*`name`*`"`
Description:	The name of the triggering interface itself.

Type:	String
Optionality:	Optional (default used if not set)
Allowed:	
Default:	beServiceTrgger
Notes:	
Example:	`triggerInterface = "beServiceTrigger"`

Output

beServiceTrigger writes error messages to the system messages file, and also writes additional output to the location indicated in the startup script, which will usually be set to:

/IN/service_packages/E2BE/tmp/beServiceTrigger.log

Notification requests

The VWS directs all notification requests to a SLC OSD interface through the beServiceTrigger (flow 2 in diagram).

Notification overview

The OSD interface triggers ACS, which loads a control plan containing the notification node in order to perform delivery (flow 6 in diagram).

If no OSD nodes are available, are unresponsive, or the notification has a Time of Day associated, then it will be stored locally for subsequent delivery.

Note: The Wallet Information will be omitted from this, as it will be stale before the notification is sent.

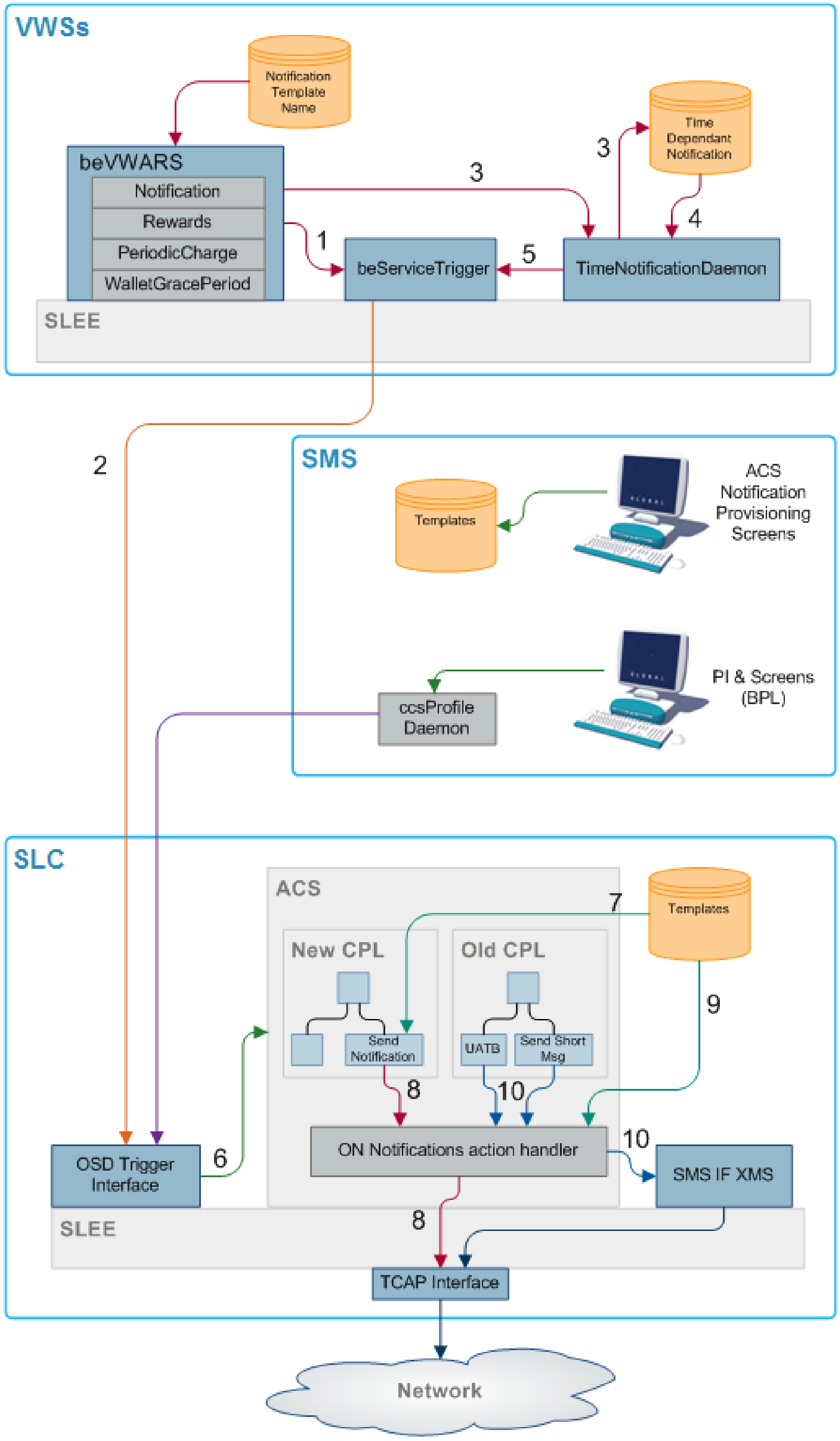
The Time Daemon will poll the Time Dependant notifications stored on the VWSs and trigger OSD requests according to the time. These requests will be throttled and load balanced in order to not overload SLC nodes with large numbers of Control Plan requests.

The VWSs will operate in isolation within the pair, therefore if notifications are delayed and stored on the Primary VWS, they will not be synchronized to the secondary.

If there is a failure or outage in the primary, notifications to be generated will be stored locally on the secondary during the outage. After the primary is operational again it will process all relevant notifications, while the secondary retains responsibility for notifications generated during the primary outage.

Notification flows

This diagram shows the various notification flows across the NCC platform.



Flow 1

The beVWARS plug-ins send SMS information to the beServiceTrigger.

Flow 2

Notification XML messages from the beServiceTrigger to the OSD interface on the SLC.

Flow 3

If a notification cannot be delivered immediately, either because it has an associated time period when it can be delivered, or because the delivery attempt failed, then persistent storage of the notification is provided in a database table.

Flow 4

When the time notification daemon examines the notification entries in the database, it retrieves the notifications that can now be sent either because their allowable delivery time has been met or because it is a message retry.

Flow 5

The time notification daemon deletes the active entries from the database and sends delivery request messages to the beServiceTrigger for each of the active entries.

Flow 6

The OSD interface triggers ACS, which then loads the control plan containing the notification feature node that will perform delivery of the notification.

Flow 7

The notification template to use is determined by the notification feature node, based on:

- Language ID
- Template ID
- Customer ID

Flow 8

The notification feature node delivers a USSD notification through the TCAP interface.

If the message class is "USSD push", then an internal message is sent through the USSD push action handler to the TCAP interface after the notification feature node has performed all the parameter substitutions.

Flow 9

Chassis action to construct message from template.

Flow 10

Other send message feature nodes use new chassis actions to deliver notifications using Messaging Manager.

beVWARS

Purpose

beVWARS caches and holds the state of:

- Wallets and their associated reservations
- Vouchers and their associated reservations

The database cannot reflect the state of the running system, because updates are usually pending in the Writer. To maintain a single consistent view of an individual record's state in the database, use a beVWARS instance to access all wallets or vouchers. beVWARS is responsible for all updates to database fields changed during a resync.

beVWARS also handles COMMITing the database updates and writing EDRs. For more information, see *beVWARS data updates* (on page 18).

Plug-ins

beVWARS can be extended by:

- Event handlers specified in the `plugins` (on page 99) parameter
- Message handlers specified in the *handlers* (on page 98) parameter

VWS provides a set of standard beVWARS plug-ins to handle standard interactions. These include *beVWARSMergeBuckets* (on page 121).

Other handlers are provided by other applications such as CCS to provide application-specific functions such as named event charges. For more information about the plug-ins provided by CCS, see *CCS Technical Guide*.

Activating Used Units Confirmation (UUC) Features

Follow these steps to activate Used Units Confirmation (UUC) features. The installation instructions tell you when to perform these steps.

Note: Perform this procedure on the Primary BE only. Under normal operation, a reservation expires at exactly the same time on both VWSs. Installing the ccsVWARSReservationExpiry plug-in on the Secondary BE causes the user to be double-charged.

Step	Action
1	On the primary BE, open the **/IN/service_packages/eserv.config** file in a text editor.

2	In the `plugins` section, add the `ccsVWARSReservationExpiry.so` entry.
	`BE = {`
	`    beVWARS = {`
	`        plugins = [`
	`            # ... Existing plug-ins here ...`
	`            "ccsVWARSReservationExpiry.so"  # <-- New plugin entry`
	`        ]`
	`    }`
	`}`

3	Save and close the file.

Startup

The SLEE starts this task through the following line in **SLEE.cfg**:

```
INTERFACE=beVWARS beVWARSStartup.sh /IN/service_packages/E2BE/bin
instance_count EVENT
```

Where `instance_count` is the number of instances to run of the beVWARS process.

Note: beVWARS usually uses more than one beVWARS process. Each beVWARS process has the value of `instance_count` - 1 appended to the process name. Thus, the first beVWARS process is beVWARS0, and subsequent beVWARS processes are named beVWARS1, beVWARS2, and so on. If you configure only one instance of beVWARS, nothing is appended to the process name.

For more information about configuring SLEE interfaces, see *SLEE Technical Guide*

Wallet Time Configuration

By default, beVWARS applies the OS time to every wallet as transactions are applied. You can configure beVWARS to read the time from a wallet time configuration file (**/IN/service_packages/E2BE/etc/VWARS_sysdate.cfg**). The file contains a mapping of times to wallet IDs. This allows you to manually change the time for a wallet ID on the fly.

Note: beVWARS uses the time zone of the VWS host on which the beVWARS process is running.

The following shows the format for each line in the **VWARS_sysdate.cfg** file:

```
WalletID YYYY/MM/DD HH:MM:SS
```

For example:

```
12345 2011/11/01 12:00:00
```

To configure beVWARS to read times from **VWARS_sysdate.cfg**, set the `useTimeFromConfigFile` parameter to true in the **eserv.config** file.

Configuration

beVWARS accepts the following parameters from **eserv.config**.

```
beVWARS = {

    voucherReservationPeriodSeconds = seconds
    useTimeFromClient = true|false
    maxTransactionsPerSet = num
    maxOpenDialogTime = seconds
    maxDownstreamQueueLength = num
    downstreamOverloadSleepUSec = microsecs
    minResyncReservationLength = seconds
    createBucketExpiryDays = days
    reservationExpiryCheckMilliseconds = millisecs
    walletConfigFileReReadTime = seconds
    setLastUseDateOnActivation = true|false
    maxSendReservationsToSync = num
    useTimeFromConfigFile = true|false
    waltResvnExpiryToleranceSeconds = num
    removeExpiredNotRemoved = true|false
    pluginSkipTimeOnStartup = seconds
    gapBeforeRestartingPluginSkip = seconds
    clearEmptyBuckets = true|false

    walletCache = {
        maxSize = num
        maxLoopSize = num
        checkBeforeFlush = true|false
    }

    voucherCache = {
        maxLoopSize = num
        checkBeforeFlush = true|false
        flushPeriodSeconds = 60 # -1
        maxSize = num
        voucherRevokeOnTimeout = false
            # when a voucher reservation is expired revoke it if set to true
            # this takes precedence over voucherCommitOnTimeout
        voucherCommitOnTimeout = false
            # when a voucher reservation is expired commit it if set to true
            # however voucherRevokeOnTimeout takes precedence if set
    }

    groveller = {
        periodMsec = millisecs
        requestHighWaterMark = num
        walletLowWaterMark = num
        requestTimeout = seconds
        peerDatabaseLogin = "login"
        peerWalletCheckRetrySeconds = seconds
        secondaryConnectionDelaySeconds = seconds
    }

    duplicateDetection = {
        keepDirectSeconds = seconds
        keepSyncSeconds = seconds
        directMaxDelaySeconds = seconds
        syncMaxDelaySeconds = seconds
    }

    setLastActivationDateStates = [states]

    plugins = [
        "lib"
        [...]
    ]

    handlers = [
        "lib"
        [...]
    ]

    syncWriter = {
        maxRecordsPerFile = num
        maxSecondsPerFile = seconds
    }

    dbWriter = {
        flushPeriod = seconds
        cdrOutputDirectory = "dir"

        balanceCreateBufferSize = num
        balanceUpdateBufferSize = num
        balanceDeleteBufferSize = num
        bucketCreateBufferSize = num
        bucketUpdateBufferSize = num
        bucketDeleteBufferSize = num
        walletCreateBufferSize = num
        walletUpdateBufferSize = num
        walletDeleteBufferSize = num
        voucherCreateBufferSize = num
        voucherUpdateBufferSize = num
        voucherDeleteBufferSize = num
    }
}
```

Parameters

Here are the available parameters in the `beVWARS` section of **eserv.config**.

`clearEmptyBuckets`

Syntax:	`clearEmptyBuckets = `*`true`*`|`*`false`*
Description:	Controls the empty bucket deletion. Mainly used in case of balance update flow for commit sequence of normal call.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	• true – Existing functionality will be retained i.e. empty buckets will be deleted. • false – Empty buckets won't be deleted.
Default:	true
Notes:	Other flows like Expiry flow and WalletUpdate flow can also be controlled using this flag i.e. when this flag value is set to false, it will override deleteEmptyBuckets and removeEmptyBuckets flag value, irrespective of what value is configured for them.
Example:	`clearEmptyBuckets = false`

`createBucketExpiryDays`

Syntax:	`createBucketExpiryDays = `*`days`*
Description:	In rare cases, beVWARS must spontaneously create a new bucket to preserve a wallet's last use date. This occurs, for example, when: • The last bucket is deleted. • A call is made when a wallet does not contain any buckets. • A recharge occurs against a Balance Type with no buckets. This parameter defines the new bucket's Balance Expiry Date.
Type:	Integer
Units:	Days
Optionality:	Optional (default used if not set).
Allowed:	• 0 – Creates the bucket with no expiry date. • A positive integer – Sets the bucket's Balance Expiry Date to this many days in the future.
Default:	30
Notes:	
Example:	`createBucketExpiryDays = 30`

`downstreamOverloadSleepUSec`

Syntax:	`downstreamOverloadSleepUSec = `*`microsecs`*
Description:	When a downstream process (beSync or beServer) is overloaded, this parameter specifies the amount of time to sleep, in microseconds, before rechecking the downstream process.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	100000

Notes:	This value must be shorter than the SLEE watchdog timeout period.
Example:	`downstreamOverloadSleepUSec = 100000`

handlers

Syntax:

```
handlers = [
    "lib"
    [...]
 ]
```

Description:	Lists the beVWARS message handler plug-ins to load.
Type:	Array
Optionality:	Optional (default used if not set).
Allowed:	
Default:	
Notes:	This array must include handlers for messages from processes requesting billing actions. For more information about handlers from other applications, see the associated technical guide.

Example:

```
handlers = [
    "beVWARSCCDRHandler.so"
 ]
```

maxDownstreamQueueLength

Syntax:	`maxDownstreamQueueLength = `*`num`*
Description:	Specifies the maximum number of pending events on beSync or beServer.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	10000
Notes:	When this number is exceeded, if events are queued on either of the processes, beVWARS sleeps.
Example:	`maxDownstreamQueueLength = 10000`

maxOpenDialogTime

Syntax:	`maxOpenDialogTime = `*`seconds`*
Description:	Specifies how long, in seconds, to try to open dialogs to the other SLEE processes.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	5.0
Notes:	
Example:	`maxOpenDialogTime = 5.0`

maxSendReservationsToSync

Syntax:	`maxSendReservationsToSync = `*`num`*
Description:	When beSync has requested all reservations, this is the number to send in one pass.
Type:	Integer

Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	This parameter controls the number of reservation contexts to send to the remote beSync (sink) in one pass. Between each `maxSendReservationsToSync` number of reservations sent, a single real time event will be processed. Lowering this will alter the ratio of reservation contexts sent to real time events processed by the active beVWARS.
Example:	`maxSendReservationsToSync = 1000`

maxTransactionsPerSet

Syntax:	`maxTransactionsPerSet =` *`num`*
Description:	Specifies the number of transactions and EDRs to try initially to fit into a TransactionSet written to the sync files.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	7
Notes:	This TransactionSet will be passed across the SLEE on the other VWS, so must fit inside a SLEE event. If this number of transactions does not fit the message, the message is re-encoded with fewer and fewer Transactions and EDRs per TransactionSet. 7 is used because 7.75 132 byte Transactions fit into a 1024 bytes SleeEvent.
Example:	`maxTransactionsPerSet = 7`

minResyncReservationLength

Syntax:	`minResyncReservationLength =` *`seconds`*	
Description:	Minimum reservation length (in seconds) before passing the reservation to the other Voucher and Wallet Server in a pair.	
Type:	Integer	
Optionality:	Optional (default used if not set).	
Allowed:	positive integer	Minimum
	0	Resync all reservations.
Default:	5	
Notes:	Set this parameter to reduce the amount of reservations which are sent where they will have expired by the time they are received by the other VWS.	
Example:	`minResyncReservationLength = 5`	

plugins

Syntax:

```
plugins = [
    "lib"
    [...]
 ]
```

Description:	Lists the beVWARS event handler plug-ins to load.
Type:	Parameter array
Optionality:	
Allowed:	

Default:	
Notes:	Where plug-ins are triggered by the same event, they will operate in the order they appear in this list. For more information about plug-ins from other applications, see the associated technical guide.
Example:	`plugins = [ "beVWARSExpiry.so" ]`

```
plugins = [
    "beVWARSExpiry.so"
 ]
```

pluginSkipTimeOnStartup

Syntax:	`pluginSkipTimeOnStartup = `*`seconds`*
Description:	The number of seconds to hold/stop periodic charge plugin executions, expiry bucket processing and grovelling, whenever beVWARS starts and first client event is received or, when a first client event is received after a failover from primary to secondary VWS.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	0
Notes:	In primary VWS, this parameter is used only during start-up. After start-up, as soon as first client event is received, beVWARS will start skipping periodic charge plugin executions, expiry bucket processing and grovelling for `pluginSkipTimeOnStartup` seconds. It will resume after the configured seconds. In secondary VWS, it is used during start-up as well as during a failover from primary to secondary VWS. To identify, that a client event is received in secondary because of primary failover, configuration parameter `gapBeforeRestartingPluginSkip` is checked.
Example:	`pluginSkipTimeOnStartup = 30`

gapBeforeRestartingPluginSkip

Syntax:	`gapBeforeRestartingPluginSkip = `*`seconds`*
Description:	The delay in number of seconds after which any client event is treated as failover so that `pluginSkipTimeOnStartup` can restart.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	600
Notes:	Used only in secondary VWS. When secondary VWS starts serving client traffic after a gap of `gapBeforeRestartingPluginSkip` seconds, then beVWARS will skip PC plugin executions, expiry bucket processing and grovelling again for the configured `pluginSkipTimeOnStartup` seconds.
Example:	`gapBeforeRestartingPluginSkip = 60`

`removeExpiredNotRemoved`

Syntax:	`removeExpiredNotRemoved =` *`true`* `|` *`false`*
Description:	Controls whether to remove the buckets with extended expiry from expiredNotRemoved list or not. Whenever there is an active reservation on a bucket and it expires, the bucket is kept in expiredNotRemoved list till the existing reservation is active. Any new call will not be granted from a bucket which is there in expiredNotRemoved list. There are cases where the expiry can be extended for a bucket while it is there in expiredNotRemoved list. In such cases, as soon as the expiry is extended, we may want to remove the bucket from expiredNotRemoved list so that it can be used in new calls as the expiry is extended. This flag can be utilized for such scenarios, where we would want to grant from a bucket whose expiry has been extended.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	• true – Remove the buckets from expiredNotRemoved list whose expiry has been extended. • false – Do not remove buckets from expiredNotRemoved list even if the expiry is extended.
Default:	false
Notes:	
Example:	`removeExpiredNotRemoved = true`

`reservationExpiryCheckMilliseconds`

Syntax:	`reservationExpiryCheckMilliseconds =` *`millisecs`*
Description:	The frequency (milliseconds) that the reservation expiry check occurs.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	10000
Notes:	
Example:	`reservationExpiryCheckMilliseconds = 5000`

`setLastActivationDateStates`

Syntax:	`setLastActivationDateStates = [`*`states`*`]`
Description:	A list of all old wallet states which will cause the 'Last Activation Date' for the wallet to be updated. The new wallet state in all these cases will be "ACTV" after the call has completed.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	Valid values are: PREU, FROZ, DORM, SUSP, and TERM
Default:	PREU
Notes:	
Example:	`setLastActivationDateStates = [PREU,DORM]`

`setLastUseDateOnActivation`

Syntax:	`setLastUseDateOnActivation = `*`true | false`*
Description:	Specifies whether beVWARs creates a new bucket for all balance types.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	• true – When a wallet is activated, beVWARs stores balances in one bucket and sets each balance's last use date. • false – When a wallet is activated, beVWARs creates a new bucket for all balance types and sets the wallet's last use date.
Default:	true
Notes:	
Example:	`setLastUseDateOnActivation = true`

`useTimeFromClient`

Syntax:	`useTimeFromClient = `*`true|false`*
Description:	Specifies whether beVWARS retrieves the time for every wallet from either the incoming message or the OS.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	• true – Uses the time from the incoming message (client date and usec). • false – Applies the OS time to every wallet as transactions are applied.
Default:	true
Notes:	This ensures that a given client message is treated identically on both VWSs, when it is re-sent. Otherwise, duplicate detection will cause the VWSs to get out of sync.
Example:	`useTimeFromClient = true`

`useTimeFromConfigFile`

Syntax:	`useTimeFromConfigFile = `*`true|false`*
Description:	Specifies whether beVWARS reads the time from the `useTimeFromClient` parameter or the **VWARS_sysdate.cfg** file. This parameter allows beVWARS to read a configuration file on disk for every call to detect time mapping changes and apply them. This allows you to change the time applied to wallets on the fly without an application restart.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	• true – Use the time from the **/IN/service_packages/E2BE/etc/VWARS_sysdate.cfg** file. • false – Use the time set in the `useTimeFromClient` parameter. See *useTimeFromClient*.
Default:	false
Notes:	Set this parameter to true for functional tests only. This parameter should be disabled for performance tests and production systems.
Example:	`useTimeFromConfigFile = true`

`voucherReservationPeriodSeconds`

Syntax:	`voucherReservationPeriodSeconds = `*`seconds`*
Description:	The number of seconds that vouchers remain ‘reserved’.

Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	120
Notes:	
Example:	`voucherReservationPeriodSeconds = 120`

`walletConfigFileReReadTime`

Syntax: `walletConfigFileReReadTime =` *`seconds`*

Description: Specifies how often, in seconds, beVWARS reads/parses the wallet time configuration file (**VWARS_sysdate.cfg**). beVWARS saves parsed values in a map, which is then queried until the time specified in `walletConfigFileReReadTime` elapses.

To use this parameter, the `useTimeFromConfigFile` parameter must be set to **true**.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

- 0 – beVWARS does not read/parse the wallet time configuration file.
- A positive integer – beVWARs reads/parses the wallet time configuration file at the specified interval. A value between 30 and 300 is recommended.

Default: 300

Notes:

Example: `walletConfigFileReReadTime = 300`

`waltResvnExpiryToleranceSeconds`

Syntax: `waltResvnExpiryToleranceSeconds =` *`num`*

Description: Specifies the number of seconds to add to the expiration time of wallet reservations during the data synchronization process.

The data synchronization process introduces a slight delay between the time a wallet reservation is sent from the remote VWS server to the local VWS server. This means a wallet reservation could expire before it is received by the local VWS server. Use this parameter to ensure that wallet reservations expire sometime after the data synchronization process completes.

Type: Integer

Optionality: Optional (default used if not set)

Allowed:

- 0 – Does not add any time to wallet reservation expirations.
- A positive integer – The number of seconds to add to the expiration time of wallet reservations.

Default: 30

Notes:

Example: `waltResvnExpiryToleranceSeconds = 20`

duplicateDetection parameters

Duplicate messages are possible, as the BeClient can switch to the auxiliary beServer after the original beServer has processed the message, but failed to return a response. The beVWARS detects duplicates by keeping a list of the client, clientMessageId and clientMsgTimestamps from messages received directly from the local beServer or received via beSync.

It is not necessary to store message identifiers permanently, as the BeClient switchover time is finite. For a given stream of messages, it is possible to tell that the client has not switched over.

Example: If the client is set to switch over at 1:00, but a message is received dated 3:00 from the beServer, we know that the BeClient has not switched over (or messages would not continue to be received via the beServer).

If we receive a message through the beServer dated 2:00, we know that we will not receive any other messages (from the beClient) through the beServer dated 1:30 or earlier. Due to wallet locks, messages are not always received exactly in the beClientMessageTimestamp order.

`directMaxDelaySeconds`

Syntax:	`directMaxDelaySeconds = `*`seconds`*
Description:	The maximum delay (in seconds) before IDs received directly are removed from the main stream. IDs are kept for this time, in order to compare them with IDs from the converse stream.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1.0
Notes:	It is recommended that the delay be kept to a minimum, so that the timestamp order is not affected too greatly.
Example:	`directMaxDelaySeconds = 1.0`

`keepDirectSeconds`

Syntax:	`keepDirectSeconds = `*`seconds`*
Description:	The time (in seconds) to keep IDs from messages received directly (through beServer), for comparison later with IDs from messages received through sync (through beSync).
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	60.0
Notes:	If a message has been kept much longer than the latest message received, we know that the BeClient has not switched beServers, so duplicates are not possible.
Example:	`keepDirectSeconds = 60.0`

`keepSyncSeconds`

Syntax:	`keepSyncSeconds = `*`seconds`*
Description:	The time (in seconds) to leave the IDs in the duplicate map, to wait for a duplicate.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	60.0
Notes:	
Example:	`keepSyncSeconds = 60.0`

`syncMaxDelaySeconds`

Syntax:	`syncMaxDelaySeconds = `*`seconds`*
Description:	The maximum delay (in seconds) before IDs are removed from the duplicate map. IDs are kept for this time, in order to compare them with IDs from the converse stream.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1.0
Notes:	It is recommended that the delay be kept to a minimum, so that the timestamp order is not affected too greatly.
Example:	`syncMaxDelaySeconds = 1.0`

groveller parameters

The `groveller` sub-section of the `beVWARS` provides the configuration for the groveller subsystem in beVWARS to query wallets and run plug-ins against them. Wallets are supplied by wallet ID from beGroveller. For more information about how wallets are groveled, see *Background processing* (on page 6).

`peerDatabaseLogin`

Syntax:	`peerDatabaseLogin = "`*`login`*`"`
Description:	If peerDatabaseLogin is not "", the groveller will perform a remote database query on the other VWS's database for information on the wallet. If inconsistent, another check is attempted after the number of seconds configured in peerWalletCheckRetrySeconds. This gives the remote VWS time to process and commit transactions. If the wallets are still inconsistent, a syslog message is produced.
Type:	
Units:	
Optionality:	Optional (default used if not set).
Allowed:	
Default:	""
Important:	This should not be used in production due to impact on performance.
Example:	`peerDatabaseLogin = ""`

`peerWalletCheckRetrySeconds`

Syntax:	`peerWalletCheckRetrySeconds = `*`seconds`*
Description:	How long (in seconds) to wait after the peer VWS's information on a wallet, if found to be inconsistent. If the wallet is still inconsistent after this period a syslog message is produced.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	60
Notes:	This period should be long enough for locally generated operation message to be sent to the remote VWS, and committed to the database.
Example:	`peerWalletCheckRetrySeconds = 60`

`periodMsec`

Syntax:	`periodMsec = `*`millisecs`*
Description:	The minimum number of milliseconds between groveling wallets.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	0 — Send a new request as soon as possible. positive integer
Default:	1000
Notes:	This parameter is restricted by `requestHighWaterMark` (on page 106). Setting request frequency to 0 will impact the normal VWS processing speed. This parameter will have no effect if groveling is disabled (for example, if `enableGrovelling` (on page 41)is set to false).
Example:	`periodMsec = 1200`

`requestHighWaterMark`

Syntax:	`requestHighWaterMark = `*`num`*
Description:	The maximum number of outstanding requests queued on the SLEE waiting on this beVWARS instance before no requests are sent to beGroveller for wallets to grovel.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1

Notes:

If the number of outstanding requests in the SLEE queue is less than or equal to half of requestHighWaterMark, then this beVWARS instance will try to grovel a wallet between every request it processes.

If the number of outstanding requests in the SLEE queue is more than half of requestHighWaterMark, wallets will be groveled with decreasing frequency proportional to the queue length, until only one wallet is groveled every for every 10 requests from beServer. The proportion is calculated to approximately increase the total number of requests (SLEE queue + grovel requests) to requestHighWaterMark.

Examples: These examples assume requestHighWaterMark = 100.

- If queue length = 25, beVWARS will process one grovel request for each SLEE queue request (that is , 50 of every 100 requests will be grovel requests).
- If queue length = 50, beVWARS will process one grovel request for each SLEE queue request (that is, 50 of every 100 requests will be grovel requests).
- If queue length = 80, beVWARS will process two grovel requests for every eight SLEE queue requests (that is, 20 of every 100 requests will be grovel requests).
- If queue length = 90, beVWARS will process one grovel request for every 10 SLEE queue requests (that is, 10 of every 100 requests will be grovel requests).
- If queue length = 100, beVWARS will not process any grovel requests.

Example: `requestHighWaterMark = 100`

`requestTimeout`

Syntax:	`requestTimeout =` *`seconds`*
Description:	The maximum number of seconds to wait for a response after sending a request to beGroveller for another batch of wallet IDs to grovel, before timing out the request.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	positive integer
	-1 Do not time out requests.
Default:	30
Notes:	If a request is timed out, beVWARS will log a Warning level error to syslog and will resend the request. If errors are being logged to the syslog indicating timeouts, try: • Setting *walletLowWaterMark* (on page 11) to a higher value • Setting requestTimeout to a higher value • If there are not as many beGroveller processes as beVWARS processes, adding the number of beGroveller processes (this can be done until there are as many beGroveller processes as beVWARS processes) • Setting *maxIDsPerResponse* (on page 62) to a higher value
Example:	`requestTimeout = 30`

`secondaryConnectionDelaySeconds`

Syntax:	`secondaryConnectionDelaySeconds =` *`seconds`*
Description:	The number of seconds to wait before enabling the beGroveller to process wallets on the secondary VWS if the primary VWS cannot be contacted. Ensures that wallet processing by the beGroveller is not started on the secondary VWS if the primary VWS is down for a short period of time only.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	900
Notes:	The configured delay is applied when the secondary VWS beGroveller is started and when the client connection from the secondary VWS beGroveller to the primary beServer is lost.
Example:	`secondaryConnectionDelaySeconds = 900`

`walletLowWaterMark`

Syntax:	`walletLowWaterMark =` *`num`*
Description:	The number of outstanding wallet IDs to grovel, before sending a request to beGroveller for another batch of wallet IDs to grovel.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	100
Notes:	
Example:	`walletLowWaterMark = 100`

voucherCache parameters

The `voucherCache` sub-section of the `beVWARS` parameters defines the voucher cache.

`checkBeforeFlush`

Syntax:	`checkBeforeFlush =` *`true|false`*
Description:	If true, vouchers are checked against the database before flushing. If they are different, an error is sent to syslog.
Type:	Boolean
Units:	
Optionality:	Optional (default used if not set).
Allowed:	true, false
Default:	false
Important:	This should not be used in production due to impact on performance.
Example:	`checkBeforeFlush = false`

`maxLoopSize`

Syntax:	`maxLoopSize =` *`num`*	
Description:	The number of vouchers in the voucher cache for beVWARS to process at one time before pausing. This enables beVWARS to pause and respond to other requests, instead of attempting to process the whole cache at once.	
Type:	Integer	
Optionality:	Optional (default used if not set).	
Allowed:	Positive integers	Number of vouchers to process in a batch.
	-1	Process whole cache at once.
Default:	10000	
Notes:	This parameter should be set if `maxSize` and `maxAgeSeconds` are set to -1 (which is likely to cause a large cache). If this number is set too high (or to -1) the SLEE watchdog can restart beVWARS unnecessarily. Too low, and beVWARS will waste CPU polling the SLEE unnecessarily.	
Example:	`maxLoopSize = 5000`	

`maxSize`

Syntax:	`maxSize =` *`MB`*
Description:	The maximum size of the beVWARS voucher cache.
Type:	Integer
Units:	MB
Optionality:	Optional (default used if not set).
Allowed:	0 or a positive integer.
Default:	10000
Notes:	
Example:	`maxSize = 10000`

`voucherCommitOnTimeout`

Syntax:	`voucherCommitOnTimeout =` *`true|false`*
Description:	If true, vouchers are committed when voucher reservation expires. `voucherRevokeOnTimeout` takes precedence if set.

Type:	Boolean
Units:	
Optionality:	Optional (default used if not set).
Allowed:	true, false
Default:	false
Notes:	
Example:	`voucherCommitOnTimeout = true`

`voucherRevokeOnTimeout`

Syntax:	`voucherRevokeOnTimeout = true|false`
Description:	If true, vouchers are revoked when voucher reservation expires.
Type:	Boolean
Units:	
Optionality:	Optional (default used if not set).
Allowed:	true, false
Default:	false
Notes:	
Example:	`voucherRevokeOnTimeout = true`

walletCache parameters

The `walletCache` sub-section of the `beVWARS` parameters defines the wallet cache.

`checkBeforeFlush`

Syntax:	`checkBeforeFlush = true|false`
Description:	If true, wallets are checked against the database before flushing. If they are different, an error is sent to syslog.
Type:	Boolean
Units:	
Optionality:	Optional (default used if not set).
Allowed:	
Default:	false
Important:	This should not be used in production due to impact on performance.
Example:	`checkBeforeFlush = false`

`maxLoopSize`

Syntax:	`maxLoopSize = num`	
Description:	The number of wallets in the wallet cache for beVWARS to process at one time before pausing. This enables beVWARS to pause and respond to other requests, instead of attempting to process the whole cache at once.	
Type:	Integer	
Optionality:	Optional, default value will be used if not set.	
Allowed:	Positive integers	Number of wallets to process in a batch.
	-1	Process whole cache at once.
Default:	100000	

Notes:	This parameter should be set if `maxSize` and `maxAgeSeconds` are set to -1 (which is likely to cause a large cache). If this number is set to high (or to -1) the SLEE watchdog can restart beVWARS unnecessarily. Too low, and beVWARS will waste CPU polling the SLEE unnecessarily.
Example:	`maxLoopSize = 5000`

`maxSize`

Syntax:	`maxSize =` *`MB`*
Description:	The maximum size of the beVWARS wallet cache.
Type:	Integer
Units:	MB
Optionality:	Optional (default used if not set).
Allowed:	0 or a positive integer.
Default:	100000
Notes:	
Example:	`maxSize = 100000`

syncWriter parameters

The `syncWriter` sub-section of the `beVWARS` parameters defines how the beVWARS syncWriter writes sync files.

`maxRecordsPerFile`

Syntax:	`maxRecordsPerFile =` *`num`*
Description:	The maximum number of records in a sync file.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	100
Notes:	
Example:	`maxRecordsPerFile = 200`

`maxSecondsPerFile`

Syntax:	`maxSecondsPerFile =` *`seconds`*
Description:	The maximum number of seconds to hold a sync file open.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	2
Notes:	
Example:	`maxSecondsPerFile = 4`

dbWriter parameters

The `dbWriter` sub-section of the `beVWARS` parameters defines how the beVWARS dbWriter buffers and writes:

- Data updates to the E2BE database
- EDRs to the filesystem

Note: All buffers and the EDR cache are flushed whenever one of the following conditions occurs:

- One of the buffers is full
- The EDR cache is full
- The current flush period has ended
- The beVWARS writer subsystem is told to flush and commit (on shutdown for example)

`balanceCreateBufferSize`

Syntax:	`balanceCreateBufferSize =` *`num`*
Description:	The number of items in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`balanceCreateBufferSize = 1500`

`balanceDeleteBufferSize`

Syntax:	`balanceDeleteBufferSize =` *`num`*
Description:	The number of deletes in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`balanceDeleteBufferSize = 1000`

`balanceUpdateBufferSize`

Syntax:	`balanceUpdateBufferSize =` *`num`*
Description:	The number of updates in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`balanceUpdateBufferSize = 1000`

`bucketCreateBufferSize`

Syntax:	`balanceCreateBufferSize =` *`num`*
Description:	The number of bucket creates in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	

Default:	1000
Notes:	
Example:	`balanceCreateBufferSize = 1000`

`bucketDeleteBufferSize`

Syntax:	`bucketDeleteBufferSize =` *`num`*
Description:	The number of bucket deletes in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`bucketDeleteBufferSize = 1000`

`bucketUpdateBufferSize`

Syntax:	`bucketUpdateBufferSize =` *`num`*
Description:	The number of bucket updates in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`bucketUpdateBufferSize = 1000`

`cdrOutputDirectory`

Syntax:	`cdrOutputDirectory = "`*`dir`*`"`
Description:	Directory where EDRs are written to.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"/IN/service_packages/E2BE/logs/CDR"
Notes:	EDRs will be stored in this directory until they are moved by another process.
Example:	`cdrOutputDirectory = "/var/EDRs/UBE/"`

`flushPeriod`

Syntax:	`flushPeriod =` *`seconds`*
Description:	The maximum number of seconds between flushes.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	10
Notes:	
Example:	`flushPeriod = 20`

voucherUpdateBufferSize

Syntax:	`voucherUpdateBufferSize = `*`num`*
Description:	The number of voucher updates in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`voucherUpdateBufferSize = 1000`

voucherCreateBufferSize

Syntax:	`voucherCreateBufferSize = `*`num`*
Description:	The number of voucher creates in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`voucherCreateBufferSize = 1000`

voucherDeleteBufferSize

Syntax:	`voucherDeleteBufferSize = `*`num`*
Description:	The number of voucher deletes in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`voucherDeleteBufferSize = 1000`

walletCreateBufferSize

Syntax:	`walletCreateBufferSize = `*`num`*
Description:	The number of wallet creates in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`walletCreateBufferSize = 1000`

walletDeleteBufferSize

Syntax:	`walletDeleteBufferSize = `*`num`*
Description:	The number of wallet deletes in a buffer before beVWARS will flush it.
Type:	Integer

Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`walletDeleteBufferSize = 1000`

`walletUpdateBufferSize`

Syntax:	`walletUpdateBufferSize =` *`num`*
Description:	The number of wallet updates in a buffer before beVWARS will flush it.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	1000
Notes:	
Example:	`walletUpdateBufferSize = 1000`

Tracing parameters

The tracing parameters allow tracing to be performed for individual wallet IDs on selected be clients.

Where more than one criteria (wallet and client) is configured for tracing then the message must satisfy all criteria (logical AND) for tracing/debug to activate.

`beClients`

Syntax:	`beClients = ["`*`client1`*`", "`*`client2`*`", ...]`
Description:	List of BE client names to trace.
Type:	Array, String
Optionality:	Optional if walletIDs parameter supplied, mandatory if walletIDs not supplied.
Allowed:	Any beClient.
Default:	None
Notes:	The names are converted to a unique BE client hash ID - which is the same mechanism employed by the beVWARS for referencing BE clients.
Example:	see below

```
beClients = [
   "ccsBeOrb",
   "PIbeClient"
]
```

`debugLevel`

Syntax:	`debugLevel = "`*`level`*`"`
Description:	The debug level/filter, is equivalent to DEBUG environment variable.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	Any of the DEBUG options.
Default:	"all"
Notes:	This is a comma separated string. See traceDebugLevel in *ACS Technical Guide* for more information.
Example:	`debugLevel = "all"`

enabled

Syntax:	`enabled = `*`true|false`*
Description:	The tracing activation switch to allow tracing of selected wallet and/or be client activity.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true, false
Default:	false
Notes:	
Example:	`enabled = true`

walletIds

Syntax:	`walletIds = `*`[ID1, ID2,`*` ...]`
Description:	List of subscriber wallet ids we want to trace.
Type:	Array, Integer
Optionality:	Optional if beClient parameter supplied, mandatory if beClient not supplied.
Allowed:	Any valid wallet ID.
Default:	None
Notes:	To obtain the wallet id(s) for a given CLI/subscriber use the showCLI.sh script on the BE where tracing is to occur.
Example:	`walletIds = [ 382, 385 ]`

Example configuration

This is an example `beVWARS` section **eserv.config** on a VWS node (comments have been removed).

```
beVWARS = {

    voucherReservationPeriodSeconds = 120
    useTimeFromClient = true
    maxTransactionsPerSet = 7
    maxOpenDialogTime = 5.0
    maxDownstreamQueueLength = 10000
    downstreamOverloadSleepUSec = 100000
    minResyncReservationLength = 5
    createBucketExpiryDays = 30
    maxSendReservationsToSync = 1000
    removeExpiredNotRemoved = true
    reservationExpiryCheckMilliseconds = 10000
    setLastUseDateOnActivation = true
    pluginSkipTimeOnStartup = 30
    gapBeforeRestartingPluginSkip = 60
    clearEmptyBuckets = true

    walletCache = {
        maxSize = 10000
        checkBeforeFlush = false
        maxLoopSize = 500
    }
```

```
voucherCache = {
    checkBeforeFlush = false
    maxLoopSize = 500
    flushPeriodSeconds = 60 # -1
    maxSize = 2
    voucherRevokeOnTimeout = false
        # when a voucher reservation is expired, revokes it if set to true
        # this takes precedence over voucherCommitOnTimeout
    voucherCommitOnTimeout = false
        # when a voucher reservation is expired, commits it if set to true
        # however voucherRevokeOnTimeout takes precedence if set
}

groveller = {
    periodMsec = 1200
    requestHighWaterMark = 1
    walletLowWaterMark = 100
    requestTimeout = 300
    peerDatabaseLogin = ""
    peerWalletCheckRetrySeconds = 60
    secondaryConnectionDelaySeconds = 900
}

duplicateDetection = {
    keepDirectSeconds = 60.0
    keepSyncSeconds = 60.0
    directMaxDelaySeconds = 1.0
    syncMaxDelaySeconds = 1.0
}

setLastActivationDateStates = [
    [PREU]
]

plugins = [
    "beVWARSExpiry.so"
]

handlers = [
    "beVWARSCCDRHandler.so"

]

syncWriter = {
    maxRecordsPerFile = 100
    maxSecondsPerFile = 2
}

dbWriter = {
    flushPeriod = 10
    cdrOutputDirectory = "/IN/service_packages/E2BE/logs/CDR"

    balanceCreateBufferSize = 1000
    balanceUpdateBufferSize = 1000
    balanceDeleteBufferSize = 1000
    bucketCreateBufferSize = 1000
    bucketUpdateBufferSize = 1000
    bucketDeleteBufferSize = 1000
    walletCreateBufferSize = 1000
    walletUpdateBufferSize = 1000
    walletDeleteBufferSize = 1000
    voucherCreateBufferSize = 1000
    voucherUpdateBufferSize = 1000
    voucherDeleteBufferSize = 1000
```

```
        }
        tracing = {
            enabled = true
            debugLevel = "all"
            walletIds = [
                382,
                385
            ]
            beClients = [
                "ccsBeOrb",
                "PIbeClient"
            ]
        }

        setLastActivationDateStates = [
            "PREU"
        ]

    } # BE.beVWARS
```

Output

Each beVWARS writes error messages to the system messages file, and also writes additional output to its own log file. By default this is:

/IN/service_packages/E2BE/tmp/beVWARS0.log

Note: The actual name will be different for each beVWARS process.

beVWARSCCDRHandler

Purpose

The beVWARSCCDRHandler provides a specific EDR-generating function. This is generally used where no other process in an interaction will produce an EDR, but an EDR should still be generated.

Example: If a voucher redeem fails because the voucher cannot be found on any Voucher and Wallet Server, the client process will send a request to the beVWARSCCDRHandler to write an EDR for the failed voucher redeem.

Startup

beVWARSCCDRHandler.so is included in the beVWARS by specifying it in the handlers array.

For more information about the handlers array, see *handlers* (on page 98).

Configuration

This binary has no specific configuration.

beVWARSExpiry

Purpose

beVWARSExpiry monitors subscriber accounts and wallets, checking for subscriber accounts and wallets which have passed their expiry date. If it finds a subscriber account or wallet which requires expiring, it processes the record as configured.

Startup

If beVWARSExpiry is configured in **eserv.config**, it is started by beVWARS when beVWARS is initialized. It is included in the beVWARS handlers section.

```
handlers = [
    "beVWARSExpiry.so"
]
```

For more information about beVWARS:

- Plug-ins, see *Plug-ins* (on page 94).
- Handlers section, see *handlers* (on page 98)

Note: Other handlers can also be included in the handlers list.

Configuration

beVWARSExpiry accepts the following parameters from **eserv.config**.

```
beVWARSExpiry = {
    expireNegativeBuckets = true|false
    removeEmptyBuckets = true|false
    expireBucketsForExpiredWallets = true|false
    expireAtMidnightTZ = "timezone"
    terminatedWalletConsistencyCheck = true|false

}
```

Parameters

Here are the available parameters in the `beVWARSExpiry` section of the **eserv.config**.

expireAtMidnightTZ

Syntax:	`expireAtMidnightTZ = "`*`timezone`*`"`
Description:	Sets wallets and buckets to expire at midnight for the time zone specified.
Type:	String
Optionality:	Optional (default used if not set)
Allowed:	The time zone part of the parameter must be typed in a form that the operating system recognizes. Alternatively you can select a time zone from the operating system's list. To view top-level time zone names, type `ls /usr/share/lib/zoneinfo` from a shell. To see second-level time zone names type `ls /usr/share/lib/zoneinfo` *`TopLevelName`*`/`. For example, to verify that the operating system recognizes a time zone name for DeNoranha, in Brazil, you would type `ls /usr/share/lib/zoneinfo/Brazil/`. DeNoranha is listed, so the time zone name would be "`Brazil/DeNoranha`".
Default:	GMT
Notes:	A list of time zones can be found in the Time Zones appendix of *ACS Technical Guide*. This parameter does not affect the expiry calculations of periodic charge buckets.

Example:	An account is created at 2 p.m. on 5 September 2014 and is set to have a life span of 24 days. If `expireAtMidnightTZ = "Asia/Vladivostok"` is included, the account will expire on 29 September 2014 at midnight, Vladivostok time. If this parameter is omitted, the account will expire on 29 September 2014 at 2:00 PM GMT.

expireBucketsForExpiredWallets

Syntax:	`expireBucketsForExpiredWallets =` *`true`*`|`*`false`*
Description:	Controls whether wallet expiry triggers bucket expiry. If true, any buckets under the wallet will be expired when the wallet expires.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true — All buckets with a positive or zero value in a wallet will be expired when the wallet is expired, even if the buckets are not due to expire yet. false — Buckets are expired when their own expiry date passes. **Note:** This means the wallet will not be deleted from the system, but will instead be set to Removed state and kept until the last bucket is expired.
Default:	false
Notes:	If `expireNegativeBuckets` (on page 119) is set to false, buckets with a negative value will not be deleted, regardless of the value of this parameter. Using this parameter will remove any positive value the wallet holds when the wallet expires.
Example:	`expireBucketsForExpiredWallets = false`

expireNegativeBuckets

Syntax:	`expireNegativeBuckets =` *`true`*`|`*`false`*
Description:	Whether or not to expire buckets which have a negative value.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	true — Buckets with negative values are expired when their expiry date passes. Buckets with negative values (where the subscriber is in debit), are expired when their expiry date passes. false — Buckets are expired when their expiry date passes and they have a positive or 0 balance. **Note:** This means wallets with negative balances will not be deleted from the system, but will instead be set to Removed state and kept until the last bucket is expired.
Default:	false
Notes:	This parameter is designed to enable the Telco to keep the wallet until all outstanding money has been recovered from the subscriber.
Example:	`expireNegativeBuckets = false`

`removeEmptyBuckets`

Syntax:	`removeEmptyBuckets = true|false`
Description:	Whether or not to remove buckets when they have a value of 0.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	false — Leave buckets to expire as normal. true — If true, any buckets with 0 value will be removed.
Default:	false
Notes:	Setting this to false does not stop beVWARSExpiry removing buckets for expired wallets if `expireBucketsForExpiredWallets` (on page 119) is set to true. When `clearEmptyBuckets` flag is set to false, `removeEmptyBuckets` flag will be disabled.
Example:	`removeEmptyBuckets = false`

`terminatedWalletConsistencyCheck`

Syntax:	`terminatedWalletConsistencyCheck = true|false`
Description:	Whether to check the wallet cache against the database for terminated wallets. If `terminatedWalletConsistencyCheck` is set to true, then beVWARSExpiry checks for terminated wallets in the wallet cache and if the status is different in the database, updates the wallet status to terminated in the database.
Type:	Boolean
Optionality:	Optional (default used if not set)
Allowed:	true (perform database consistency check on terminated wallets) false (do not perform database consistency check)
Default:	false
Notes:	Set this parameter to true if wallet expiry transactions are incorrectly synchronized with the database; for example, if the database failed when a wallet expired for the first time.
Example:	`terminatedWalletConsistencyCheck = true`

Example configuration

This is an example of the `beVWARSExpiry` section of an **eserv.config** file from a VWS (comments have been removed).

```
beVWARSExpiry = {
    expireNegativeBuckets = false
    removeEmptyBuckets = false
    expireBucketsForExpiredWallets = false
    expireAtMidnightTZ = "Asia/Vladivostok"
}
```

Failure

If beVWARSExpiry fails, it will not trigger expiry events for any Expiry plug-in. When beVWARSExpiry recovers, it will process as normal, and will catch up with any expired wallets or buckets.

Output

The beVWARSExpiry writes error messages to the system messages file, and also writes additional output to the following default (can vary as per configuration):

/IN/service_packages/E2BE/tmp/beVWARSExpiry.log

beVWARSMergeBuckets

Purpose

beVWARSMergeBuckets is a plug-in library for beVWARS.

This beVWARS plug-in merges buckets in the same balance when there are too many buckets in the wallet. If there are too many buckets the message detailing the wallet contents will not fit in a 1024 byte SLEE event and can cause errors.

Merging begins with the balances that have the most buckets. In each balance, the bucket with the earliest expiry has its value added to the next bucket, then it is removed. This is repeated until the wallet has the maximum allowed number of buckets left.

Configuration

beVWARSMergeBuckets accepts the following parameters from **eserv.config**.

```
maxBuckets = num
triggerPlugins = true|false
```

Parameters

Here are the available parameters in the `beVWARSMergeBuckets` section of the **eserv.config**.

maxBuckets

Syntax:	`maxBuckets = num`	
Description:	The maximum number of buckets a wallet can have.	
Type:	Integer	
Units:		
Optionality:	Optional (default used if not set).	
Allowed:	-1	No maximum.
	positive integer	Maximum number of buckets.
Default:	-1	
Notes:		
Example:	`maxBuckets = -1`	

triggerPlugins

Syntax:	`triggerPlugins = true|false`
Description:	When we merge buckets (update the value of one and delete the other), should we trigger other beVWARS plug-in.
Type:	Boolean
Units:	
Optionality:	Optional (default used if not set).
Allowed:	
Default:	false
Notes:	
Example:	`triggerPlugins = false`

Example configuration

This is an example of the `beVWARSExpiry` section of an **eserv.config** file on a VWS (comments have been removed).

```
beVWARSMergeBuckets = {
    maxBuckets = -1
    triggerPlugins = false
}
```

cmnPushFiles

Purpose

cmnPushFiles transfers files to specific directories on the SMS from SLCs and VWSs. The files transferred include:

- EDRs
- PIN logs

Note: Other Oracle applications also use their own instances of this process.

Startup

This task is started by entry scp1 in the inittab, using the shell script:

```
/IN/service_packages/SMS/bin/cmnPushFilesStartup.sh
```

Configuration

cmnPushFiles accepts the following command-line options:

Usage:

```
cmnPushFiles -d dir [-o dir [-a age]] [-f dir] [-F] [-P prefix] [-S suffix] -h host
[-r prefix] [-p port] [-s seconds] [-R seconds] [-M seconds] [-C seconds] [-t
bitrate] [-T] [-x] [-e] [-w seconds]
```

The available parameters are:

Parameter	Default	Description
`-d`		Destination directory for files on remote machine. **Example:** The directory on SLC where the cmnPushFiles looks for the files to be sent to the SMS.
`-o`	File deleted	Transferred directory.
`-a`	Never delete files	Age of transferred files before being deleted. This parameter only relevant when `-o` option is specified.
`-f`	none	Retry directory.
`-F`	Do not use	Use fuser to not move files in use.
`-P`	none	File prefix.
`-S`	none	File suffix.
`-h`	none	Remote hostname.
`-r`	none	Remote directory prefix. **Note:** Required if `-d` is relative directory.

Parameter	Default	Description
`-p`	2027	Port on remote machine on which the cmnReceiveFiles will listen for receiving files. **Note:** -1 for stdin/stdout.
`-s`	15	Sleep period in seconds.
`-R`	15	Seconds before Initial retry period in seconds.
`-M`	900	Maximum retry period in seconds.
`-C`	1800	Cleanup period in seconds.
`-t`	none (no throttling)	Throttles transfer to *nnn* bits per second.
`-T`	off (non-recursive)	Tree move: recursive into subdirectories.
`-x`	On (use prefixing)	Do not use hostname-prefixing on remote filenames.
`-e`	Daemon mode	Non-daemon mode. Run file transfer only once, then exit.
`-w`	30	Time to wait for success in seconds.

Example:

```
cmnPushFiles -d /IN/service_packages/SMS/cdr/closed -f
/IN/service_packages/SMS/cdr/retry -r /IN/service_packages/SMS/cdr/received -h
prodsmp1.telcoexample.com -s 10 -p 2028 -S cdr -w 20
```

Parameters

Here are the available parameters in the `cmnPushFiles` section of the **eserv.config** file.

`CDR`

Syntax:

```
CDR = [
     "param"[, "value"]
     [...]
 ]
```

Description: Arguments to cmnPushFiles when used to send EDRs to SMS.

Type: Array

Optionality: Optional (default used if not set).

Allowed: See cmnPushFiles documentation in *SMS Technical Guide*.

Default:

Notes:

Example:

```
CDR = [
     "-d", "/IN/service_packages/E2BE/logs/CDR-out"
     "-r", "/IN/service_packages/CCS/logs/CDR-in"
     "-h", "smp1prod"
     "-F"
]
```

`-d`

Syntax: `"-d", "dir"`

Description: Local source directory.

Type: String

Optionality:	
Allowed:	
Default:	
Notes:	
Example:	`"-d", "/IN/service_packages/E2BE/logs/CDR-out"`

`-r`

Syntax:	`"-r", "`*`dir`*`"`
Description:	Remote destination directory.
Type:	String
Optionality:	
Allowed:	
Default:	
Notes:	
Example:	`"-r", "/IN/service_packages/CCS/logs/CDR-in"`

`-h`

Syntax:	`"-h", "`*`host`*`"`
Description:	Full host name and domain of the SMS machine.
Type:	String
Optionality:	
Allowed:	
Default:	
Notes:	
Example:	`"-h", "smp1prod"`

`-F`

Syntax:	`-F`
Description:	Do not send the file if a process is currently using it.
Type:	Boolean
Optionality:	Optional (file sent if not set).
Allowed:	
Default:	
Notes:	
Example:	`-F`

Example configuration

This is an example of the `cmnPushFiles` section of an **eserv.config** file on a VWS (comments have been removed).

```
cmnPushFiles = {
    CDR = [
        "-d", "/IN/service_packages/E2BE/logs/CDR-out"
        "-r", "/IN/service_packages/CCS/logs/CDR-in"
        "-h", "smp1hostname"
        "-F"
    ]
}
```

Failure

If cmnPushFiles fails, EDRs will accumulate in:

/IN/service_packages/SMS/cdr/current/

cmnPushFiles will send error messages to the syslog and the cmnPushFiles log.

Output

The cmnPushFiles writes error messages to the system messages file, and also writes additional output to this default location:

/IN/service_packages/SMS/tmp/cmnPushFiles.log

Event Storage Interface

Overview

The event storage interface stores events to be sent to a different SLEE interface at a future time.

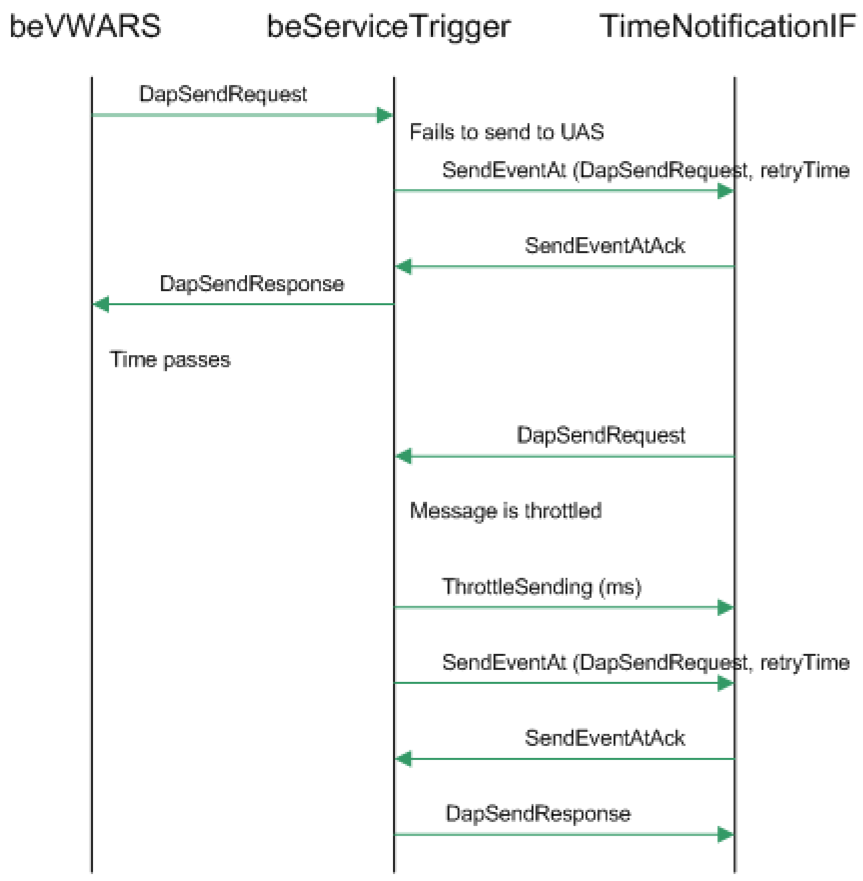
When it is time to send an event, the event storage interface sends the event to the specified SLEE interface and waits for a response. The response can be one of the following:

- DIALOG_CLOSED: An error occurred, and the event will be retried later.
- Any event other than DIALOG_CLOSED: The event was delivered successfully. The event storage interface removes the event from the queue.

To prevent it from spamming the outbound interface with events, the storage interface accepts a throttle message, which inserts a gap between events sent to the interface.

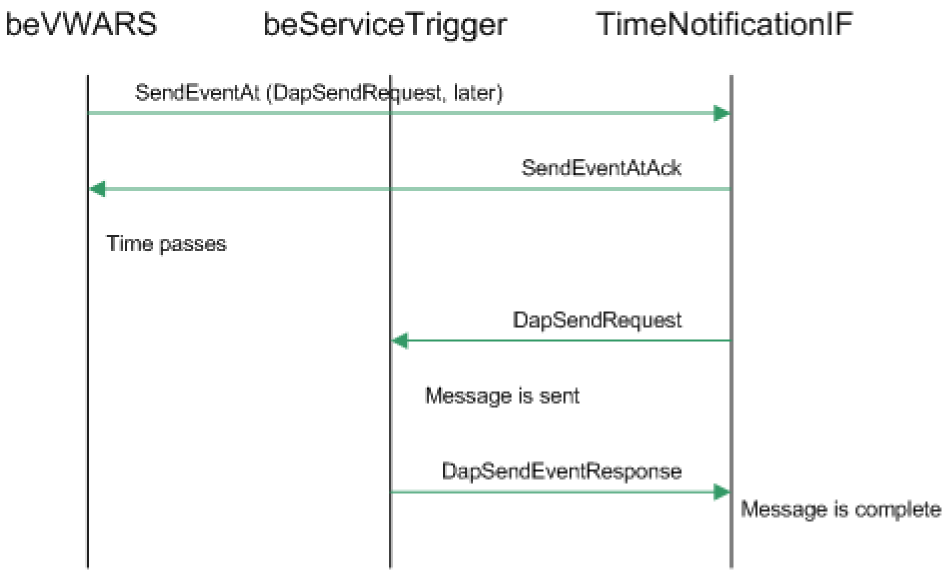
Error and throttle flow

Here is an example flow showing the interaction between the plug-in and beServiceTrigger when an event that is to be sent immediately encounters multiple failures: first the SLC is down, and then a second failure occurs due to throttling.



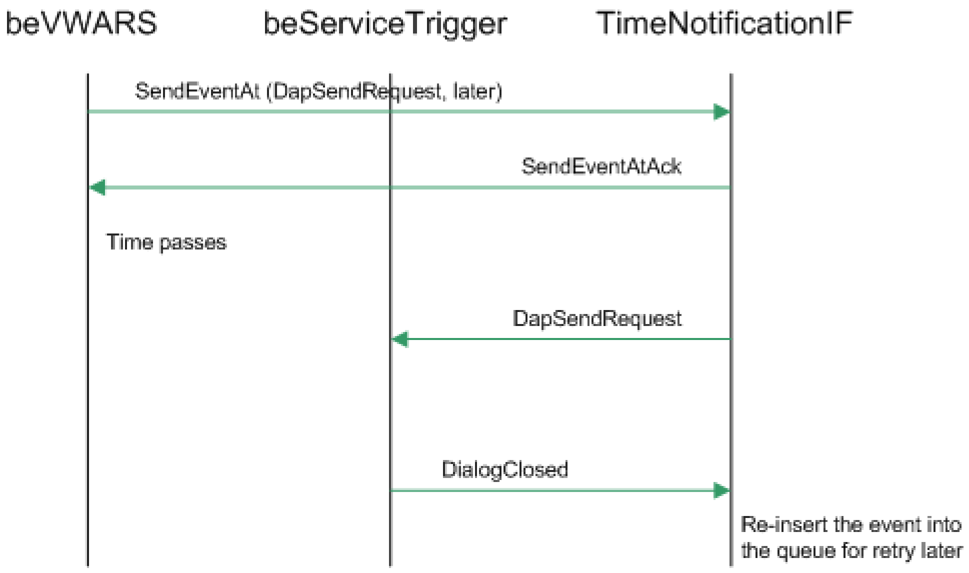
Send later flow

Here is an example flow showing the interaction between the plug-in and beServiceTrigger for an event that is to be sent later.



Crash flow

Here is an example flow showing the interaction between the plug-in and beServiceTrigger when an event that is to be sent immediately encounters a beServiceTrigger failure.



Event Storage SLEE Events

Trigger events for event storage, plus parameters are:

- SenSendEventAt:
  When
  InterfaceName

EventToSend

- SendEventAck
  Success
- ThrottleSending
  TimeBetweenEvents
  InstanceToThrottledEventAt

Configuration

The event storage interface accepts the following parameters from **eserv.config**.

```
eventStorage = {
    nextEventWindowTime = millisecs
    NumberOfRows = num
    sleepTime = millisecs
}
```

Parameters

Parameters of the `eventStorage` group are listed below.

nextEventWindowTime

Syntax:	`nextEventWindowTime = millisecs`
Description:	The timeout interval, in milliseconds, for the next event that is sent. This allows time for the event storage interface to check for further events, rather than get caught processing timeouts.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	A positive integer
Default:	100
Notes:	The event timeout is set to one of the following, depending on whichever is greatest: • The current time plus `nextEventWindowTime` • Next event time
Example:	`nextEventWindowTime = 75`

NumberOfRows

Syntax:	`NumberOfRows = num`
Description:	The number of rows that the event storage interface returns from a single re-read iteration, before the SLEE is checked for a management event.
Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	A positive integer
Default:	10
Notes:	
Example:	`NumberOfRows = 15`

sleepTime

Syntax:	`sleepTime = millisecs`
Description:	The amount of time, in milliseconds, that the event storage interface sleeps when waiting for new events. This time is used when no new events are due before the specified time interval elapses.

Type:	Integer
Optionality:	Optional (default used if not set)
Allowed:	Any positive integer
Default:	1
Notes:	The amount of time that the event storage interface sleeps is set to one of the following, depending on whichever is shortest: • `sleepTime` • Next event time
Example:	`sleepTime = 2`

libbeMsgRouterDefault

Purpose

libbeMsgRouterDefault is a beServer plug-in which determines which beVWARS to direct ESCHER messages to.

Startup

libclientBcast is used by beServer process if its configuration includes the library. To be used, it must be included in the `messageRoutingPlugins` array as shown:

```
messageRoutingPlugins = [
    "libbeMsgRouterDefault.so"
]
```

For more information about the beServer messageRoutingPlugins section, see *messageRoutingPlugins* (on page 71).

Configuration

The libbeMsgRouterDefault's configuration is read from the `beServer` section of **eserv.config**. libbeMsgRouterDefault supports the following parameters:

- All parameters in `msgRouterDefault` (on page 71)
- `routingVoucherNumberLength` (on page 74)

libBeClientIF

Purpose

The libBeClientIF provides an interface to one or more VWS Voucher and Wallet Servers.

Startup

The libBeClientIF is a runtime dependency of processes which need to talk to the VWS Voucher and Wallet Servers.

Configuration

The libBeClientIF's configuration is usually read from the section which configures the process which is using it. libBeClientIF supports the following parameters.

- `beLocationPlugin` (on page 41)
- *clientName* (on page 54)
- *heartbeatPeriod* (on page 55)
- *messageTimeoutSeconds* (on page 56)
- *maxOutstandingMessages* (on page 55)
- *reportPeriodSeconds* (on page 58)
- *connectionRetryTime* (on page 55)
- *plugins* (on page 57)
- *notEndActions* (on page 56)

Notes:

- Any process which is using the libBeClientIF can use these parameters.
- `beLocationPlugin` is located in the shared parameters section.

libclientBcast

Purpose

libclientBcast is used by BeClient processes to send messages which may be answered by any Voucher and Wallet Server. The request is sent to one of the VWSs in all Voucher and Wallet Server pairs at once. The plug-in is activated by sending a message to BE ID 0.

This process is specifically used to redeem vouchers, and vouchers cannot be redeemed if this library is not loaded.

Startup

libclientBcast is used by BeClient processes if their configuration includes the library.

If libclientBcast is used by BeClient and ccsBeOrb, it is included in the `plugins` array as shown:

```
plugins = [
     {
         config=""
         library="libclientBcast.so"
         function="makeBroadcastPlugin"
     }
 ]
```

For more information about the BeClient `plugins` section, see *plugins* (on page 57).

Configuration

The libBeClientIF's configuration is usually read from the section which configures the process which is using it. libBeClientIF supports the `aggregateNAckCodes` (on page 54) parameter.

libbeEventFactory

Purpose

libbeEventFactory is the common library used to create SLEE events from ESCHER messages. It is required by the system and should not be removed.

Startup

libbeEventFactory is used by a number of processes on the VWS. No startup configuration is required for this library to be used.

Configuration

This binary has no specific configuration.

Chapter 4

Tools and Utilities

Tools and Utilities Overview

Introduction

This chapter provides a description of the operational programs or executables that can used on the VWS. You can run these processes as needed.

In this chapter

This chapter contains the following topics.

VWS Correction Tool

Purpose

Use the correction tool to restore consistent data in the system following a software fault or configuration error, without causing further outage or lost service for any node in the system.

Use the correction tool for making a small number of changes to fields that cannot be maintained via the User Interface or Provisioning Interface.

Rollback

There is no rollback as such. The commands can be run again with pre-corrected data to reach the previous state.

About minimizing tool impact

Any changes to the VWS database will affect the performance of the server. To mitigate performance degradation, several of the configuration parameters can be used to effectively throttle the tool.

The operator can:

- Limit number of concurrent commands executing in one binary by using the **maxQueueSize** configuration parameter. The tool will not issue new requests unless there are currently less than or equal to maxQueueSize commands in flight.
- Impose time delay between successive commands by using the **maxCommandsPerSecond** configuration parameter. The tool will issue this maximum number of commands per second. When used in conjunction with the maxQueueSize parameter, allows an even tighter throttle if required. For example, with a maxQueueSize of 1, the tool can still direct a strong volume commands, but if it is further moderated by a setting such as maxCommandsPerSecond=(say) 2 or even 1, then the traffic can be brought under tighter control.
- Queue or reject concurrent requests to same business object by using the **queueUpdatesToSameObject** configuration parameter. The tool can either allow concurrent or serial commands to run against the

same business object. The default is **false**, which means that a file of commands for the same wallet will be run serially, **true,** the commands are run in parallel.

Starting the commands

For individual changes, the commands can be typed directly into the command line.

For many changes, the command lines can be entered into a batch file which is then executed from the command line.

eserv.config parameters

`clientName`

Syntax:	`clientName = "`*`value`*`"`
Description:	The unique client name to connect to the database with.
Type:	String
Optionality:	Optional (default used if not set).
Allowed:	
Default:	"ccsAccount"
Notes:	Only one connections with the same name is allowed.
Example:	`clientName = "nccdemo-dev-ccsVWSCorrection"`

`heartbeatPeriod`

Syntax:	`heartbeatPeriod = `*`value`*
Description:	The number of microseconds since previous message before fail over to the other VWS.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	0 for no heartbeating.
Default:	30000000 (30 seconds)
Notes:	If no heartbeat or other messages received in this period we switch to the other VWS in the pair on the assumption that the current VWS has failed.
Example:	`heartbeatPeriod = 10000000`

`connectionRetryTime`

Syntax:	`connectionRetryTime = `*`value`*
Description:	The number of seconds before we try to reconnect.
Type:	Integer
Optionality:	Optional (default used if not set).
Allowed:	
Default:	5
Notes:	
Example:	`connectionRetryTime = 2`

`plugins`

Syntax:	`plugins = `*`value`*
Description:	Identifies which plugins to load.

Type:	Array
Optionality:	Optional (default used if not set).
Allowed:	
Default:	[] (empty, no plugins)
Notes:	Not currently used, for future potential use.
Example:	`plugins = []`

Syntax:	`billingEngines =` *`value`*
Description:	beLocationPlugin values override.
Type:	Array
Optionality:	Optional (default used if not set).
Allowed:	
Default:	beLocationPlugin billing engine values
Notes:	Used to override the beLocationPlugin that would normally load the connection details from the DB.
Example:	see below

```
billingEngines = [
    {id = 1,primary = { ip="PRIMARY_BE_IP", port=1500 },secondary
    = { ip="SECONDARY_BE_IP", port=1500 }
    }
]
```

maxQueueSize

Syntax:	`maxQueueSize =` *`value`*
Description:	Number of concurrent commands executing in one binary.
Type:	Integer
Optionality:	Mandatory.
Allowed:	
Default:	
Notes:	The tool will not issue new requests unless there are currently less than or equal to maxQueueSize commands in flight. In conjunction with maxCommandsPerSecond can be used to control impact on the VWS.
Example:	`maxQueueSize = 10`

queueUpdatesToSameObject

Syntax:	`queueUpdatesToSameObject =` *`value`*
Description:	Controls whether multiple updates to a single business object are done in parallel or serially.
Type:	Boolean
Optionality:	Optional (default used if not set).
Allowed:	• true – Allow parallel updates • false – Serial updates only
Default:	false
Notes:	
Example:	`queueUpdatesToSameObject = true`

Syntax:	`notificationInterval =` *`value`*
Description:	The number of seconds between reporting progress status to the log file.

Type:	Integer
Optionality:	Mandatory
Allowed:	
Default:	
Notes:	
Example:	`notificationInterval = 6`

Syntax:	`maxCommandsPerSecond =` *`value`*
Description:	The maximum number of commands allowed in flight per second.
Type:	Integer
Optionality:	Mandatory
Allowed:	
Default:	
Notes:	
Example:	`maxCommandsPerSecond = 10`

Syntax:	`pollTimeUsecs =` *`value`*
Description:	The number of microseconds to wait for a message from a Billing Engine client before polling.
Type:	Integer
Optionality:	Mandatory
Default:	1000000 (one second)
Example:	`pollTimeUsecs = 100000`

Example of VWS correction tool section

This is an example of the **eserv.config** file correction tool section.

```
ccsVWSCorrection = {

    ClientIF = {

        clientName = "nccdemo-dev-ccsVWSCorrection"

        heartbeatPeriod = 10000000

        connectionRetryTime = 2

        plugins = []

        billingEngines = [
            {id = 1,
            primary   = { ip="PRIMARY_BE_IP", port=1500 },
            secondary = { ip="SECONDARY_BE_IP", port=1500 }
            }
        ]

    }

    maxQueueSize = 10
    queueUpdatesToSameObject = false
    notificationInterval = 6
    maxCommandsPerSecond = 10
    pollTimeUsecs = 100000
} # CCS.ccsVWSCorrection section
```

Command line parameters

The commands that can be used are:

- delete_balance:
  This deletes the balance from the wallet ID and balance type ID.
- delete_bucket:
  This updates the supplied bucket ID by zeroing the current value of the bucket. The mechanism to physically delete the bucket is up to other (pr-existing) configuration on the VWS as to whether or not zero value buckets are retained or deleted.
- update_balance:
  This updates the supplied balance fields with the new values.
- update_bucket:
  This updates supplied bucket fields with the new values.
- update_wallet:
  This updates the supplied wallet fields with the new values.

See Command line examples.

Update balance parameters

The **update_balance:** command has the ability to modify the following fields to schema and business rule acceptable values against a specified balance for a specified wallet:

- limit_type
- minimum_credit

The balance and wallet key data is supplied in these fields:

- wallet_id
- balance_type

An example of the update_balance command is:

```
update_balance:wallet_id=4,balance_type=9,limit_type=LCRD,minimum_credit=888
81000
```

Update bucket parameters

The **update_bucket:** command has the ability to modify the following fields to schema and business rule acceptable values against a specified bucket for a specified balance and wallet:

- expiry
- value
- value_delta
- reference
- start_date
- last_use
- never_expires
- never_used

The balance and wallet key data is supplied in these fields:

- wallet_id
- balance_type

- bucket_id

Examples of the update_bucket command is:

```
update_bucket:wallet_id=4,balance_type=9,bucket_id=2,expiry=20160101115500
```

```
update_bucket:wallet_id=4,balance_type=9,bucket_id=2,value=5,value_delta=1
```

Update wallet parameters

The **update_wallet:** command has the ability to modify the following fields to schema and business rule acceptable values against a specified wallet:

- max_concurrent state
- never_expires
- expiry
- never_activated
- activation_date
- state

The wallet key data is supplied in this field:

- wallet_id

Examples of the update_bucket command is:

```
update_wallet:wallet_id=4,never_expires=true,expiry=20110101115600
```

```
update_wallet:wallet_id=47,state=ACTV
```

Delete balance parameters

The **delete_balance:** command deletes the balance.

The balance key data is supplied in these fields:

- wallet_id
- balance_type

Example of the delete_balance command is:

```
delete_balance:wallet_id=4,balance_type=99
```

Delete bucket parameters

The **delete_bucket:** command modifies the bucket value field to zero, allowing the VWS to retain or delete the bucket.

The bucket key data is supplied in these fields:

- wallet_id
- balance_type
- bucket_id

Example of the delete_bucket command is:

```
delete_bucket:wallet_id=44,balance_type=13,bucket_id=30
```

Command line examples

The commands can be run singularly by typing in at the command line prompt, or as a batch in a file.

This is an example of a file of commands that will do a set of updates (picture a file with 450 lines of the following) that generated the Progress reporting and Audit reporting examples.

update_bucket:wallet_id=4,balance_type=9,bucket_id=2,value=5,value_delta=1

```
update_bucket:wallet_id=4,balance_type=9,bucket_id=2,value=5,value_delta=1
```

lots of lines (447) deleted for conciseness.

```
update_bucket:wallet_id=4,balance_type=9,bucket_id=2,value=5,value_delta=1
```

Other examples are:

- update_wallet:wallet_id=4,never_expires=true
- update_balance:wallet_id=4,balance_type=9,limit_type=LCRD
- delete_balance:wallet_id=44,balance_type=9
- delete_bucket:wallet_id=4,balance_type=13,bucket_id=30

Progress reporting

The tool sends report information to the logfile, including, the parameters at the start, the status periodically as it executes, and the details of each command processed.

Here is a an example showing the expected reporting. The tool is executing a series of commands that add 5c to a particular bucket repeatedly (450 times).

To see the report, on the command line type (for example):

```
-bash-3.00$ ./ccsVWSCorrection -i commandFile -o logFile
```

The logfile report will look something like this:

```
Aug  2 14:54:37.208548 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection processing starting
Aug  2 14:54:37.213175 ccsVWSCorrection(29583) NOTICE: Connection to BE 1:192.168.10.217-1500 is established.
Aug  2 14:54:37.547017 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 0 of 450 commands: 0.0% complete
Aug  2 14:54:43.091905 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 30 of 450 commands: 6.7% complete
Aug  2 14:54:49.107811 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 72 of 450 commands: 16.0% complete
Aug  2 14:54:55.031967 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 106 of 450 commands: 23.6% complete
Aug  2 14:55:01.058072 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 145 of 450 commands: 32.2% complete
Aug  2 14:55:07.002602 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 179 of 450 commands: 39.8% complete
Aug  2 14:55:13.107238 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 204 of 450 commands: 45.3% complete
Aug  2 14:55:19.081310 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 238 of 450 commands: 52.9% complete
Aug  2 14:55:25.046720 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 278 of 450 commands: 61.8% complete
Aug  2 14:55:31.141610 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 310 of 450 commands: 68.9% complete
Aug  2 14:55:37.082081 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 346 of 450 commands: 76.9% complete
Aug  2 14:55:43.022000 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 386 of 450 commands: 85.8% complete
Aug  2 14:55:49.096070 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 418 of 450 commands: 92.9% complete
```

Aug   2 14:55:54.407038 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection Tool status: processed 450 of 450 commands: 100.0% complete

Aug   2 14:55:54.407308 ccsVWSCorrection(29583) NOTICE: ccsVWSCorrection processing complete

Audit reporting

The audit log contains structured fields (keyed by command number, time stamp and log record type. This is to permit convenient grepping, filtering, sorting and analysis of the log records after the run.

A log record can be of type:

- COMMAND: dumping the command being called
- INFO: displaying any informational message
- WARNING: displaying some warning condition
- ERROR: displaying an error in order to explain why the command did not run
- AUDIT: for commands that got as far as an update request, one or more of these show what fields were modified. Commands that delete business objects will display the current value of that object and any children it contains, to assist with recovery should it be necessary.

This is the log from the Command line examples and Progress reporting examples.

```
00000001 [20110802145437.549545] COMMAND:
update_bucket:wallet_id=4,balance_type=9,bucket_id=2,value=5,value_delta=1
00000002 [20110802145437.652071] COMMAND:
update_bucket:wallet_id=4,balance_type=9,bucket_id=2,value=5,value_delta=1
00000002 [20110802145437.652117] WARNING: We're already executing a command for wallet ID 4
(we'll retry shortly..)
00000001 [20110802145437.671977] AUDIT  : update_bucket:
wallet_id=4,balance_type=9,bucket_id=2,old_value=6330,new_value=6335,old_reference=,new_referen
ce=,old_start_date=19700101000000,new_start_date=19700101000000,old_never_expires=1,new_never_e
xpires=1,old_expiry=19700101000000,new_expiry=19700101000000,old_never_used=0,new_never_used=0,
old_last_use=20110802025232,new_last_use=20110802025232
00000003 [20110802145437.882058] COMMAND:
update_bucket:wallet_id=4,balance_type=9,bucket_id=2,value=5,value_delta=1
00000003 [20110802145437.882146] WARNING: We're already executing a command for wallet ID 4
(we'll retry shortly..)
00000003 [20110802145437.990985] WARNING: We're already executing a command for wallet ID 4
(we'll retry shortly..)
00000002 [20110802145438.045385] AUDIT  : update_bucket:
wallet_id=4,balance_type=9,bucket_id=2,old_value=6335,new_value=6340,old_reference=,new_referen
ce=,old_start_date=19700101000000,new_start_date=19700101000000,old_never_expires=1,new_never_e
xpires=1,old_expiry=19700101000000,new_expiry=19700101000000,old_never_used=0,new_never_used=0,
old_last_use=20110802025437,new_last_use=20110802025437
00000003 [20110802145438.172316] AUDIT  : update_bucket:
wallet_id=4,balance_type=9,bucket_id=2,old_value=6340,new_value=6345,old_reference=,new_referen
ce=,old_start_date=19700101000000,new_start_date=19700101000000,old_never_expires=1,new_never_e
xpires=1,old_expiry=19700101000000,new_expiry=19700101000000,old_never_used=0,new_never_used=0,
old_last_use=20110802025438,new_last_use=20110802025438
```

lots of lines removed for conciseness.

```
00000449 [20110802145553.946776] COMMAND:
update_bucket:wallet_id=4,balance_type=9,bucket_id=2,value=5,value_delta=1
00000449 [20110802145554.013305] AUDIT  : update_bucket:
wallet_id=4,balance_type=9,bucket_id=2,old_value=8570,new_value=8575,old_reference=,new_referen
ce=,old_start_date=19700101000000,new_start_date=19700101000000,old_never_expires=1,new_never_e
xpires=1,old_expiry=19700101000000,new_expiry=19700101000000,old_never_used=0,new_never_used=0,
old_last_use=20110802025553,new_last_use=20110802025553
00000450 [20110802145554.122045] COMMAND:
update_bucket:wallet_id=4,balance_type=9,bucket_id=2,value=5,value_delta=1
00000450 [20110802145554.122104] WARNING: We're already executing a command for wallet ID 4
(we'll retry shortly..)
00000450 [20110802145554.295870] AUDIT  : update_bucket:
wallet_id=4,balance_type=9,bucket_id=2,old_value=8575,new_value=8580,old_reference=,new_referen
ce=,old_start_date=19700101000000,new_start_date=19700101000000,old_never_expires=1,new_never_e
xpires=1,old_expiry=19700101000000,new_expiry=19700101000000,old_never_used=0,new_never_used=0,
old_last_use=20110802025554,new_last_use=20110802025554

Statistics:
   Completed commands = 450
   Information acks received = 450
   Primary information acks received = 450
   Primary update acks received = 450
   Total commands = 450
   Update acks received = 450
   Wallet Info Requests sent to Primary BE = 450

 ccsVWSCorrection stopped at Tue Aug  2 14:55:54 2011
```

The log report shows that all commands were (eventually, there were some cases of the tool waiting for a previous update for the same wallet ID to finish, but this is normal in a file with multiple commands against the same wallet) successful, and the bucket value grew from an initial balance of 6330 (old value on first audit record) to 8580 (new value on last audit record). The difference is 450 x 5, so all updates were applied correctly.

beEventStorageIFDump

Purpose

The beEventStorageIFDump utility is a diagnostic tool for finding bottlenecks in the BE_EVENT_STORAGE database table. It parses through the table and lists the number, type, and contents of the storage events.The utility parses data exported from the database.

This tool allows decoding of notifications in the BE_EVENT_STORAGE table. Decoding can occur via direct query of the database, or by parsing of the exported table.

Location

The beEventStorageIFDump utility is located on the SMS node.

Syntax

You start the beEventStorageIFDump utility from the command line by using the following syntax:

`beEventStorageIFDump -a|-m` *`Value`* `[-b` *`FieldFile`*`] [-c` *`EventFieldName`*`] [-d] [-f` *`DumpFile`*`] [-g] [-h] [-l` *`ListFile`*`] [-n] [-r` *`MSISDN`*`] [-R` *`Date`*`] [-s` *`StartEvent`*`] [-v 1|2|3]`

The following table describes the beEventStorageIFDump command line parameters.

Parameter	Description
-a	Lists all events that are in the BE_EVENT_STORAGE table.
-m *Value*	Writes all events that match the specified value to a file. • ALL – Writes all events from the BE_EVENT_STORAGE table to a file named **BE_EVENT_STORAGE_***EventFieldName***_allevents**. • *FieldValue* – Finds all events that have a matching event field value and writes them to a file named **BE_EVENT_STORAGE_***EventFieldNameValue*. where *EventFieldName* is the name of a field in the BE_EVENT_STORAGE table, and *FieldValue* is the value of any field in the BE_EVENT_STORAGE table.
-b *FieldFile*	The default file name is **beEventHeaders.txt**.

Parameter	Description
-c *EventFieldName*	Finds all events in the table that have a matching event field name. **Note:** You can further narrow the results by adding the -m *FieldValue* option to specify the required value of the field.
-f *DumpFile*	Specifies to write the event contents to the specified file. The default file name is **BE_EVENT_STORAGE.dmp**.
-g	Displays all event details.
-h	Displays the command-line syntax and parameters.
-l *ListFile*	Specifies to write the output of the utility to the specified file. The default file name is **beEventHeaders.txt**.
-n	Don't show event data as it finds it????
-r *MSISDN*	Match all entries based on <msisdn> in RequestTime window, use with -R
-R *Value*	match all entries with request times that occur in the interval |<date> - <n> days to <date>| where <date> is the value specified by -m<value> An msisdn match can be specified with -r<msisdn> this option requires -c <time based event field name> e.g. -cRequestTime
-s *StartEvent*	Use eventFieldName <startEvent> as the start of the event dump file, default='beServiceTrigger' use in conjunction with -m
-v 1|2|3	Verbose: 1 (minimum), 2(mid), 3 (max)

Command line examples

The commands can be run singularly by typing in at the command line prompt, or as a batch in a file.

The following example shows the syntax to list all MSISDN in exported table.

- show all MSISDN in exported table

(i) show all msisdn in exported table
  (a) list event field names to match against : ../beEventStorageIFDump -l
  (b) list Calling_Party_id for all events, show frequency: ../beEventStorageIFDump -cCalling_Party_id
  (c) list matched events, e.g.: ../beEventStorageIFDump -cCalling_Party_id -m12345678
  (d) to reduce display, add -n

To show all MSISDN in an exported table

Step	Action
1	List all events and their field names from the BE_EVENT_STORAGE table by entering the following command: `beEventStorageIFDump -l`
2	List the value of each event's CALLING_PARTY_ID field by entering the following command: `beEventStorageIFDump -c CALLING_PARTY_ID`
3	List all events that have a value of 12345678 in the CALLING_PARTY_ID field: `beEventStorageIFDump -c CALLING_PARTY_ID -m 12345678`

```
beEventStorageIFDump -l -c CALLING_PARTY_ID -m 12345678 -n
```

The following example shows the syntax

(a) ../beEventStorageIFDump -cCalling_Party_id -m12345678 -g
(b) to reduce display, add -n
(c) to show all events, regardless of -c/-m match, add -a

```
beEventStorageIFDump -c Calling_Party_id -m 12345678 -g
```

(iii)match all MSISDN with direct read from database
(a) ../beEventStorageIFDump -cCalling_Party_id -m12345678 -S
generates result in BE_EVENT_STORAGE_Calling_Party_id12345678
(b) to output to debug, rather than file, add -d

(iv) match all events on, or previous to <date> in DB dump
(a) ../beEventStorageIFDump -fBE_EVENT_STORAGE2.dmp -cRequestTime -m2012-02-01 -r<> -n
(b) to match all events between -m<date> and the previous n days, add -R<n>
Note: date is expected to be 'YYYY-MM-DDTHH-MM-SSZ'
e.g.: -mYYYY, -mYYYY-MM, -mYYYY-MM-DD etc accepted

```
beEventStorageIFDump -f BE_EVENT_STORAGE2.dmp -cRequestTime -m2016-02-01 -r -n
```

beServiceTriggerUser

Purpose

The beServiceTriggerUser utility sets the user name and password that beServiceTrigger uses to log in to external systems remotely; for example, when sending service requests to a client ASP through the NCC Open Services Development (OSD) component. The beServiceTriggerUser utility stores the user name and password in a secure credentials vault on the SMS node.

Location

The beServiceTriggerUser utility is located on the SMS node.

Startup

You start the beServiceTriggerUser utility from the command line by using the following syntax:

```
beServiceTriggerUser [-d user/password] [-u STUsername] [-p STpassword] [-r]
```

The following table describes the beServiceTrigger command line parameters.

Parameter	Description
`-d` *`user/password`*	(Optional) The oracle user and password to use to log in to the database on the SMS. If you omit the `-d` option, then beServiceTriggerUser uses the database login specified in the oracleUserAndPassword parameter in the BE section of **eserv.config**. Defaults to '/' if `-d` is not specified and oracleUserAndPassword is not set.
`-u` *`STUsername`*	(Optional) The name of the beServiceTrigger user. If you omit the `-u` option, beServiceTriggerUser prompts for a name.
`-p` *`STpassword`*	(Optional) The password for the beServiceTrigger user. If you omit the `-p` option, beServiceTriggerUser prompts for a password.
`-r`	(Optional) Specifies to delete the password.

Setting the beServiceTrigger User and Password

Follow these steps to set the username and password for the `beServiceTrigger` process by using the `beServiceTriggerUser` utility.

Step	Action
1	Log in to the SMS as user `smf_oper`.
2	Go to the directory where `beServiceTriggerUser` is located.

3	Enter the following command to set the username and password for `beServiceTrigger`:
	`beServiceTriggerUser [-d user/password] [-u STusername] [-p ST_password]`
	Where:
	• *`user/password`* is the login ID for the Oracle database. The login specified in the `oracleUserAndPassword` parameter is used if you omit the `-d` option. If this is not set, then "/" is used.
	• *`STusername`* is the remote login name for the `beServiceTrigger` user. If you omit the `-u` option, then `beServiceTriggerUser` prompts for a name.
	• *`ST_password`* is the new password for the `beServiceTrigger` user. If you omit the `-p` option, then beServiceTriggerUser prompts for a password.
	Tip: To remove the `beServiceTrigger` user and password, enter the following command:
	`beServiceTriggerUser -r`

Chapter 5

Troubleshooting

Overview

Introduction

This chapter explains the important processes on each of the server components in NCC, and describes a number of example troubleshooting methods that can help aid the troubleshooting process before you raise a support ticket.

In this chapter

This chapter contains the following topics.

Common Troubleshooting Procedures

Introduction

Refer to *System Administrator's Guide* for troubleshooting procedures common to all NCC components.

Possible Problems

Introduction

This topic lists common problems and actions you can take to investigate or solve them. This list enables you to check for alarms based on the overall behavior you are experiencing.

Database failure

Upon network failure, any request or response may be lost.   Pending Database (DB) write and EDRs will be lost.

Failure scenarios

This table lists a range of failure scenarios and a description of the events that will happen as a result.

For more information about resynchronization, see *Resynchronizations* (on page 22).

Scenario	Resulting Events
VWS is running. beServer core dumps, losing all contexts and BeClient connections.	**1** beServer recovers, finds the current state from beVWARS, determines that the VWS should be in Recovery state and makes it so. **2** beServer prepares to receive contexts from beServer on the other VWS. **3** beGroveller detects the dropped connection and a failover is triggered.

Scenario	Resulting Events
	4 beSync starts attempting recovery. **5** Remote beServer starts sending contexts to the local beServer (and receives Operations from beSync). **6** beSync should complete its recovery quickly (it should have already been in Sync). beSync will tell beVWARS to move to Running state, when the beVWARS has finished sending contexts it will move to Running. **7** beServer starts accepting connections from BeClient processes. **8** beGroveller establishes connection with beServer and starts grovelling.
beVWARS core dumps, loses all reservations and cached wallets, then restarts.	**1** beServer and beSync are informed that the beVWARS process had died. **2** beServer closes all open connections as nicely as it can. **3** beVWARS restarts, prepares to send contexts to beServer (and receive Operations from beSync). **4** beServer and beSync recognise beVWARS recovery. **5** beSync initiates recovery (includes getting all reservation details from other VWS). **6** beGroveller detects the dropped connection and a failover is triggered. **7** When beSync completes recovery, the process completes as above (in beServer failure).
VWS state is running. beSync core dumps and restarts.	**1** beSync restarts, gets current status from beVWARS. **2** If the beVWARS is in Recovery or Running state it starts recovery. **3** beSync processes as normal, but does not force the system into Recovery (which would deny connections, and this is not required) groveller proceeds as it was. **4** beSync proceeds, when the inSync threshold is reached it tells the beVWARS to go to Running state (which it may already be in).
Primary VWS has power turned off. BeClient detects failure of primary VWS.	**1** Primary VWS is turned back on. **2** SLEE starts up, all SLEE processes start. **3** beServer starts disabled, refuses BeClient connections. **4** beGroveller attempts to connect to beServer and fails. It doesn't start processing. **5** beSync starts, reads in existing sync file repository. At this point beSync source will not accept connections from the remote beSync sink as we do not want to send anything. **6** BeClient swaps to sending messages to Secondary VWS, resending any it does not have responses for (and marks them as duplicates). **7** beSync asks each local beVWARS for their last written Sequence Number. **8** beSync looks to see if it needs to process files locally to write updates that are in the sync files but not in the database. This is done by looking to see if there is any later sequence numbers in the files. **9** If later sequence numbers are found, they are read and sent to the beVWARS. **10** While this is proceeding, the beSync source will start accepting connections from the remote beSync sink. **11** When all local updates have been performed, beSync sink requests all reservations from the remote VWS to populate the beVWARS. **Note:** This is an extra step only performed on full recovery.

Scenario	Resulting Events
	12 The local beSync sink establishes a link to the remote VWS and requests all updates since the last remote update we have recorded on our database. **13** Updates stream to us, and we confirm them in chunks. **14** The timestamp on every update is checked against the wall clock. If the difference is less than the (configurable) inSyncThreshold then we consider ourselves to be inSync and tell beServer to start accepting connections again. **15** beServer starts accepting connections again. **16** beGroveller establishes connection with beServer and starts grovelling.

For more information about beGroveller failover, see *beGroveller quorum* (on page 148).

Process Failure Recovery

Startup checks

On startup, or failure (and restart) of the beServer or beVWARS, we must get all reservations and server contexts from the peer VWS.   Both the beServer and beVWARS must be present if this is to be successful.

Startup process

After you start the SLEE, the following events occur.

Stage	Description
1	All processes startup in disabled/startup state.
2	beSync waits until it can contact beVWARS.
3	Once beSync can, it starts local recovery, by sending updates to the beVWARS.
4	The local beSync establishes a connection to the beSync on the remote VWS, and asks for contexts and reservations.
5	After the local beSync has all the remote contexts and updates, it requests the remote updates from beSync.
6	After remote updates are within a couple of seconds of the current time, beSync tells the beVWARS to change to running state.
7	beVWARS passes the state change message to beServer.
8	After beServer has Running messages from all beVWARS, it goes into running mode, and opens for client connections.
9	If grovelling is available at this time, beGroveller can now grovel, and will respond to requests for more wallets to grovel from each of the beVWARS processes.

Restarts while in state Recovery

If you restart while in state recovery, the following events will occur.

For beSync:

- Queries the beVWARS, finds it is recovering, and starts requesting everything from the other beSync again.
- Reservations in the beVWARS are overwritten.

- Contexts in the beServer are overwritten.

For beVWARS:

- Comes up in disabled state.
- beServer gets a dialog closed event and disables itself, then it tries to contact beVWARS to put it in Recovery mode.

For beServer:

- Same as SLEE startup; contacts the beVWARS and resets Recovery mode.

Restarts while in state Running

If you restart while in state running, the following events will occur.

- beSync queries the beVWARS and starts up running.
- beVWARS comes up disabled.
- beServer will get a dialog closed, disable itself, and then same as SLEE startup. Then it will follow the same process as a SLEE restart.
- beGroveller will start disabled, and will start processing when beServer starts accepting connections.
- We have lost all of the updates between the committed database's sequence number and those in the beVWARS, however the sync files still record all of these (they have not been removed as beSync hasn't received a COMMIT message yet).
- Each beVWARS clears its cache, reads the local and remote sequence number from the database, goes into recovery mode - broadcasting these SSEQ numbers.
- beVWARS will then ignore all operations from beSync until it sees one with the SessionNumber set to the ID of the control message it sent to set the VWS state to Recovery - beSync may have operations queued on the beVWARS with SSEQs AFTER those of the database (we do not want to skip those in between).
- beSync sees these new SSEQ numbers and sends all of the local and remote transactions it has to the beVWARS. The first operation message has the SessionNumber set to the ID of the control message that set the state to Recovery.

beGroveller quorum

beGroveller is designed to only run on the primary Voucher and Wallet Server in a pair. However, groveling activity will failover to the secondary Voucher and Wallet Server if the Voucher and Wallet Server fails.

beGroveller determines whether it should pass groveling work to beVWARS processes, by checking whether it is on the primary and whether it can connect to the:

- Local beServer process
- beServer on the other VWS in the pair
- SMS specified in *quorumHost* (on page 64)

Establishing quorum on primary VWS

beGroveller determines whether it is running on a primary VWS by checking the value of `amPrimary` (on page 41). If `amPrimary` is set to true, the beGroveller is running on a primary VWS.

If the beGroveller on a primary VWS can connect to the local beServer, beGroveller will respond to beVWARS grovel requests with lists of wallet IDs to grovel. If beGroveller cannot connect to the local beServer, it assumes the VWS is disabled or recovering and will not return work to beVWARS processes.

Establishing quorum on secondary VWS

beGroveller determines whether it is running on a secondary VWS by checking the value of `amPrimary` (on page 41). If `amPrimary` is set to false, the beGroveller is running on a secondary VWS.

If beGroveller:

- Cannot connect to the local beServer, it assumes the VWS is disabled or recovering and will not return work to beVWARS processes.
- On a secondary VWS can connect to the local beServer, beGroveller will check whether it can connect to the beServer on the other VWS in the pair. If it can connect to the remote beServer, it will assume the primary is running and will not respond to beVWARS grovel requests with lists of wallet IDs to grovel.
- Can connect to the local beServer, but cannot connect to the beServer on the VWS, it will check whether it can ping the remote VWS. If it can ping the remote VWS it assumes the remote VWS is disabled and will start groveling.
- Cannot ping the remote VWS it will attempt to ping the SMS specified in *quorumHost* (on page 64). If it can ping quorumHost, it will assume the VWS pair has failed over, and will start to respond to beVWARS requests with lists of wallet IDs to grovel. If it cannot ping quorumHost, it will assume it is not on the main network, and will not respond to beVWARS requests with lists of wallet IDs to grovel.

Chapter 6

About Installation and Removal

Overview

Introduction

This chapter provides information about the installed components for the Oracle Communications Network Charging and Control (NCC) application described in this guide. It also lists the files installed by the application that you can check for, to ensure that the application installed successfully.

In this Chapter

This chapter contains the following topics.

Installation and Removal Overview

Introduction

For information about the following requirements and tasks, see *Installation Guide*:

- NCC system requirements
- Pre-installation tasks
- Installing and removing NCC packages

Voucher and Wallet Server packages

An installation of Voucher and Wallet Server includes the following packages, on the:

- SMS:
  - beSms
- SLC:
  - beScp
- VWS:
  - beBe

Checking the Installation

Introduction

Refer to this checklist to ensure that VWS has installed correctly.

The end of the package installation process specifies a script designed to check the installation just performed.   They must be run from the command line.

Checklist

Follow these steps in this checklist to ensure VWS has been installed on an VWS machine correctly.

Step	Action
1	Log into the VWS machine as root.
2	Check the following directory structure exists with the subdirectory:
	/IN/service_packages/E2BE
3	Check the directory contains subdirectories and that all are owned by:
	ebe_oper user (group esg)
4	Log into the system as ebe_oper.
	Note: This step is to check that the ebe_oper user is valid.
5	Type `sqlplus /`
	No password is required.
	Note: This step is to check that the ebe_oper user has valid access to the database.
	Ensure that the ORACLE_SID is set.
6	Ensure that VWS and CCS tables have been added to the database.
7	Check the entries of the following file:
	/etc/inittab
	Inittab Entries Reserved for VWS on VWS:
	a. `be_1   bin/beCDRMoverStartup.sh`
	(Runs beCDRMoverStartup, which moves completed EDR files into an output directory for later processing.)
	b. `be_2   bin/cmnPushFilesStartup.sh`
	(Runs cmnPushFiles, which moves the EDRs to a configured destination machine (usually the SMS).)