



Service Control Application Suite for Broadband

API Programmer's Guide

Ver. 2.5.5

OL-7207-01

Corporate Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
<http://www.cisco.com>
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 526-4100

Customer Order Number: DOC-7207-01=
Text Part Number: OL-7207-01



THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

The following information is for FCC compliance of Class A devices: This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio-frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case users will be required to correct the interference at their own expense.

The following information is for FCC compliance of Class B devices: The equipment described in this manual generates and may radiate radio-frequency energy. If it is not installed in accordance with Cisco's installation instructions, it may cause interference with radio and television reception. This equipment has been tested and found to comply with the limits for a Class B digital device in accordance with the specifications in part 15 of the FCC rules. These specifications are designed to provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation.

Modifying the equipment without Cisco's written authorization may result in the equipment no longer complying with FCC requirements for Class A or Class B digital devices. In that event, your right to use the equipment may be limited by FCC regulations, and you may be required to correct any interference to radio or television communications at your own expense.

You can determine whether your equipment is causing interference by turning it off. If the interference stops, it was probably caused by the Cisco equipment or one of its peripheral devices. If the equipment causes interference to radio or television reception, try to correct the interference by using one or more of the following measures:

- Turn the television or radio antenna until the interference stops.
- Move the equipment to one side or the other of the television or radio.
- Move the equipment farther away from the television or radio.
- Plug the equipment into an outlet that is on a different circuit from the television or radio. (That is, make certain the equipment and the television or radio are on circuits controlled by different circuit breakers or fuses.)

Modifications to this product not authorized by Cisco Systems, Inc. could void the FCC approval and negate your authority to operate the product.

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED "AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

CCSP, the Cisco Square Bridge logo, Follow Me Browsing, and StackWise are trademarks of Cisco Systems, Inc.; Changing the Way We Work, Live, Play, and Learn, and iQuick Study are service marks of Cisco Systems, Inc.; and Access Registrar, Aironet, ASIST, BPX, Catalyst, CCDA, CCDP, CCIE, CCIP, CCNA, CCNP, Cisco, the Cisco Certified Internetwork Expert logo, Cisco IOS, Cisco Press, Cisco Systems, Cisco Systems Capital, the Cisco Systems logo, Cisco Unity, Empowering the Internet Generation, Enterprise/Solver, EtherChannel, EtherFast, EtherSwitch, Fast Step, FormShare, GigaDrive, GigaStack, HomeLink, Internet Quotient, IOS, IP/TV, iQ Expertise, the iQ logo, iQ Net Readiness Scorecard, LightStream, Linksys, MeetingPlace, MGX, the Networkers logo, Networking Academy, Network Registrar, Packet, PIX, Post-Routing, Pre-Routing, ProConnect, RateMUX, ScriptShare, SlideCast, SMARTnet, StrataView Plus, SwitchProbe, TeleRouter, The Fastest Way to Increase Your Internet Quotient, TransPath, and VCO are registered trademarks of Cisco Systems, Inc. and/or its affiliates in the United States and certain other countries.

All other trademarks mentioned in this document or Website are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (0501R)

Printed in the USA on recycled paper containing 10% postconsumer waste.

Service Control Application Suite for Broadband API Programmer's Guide Ver. 2.5.5

Copyright © 2002-2005 Cisco Systems, Inc.
All rights reserved.



Introduction v

- Audience v
- Purpose vi
- Document Content vii
- Document Conventions vii
- Related Publications viii
- Obtaining Technical Assistance viii
 - Cisco TAC Website viii
 - Opening a TAC Case viii
 - TAC Case Priority Definitions ix

Overview 1-1

- The Cisco Service Control Concept 1-2
 - Service Control Capabilities 1-2
- The SCE Platform 1-3

The Service Control Solution 2-1

- System Components 2-1
- Subscribers and Subscriber Modes 2-3
 - Subscriber-less mode 2-3
 - Anonymous subscriber mode 2-4
 - Static Subscriber Mode 2-4
 - Subscriber-aware mode – Dynamic Subscribers 2-4
 - Subscriber Modes – Summary 2-5
- Service Configuration 2-5
 - SCAS BB Console 2-6
 - Service Configuration Utility 2-6
 - SCAS BB Service Configuration APIs 2-7

System Architecture for Developers 3-1

Integration Factors and Motivation 3-1

Essential Components 3-2

 SCE Platform 3-3

 smartSUB Manager 3-3

 Collection Manager 3-4

SCAS BB Licenses 3-4

Integration Points 3-5

 Service Configuration API 3-6

 SCAS Reporter Command Line Interface 3-6

 Flat Files 3-6

Flow of Information 3-7

Service Configuration Entities 4-1

Logical Entities 4-2

Service Configurations 4-2

Services and Service Elements 4-3

Protocols 4-4

Dynamic Signatures 4-5

Initiating Side 4-5

Lists 4-6

Rules 4-6

Packages 4-7

Global Controllers 4-8

Subscriber BW Controllers 4-8

Subscriber Quota Buckets 4-9

Service Configuration API 5-1

Overview of Service Configuration API 5-1

SCAS API Base Classes 5-2

SCAS Client/Server Connectivity 5-2

 Including the SCAS Libraries 5-3

 Connecting to the SCE Platform 5-3

Managing Service Configurations with SCAS Service Configuration API 5-4

 Retrieving and Applying Service Configurations 5-5

- Importing, Exporting, and Creating Service Configurations 5-6
- Lists 5-7
- Protocols 5-8
- Service and Service Elements 5-11
- Packages 5-13
- Example - Adding a Service and Applying a Service Configuration 5-22

Subscriber Integration 6-1

- Subscriber Modes 6-1
- Subscriber-less Mode 6-2
- Anonymous Subscriber Mode 6-3
- Static Subscriber-Aware Mode 6-3
- Dynamic Subscriber-Aware Mode and smartSUB Manager (SM) 6-4
 - SM General Functions 6-4
 - Pull-mode 6-4
 - Subscriber State 6-5
 - Subscriber-Integration: PRPC Protocol 6-6
 - Subscriber-Integration: CNR (DHCP) Plug-in 6-6

Quota Provisioning API 7-1

- External Quota Provisioning 7-2
 - Service Configuration for External Quota Provisioning 7-2
 - Quota Bucket States 7-3
- Quota Provisioning Life Cycle 7-3
- Limitations 7-4
- Installing the External Quota Provisioning APIs 7-4
- QP API (Java) Methods 7-5
 - addSubscriberQuota 7-5
 - addSubscriberQuota 7-6
 - getSubscriberQuota 7-6
 - setSubscriberQuota 7-7
- QP API (Java) Code Examples 7-8
- QP API (C) Methods 7-9
 - addQuota 7-10
 - getRemainingQuota 7-10

setQuota 7-10

QP API (C) Code Examples 7-11

Error Codes and Exception Handling 7-14

Quota Provisioning API Error Codes 7-14

Managing Exceptions in the Java API 7-14

Managing Error Codes in the C/C++ API 7-15

Reporter Command Line interface 8-1

Overview of Reporter Command Line Interface 8-1

Syntax and Usage 8-1

Command-Line Usage 8-2

Command-Line Syntax 8-2

Command-Line Options 8-2

Command-File Usage 8-3

Command-File Syntax 8-3

Glossary of Terms 1

Index 1



Introduction

This guide explains how to use and write programs using the set of APIs of the SCAS BB application.

Audience

This guide is written for experienced Java programmers. Working knowledge of network protocols and topologies is assumed.

The SCAS BB developer can use the SCAS Service Configuration API classes, the SCAS Reporter Command-line Interface, and the maintenance scripts, database scripts, and other SCAS BB tools for creating value-added applications, to integrate with existing hardware and software on the provider's network, and to provide value-added software solutions and services.

Purpose

The *Service Control Application Suite for Broadband API Guide* explains how to use and write programs using Service Control technology and solution, which enables Service Providers (SPs) to monitor and provision value-added services for their network subscribers.

The Service Control solution, based on the Service Control Engine Platform (SCE platform), offers a flexible and simple-to-use environment to service providers for performing detailed network analysis, traffic shaping and prioritization, and transaction-level control, all at wire-speed rates and in a fully programmable and extendable framework. These capabilities can be used by the SP to understand how the network is used, develop advanced usage-based billing schemes, control network abuse, and offer tiered-access services.

The SCAS BB API technology enables SPs to easily develop and deploy networked Java applications that hide the underlying layers of complexity and provide network users with differentiated services. SPs can enhance and augment the Service Control solution, by integrating it with new or existing OSS and back-office systems. The APIs can also be used to develop and deploy a rich set of functionality that hide the underlying layers of complexity and provide network users with differentiated services.

After working through the chapters in this guide you will be able to develop applications using the SCAS BB API. The SCAS BB APIs are implemented in Java, offering a simple, platform-independent, and interoperable programming environment.

To use the SCAS BB API software product, you (or the system administrator) must have previously performed certain administrative tasks, such as installing the Service Control system hardware and software on the provider network. Note that the SP network administrator is responsible for certain aspects of the work; for example, changing IP addresses or adding additional computers and Service Control hardware devices to the network.



Note

To develop applications using the SCAS BB Service Configuration API, you must install and use **JDK** version 1.3.

Document Content

This manual contains the following chapters:

Chapter 1, *Overview*, provides a general overview of the Service Control solution and the Service Control concept.

Chapter 2, *The Service Control Solution*, describes certain aspects of Service Control technology and solution: system components, subscribers and subscriber modes, and service configuration.

Chapter 3 *System Architecture for Developers*, provides an overview of the Service Control Application Suite for Broadband architecture for programmers.

Chapter 4, *Service Configuration Entities*, provides definitions and outlines the principles for developing a Service Control Application Suite for Broadband application.

Chapter 5, *Service Configuration API*, explains how the SCAS BB Service Configuration API is used for automating the SCAS BB service configuration activities.

Chapter 6, *Subscriber Integration*, discusses subscriber integration in a Service Control Application Suite for Broadband application.

Chapter 7, *Quota Provisioning API*, describes the External Quota Provisioning (QP) API for Java and C.





Chapter 8, *Reporter Command Line Interface*, explains how the Reporter Command Line Interface may be used to generate reports.

Glossary

Index

Document Conventions

The following typographic conventions are used in this guide:

Typeface or Symbol	Meaning
<i>Italics</i>	References, new terms, field names, and placeholders.
Bold	Names of menus, options, and command buttons.
Courier	System output shown on the computer screen in the Telnet session.
Courier Bold	CLI code typed in by the user in examples.
<i>Courier Italic</i>	Required parameters for CLI code.
<i>[italic in brackets]</i>	Optional parameters for CLI code.
	Note.
	Notes contain important information.
	Warning.
	Warning means danger of bodily injury or of damage to equipment.

Related Publications

The *SCAS BB API Guide* should be used in conjunction with SCE Platform user guides (*SCE1000 User Guide*; *SCE3000 User Guides*) and the other Management Suite user guides (*Collection Manager User Guide*, *SCAS BB User Guide*, *smartSUB User Guide*, and *SM API Guides*).

Obtaining Technical Assistance

For all customers, partners, resellers, and distributors who hold valid Cisco service contracts, the Cisco Technical Assistance Center (TAC) provides 24-hour, award-winning technical support services, online and over the phone. Cisco.com features the Cisco TAC website as an online starting point for technical assistance.

Cisco TAC Website

The Cisco TAC website (<http://www.cisco.com/tac> (<http://www.cisco.com/tac>)) provides online documents and tools for troubleshooting and resolving technical issues with Cisco products and technologies. The Cisco TAC website is available 24 hours a day, 365 days a year.

Accessing all the tools on the Cisco TAC website requires a Cisco.com user ID and password. If you have a valid service contract but do not have a login ID or password, register at this URL:

<http://tools.cisco.com/RPF/register/register.do> (<http://tools.cisco.com/RPF/register/register.do>)

Opening a TAC Case

The online TAC Case Open Tool (<http://www.cisco.com/tac/caseopen> (<http://www.cisco.com/tac/caseopen>)) is the fastest way to open P3 and P4 cases. (Your network is minimally impaired or you require product information). After you describe your situation, the TAC Case Open Tool automatically recommends resources for an immediate solution.

If your issue is not resolved using these recommendations, your case will be assigned to a Cisco TAC engineer.

For P1 or P2 cases (your production network is down or severely degraded) or if you do not have Internet access, contact Cisco TAC by telephone. Cisco TAC engineers are assigned immediately to P1 and P2 cases to help keep your business operations running smoothly.

To open a case by telephone, use one of the following numbers:

Asia-Pacific: +61 2 8446 7411 (Australia: 1 800 805 227)

EMEA: +32 2 704 55 55

USA: 1 800 553-2447

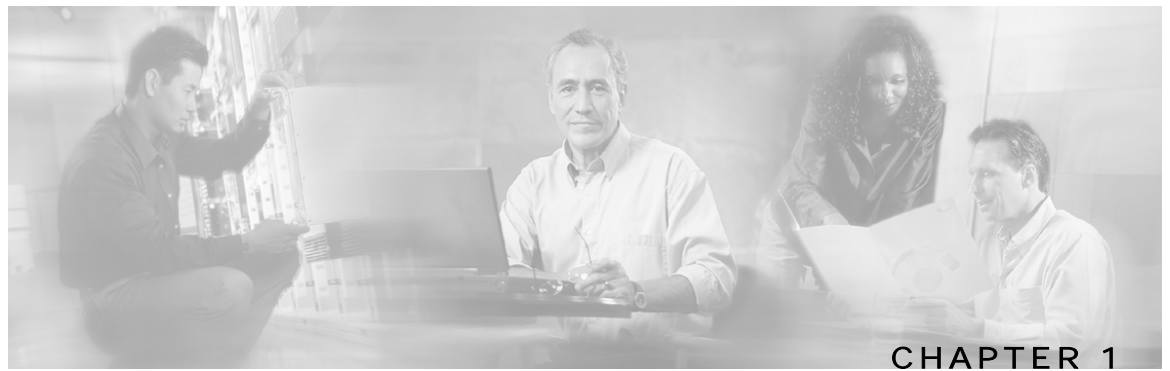
For a complete listing of Cisco TAC contacts, go to this URL:

<http://www.cisco.com/warp/public/687/Directory/DirTAC.shtml>
(<http://www.cisco.com/warp/public/687/Directory/DirTAC.shtml>)

TAC Case Priority Definitions

To ensure that all cases are reported in a standard format, Cisco has established case priority definitions.

- **Priority 1 (P1)**—Your network is “down” or there is a critical impact to your business operations. You and Cisco will commit all necessary resources around the clock to resolve the situation.
- **Priority 2 (P2)**—Operation of an existing network is severely degraded, or significant aspects of your business operation are negatively affected by inadequate performance of Cisco products. You and Cisco will commit full-time resources during normal business hours to resolve the situation.
- **Priority 3 (P3)**—Operational performance of your network is impaired, but most business operations remain functional. You and Cisco will commit resources during normal business hours to restore service to satisfactory levels.
- **Priority 4 (P4)**—You require information or assistance with Cisco product capabilities, installation, or configuration. There is little or no effect on your business operations.



Overview

This chapter provides a general overview of the Service Control concept. It describes the functional topology of the SCE platform, and gives a general understanding of how the platform works and how information flows through the system.

The SCE platform is designed to support observation, analysis, and control of Internet/IP traffic. The SCE platform provides the raw data for a rich range of monitoring, analysis and traffic control functions and tools to contribute to the Service Provider's (SP) profitability and for reducing its OPEX (operational expenses). It supports the Service Control applications that enable the SP to achieve a high level of differentiation, quality of service, market segmentation abilities, and customer satisfaction.

This chapter contains the following sections:

- [The Cisco Service Control Concept](#) 1-2
- [The SCE Platform](#) 1-3

The Cisco Service Control Concept

The Cisco Service Control concept is delivered through a combination of purpose-built hardware and specific software solutions that address various Service Control challenges faced by service providers. The SCE Platform is designed to support observation, analysis, and control of Internet/IP traffic.

Service Control enables service providers to create profitable new revenue streams while capitalizing on their existing infrastructure. With the power of Service Control, service providers have the ability to analyze, charge for, and control IP network traffic at multi-Gigabit wire line speeds. The Cisco Service Control solution also gives service providers the tools they need to identify and target high-margin, content-based services.

As the downturn in the telecommunications industry has shown, IP service provider business models need to be reworked in order to make them profitable. Having spent billions of dollars to build ever larger data links, providers have incurred massive debts and rising costs. During the same time, access and bandwidth became a commodity where prices continually fell and profits disappeared. Service providers now realize that they must offer value-added services to derive more revenue from the traffic and services running on their networks. However, capturing real profits from IP services requires more than simply running those services over data links; it requires detailed monitoring and precision, real-time control and awareness of services as they are delivered. Cisco provides Service Control solutions that allow the service provider to bridge this gap.

Service Control Capabilities

At the core of the Cisco Service Control Platform stands the purpose-built network hardware device: the Service Control Engine (SCE). Implementing a complete Service Control solution requires that the Service Control Engine provide certain functionalities and capabilities. The following are the core capabilities of the Cisco Service Control Engine, which support a wide range of applications for delivering Service Control solutions:

- Subscriber and application awareness: Application-level drilling into IP traffic for real-time understanding and controlling of usage and content at the granularity of a specific subscriber.
 - Subscriber awareness: The ability to map between IP flows and a specific subscriber for maintaining the state of each subscriber transmitting traffic through the platform, and enforcing the appropriate policy on this subscriber traffic.

Subscriber awareness is achieved using dedicated integrations with subscriber management repositories, such as a DHCP or a Radius server.

- Application awareness: The ability to understand and analyze traffic up to the application protocol layer (Layer 7).

For an application protocol that is implemented using bundled flows (such as FTP, which is implemented using Control and Data flows), the SCE Platform understands the bundling connection between the flows and treats them accordingly.

- Stateful, real time traffic control: The ability to perform advanced control functions, including granular BW metering and shaping, quota management and redirection, utilizing stateful real-time traffic transaction processing. This requires highly adaptive protocol and application level intelligence.

- **Programmability:** The ability to quickly add new protocols and easily adapt to new services and applications in the ever-changing service provider environment. Programmability is achieved using the SML language.
Programmability means that new services can be deployed quickly and provides an easy upgrade path for network, application, or service growth.
- **Robust and flexible back office integration:** The ability to integrate with existing 3rd party systems at the Service Provider, such as provisioning systems, subscriber repositories, billing systems, and OSS systems. The Service Control Engine provides a set of open and well-documented APIs that allows a quick and robust integration process.
- **Scalable High-Performance Service Engines:** The ability to execute all operations described above at wire speed.

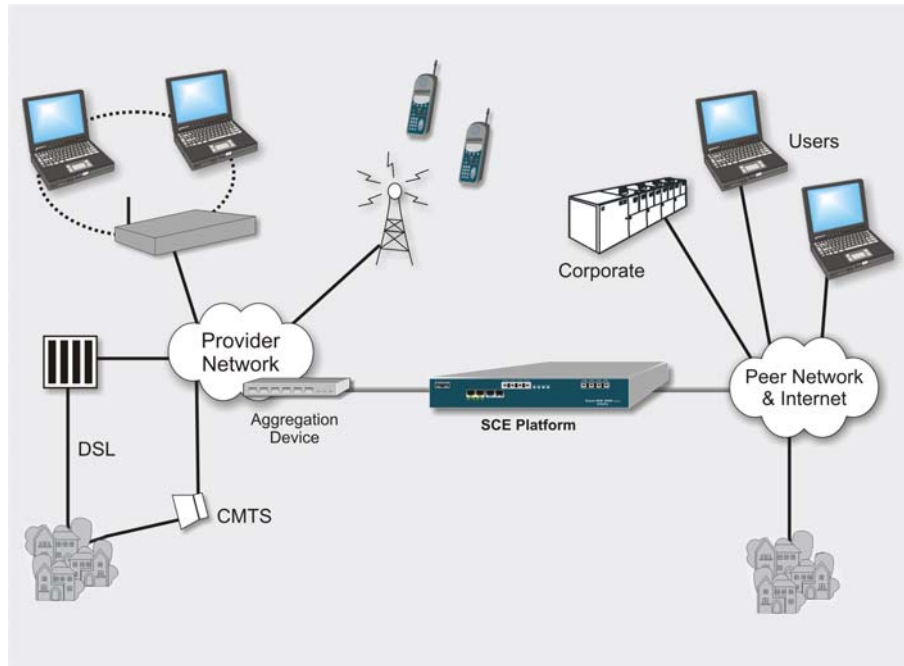
The SCE Platform

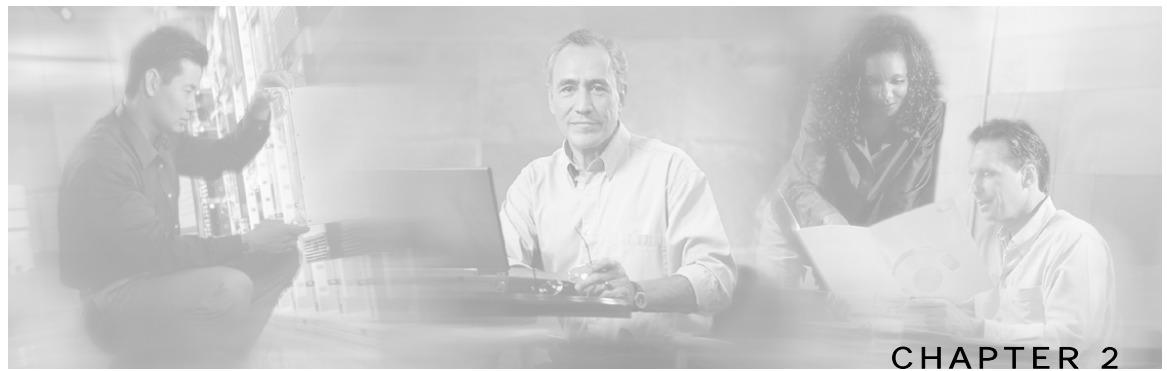
The Service Control Engine family of programmable network devices is capable of performing stateful flow inspection of IP traffic, and controlling that traffic based on configurable rules. The Service Control Engine is a purpose-built network device making use of ASIC components and RISC processors to go beyond packet counting and delve deeper into the contents of network traffic. Providing programmable, stateful inspection of bi-direction traffic flows and mapping these flows with user ownership, the Service Control Engine platforms provide a real-time classification of network usage. This information provides the basis of the Service Control Engine advanced traffic control and bandwidth shaping functionality. Where most bandwidth shaper functionality ends, the Service Control Engine provides more control and shaping options including:

- Layer 7-3 stateful wire-speed packet inspection and classification
- Robust support for over 600 protocol/applications including:
 - General: HTTP, HTTPS, FTP, TELNET, NNTP, SMTP, POP3, IMAP, WAP, and others
 - P2P: FastTrack-KazaA, Gnutella, WinMX, Winny, Hotline, eDonkey, DirectConnect, Piolet, and others
- Streaming & Multimedia: RTSP, SIP, HTTP-STREAMING, RTP/RTCP, and others
- Programmable system core for flexible reporting and bandwidth control
- Transparent network and BSS/OSS integration into existing networks
- Subscriber awareness for relating traffic and usage to specific customers

The following diagram demonstrates a deployment of an SCE Platform in the network.

Figure 1-1: SCE Platform in the Network





The Service Control Solution

SCAS BB (SCAS) is the Service Control solution that allows broadband service providers to gain visibility into and control over the distribution of network resources, and thereby to optimize traffic in accordance with their business strategies. It enables service providers to reduce network costs, improve network performance and customer experience, and create new service-offerings and packages.

This chapter contains the following sections:

- [System Components](#) 2-1
- [Subscribers and Subscriber Modes](#) 2-3
- [Service Configuration](#) 2-5

System Components

The Service Control Application Suite for Broadband solution consists of three main components:

- The SCE Platform: A flexible and powerful dedicated network usage monitor that is purpose-built to analyze and report on network transactions at the application level.

For complete information regarding the installation and operation of the SCE Platform, see the *SCE 1000 and SCE 2000 User Guides*.

- The smartSUB Manager (SM): A middleware software component used in cases where dynamic binding of subscriber information and service configurations is required. The SM manages subscriber information and provisions it in real time to multiple SCE Platforms. The SM can store subscriber service configurations information internally, and act as a state-full bridge between the AAA system (for example, RADIUS and DHCP) and the SCE Platforms

For complete information regarding the installation and operation of the smartSUB Manager, see the *smartSUB Manager User Guide*.

- The Collection Manager (CM): An implementation of a collection system, listening in on RDRs from one or more SCE Platforms. It collects usage information and statistics, stores them in a bundled database, and provides a set of insightful reports from this data. The DC also converts subscriber usage information and statistics into simple text-based files for further processing and collection by external systems.

For complete information regarding the installation and operation of the Collection Manager, see the *Collection Manager User Guide*.

Together, the SCE Platform, the Collection Manager, and the smartSUB Manager are designed to support detailed observation, analysis, reporting, and control of IP network traffic. Note that the Collection Manager and smartSUB Manager are optional components, and are not required in all deployments of the solution. Sites that employ third party collection and reporting applications and/or do not require dynamic subscriber-aware processing may not require these components.

The following figure illustrates the flow of information within the SCAS BB solution.

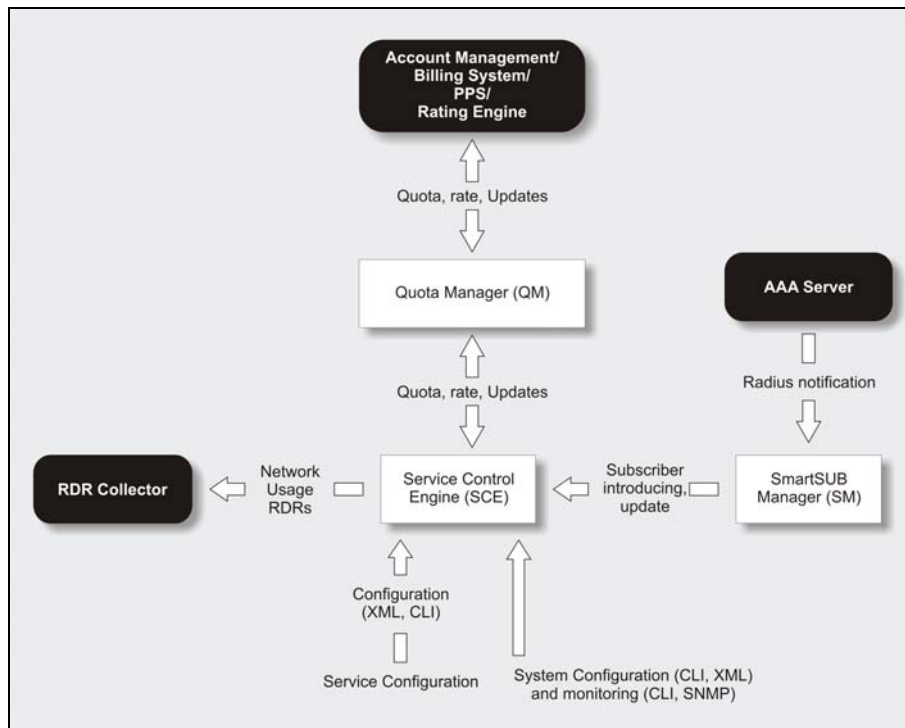
- Horizontal flow: Represents traffic between subscribers and IP network.

The SCE Platform monitors traffic flow.

- Vertical flow: Represents transmission of the Raw Data Records (RDRs) from the SCE Platform to the Collection Manager.

The *smartSUB Manager* may be added to the control flow to provide subscriber data. This enables the SCAS BB to conduct dynamic subscriber integrations.

Figure 2-1: Logical Components of the SCAS System



Subscribers and Subscriber Modes

One of the fundamental entities in the SCAS BB solution is a *subscriber*. A subscriber is the most-granular entity that the SCAS BB solution can individually monitor, account, and enforce a service configuration on. In the most granular instance of the SCAS BB system, a subscriber is an actual customer of the service-provider on whom an individual service configuration is implemented. However, it is also possible to use the SCAS BB solution to monitor and control traffic at a higher granularity, such as when monitoring controlling traffic by subnets or aggregation devices.

One of the most important decisions to be made when designing a SCAS BB solution is what will be defined as a subscriber in the system. This determines what subscriber-mode will be used, which in turn determines what (if any) integrations are required, as well as what actual service configuration to define. The following section describes the different subscriber-modes supported, what functions are supported for each mode, and what prerequisites and required components are needed.

SCAS BB supports the following subscriber modes:

- **Subscriber-less mode:** No subscribers are defined.
- **Anonymous subscriber mode:** IP addresses are controlled and monitored individually. The SCE Platform automatically identifies IP addresses as they are used and assigns them the default service configuration.
- **Subscriber-aware mode – Static Subscribers:** Incoming IP addresses are bound and grouped statically into subscribers, as configured by the system operator.
- **Subscriber-aware mode – Dynamic Subscribers:** subscriber information is dynamically bound to the IP address currently in use by the subscriber through an integration with the system that assigns IP addresses to subscribers (RADIUS, DHCP). Service configuration information either is administered to the Service Control solution directly, or is also provisioned dynamically through an integration.

Subscriber-less mode

Subscriber-less mode is the choice for sites where control and level analysis functions are required only at a global device resolution. It can be used, for example, to monitor the total amount of P2P traffic over the link.

Since subscriber-less mode requires no integration, the smartSUB Manager component is not required. Note that, since subscriber-less mode is not influenced by the number of subscribers or inbound IP addresses, the total amount of subscribers utilizing the monitored link is unlimited from the perspective of the SCE Platform.

Anonymous subscriber mode

Anonymous subscriber mode provides the means to analyze and control network traffic at a subscriber-inbound IP address granularity. Use this mode when no subscriber-differentiated control or subscriber-level quota tracking is required, when analysis on an IP level is sufficient, or when offline IP-address/subscriber binding can be performed. For example, it is possible to identify which subscribers generate the most P2P traffic by identifying the top IP addresses and correlating them to individual subscribers manually/offline via RADIUS/DHCP logs. The total bandwidth of P2P traffic allowed for each subscriber can be limited as well.

Since anonymous mode requires no integration or static configuration of the IP addresses used, the smartSUB Manager component is not required. Rather, ranges of IP addresses are configured directly on the SCE Platform, for which the system will dynamically create ‘anonymous’ subscribers, using the IP address as the subscriber-name. Note that the total number of concurrently active anonymous subscribers supported by the SCE Platform is the same as the total number of concurrently active subscribers.

Static Subscriber Mode

Static subscriber mode binds together incoming IP addresses into groups, so that traffic from/to a defined subscriber can be controlled as a group. For example, with this mode, all traffic from/to a particular network subnet (used by multiple subscribers concurrently) can be defined as a ‘virtual subscriber’ and controlled/viewed as a group.

Static subscriber mode supports cases in which the entity controlled by the Service Control solution uses a constant IP address or address-range that does not change dynamically, such as:

- Environments where the subscriber IP address(es) do not change dynamically via DHCP, RADIUS, etc.
- Deployments in which a group of subscribers using a common pool of IP addresses, such as all those served by a particular CMTS, BRAS, etc., are to be managed together to provide a shared bandwidth to the entire group.

The system supports the definition of static subscribers directly on a SCE Platform, and does not require external management software (smartSUB Manager). This is achieved by using the SCE Platform CLI to define the list of subscribers, their IP addresses and associated package.

Subscriber-aware mode – Dynamic Subscribers

In dynamic subscriber mode, the SCE is populated by subscriber information (OSS ID and service configuration) that is dynamically bound to the (IP) address currently in use by the subscribers. This provides differentiated and dynamic control per subscriber and subscriber-level analysis, regardless of IP address in use. This mode is used to control/analyze traffic on a subscriber level and monitor subscriber-usage, regardless of IP addresses. It also enables assigning and enforcing different service configuration or packages for different subscribers.

In this mode, the smartSUB Manager (SM) needs to be used to perform device provisioning with subscriber information. The SM is a server application that maintains the above association, and provisions it to SCE Platforms in real time.

Subscriber Modes – Summary

The following table summarizes the different subscriber modes supported by the system.

Table 2-1 Subscriber-Mode Summary Table

Mode	Features Supported	Main Advantages	When to Use
Subscriber-less	Global (device-level) analysis and control	No subscriber configuration required	For global control solution or subscriber level analysis. Examples: <ul style="list-style-type: none"> Controlling P2P uploads at peering points Limiting total amount of P2P to a fixed percentage
Anonymous subscriber	Global analysis and control Individual IP address level analysis and control	No subscriber configuration required Need only to define the subscriber IP address ranges used Provides subscriber-level control without integration	For IP level analysis or control that is not differentiated per subscriber, and where offline IP-address/subscriber binding is sufficient. Examples: <ul style="list-style-type: none"> Limiting per subscriber P2P to 64 Kbps (kilobits per second) Identifying top subscribers by identifying top IP addresses and correlating manually/offline with RADIUS/DHCP logs
Static Subscriber	Global Analysis and Control Control based on individual IP addresses/groups as configured statically to the SCE Platform	On-time static subscriber configuration, with no integration requirements Manages subscriber traffic in logical groups	For controlling traffic of groups of subscribers. Example: <ul style="list-style-type: none"> Assigning a 5 Mbps (megabits per second) limit of P2P traffic for each group of subscribers using a single CMTS device
Dynamic Subscriber	Full system functionality	Differentiated and dynamic control per subscriber Subscriber-level analysis, regardless of IP address in use	For <ul style="list-style-type: none"> Controlling/analyzing traffic on a subscriber level. Monitoring subscriber-usage, regardless of IP addresses Assigning different service configuration or packages to different subscribers, and changing packages dynamically

Service Configuration

Service configuration defines the way an SCE Platform analyses and controls traffic. In very general terms, Service Configuration defines the following:

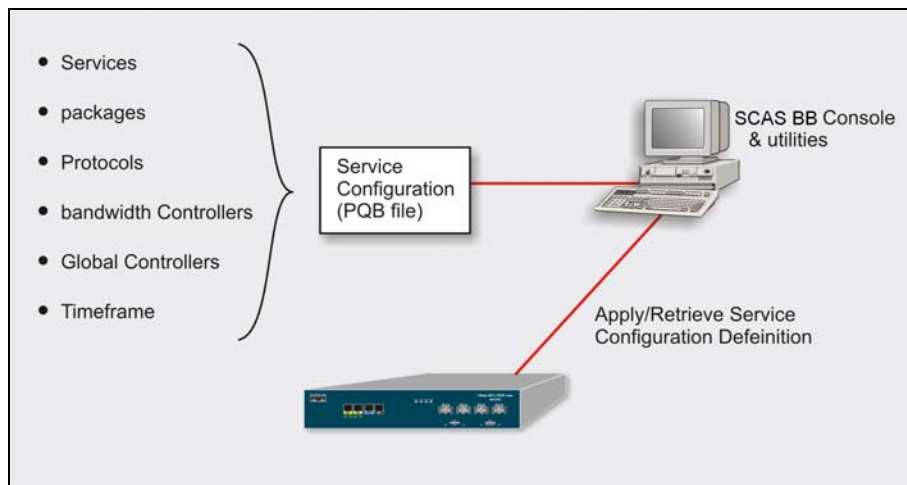
- protocol and service classification

- packages and service configuration
- bandwidth controllers
- global controllers

Service configuration is accomplished using one of the following:

- SCAS BB Console
- Service Configuration Utility
- SCAS BB APIs

Figure 2-2: Service Configuration



SCAS BB Console

The SCAS BB Console is the SCAS BB GUI, used to create, modify, and apply the service configuration. The SCAS BB Console lets you define services, packages, protocols, bandwidth control and other entities in the configuration. The SCAS BB Console creates a service configuration file (*.pqb*), which can then be saved and/or applied to the SCE Platform(s).

You can also access the smartSUB Manager from the SCAS BB Console to manage subscribers. In addition, you can access the Reporter feature of the Collector Manager to create and output reports.

The SCAS BB Console is fully documented in the *SCAS BB User Guide*.

Service Configuration Utility

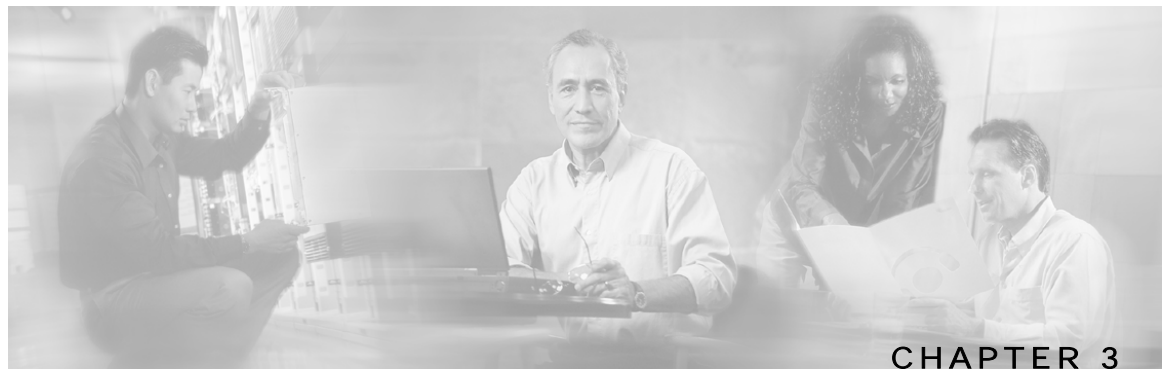
The Service Configuration Utility is a simple command line tool that can be used to apply **.pqb** configuration files onto SCE Platforms, or retrieve the current configuration from an SCE Platform and save as a **pqb** file. The tool can be installed and executed on either Windows or Solaris environments, and configures SCE Platforms with the service configuration in a **.pqb** file.

The Service Configuration Utility (`servconf`) helps with the automation of applying/retrieving service configurations, by providing a command-line interface for performing these operations.

The Service Configuration Utility is fully documented in the **SCAS BB User Guide**.

SCAS BB Service Configuration APIs

The SCAS BB Service Configuration API is a set of Java classes used for programming and managing Service Configurations, and for applying these Service Configurations to the SCE Platforms. In addition, applications using the SCAS API can be integrated with third-party systems, allowing service providers to automate and simplify management and operational tasks.



System Architecture for Developers

This chapter:

- Outlines the SCAS BB system architecture for SCAS developers.
- Describes the main components of the SCAS BB hardware and software.
- Introduces SCAS API programming definitions and concepts.
- Describes the different levels of licensing for SCAS BB.
- Describes the connections between the SCAS BB system's various logical entities.
- Describes information flow between the SCAS BB hardware and software components.

The SCAS BB Java API classes provide a rich programming environment in which to develop value-added Internet services and network traffic monitoring applications for SPs.

This chapter contains the following sections:

- [Integration Factors and Motivation](#) 3-1
- [Essential Components](#) 3-2
- [SCAS BB Licenses](#) 3-4
- [Integration Points](#) 3-5
- [Flow of Information](#) 3-7

Integration Factors and Motivation

Using the SCAS API, service providers will be able to:

- **Make SCAS BB part of a larger SP solution.** As part of a larger SP solution, providers may not be able to allow dual operation and maintenance of similar systems. The need to integrate more compactly, natively, and transparently with existing systems such as AAA and billing should be satisfied.
- **Provide added revenue sources for business growth.** By utilizing the API's programming capabilities, sophisticated and advanced customized applications and solutions can be delivered. SPs can develop applications for in-house systems, or software applications and services for external consumption. For example, if an SP wants to develop a web interface through which customers can select their preferred service package, the API provides the SP with the software platform and programming tools with which to create such an application.

- **Simplify operational tasks.**
- **Keep legacy systems GUI look and feel.**

In addition, when the SCAS BB system is integrated into the provider's network, it is not necessary to interfere with or to displace existing hardware and infrastructure. The SCAS BB system comes with its hardware and software pre-installed, and is ready to be enabled and deployed on the provider's network with minimum configuration.

Essential Components

Running on a Management LAN, the SCAS BB system components are situated between the network subscriber and the Internet.

The SCAS BB system is responsible for performing three major functions on the network flow:

- **Analyzing Network**- Analyzes network traffic and determines the type of transaction passing through the network.
- **Analyzing Subscribers** - Analyzes the characteristics of the subscriber or organization whose transaction is passing through the network.
- **Enforcing Service Configuration** - Enforces the Service Configuration on the network based upon its traffic analysis. It enables provisioning of network services, differentiating between subscribers.

The SCAS BB system's core components include:

- SCE Platforms (SCE 1000 and SCE2 000)
- smartSUB Manager (SM)
- (Optional) One or more Collection Manager systems (CMs)
- SCAS Clients (SCAS BB Console, SCAS smartSUB Manager, SCAS Reporter)

SCE Platform

The SCE Platform is a purpose-built service component and active enforcing system designed for enhancing service providers and backbone carrier networks. By identifying, classifying, and manipulating complex traffic flows at wire-speed, the SCE Platform transforms simple transport networks into differentiated service delivery infrastructures for a wide variety of value-added IP applications, such as video streaming, VoIP, tiered services, and bilateral application-level SLAs.

The SCE Platform seamlessly interfaces with existing network elements—including routers, switches, aggregators, subscriber management devices, and operational support systems—using industry standard interfaces and communications protocols.

The need to guarantee that packets passing through the network are processed at the rate they arrive makes it necessary to provide a custom-made hardware solution.

The SCE Platform comes in three models: SCE 1000, SCE 2000 4xGBE, and SCE 2000 4/8xFE. There may be one or more of the SCE Platforms in the provider network. Within the SCE Platforms, network transactions are analyzed and mapped to services that enforce the provider's policies.

In addition, the SCE Platform implements the business logic of the system solution and performs the transaction analysis in real time. When so instructed, the SCE Platform creates a Raw Data Record (RDR) to be sent for storage to the system's data repository, the Collection Manager (CM); or carries out some other operation such as bandwidth and volume control.

smartSUB Manager

The smartSUB Manager (SM) is a software solution that addresses three issues in the handling of subscribers by the SCE Platform when operating in subscriber-aware mode:

- **Capacity:** The SCE Platform(s) may need to process (over time) more subscribers than it can statically hold.

The smartSUB Manager is a repository for subscriber names and information.

- **Mapping:** The SCE Platform encounters flows with network IDs (IP addresses), and it requires mapping between those network IDs and the subscriber IDs.

The smartSUB Manager database contains the network IDs that map to the subscribers.

- **Location:** The system may not be able to predict which SCE Platform will handle which subscriber traffic.

The smartSUB Manager allows the system to be configured to introduce subscribers in pull mode, and detect which SCE Platform handles which subscriber at runtime.

The SM database can function in one of two ways:

- As the only source for subscriber information: when the SM works in standalone mode.
- As a subscriber information cache: when the SM serves as a bridge between a group of SCE devices and the customer AAA and OSS systems.

Collection Manager

The Collection Manager (CM) is the software component that is responsible for receiving usage records for processing, such as Raw Data Records (RDRs) from the SCE Platform. The Collection Manager processes the RDRs and sends them to be stored in storage devices, such as in databases or CSV (Comma Separated Value) flat files. A typical system integration may include periodical visits to the Data Collector and processing of the stored usage files.

The Collection Manager may either reside on the server platform together with the Sybase database to which it sends records, or they can be separated and reside on different platforms. For sample Sybase database scripts, refer to the *Collection Manager User Guide*.

SCAS BB Licenses

SCAS BB offers three different levels of licensing to suit the needs of different sites:

- **SCAS BB View:** This is the basic form of SCAS BB. It has the following capabilities:
 - Monitoring and Reporting
 - No control capabilities
- **SCAS BB Capacity Control:** This license adds traffic control functionality. It has the following capabilities:
 - Monitoring and Reporting
 - Capacity Control, for example by assigning traffic of different applications to different Global Controllers
 - One package only (default package) - no differentiation between subscribers
 - Requires a key
- **SCAS BB Tiered Control:** This license permits the differential control of traffic flows based on package. It has the following capabilities:
 - Monitoring and Reporting
 - Capacity Control
 - Multiple packages - allows differentiation between subscribers; for example, by allowing greater BW or greater daily volume quota to subscribers of a certain package
 - Requires a key

To register for a higher-level license:

Step 1 From the **Help** menu, click **License Manager**.

The *License Manager* dialog appears.



Step 2 Check the **Enter new license key** check box.

The **Customer ID** and **Key** fields become available.

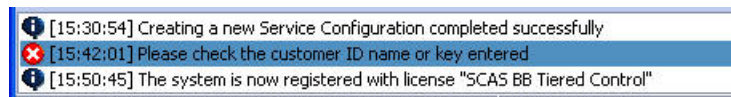


Step 3 Type your Customer ID and Key in the appropriate fields.

Step 4 Click **OK**.

The new license is displayed in the message band and status bar.

Figure 3-1: Displaying the New License



Integration Points

Example of integration scenarios with the SCAS BB Solution are:

- **SM API** - Applications integrated with the SM API may provision subscribers, assign them to packages, and change a subscriber's package dynamically.
- **Quota Provisioning API** - Applications integrated with the QP API may provision quota to subscribers, as well as query on the subscriber's remaining quota.

- **Service Configuration API** - Applications integrated with the Service Configuration API may perform list updates, Service Configuration editing, and deployment scripts.
- **Reporter Command Line Interface** - Applications integrated with the SCAS Reporter Command Line Interface can visit the SQL database periodically to execute batch queries and produce image files to be used in third-party usage billing systems or statistical analysis packages.
- **Flat Files** - CSV (Comma Separated Value) flat files may be used for integrating their data with third-party billing systems or statistical analysis packages.

Service Configuration API

The *Service Configuration API* is a set of Java classes used to program and manage Service Configuration, and to configure the functionality of the SCE Platforms responsible for enforcing the Service Configuration business rules.

Using the Service Configuration API, GUI applications can be developed for enhancing the capabilities provided by the Service Configuration Management Module. Through server applications created with the Service Configuration API, SPs can roll out network solutions and deliver customized applications, providing value-added services that satisfy customer needs.

Besides the aforementioned advantages, applications using the Service Configuration API can be integrated with third-party systems.

SCAS Reporter Command Line Interface

The SCAS Reporter Command Line Interface is the SCAS BB system software counterpart of the Service Control Reporter GUI client. It is responsible for executing SQL queries to the Collection Manager and for filtering the output to generate network transaction usage reports in tabular or chart format.

The SCAS Reporter Command Line Interface can be executed as a standalone program from the command line, or spawned from various applications. In addition, the SCAS Reporter Command Line Interface provides all the capabilities of the Service Control Reporter GUI client, and can readily be integrated and used from various systems.

Flat Files

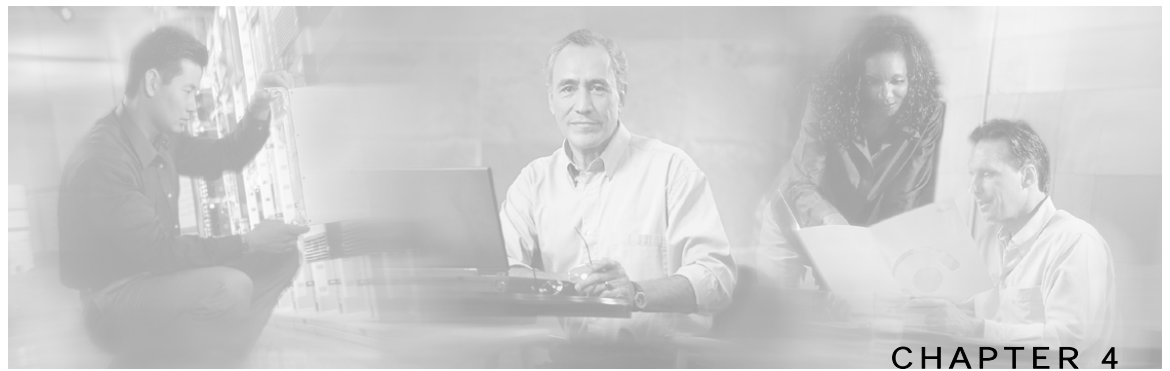
Applications using Comma Separated Value (CSV) flat files may apply their own formats and templates for reviewing and using the data. For example, applications such as Excel are capable of importing CSV files into their spreadsheets.

Flow of Information

This section describes the information flow between the various hardware and software components of SCAS BB. The SCAS BB system consists of one or more SCE Platforms, and software components such as the SM, Collection Manager, and other SCAS modules, which provide each other with services. Understanding the relationships between the components and the direction in which their data travels in the system will provide the developer with a snapshot of the system.

The important relationships between the SCAS BB system components can be summarized as follows:

- **Retrieving a Service Configuration** - Gets a copy of the active Service Configuration in the SCE device.
Service Configuration retrieval is generally done for the purpose of performing Service Configuration editing: a static process involving modifications to the business rules of the provider.
- **Applying a Service Configuration** - The application executing the Service Configuration API *apply* function call provides Service Configuration modification information that replace the active Service Configuration in the SCE device. Once the Service Configuration transfer has been carried out, the Service Configuration goes live and the new business rules of the provider become active and are enforced on the network traffic.
- **SCE Platforms Trigger RDRs** - An SCE Platform creates an RDR as a result of an action triggered when a system-defined condition on a subscriber's network transaction is met. The information contained in the RDR flows from the SCE Platform (where it was created) to the Collection Manager, where the Collection Manager Adapter adapts and prepares the RDR to be inserted in database tables or placed in CSV flat files, depending on how the Collection Manager is configured.
- **SCAS Reporter Execution** - When a report is executed using the SCAS Reporter, either interactively or through the Reporter Command Line Interface, an SQL query is executed through an ODBC driver to its Sybase database. The results of the SQL query are then translated into the format specified by the executing report, and can either be displayed or saved in a file.



Service Configuration Entities

This chapter:

- Provides a technical overview of the system.
- Introduces SCAS API programming definitions and concepts.

This chapter provides definitions for the main elements (Service Configuration Entities) of the SCAS BB system, which are the building blocks of the SCAS BB Java application. An understanding of the function of these elements is therefore a prerequisite for developing applications that can use the full potential of the SCAS API, with its programmable objects of subscribers, service configurations, packages, services, and rules.

This chapter contains the following sections:

- [Logical Entities](#) 4-2
- [Service Configurations](#) 4-2
- [Services and Service Elements](#) 4-3
- [Protocols](#) 4-4
- [Dynamic Signatures](#) 4-5
- [Initiating Side](#) 4-5
- [Lists](#) 4-6
- [Rules](#) 4-6
- [Packages](#) 4-7
- [Global Controllers](#) 4-8
- [Subscriber BW Controllers](#) 4-8
- [Subscriber Quota Buckets](#) 4-9

Logical Entities

The following sections describe the logical entities that make up the SCAS BB system at the programming granularity level. The SCAS Java API builds on these definitions to deliver robust client/server applications. The developed applications can run over the network, or be used as standalone applications, or be integrated into third-party systems delivered over the Internet.

These sections cover the following entities:

- *Service Configuration* ("[Service Configurations](#)" on page [4-2](#))
- *Services and Service Elements* (on page [4-3](#))
- *Protocols* (on page [4-4](#))
- *Dynamic Signatures* (on page [4-5](#))
- *Initiating Side* (on page [4-5](#))
- *Lists* (on page [4-6](#))
- *Rules* (on page [4-6](#))
- *Packages* (on page [4-7](#))
- *Global Controllers* (on page [4-8](#))
- *Subscriber BW Controllers* (on page [4-8](#))
- *Subscriber Quota Buckets* (on page [4-9](#))

Note that these entities are closely related to one another, therefore occasionally an entity is referred to in a section that comes before the section in which the entity is fully described.

Service Configurations

A service configuration implements and enforces the business strategy and vision of the provider, by offering the subscriber a variety of packages to choose from in order to meet the subscriber's personal or business needs. Packages can be created using the Service Configuration Editor, or be custom made to the requirements of the network provider using the Service Configuration API. The system places a limit of 64 packages and 31 services per service configuration.

Before a service configuration can be used, the service configuration needs to be propagated to the appropriate SCE Platforms for service configuration enforcement and activation. The active service configuration residing on the SCE Platforms enforces the service configuration by analyzing the network traffic passing through them.

A service configuration consists of:

- **Services** - A service configuration contains *services*, such as Web Browsing, Video Streaming, and Video Conferencing. Each service consists of transaction mappings that define how network activity is mapped to that service.
- **Packages** - A service configuration contains *packages*, each package consisting of a set of rules—such as bandwidth rate limit, volume limits, and admission control—defined for different *services*. Each set of rules comprising a package should be tailored to be a complete solution targeted for a different subscriber audience.

Services and Service Elements

From a provider's perspective, a service is a network product sold to a customer. From a programming perspective, a service consists of one or more service elements, each element enabling a decision regarding the service associated with a network traffic transaction type.

These transaction mappings are used by the SCE Platform for mapping each network connection passing through it to a service. Afterward, rules can be applied to the different services for implementing the service configuration. The classification rules can contain L3/4 parameters (such as port numbers and IP addresses), as well as L7 parameters (such as host name and user agent for HTTP connections).

The following table contains examples of services and their network parameters.

Table 4-1 Examples of Services and Service Parameters

Service Name	Protocol	Initiating Side	Address List
Web Browsing	HTTP HTTPS FTP	Subscriber-Initiated	None
Web Hosting	HTTP HTTPS	Network-Initiated	None
Local Streaming	RTSP MMS	Subscriber-Initiated	215.53.64.43 213.53.64.53

More than 600 protocols are supported by SCAS BB. Arranging them by purpose and providing them with appropriate names—for example, Web Browsing, Web Hosting, Video Conferencing, Video Streaming, and Local Content—which can later be used as part of the subscriber's purchased package, is a useful way for marketing services.

Following are examples of services:

- **Web Browsing** - Used for enabling and restricting Internet web browsing to outgoing HTTP protocol transactions.
- **Web Hosting** - Used for providing Internet web hosting to incoming HTTP transactions.
- **Video Conferencing** - Used for providing video conferencing to RTSP transactions.
- **Gaming Service** - Provides a particular network service, such as a gaming application, access to unrestricted bandwidth.
- **Local Content Gaming** - Gives subscribers an incentive to use a gaming application—such as Doom hosted on local servers within the provider's network, as opposed to Doom hosted on servers outside the provider's local network—by providing them with favorable gaming service usage rates.
- **Stock Quote Service** - Charges a fee for a popular stock quote service used by business investors.

Using the SCAS BB object-oriented technology, you can define program elements, such as service objects, with equivalent names, and can seamlessly integrate their functionality into networked system-level applications. Other benefits of working with programmable services include having greater control over how the network services are delivered.

An example of a service consisting of more than one service element might be a Gaming Service, with one service element defining a network classification entity for the program Doom on port 666, and a second service element classification entity for the program Quake on port 333. Because of the heavy bandwidth consumption of gaming applications, the appropriate service rule defined on the particular Gaming Service might be to measure the bandwidth volume the subscriber uses while connected to the service and bill accordingly.

Since rules are defined on the service, and not on its composite service elements, an important design consideration when planning your system is to place service elements with common rule-based functionality and characteristics together.

In practice, a system service consists of an array of service elements. Following are the main components of service elements:

- **Protocol** - The name of protocol to be associated with the service element.
- **Initiating Side** - Whether there should be a restriction on the direction in which the transaction is traveling. The possible directions are Network-initiated, Subscriber-initiated, or both of them.
- **(Optional) List** - Whether the transaction should be associated with the IP addresses or host names specified in the list.
- Each of the components of a service element mentioned above is described in the following sections.

Protocols

The protocol of the transaction is determined by its port number and transport type. Based on these parameters, further analysis (layer 7) is performed on the network transaction. For example, if the port number is 80 and transport type is TCP and content matches the protocol's signature, the system checks its tables and maps the transaction to the HTTP protocol. Or, if the port number is 1755 and transport type is UDP and content matches the protocol's signature, the system maps the transaction as RTSP, a protocol used for video streaming.

In future versions of SCAS BB, it is likely that existing network transaction for servicing mapping schemes will be extended. These additional mapping schemes could be applied, for example, to match the subject line of an e-mail containing the text "I love you" and forward the transaction to be processed by a virus-scanning application before it is delivered to the subscriber. This would trap a computer virus before it is capable of reaching the computer system of the provider or subscriber and causing damage. Alternately, these additional mapping schemes could be used to help prevent viruses from reaching the Internet, since the SCAS BB system is capable of analyzing both incoming and outgoing traffic.

The following points summarize the main aspects of protocols in the SCAS BB system:

- A protocol, as defined in the system, is a combination of port number(s) and transport type.

- The service configuration contains a list of protocols, each with a protocol name, transport type, and port number(s) if they are UDP or TCP protocols or with an IP protocol number. The protocol of the network transaction is identified according to these parameters.
- When the port number of a TCP or UDP transaction is not defined in any of the service configuration's protocols, the system identifies the transaction's protocol as a "generic TCP" or "generic UDP" protocol.
- When the IP protocol number is not defined in any of the service configuration's protocols, the system identifies the transaction's protocol as a "generic IP" protocol.

Dynamic Signatures

Dynamic signatures are a mechanism through which classification for new protocols can be added to a configuration. This is useful for cases where a new protocol is released and a customer would like to be able to classify its traffic (for example, a new P2P protocol in a P2P-Control solution). Dynamic signatures are provided in special Dynamic Signature Script (DSS) files, which can be added to a PQB file using the SCAS BB Console or API. After a DSS file is loaded into a PQB, the new protocols it supports are available in the protocol list, can be added to Services as appropriate, and are used when viewing reports. DSS files are periodically released by Cisco or its partners in accordance with customer requirements and market needs.

Initiating Side

The initiating side of a network transaction may be either Network-Initiated or Subscriber-Initiated. Subscriber-Initiated transactions are those initiated by the subscriber toward the network, while Network-Initiated transactions are those initiated from the network toward the subscriber.

The system can specify that the network transaction defined by the service element should be restricted to: (a) Network-Initiated direction, (b) Subscriber-Initiated direction, or (c) the transaction should be unrestricted in either direction.

Examining a few protocols will best explain how the "direction" process works.

- Looking at the ICQ protocol, it is evident why it does not matter which direction the transaction takes, since instant messages should be both incoming (Network-Initiated) and outgoing (Subscriber-Initiated).
- With HTTP, it is sensible to restrict the direction of the transaction to Subscriber-Initiated, since HTTP is always Subscriber-Initiated when the subscriber ventures outward to surf the Internet. If the direction of the HTTP transaction is Network-Initiated, it probably means that a web server has been opened on the network subscriber's local machine for receiving incoming HTTP traffic. The provider may want to forbid the use of HTTP in this way because it strains network resources; in addition, this usage could be considered a form of network abuse and a breach of the subscriber-provider network service agreement.

Lists

When network addresses, such as IP addresses or host names, are arranged in groups connected by a common purpose and on which a subscriber's network transaction mapped to a service may be applied, it is called a *list*. The system can contain multiple lists. The master list, which connects the multiple lists together, is referred to as the list array.

Examples of lists are:

- A list of web sites offering offensive content that the provider specifies as undesirable. Access to these sites can be blocked by defining a "Parental Watch Service."
- A list of ftp sites containing network addresses, which the provider wants to limit download transfer rates to a certain bandwidth rate, or wants to limit the number of concurrent sessions it is allowed to use.

Lists containing network addresses may be specified as shown in following table:

Table 4-2 Examples of Types of Network Addresses

Network Address	Example
IP address	123.123.3.2
IP address range (and mask)	A range of IP addresses including mask can be of the form 123.3.123.0/24. This means that the first 24 bits of the IP address should be included as specified, and the remaining 8 bits or 256 IP addresses included in the range.
Host name	www.cnn.com

Rules

Components of a service, such as lists, protocols, and initiating side, give the system only the instructions on how to interpret the transaction passing through the network. They do not provide instructions on the action that should be applied to the service. A *rule* is defined as a condition on a service that specifies the action to be taken when the rule's condition is met.

Since services need to operate on time-based information also, the system specifies that the developer or programmer can define four sub-rules, one for each of the time frames defined.

Referring back to the section that described services, it was mentioned that services are made up of service elements. It is important to remember that rules apply to the entire service and not just to its individual elements; this is a system analysis and service design consideration that the developer or programmer should take into account when planning the service configuration.

In general, a rule is a set of instructions to the SCE platform about how to treat network transactions of a specific service. A rule may specify that a transaction should be blocked, or granted a certain amount of bandwidth. It may also define an aggregate volume or session limit, after which a set of different restrictions may be enforced on the transaction. A rule may also specify how a transaction should be reported for billing or analysis purposes.

Packages

A package defines the group of services delivered to the subscriber. It contains the definitions of the system's behavior per service, such as any restrictions on network transactions, guidelines for the transaction's prioritization, or instructions regarding how the transaction should be reported. This behavior is defined in a *Rule*.



Note The handling of Packages is license dependent.

Each subscriber in the network is provided a reference to a package to which that subscriber belongs. The system maps the network transaction to a certain service if it fits the definition of one of its service elements. In addition, the system identifies the subscriber to whom the transaction pertains, according to the transaction's network ID. Given that the system knows the transaction's network ID, the package the subscriber belongs to can be determined, and the correct rule can be applied to the service of the subscriber's network transaction.

The following table provides examples of packages, their services, and service parameters.

Table 4-3 Examples of Packages, Services, and Their Parameters

Package Name	Service	Access	Bandwidth Rate	Volume Quota	Concurrent Sessions per Transaction
adsl-bronze	Browsing	Admit	Unlimited	200 MB/Wk (mega-bytes per week)	Unlimited
	Web Hosting	Admit	20 Kb/s (kilo-bits per second)	Unlimited	3
	Local Streaming	Block	N/A	N/A	N/A
	FTP	Admit	30 Kb/s	Unlimited	5
	Video Conferencing	Admit	30 Kb/s	600 MB/Wk	5
adsl-gold	Browsing	Admit	Unlimited	Unlimited	1
	Web Hosting	Admit	80 Kb/s	500 MB/M (mega-bytes per month)	3
	Local Streaming	Admit	Unlimited	Unlimited	10
	Video Conferencing	Admit	Unlimited	Unlimited	30

Global Controllers

Bandwidth control in the SCAS BB solution is accomplished in two stages: global control and subscriber bandwidth control.

Bandwidth is controlled in the SCE Platforms by the use of virtual queues, or Global Controllers. You can configure a maximum of 16 Global Controllers per interface (upstream/downstream). As these are global controllers, their configuration is not linked to a package, but rather they are configured for the entire system.

The purpose of the global controllers is to provide constraints for large, global, volumes of traffic, such as “Total Gold Subscriber Traffic”, or “Total P2P Traffic”, as opposed to controlling bandwidth at the subscriber level. Each global controller represents the percentage of total system bandwidth that you want to allot to all traffic of a particular type. P2P traffic provides a good illustration, as the volume of P2P traffic has increased to the point where it is causing significant problems for many ISPs. Using the global controller, you can limit total P2P traffic in the system to any desired percentage of total traffic bandwidth, keeping the amount of total traffic bandwidth consumed by P2P traffic constant and under control.

Subscriber BW Controllers

A Subscriber Bandwidth Controller (BW Controller) controls the subscriber’s entire traffic or some portion of it. A BW Controller is specified by two main parameters that define:

- The minimal bandwidth that must be granted to traffic that is controlled by the BW Controller.
- The maximal bandwidth allowed to that traffic.

Subscriber BW Controllers enforce bandwidth in two levels:

- The first level, Primary BW Controller (Total), specifies bandwidth Service Configurations that the *provider* enforces on its subscribers.
- Second level, BW Controller (Internal), specifies Service Configurations that the *subscriber* wants to enforce on its Services.

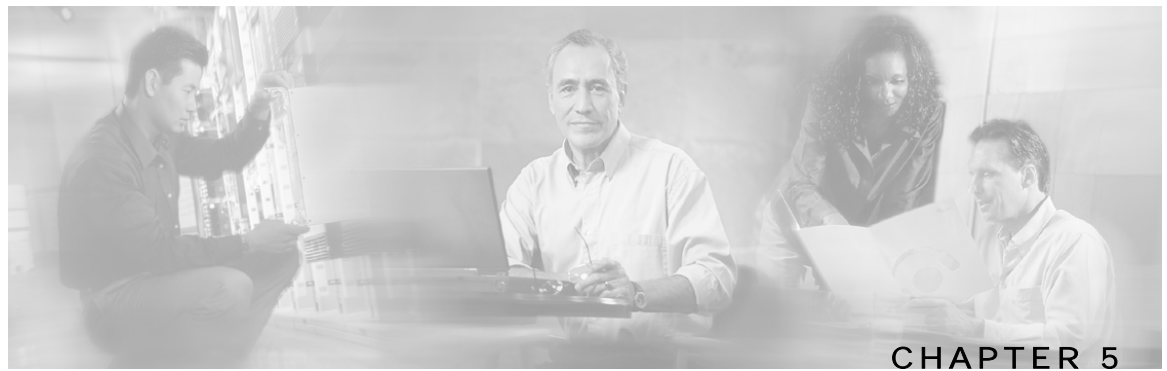
SCAS BB provides each subscriber with an independent set of BW Controllers. A single BW Controller is used to control the total bandwidth of the subscriber. This BW Controller, which provides the first-level control, is referred as the Primary BW Controller.

The other BW Controllers control the bandwidth of some Services of that subscriber. For example, one BW Controller may control the Streaming Service, while another may control the Download and Email Services together. These BW Controllers, which provide the second-level control, are referred as BWCs (Internal).

Subscriber Quota Buckets

When working in External Quota Provisioning mode, quota accounting is done using subscriber quota buckets. Each subscriber has 16 buckets, and each bucket can be defined for volume or sessions. When a subscriber uses a certain service, the amount of consumed volume or number of sessions is subtracted from one of the buckets. The service configuration determines which bucket to use for each service. In the case of volume buckets, consumption is counted in units of L3 kilobytes. In the case of session buckets, consumption is the number of sessions. For example, it is possible to define that the Browsing and E-mail services consume quota from Bucket #1, P2P service consumes quota from Bucket #2, and that all other services are not bound to any particular bucket.

External quota provisioning systems can use the *Quota Provisioning (QP) API* ("[Quota Provisioning API](#)" on page 7-1) to dynamically modify the quota in each bucket. For example, it is possible to increase the quota of a certain bucket when the subscriber purchases additional quota. These systems can also query the amount of remaining quota in each bucket. This can be used, for example, to show the subscriber (in some personal web page) how much quota is left.



Service Configuration API

This chapter discusses how the SCAS Service Configuration API contributes toward automating SCAS BB service configuration activities. This chapter:

- Describes the SCAS Service Configuration API's main elements.
- Explains and demonstrates the use of the SCAS Service Configuration APIs, service configuration creation, editing, maintenance, management, and distribution functions.
- Provides example code fragments demonstrating how to use the SCAS Service Configuration API.

This chapter contains the following sections:

- [Overview of Service Configuration API](#) 5-1
- [SCAS API Base Classes](#) 5-2
- [SCAS Client/Server Connectivity](#) 5-2
- [Managing Service Configurations with SCAS Service Configuration API](#) 5-4
- [Example - Adding a Service and Applying a Service Configuration](#)

Overview of Service Configuration API

The *Service Configuration* represents the business rules of the provider as they are to be enforced by the Service Control Application Suite for Broadband system on the provider's network, and the *Service Configuration API* is the programming end responsible for the *Service Configuration's* programming implementation.

When working with service configuration you use the Service Configuration API. The SCAS BB Console is created with the Service Configuration API programming tools. The API lets you automate routine service configuration activities, otherwise performed manually in the GUI.

From the stage of defining service configuration until the stage of activating it on the SCE Platforms, the main programming steps of the Service Configuration API are:

-
- Step 1** Defining a new service configuration, or retrieving an existing service configuration from the SCE.

- Step 2** Defining a service and assigning it a name.
- Step 3** Defining protocols and lists.
- Step 4** Defining the service elements, and assigning them a protocol, a direction, and an optional list.
- Step 5** Adding the service elements to the service.
- Step 6** Defining a package and assigning it a name.
- Step 7** Defining the package's service rules, such as pre-breach and post-breach bandwidth limitations and whether the service is enabled or disabled.
- Step 8** Activating the service configuration by propagating it to the SCE.

The primary activity performed with the SCAS Service Configuration API is service configuration creation and manipulation. This includes the definitions of services and rules, which can be done offline. Once the service configuration is ready, it can be propagated to the SCE Platforms.

Each of the steps mentioned above is explained in detail in the following sections. By working through the sections in the chapter, together with the *Subscriber Management API Reference Manual*, you will be able to create, edit, administer, and distribute SCAS BB service configurations.

SCAS API Base Classes

The main SCAS API classes that are used for creating a working application are listed in the following table.

Table 5-1 The Main SCAS API Classes

Class	Purpose
EngageAPI	Has functionality for enabling client programs to connect to the system's hardware and software. Performs network connection functions, and is the starting point for element management activities.
SCAS PolicyAPI	Has functionality for managing service configurations.

SCAS Client/Server Connectivity

A client application, such as an SCAS Java program, can connect to the SCE. The SCAS API Java class that enables the client application to connect to the SCE Platform is Engage.

- Including the SCAS Libraries
- Connecting to the SCE Platform

Including the SCAS Libraries

Working with the SCAS Java API classes requires Java JDK version 1.3 or later.

The following JAR files should be included in the Java class-path:

- `um_core.jar`
- `xerces.jar`
- `regexp.jar`
- `jdmkrt.jar`
- `log4j.jar`
- `engage.jar`

These files are installed on the workstation as part of the installation process for an SCAS client. This installation is performed using the SCAS BB *clients installation CD-ROM* provided with the SCAS BB product.

Assuming that the SCAS clients were installed in the folder `C:\Program Files\Cisco\SCAS BB x.x.x\`, all these JAR files can be located in the folder `C:\Program Files\Cisco\SCAS BB x.x.x\lib`.

**Note**

The version of the above JAR files should be coordinated with the version of other SCAS BB solution components. Whenever a new solution version is installed or upgraded, be sure to upgrade the API development JAR files.

Before you can work with SCAS Java API you need to import the SCAS Java package. Do this by adding the following lines of code at the beginning of your program:

```
import com.cisco.apps.scas.*;
import com.cisco.apps.scas.common.*;
import com.cisco.apps.scas.policy.*;
```

Connecting to the SCE Platform

To retrieve a service configuration from the SCE Platform or to apply a service configuration to the SCE Platform, it is necessary to connect to the platform. The SCAS class that makes this connection possible is responsible for the communication and coordination of networking activities.

The `login` method of the class provides a persistent connection. This method expects a user name, a password, and the host name of a SCE to which to connect. If one of the method's parameters is incorrect, an exception is returned. Notice that once the service configuration has been retrieved or applied, the connection should be closed.



Note: Always call the `logout` method when finished; do not leave an open connection to the SCE. Also, try to reuse the `Connection` object, do not create a new one each time you need to retrieve/apply.

The `login` method sets up the essential parameter values for a *Service Configuration API* call later on, after a persistent login connection has been established. This method returns an object of type `Connection`, which is a handle to the SCE used in most of the *Service Configuration API*.

The following Java code demonstrates how a connection is made to an SCE:

```
String username = "admin";
String password = "pcube";
String se = "212.47.174.32";
Connection connection = null;
try{
    connection = Engage.login(se, user, password, Connection.SE_DEVICE);
}catch (ConnectionFailedException e)
{
    // login failed - handle exception
}
```

The connection to the SCE remains open until it is closed. To break the connection with the SCE, use the `logout` method, as follows:

```
Engage.logout(connection);
```

It is worth noting that although the SCAS *Service Configuration API* performs various service configuration editing tasks, it is not necessary to be connected to the SCE while the service configuration is being edited. That is, the service configuration can be prepared offline, and when service configuration editing is complete a connection can be made to the SCE and the service configuration propagated.

Managing Service Configurations with SCAS Service Configuration API

The previous section showed how to connect (log in) and disconnect (log out) from the SCE, using the SCAS API. Other operations can be performed on service configurations, such as creating a new service configuration from scratch, or applying a service configuration: an operation that propagates the service configuration to the SCE Platform and enables it for network activation.

The Java `PolicyAPI` class serves as the starting point for the services of the SCAS Service Configuration API.

The following sections provide pseudo-code examples, illustrating how various operations may be performed on packages, services, service elements, protocols, lists, and rules. The service configuration components are finally assembled in a package to be propagated to the SCE. These sections include the following:

- Retrieving and Applying Service Configurations
- Importing, Exporting, and Creating Service Configurations
- Lists
- Protocols
- Service and Service Elements
- Packages

Retrieving and Applying Service Configurations

In the provider's network, changes that are made to service configurations are usually made to service configurations that are currently in use. For changes to be made, the service configuration needs to be *retrieved* from the SCE for service configuration editing. *Service Configuration* editing is done offline until the service configuration is ready for propagation to the SCE. *Service Configuration* propagation can also be referred to as *applying* the service configuration, which overwrites the original service configuration's contents and activates the new business rules on the SCE Platform

The two main service configuration methods used to retrieve service configurations and apply service configurations to the SCE, `retrievePolicy` and `applyPolicy`, are shown in the following code:

```
// retrieve a service configuration
Policy myPolicy =
    PolicyAPI.retrievePolicy(connection);
// apply a policy
PolicyAPI.applyPolicy (connection, myPolicy, SCAS.APPLY_FLAG_OVERWRITE);
```

One or more Service Control Platforms can be defined as the *SCE*, the second parameter of the `retrievePolicy()` member function. After `SCAS.Policy` is retrieved, you can perform various functions, such as inspecting, modifying, or saving the service configuration's contents.

While an SCAS application is being developed, the system provides some useful facilities for reviewing your SCAS Service Configuration API application. For example, if you are modifying a service configuration with your Java application, you can save the service configuration's contents as a file using the `savePolicy` method. Afterward, you can open this file using the SCAS Service Configuration Manager GUI application and inspect the service configuration elements that your application has modified.

A SCAS service configuration is applied by calling the `applyPolicy` method.

Importing, Exporting, and Creating Service Configurations

In addition to retrieving an existing service configuration from an SCE Platform, another way to retrieve a service configuration is through text files. The following code shows how to import a service configuration:

```
String filename = "policy.pqb";
BufferedReader br =
    new BufferedReader(new InputStreamReader(new
        FileInputStream(filename)));
Policy importedPolicy =
    ImportExportUtils.importPolicy(ImportExportUtils.XML_FORMATTER, br);
```

The purpose of exporting a service configuration might be for archiving purposes, or for editing the service configuration at a later time. The following code shows how to export a service configuration:

```
String filename = "policy.pqb";
PrintStream print = new PrintStream(new FileOutputStream(new
    File(filename)));
ImportExportUtils.exportPolicy(policy, ImportExportUtils.XML_FORMATTER,
    print);
```

New service configurations are usually created by modification of existing service configurations, but when necessary new service configurations can be created from scratch. The constructor of the `Policy` class uses the `new` operator to create a service configuration. The following line of code creates such a service configuration:

```
Policy myPolicy = new Policy("");
```


Lists

The following code fragment demonstrates how lists can be used. A newly created list of offensive-content web sites is to be added to the existing service configuration's list array. The list created in the sample is called *Offensive Content Website List*, and list elements are added to it. Finally, the newly created list is added to the service configuration's list array. Later, a rule can be enforced to block access to the service with the *HTTP Browsing Protocol* for the host names contained in the list.

The constructor of the `HostList` class expects as parameters a list name and an optional description. The `add` method of the `HostList` class adds new elements to the list. The `listArray` is retrieved using the `getListArray` method and the newly created list `offensiveList` is appended to it, as shown in the following code.

```
String name = " Offensive Content Website List ";
String description = " A list of offensive website hosts. ";
HostList offensiveList = new HostList(name, description);
try
{
    offensiveList.add(new HostListItem("www.offensive-content.com"));
    offensiveList.add(new HostListItem("www.more-offensive-content.com"));
} catch(DuplicateItemException e)
{
    // handle duplicate exception
}
ListArray listArray = myPolicy.getListArray();
try
{
    listArray.addList(offensiveList);
} catch(DuplicateItemException e)
{
    // handle duplicate exception
}
```

The following sections explain the different elements of the code fragment shown above. These sections cover:

- Retrieving a List Array
- Navigating a List Array
- Determining the Type of a List
- Adding Elements to a List Array

Retrieving a List Array

The following line of code retrieves the array of lists:

```
ListArray listArray = myPolicy.getListArray();
```

The `getListArray` method retrieves an array of lists of type `ListArray`. After the list array has been retrieved, you can ask questions about a list, such as list size, or retrieve a list's individual elements.

Navigating a List Array

The following code iterates through the elements of the `listArray`:

```
for (int i = 0; i < listArray.getSize(); i++)
{
    Object o = listArray.getElementAt(i);
}
```

The loop shown traverses the `listArray` elements. The `getSize` method returns the size of the `ListArray`, and the individual elements of the list's `getElementAt` method are assigned to `Object`.

Determining the Type of a List

The `getElementAt` method returns an object, which can be an instance of the `HostList` class or the `IPRangeList` class. The `IPRangeList` is a list of IP ranges. The following code shows how the method can be used for determining the type of list:

```
if (o instanceof HostList)
{
    HostList hostlist = (HostList) o;
    System.out.println("Host List");
} else // IPRangeList
{
    IPRangeList iplist = (IPRangeList) o;
    System.out.println("IP Range List");
}
```

Adding Elements to a List Array

The following line of code adds a new element to the list:

```
// To add a new host list item
hostlist.add(new HostListItem("www.cnn.com"));
```

Assuming we retrieved the first list in the group of lists, `www.cnn.com` (<http://www.cnn.com>) is added to the list.

Protocols

SCE Platforms react to the protocols of the network transaction that are mapped to services. A service configuration contains a list of services, and their protocols can be created in one of two ways: new protocols can be defined, or existing protocols can be expanded to include additional ports. In addition, dynamic signature scripts that define protocols can be imported/removed. This material is covered in the following sections:

- Defining Protocols
- Adding Ports to Protocols
- Dynamic Signature Scripts

Defining Protocols

Protocols are recognized based on accepted networking conventions. For example, when the server listens for a transaction on port 666, it is an accepted convention that port 666 listens for protocols of type DOOM. It is in the interest of both parties to recognize the convention for the service that is to be delivered.

The following code fragment shows how a protocol called quake is defined on port 26000 with a transport type of TCP:

```
// create a new protocol and pass it a reference to a service configuration
Protocol quakeProtocol = new Protocol(myPolicy);
// set the protocol name
try
{
    quakeProtocol.setName("quake");
} catch(DuplicateException de)
{
    //a Protocol with such a name already exists in the service
    configuration
    // handle exception
} catch(ItemNotFoundException infe)
{
    //item was missing while validating rename: service configuration
    corrupt
    // handle exception
}

// add new port and transport type
try
{
    quakeProtocol.add(new PortListItem(26000, Consts.TRANSPORT_TYPE_TCP));
} catch(DuplicateItemException e)
{
    // the service configuration has a Protocol with such a defined port
    // handle exception
}
// add the quake protocol to the service configuration 's protocol list
try
{
    protocols.add(quakeProtocol);
} catch(DuplicateItemException e)
{
    // the service configuration already has this Protocol
    // handle exception
}
```

The above example assumes that a reference to a service configuration called myPolicy exists, and is passed as a parameter to the constructor of the Protocol class. The quakeProtocol reference returned by the constructor is then used to set the name of the protocol.

The following line of code specifies that the protocol has a port number of 26000 and a transport type of TCP:

```
quakeProtocol.add(new PortListItem(26000, Consts.TRANSPORT_TYPE_TCP));
```

The following line of code adds the protocol to the service configuration's protocol list:

```
protocols.add(quakeProtocol);
```

This completes the definition of a protocol. The next section explains how additional ports can be added to existing protocols.

Adding Ports to Protocols

Ports that are free can be *added* to existing protocols. This essentially maps the port to the protocol and expands the protocol range.

The following code fragment retrieves the service configuration's protocol list and searches for a system-defined protocol called HTTP Browsing. Upon finding it, the protocol definition is expanded to include an additional port.

```

. . .
// get protocol list
ProtocolArray protocols = myPolicy.getProtocolList();
// get a protocol called "HTTP Browsing":
try
{
    Protocol http = protocols.getProtocol ("HTTP Browsing");
} catch (ItemNotFoundException infe)
{
    // the service configuration does not have a Protocol with such a name
    // handle exception
}
// add port 8082 to HTTP Browsing
try
{
    http.add(new PortListItem (8082, Consts.TRANSPORT_TYPE_TCP));
} catch (DuplicateItemException e)
{
    // the service configuration has a Protocol with such a defined port
    // handle exception
}
. . .

```

The example assumes that a protocol called HTTP Browsing exists. The `getProtocol` method of the `Protocol` class returns a handle that points to the protocol array, as shown in the following line of code:

```
ProtocolArray protocols = myPolicy.getProtocolList();
```

The following line of code searches by name for the specified protocol:

```
Protocol http = protocols.getProtocol ("HTTP Browsing");
```

The `add` method of class `Protocol` expects a numerical port value and a transport type. In the above example, port 8082 is introduced as having a TCP transport type, and is added to an existing service called HTTP Browsing. The following line of code adds a port to a protocol:

```
http.add(new PortListItem (8082, Consts.TRANSPORT_TYPE_TCP));
```

The service called HTTP Browsing has been expanded. This expansion is necessary if you want the service rule to service HTTP transactions delivered on port 8082.

Service and Service Elements

The next example creates a Service with two service elements. The following code fragment demonstrates how many *service elements* can be associated with a single *service*.

One service element is created for the game Doom, and the second service element is created for the game Quake. Both are well known games, and will be added to a service called Gaming Service.

```

    . . .
Service gamingService = new Service(myPolicy); // create a service
try
{
    gamingService.setName("Gaming Service"); // set service name
} catch(DuplicateException de)
{
    // a Service with such a name already exists in the service
configuration
    // handle exception
} catch(ItemNotFoundException infe)
{
    //item was missing while validating rename: service configuration
corrupt
    // handle exception
}

// create a service element for Subscriber-Initiated Doom
try
{
    gamingService.addProtocol("doom",

ServiceElement.DIRECTION_SUBSCRIBER_INITIATED);
} catch(ItemNotFoundException e)
{
    // there is no such Protocol in the service configuration
    // handle exception
} catch(DuplicateItemException e)
{
    // there is already a Service with such a Protocol and direction in the
service configuration
    // handle exception
}
// create a service element for Subscriber-Initiated Quake
try
{
    gamingService.addProtocol("quake",

ServiceElement.DIRECTION_SUBSCRIBER_INITIATED);
} catch(ItemNotFoundException e)
{
    // there is no such Protocol in the service configuration
    // handle exception
} catch(DuplicateItemException e)
{
    //There is already a Service with such a Protocol and direction in the
service configuration
    // handle exception
}

// add service to service configuration
try

```

```
{
    myPolicy.getServiceList().add(gamingService);
} catch(DuplicateItemException e)
{
    //There is already a Service in the service configuration
    // handle exception
}

. . .
```

The following sections explain the different elements of the code fragment shown above:

- Creating a Service
- Defining Service Elements
- Adding a Service to Service Configuration

Creating a Service

The class that contains the Service is instantiated in the following line of code:

```
Service gamingService = new Service(myPolicy); // create a service
```

It assumes that a Policy reference called myPolicy is passed as a parameter to the Service constructor.

The following line of code defines the name of the service:

```
gamingService.setName("Gaming Service"); // set service name
```

Defining Service Elements

The following line of code creates a new service element:

```
gamingService.addProtocol("doom",
    ServiceElement.DIRECTION_SUBSCRIBER_INITIATED);
```

The service element created above is applied to all Subscriber-Initiated network transactions of the doom protocol. However, you can specify that this service element should apply only to those network transactions carried out against specific network addresses. To do so, include a list of server addresses with the service element, as in the following example:

```
int listIndex =
    myPolicy.getListArray().getListIndex("gaming servers list");
if(listIndex == -1)
{
    // there is no such list
    // handle error
}
int serviceElementIndex =
gamingService.getServiceElementArray().getIndexOfItem("doom", ServiceElement.
DIRECTION_SUBSCRIBER_INITIATED);
if(serviceElementIndex == -1)
{
    // There is no such ServiceElement
    // handle error
}
try {
    gamingService.addList(serviceElementIndex, listIndex);
} catch(ItemNotFoundException e)
{
    // handle error
} catch(DuplicateItemException e)
{
    // handle error
}
```

Adding a Service to Service Configuration

As the final step, the service needs to be added to the service configuration. The following code shows how to add `gamingService` to the service configuration:

```
// add service to service configuration
myPolicy.getServiceList().add(gamingService);
```

Packages



Note

Defining a package is license dependant; the capacity control of a license prevents defining more packages than the license specifies

Before you can start to define rules and packages, you first need to define services and add them to the service configuration, as was described in the previous section.

The following section of code illustrates how packages and rules are defined, and how they may be used. Two packages, a Family Package and a Bachelor Package, are created for a single service configuration.

The example assumes we had already created a Parental Watch Service that contains a Parental Watch List. For the Family Package, we enable the Parental Watch Service. Subscribers to this package are blocked from accessing the censored content defined in the Parental Watch List. For the Bachelor Package we do not enable the Parental Watch Service, thus permitting access to those sites.

```

/* Define package 1 */
Package family = new Package(myPolicy); // create a package
try
{
    family.setName("Family Package"); // set package name
} catch (DuplicateItemException e)
{
    // there already is a Package in service configuration with such a name
    // handle exception
}

ServiceRule serviceRule = new ServiceRule(policy);

// adding service to rule
try
{
    serviceRule.setServiceName("Parental Watch Service");
} catch (ItemNotFoundException e)
{
    // no such service
    // handle exception
}

// get the default rule - it will apply to all the time-based rules
// since no time-based rules are created
Rule rule = serviceRule.getDefaultRule();
// set rule state to be enable
rule.setState(Rule.RULE_STATE_ENABLED);

// set rule action to be block
rule.setPreBreachAccessMode(Rule.ACCESS_BLOCK);

// get package's service rule array
ServiceRuleArray serviceRuleArray = family.getServiceRuleList();

// add service rule to the package's serviceRuleArray
try
{
    serviceRuleArray.add(serviceRule);
} catch (DuplicateItemException e)
{
    // there already is such a rule
    // handle exception
}

/* Define package 2 */
Package bachelor = new Package(myPolicy); // create a package
try

```



```

{
    bachelor.setName("Bachelor Package"); // set package name
} catch (DuplicateItemException e)
{
    // there already is a Package in service configuration with such a name
    // handle exception
}

serviceRule = new ServiceRule(policy);

// adding service to rule
try
{
    serviceRule.setServiceName("Parental Watch Service");
} catch (ItemNotFoundException e)
{
    // no such service
    // handle exception
}

// get the default rule - it will apply to all the time-based rules
// since no time-based rules are created
rule = serviceRule.getDefaultRule();
// set rule state to be enable
rule.setState(Rule.RULE_STATE_ENABLED);

// set rule action to be block
rule.setPreBreachAccessMode(Rule.ACCESS_ADMIT);

// get package service rule array
serviceRuleArray = bachelor.getServiceRuleList();

// add service rule to the package's serviceRuleArray
try
{
    serviceRuleArray.add(serviceRule);
} catch (DuplicateItemException e)
{
    // there already is such a rule
    // handle exception
}

// add the packages to the service configuration
try
{
    policy.getPackageList().add(family);
    policy.getPackageList().add(bachelor);
} catch (DuplicateItemException e)
{
    //there are already such packages
    //handle exception
}

```

The following sections explain the different elements of the code fragment shown above. These sections cover:

- Creating and Naming Packages
- Defining Service Rules
- Breaches
- Example - FTP Service Rule

- Aggregation
- Bandwidth Controller
- Breach Reports
- Default and Non-Default Time Frames
- Block and Re-Direct

Creating and Naming Packages

To create a package, use the new operator. The class that contains the `Package` is instantiated as shown in the code of the following line:

```
Package family = new Package(myPolicy); // create a package
```

After creating a new package, give the package a name, as shown in the following line of code:

```
family.setName("Family Package"); // set package name
```

Defining Service Rules

The following line of code returns a reference to the rule specified in the parameter of the residential `getRule` method:

```
ServiceRule pw =residential.getRule("Parental Watch Service");
```

Breaches

A *breach* occurs when a *limit* placed on a network transaction is exceeded. The breach is defined when the rule for the service is declared. It can be the exceeding of a limit, such as a bandwidth volume limit or a kilobits per second (Kb/s) transfer rate limit; or can be some other pre-determined limit placed on the service. The purpose of defining a breach is to specify an action to be executed when the limit has been breached.

A Rule has two modes: *pre-breach* and *post-breach*. The system performs the *pre-breach action* before the limit of the service rule is reached, and the *post-breach action* after the limit is reached and exceeded.

Examples of these actions are bandwidth volume restriction, denial of service, triggering of an RDR report, and web-site redirection. Every pre-breach function call (method) has its post-breach counterpart.

A breach needs to be made active before it can be applied, since by default a breach is inactive. The code in the following line shows how to activate the pre-breach condition by setting the flag of the `setPreBreachAccessMode` method to true:

```
rule.setPreBreachAccessMode(Rule.ACCESS_ADMIT);
```

Example - FTP Service Rule

The following example illustrates how rules for an ftp service can be set up. It assumes that a service called FTP File Downloading exists. The rules restrict bandwidth to 5 Kb/s and 100 MB daily. If either limit is reached, the service is blocked and an RDR is triggered. The code in this example specifies that the RDR should use daily aggregates for reporting. Finally, the service configuration is applied before the connection is closed.

```
// create new package
Package residential = new Package(myPolicy);

// add the packages to the service configuration
try
{
    policy.getPackageList().add(residential);
} catch(DuplicateItemException e)
{
    // there already are such packages
    // handle exception
}

// set service name
try
{
    residential.setName("residential");
} catch (DuplicateItemException e)
{
    // there already is a Package with such a name in service configuration
    // handle exception
}

ServiceRule serviceRule = new ServiceRule(policy);

// adding service to rule
try
{
    serviceRule.setServiceName("Parental Watch Service");
} catch(ItemNotFoundException e)
{
    // no such service
    // handle exception
}

// get the default rule - it will apply to all the time-based rules
// since no time-based rules are created
Rule rule = serviceRule.getDefaultRule();
// set rule state to be enable
rule.setState(Rule.RULE_STATE_ENABLED);

// set rule action to be admitted
rule.setPreBreachAccessMode(Rule.ACCESS_ADMIT);

// limit daily volume to approximately 100 MB
rule.setBreachVolumeLimit(100000);

// set rule post-breach action to be blocked
rule.setPostBreachAccessMode(Rule.ACCESS_BLOCK);

// generate RDR when limit is breached
rule.setBreachReportEnabled(true);
```

```

// get package service rule array
ServiceRuleArray serviceRuleArray = residential.getServiceRuleList();

// add service rule to the package's serviceRuleArray
try
{
    serviceRuleArray.add(serviceRule);
} catch(DuplicateItemException e)
{
    // there is already such a rule
    // handle exception
}

// tell the package to use daily aggregation periods
residential.setAggregationPeriod(Package.AGGREGATED_PERIOD_DAILY);

// apply changes to the SCE Platform
PolicyAPI.applyPolicy(connection, myPolicy, SCAS.APPLY_FLAG_OVERWRITE);
// close the connection
SCAS.logout(con);

```

As shown in the above example, the system can measure total volume either in kilobytes used or as the number of network sessions used in a service transaction. If one of these bounds is exceeded, the system can, for example, modify the subscriber's bandwidth usage rate, send out a report (RDR), or block the subscriber from using the service.

The following sections break down the program and explain its individual elements.

Aggregation

Aggregation periods are defined globally per package. The aggregate is used for reporting purposes when an RDR is created. Aggregates can be measured on a monthly, weekly, daily or hourly basis.

For the same service, different aggregate periods can be used in different packages. That is, in one package you can specify that a daily aggregate of an `FTP Download Service` should be used, and in a second package using the same `FTP Download Service` specify that an hourly total be used. The aggregate, or total usage information, can then be used in the subscriber's itemized bill or account statement.

The following line of code specifies that daily totals should be used in the generated RDR:

```
residential.setAggregationPeriod (Package.AGGREGATED_PERIOD_DAILY);
```

Quota resolution is defined per package, therefore if the package consists of two services—for example, an `FTP Download Service` and a `Video Conferencing Service`—the aggregation would reflect the total of both services. This should be a consideration when designing the package.

It is worth noticing in the above example that the order of commands does not matter, since the service configuration is edited off-line. In consequence, the commands are not executed until the modified service configuration has been propagated to the SCE Platforms of the SCE.

Bandwidth Controller

Bandwidth controllers, also known as meters, are flow-rule-based functions used for measuring the volume, in Kbps, that a particular service transaction is using. They can measure single-flow transactions or transactions that use several flows simultaneously. Bandwidth controller rules may be flow-based rules, time-based rules, or a combination of the two.

Bandwidth controllers are directional: each bandwidth controller controls the volume in either the upstream or the downstream direction. The direction is configurable through the API.

It is useful to know the *number of flows* (concurrent sessions) and their total bandwidth usage rate, since it gives the provider greater control over system services and enables the SP to have greater influence over how network resources are utilized. For example, bandwidth controller functionality enables the provider to specify that the subscriber may use an unlimited number of flows, so long as a certain bandwidth rate is not exceeded.

Having control over the number of flows being used in a network transaction would be useful, for example, in a digitally broadcast Internet video application using the RTSP protocol. RTSP is capable of keeping several concurrent flows of video traffic going at once. The bandwidth controller gives the provider the tools to specify that the subscriber may not exceed a certain rate limit.

The following code illustrates use of a bandwidth controller function at the flow level:

```
// assigning BWController number 3 to the Rule in upstream direction
ftpRule.getDefaultRule().setPreBreachUpstreamBWControllerIndex(3);
```

Breach Reports

A *breach* occurs when a system-specified limit has been exceeded. When a breach is encountered, two kinds of actions may take place. They are: (a) the service may be blocked, or (b) some sort of change may be made to the way in which a subscriber's transaction or service is delivered.

The system provides, therefore, a facility to generate two types of reports. They are: (a) block report, or (b) breach report. The system can be instructed that each time a service has been breached, the system should trigger a report that depends on the type of breach that occurred.

If a breach occurs, the following line of code triggers the creation of an RDR by setting the function's parameter to true:

```
Rule.setBreachReportEnabled(true);
```

Alternatively, instead of generating an RDR, the system can be set up such that each time a breach occurs the subscriber is notified by e-mail.

Default and Non-Default Time Frames

This section discusses time frames, default rules, and non-default rules, and when the default rule function should be used.

Through the use of time frames, a single rule may be set up to have different behavior at different times. Examples of time frames are an evening time frame, a night time frame, and a weekend time frame. The SCAS system allows up to four different time frames to be defined.

Using the API gives the capability of developing an application that handles information based on time-frame exceptions. For example, an application that changes the time frame so that a weekend-rate Video Conferencing Service allows unlimited bandwidth usage from Wednesday the 31st at 3 p.m. until 6 p.m. of the following day, as opposed to its regular day-based schedule where the weekend rate begins at Friday 4:54 p.m.

When the `getDefaultRule` method is used, it means that the rule is applied globally to all time frames. The following code is an example of the `getDefaultRule` method being used set a global time-frame rule:

```
// set rule post-breach action to block access
rule.setPostBreachAccessMode(Rule.ACCESS_BLOCK);
```

An example of a non-default rule is the following. Assume a default rule was defined where a subscriber is denied access to a service. Assume also two time frames were created: one for weekdays between 2200 till 0600, and the other for Fridays during the same hours. An exception could be created whereby access to the service is permitted during the weekday time frame while access to the service on Fridays is blocked.

The following code fragment illustrates how a single time frame called T1 is created. Since the default rule applies to all time frames, the T1 time frame is used to establish its exception and therefore permit access to the service during the period for which it is defined.

```

. . .
// creates a new Package
Package myPackage = new Package(myPolicy);

// add the packages to the service configuration
try
{
    policy.getPackageList().add(myPackage);
} catch(DuplicateItemException e)
{
    // there already are such packages
    // handle exception
}

// creates new service rule
ServiceRule serviceRule = new ServiceRule(policy);

// adds service to rule
serviceRule.setServiceName(serviceName);

// gets the default rule - it will apply to all time based rules
// since no time based rules are created
Rule defaultRule = serviceRule.getDefaultRule();
// set default rule state to be disable
defaultRule.setState(Rule.RULE_STATE_DISABLED);

// announce the need for configuring specific behavior to T1
Rule t1Rule = null;
```

```

Try
{
    t1Rule = serviceRule.addTimeFrameRule(TimeFrame.T1);
} catch(ItemNotFoundException e)
{
    // handle exception
} catch(DuplicateItemException e)
{
    // handle exception
}

// set T1 rule state enabled
t1Rule.setState(Rule.RULE_STATE_ENABLED);

// adds service rule to the package's serviceRuleArray
myPackage.setServiceRule(serviceRule);
. . .

```

Block and Redirect

Block-and-redirect functions, relevant only to services using redirectable protocols such as HTTP and RTSP, are provided so that if a subscriber attempts to access a service that is not part of the service package, the subscriber's transaction can be redirected to a different network address. The network address or web site can be used to provide useful information regarding how to gain access to the service.

```

// assign the default redirect string of the "HTTP Browsing" protocol
// to be redirected to a certain URL
try
{
    policy.setProtocolRedirectString("HTTP
Browsing","http://www.mySite.com/redirect.html");
} catch (ItemNotFoundException e)
{
    //handle exception
} catch (MalformedURLException e)
{
    //handle exception
}

// creates new service rule
ServiceRule serviceRule = new ServiceRule(policy);

// adds service to rule
serviceRule.setServiceName("http browsing service");
// gets the default rule - it will apply to all time-based rules
// since no time-based rules are created
Rule defaultRule = serviceRule.getDefaultRule();
// set default rule state to be enable
defaultRule.setState(Rule.RULE_STATE_ENABLED);

// set rule post-breach action to block and redirect
defaultRule.setPostBreachAccessMode(Rule.ACCESS_BLOCK_AND_REDIRECT);

```

Example - Adding a Service and Applying a Service Configuration

The Doom gaming application is used as a template in the following example, which explains how a network provider can set up a service on a gaming application server hosted within the subscriber's internal network. The Java application has the following features:

- Step 1** A new service configuration will be created.
- Step 2** A service element named *Local Doom* will be defined.
- Step 3** The Doom service will be Subscriber-Initiated.
- Step 4** The Java application will map a list of IP addresses of local servers within the subscriber's network for hosting the service.
- Step 5** The Java application will trigger billing records based on the amount of time the subscriber stays connected to the service.


```

import com.cisco.apps.scas.Connection;
import com.cisco.apps.scas.ConnectionFailedException;
import com.cisco.apps.scas.SCAS;
import com.cisco.apps.scas.PolicyAPI;
import com.cisco.apps.scas.common.DuplicateItemException;
import com.cisco.apps.scas.common.ItemNotFoundException;
import com.cisco.apps.scas.policy.IPListItem;
import com.cisco.apps.scas.policy.IPRangeList;
import com.cisco.apps.scas.policy.ListArray;
import com.cisco.apps.scas.policy.Package;
import com.cisco.apps.scas.policy.PackageArray;
import com.cisco.apps.scas.policy.Policy;
import com.cisco.apps.scas.policy.PortListItem;
import com.cisco.apps.scas.policy.Protocol;
import com.cisco.apps.scas.policy.ProtocolArray;
import com.cisco.apps.scas.policy.Rule;
import com.cisco.apps.scas.policy.Service;
import com.cisco.apps.scas.policy.ServiceArray;
import com.cisco.apps.scas.policy.ServiceElement;
import com.cisco.apps.scas.policy.ServiceRule;
import com.cisco.apps.scas.policy.ServiceRuleArray;

/**
 * SCAS API Example
 */
public class Example {

    public static void main(String[] args) {
        Policy policy = new Policy("");

        // get policy package list
        PackageArray packageArray = policy.getPackageList();

        byte ip_mask = 32;

        // create host list item
        IPListItem item1 = new IPListItem("1.1.1.1", ip_mask);
        IPListItem item2 = new IPListItem("2.2.2.2", ip_mask);
        IPListItem item3 = new IPListItem("3.3.3.3", ip_mask);

        // create ip list
        String listName = "doom ips";
        IPRangeList ipList =
            new IPRangeList(listName, "A list of doom ips", false);

        // add list item to list
        try {
            ipList.add(item1);
            ipList.add(item2);
            ipList.add(item3);
        } catch (DuplicateItemException e) {
            System.out.println(
                "Add ip list item to ip list failed : " +
e.getMessage());
            System.exit(1);
        }

        // get the policy list array and add to it the ip list
        ListArray policyListArray = policy.getListArray();
        try {
            policyListArray.addList(ipList);
        } catch (Exception e) {

```

Example - Adding a Service and Applying a Service Configuration

```

        System.out.println(
            "Add ip list to policy lists failed :" +
e.getMessage());
        System.exit(1);
    }

    // get Policy Service list
    ServiceArray policyServiceList = policy.getServiceList();

    // create doom protocol
    String protocolName = "doom";
    ProtocolArray policyProtocols = policy.getProtocolList();
    Protocol protocol = new Protocol(policy);

    // set port
    PortListItem doomPort =
        new PortListItem(666, PortListItem.TRANSPORT_TYPE_BOTH);

    try {
        protocol.add(doomPort);
    } catch (DuplicateItemException e) {
        System.out.println(
            "ERROR - adding port 666 failed :" +
e.getMessage());
        System.exit(1);
    }

    try {
        policyProtocols.add(protocol);
    } catch (DuplicateItemException e) {
        System.out.println(
            "ERROR - adding protocol \"
            + protocolName
            + \" failed:"
            + e.getMessage());
        System.exit(1);
    }

    // to allow doom to support IP addresses,
    // the generic TCP and UDP must support it
    try {
        policyProtocols.getProtocol(
            Protocol.GENERIC_UDP_PROTOCOL).setListElementsType(
            Protocol.LIST_SUPPORT_IP_RANGE_LIST);
    } catch (Exception e) {
        System.out.println(
            "ERROR - UDP setListElementsType threw exception"
            + e.getMessage());
        System.exit(1);
    }

    // create new Service
    Service service = new Service(policy);

    String serviceName = "Local Doom";

    try {
        // set the service name, which is its identifier
        service.setName(serviceName);
    } catch (ItemNotFoundException e) {
        System.out.println(

```

```

        "ERROR - service set name threw
ItemNotFoundException : "
                + e.getMessage());
        System.exit(1);
    } catch (DuplicateItemException e) {
        System.out.println(
            "ERROR - service set name threw
DuplicateItemException : "
                + e.getMessage());
        System.exit(1);
    }

    // add the doom protocol to the service
    try {
        service.addProtocol(protocolName,
ServiceElement.DIRECTION_BOTH);
    } catch (ItemNotFoundException e) {
        System.out.println(
            "ERROR - adding protocol threw
ItemNotFoundException : "
                + e.getMessage());
        System.exit(1);
    } catch (DuplicateItemException e) {
        System.out.println(
            "ERROR - adding protocol threw
DuplicateItemException : "
                + e.getMessage());
        System.exit(1);
    }

    // add the list to the service: the first index is
    // the service element index - that is, to which protocol -
    // and the second index is the list index in the policyListArray
    try {
        service.addList(0, ipList.getName());
    } catch (ItemNotFoundException e) {
        System.out.println(
            "ERROR - adding list threw ItemNotFoundException : "
                + e.getMessage());
        System.exit(1);
    } catch (DuplicateItemException e) {
        System.out.println(
            "ERROR - adding list threw DuplicateItemException
: "
                + e.getMessage());
        System.exit(1);
    }

    // add service to policyServiceList
    try {
        policyServiceList.add(service);
    } catch (DuplicateItemException e) {
        System.out.println(
            "ERROR - adding service to policy service list "
                + "threw DuplicateItemException : "
                + e.getMessage());
        System.exit(1);
    }

    // creating a new Package
    Package pack = new Package(policy);

```

Example - Adding a Service and Applying a Service Configuration

```

String packageName = "Game users";
try {
    pack.setName(packageName);
} catch (DuplicateItemException e) {
    System.out.println(
        "ERROR - package.setName threw
DuplicateItemException");
    System.exit(1);
}

// add package to service configuration package array
try {
    packageArray.add(pack);
} catch (DuplicateItemException e) {
    System.out.println(
        "ERROR - add package failed : " + e.getMessage());
    System.exit(1);
}

// create new service rule
ServiceRule serviceRule = new ServiceRule(policy);

// add service to rule
try {
    serviceRule.setServiceName(serviceName);
} catch (ItemNotFoundException e) {
    System.out.println(
        "ERROR - service rule set service name failed:"
        + e.getMessage());
    System.exit(1);
}

// get the default rule - it will apply to all the time-based
rules
// since no time-based rules are created
Rule rule = serviceRule.getDefaultRule();

// set rule state to enabled
rule.setState(Rule.RULE_STATE_ENABLED);

// set rule action to be blocked
rule.setBillingReportEnabled(true);

// set other Rule parameters
// . . .

// get package service rule array
ServiceRuleArray serviceRuleArray = pack.getServiceRuleList();

// add service rule to the package's serviceRuleArray
try {
    serviceRuleArray.add(serviceRule);
} catch (DuplicateItemException e) {
    System.out.println(
        "ERROR - adding service rule to the package "
        + "threw DuplicateItemException : "
        + e.getMessage());
    System.exit(1);
} catch (ItemNotFoundException e) {
    System.out.println(
        "ERROR - adding service rule to the package "
        + "threw ItemNotFoundException : "

```

```

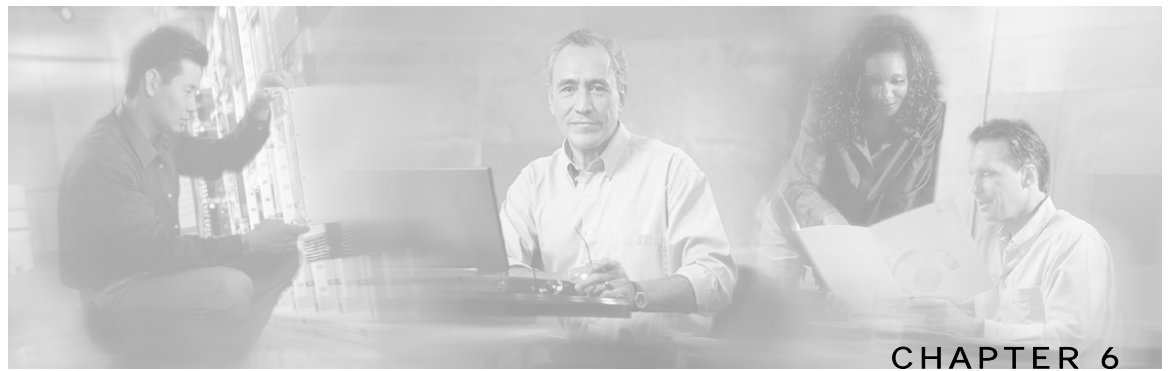
        + e.getMessage());
        System.exit(1);
    }

    // connect to the SCE Box
    String username = "admin";
    String password = "pcube";
    String se_address = "212.47.174.32";
    Connection connection = null;
    try {
        connection =
            SCAS.login(
                se_address,
                username,
                password,
                Connection.SE_DEVICE);
    } catch (ConnectionFailedException e) {
        // login failed - handle exception
    }

    // to apply the service configuration, the parameters are:
    // a connection, the new policy to apply, the SCE to apply to,
    // and a flag stating that the a connection, should be applied
    // (although you might override other applied service
configurations)
    try {
        PolicyAPI.applyPolicy(connection, policy);
    } catch (Exception e) {
        System.out.println("ERROR - first apply failed :" +
e.getMessage());
        System.exit(1);
    } finally {
        SCAS.logout(connection);
    }
    System.exit(0);
}

private Example() {}
}

```

Subscriber Integration

**Note**

The concepts in this chapter are described in more details in the *SCE 1000/SCE 2000 User Guides* and the *SM User Guide*. Please refer to these documents for more information.

This chapter discusses subscriber integration in a SCAS BB application. This chapter describes the available subscriber modes, and when to use each of them.

This chapter contains the following sections:

- [Subscriber Modes](#) 6-1
- [Subscriber-less Mode](#) 6-2
- [Anonymous Subscriber Mode](#) 6-3
- [Static Subscriber-Aware Mode](#) 6-3
- [Dynamic Subscriber-Aware Mode and smartSUB Manager \(SM\)](#) 6-4

Subscriber Modes

The SCAS BB system can operate in any of the following subscriber modes:

- subscriber-less
- anonymous subscriber
- subscriber-aware; in addition, the subscriber-aware mode can be:
 - static
 - dynamic

**Note**

The different subscriber modes are implicit, as there is no global setting controlling the mode in effect. Refer to the system manuals to see how to configure the system to work in the desired mode.

After presenting (in the following table) a summary of the subscriber modes, features, advantages, and situations in which to use each mode, this section discusses each subscriber mode in further detail.

Table 6-1 Subscriber-Mode Summary Table

Mode	Feature Supported	Main Advantages	Situations in Which to Use
Subscriber-less	Global (device-level) analysis and control	Turn-key: No subscriber configuration required Not influenced by the number of subscribers (or inbound IP addresses)	For global control solution or subscriber level analysis. Examples: <ul style="list-style-type: none"> • Controlling P2P uploads at peering points • Limiting total amount of P2P to a specified percent
Anonymous subscriber	Global analysis and control Individual IP address level analysis and control	Turn-key: Need only to define IP ranges possible for subscribers Provides subscriber-level control without integration	For IP level analysis or control that is not differentiated per subscriber, and where offline IP-address/subscriber binding is sufficient. Examples: <ul style="list-style-type: none"> • Limiting per subscriber P2P to 64 Kbps (kilobytes per second) • Identifying top subscribers by identifying top IP addresses and correlating manually/offline with RADIUS/DHCP logs
Subscriber-aware	Full system functionality	Differentiated and dynamic control per subscriber Subscriber-level analysis, regardless of IP address in use Grouping of traffic for control/analysis into statically defined IP ranges	For <ul style="list-style-type: none"> • Controlling/analyzing traffic on a subscriber level. • Monitoring subscriber-usage, regardless of IP addresses • Assigning different service configuration or packages to different subscribers, and changing packages dynamically

Subscriber-less Mode

Subscriber-less mode provides control and link level analysis functions at a global device resolution. In subscriber-less mode, no integration is required, and the total number of subscribers utilizing the monitored link is unlimited from the perspective of the SCE Platform.

SCAS BB is useful also when there are no subscribers at all: The user can view network activity using *traffic discovery*, and perform capacity control using global BW limiters and the package for unknown subscribers.

Anonymous Subscriber Mode

Anonymous subscriber mode provides the means for analyzing and controlling network traffic at a subscriber-inbound IP address. (For example, it is possible to analyze network activity to identify the Top-P2P-IP-addresses, or limit each subscriber's P2P traffic to 64 Kbps.) Anonymous subscriber mode requires no integration or static configuration of the IP addresses used. Instead, ranges of IP addresses are configured (on the SCE Platform), for which the system will dynamically create anonymous subscribers (using the IP address as the subscriber-name).

Use anonymous subscriber mode when no subscriber-differentiated control or subscriber-level quotas tracking is required, and when analysis on an IP level is sufficient.



Note

The total number of concurrently active anonymous subscribers is identical to the total number of concurrently active subscribers, and is therefore subject to a similar license.

Anonymous subscriber mode can support the assignment of different service configuration to different subscribers by ensuring that subscribers are assigned IP addresses from different pools. This uses the SCE Platform's service configuration -template function, through which the system can be instructed to assign a different package to different anonymous subscribers, depending on the range from which their IP address is assigned.



Note

Anonymous subscriber mode assumes that the IP-address recycling time (the time between a subscriber's logging off the network and the subscriber's IP address being reassigned) is sufficiently long. Normally, this is the case in broadband networks (Cable, DSL).

Static Subscriber-Aware Mode

SCAS BB supports a mode of operation in which incoming IP addresses can be statically bound and grouped into subscribers. With this binding, traffic from/to a defined subscriber can be controlled as a group (for example, to limit the P2P traffic to/from that subscriber), in addition to the usage reports that can be provided for that range.

Static subscriber-aware mode supports cases in which the entity using a particular IP address or address-range does not change dynamically. This includes situations such as:

- Deployment in environments where a subscriber's IP address or addresses do not change dynamically via DHCP, RADIUS, etc.
- Deployment in which a group of subscribers that use a common pool of IP addresses (such as all those served by a particular CMTS, BRAS, etc.) are to be managed together (to provide a shared bandwidth to the entire group).

The system supports the definition of static-subscribers, directly on an SCE Platform, and does not require external management software (SM). This is achieved by using the device's CLI, and by defining the list of subscribers, their IP addresses, and associated package (support is provided for interactive configuration, as well as for import/export operations).

Dynamic Subscriber-Aware Mode and smartSUB Manager (SM)

When operating in dynamic subscriber aware mode, the SCE Platform is populated by subscriber information (OSS ID and service configuration) that is dynamically bound to the (IP) address currently in use by the subscribers. In this mode, the smartSUB Manager (SM) must be used for performing device provisioning with the subscriber information. The SM is a server application that maintains the above association, and provisions it to SCE Platforms in real time.

SM General Functions

The SM supports up to 500,000 subscribers and up to 20 SCE Platforms. Cisco also provides a sizing tool to assist in the selection of the correct platform, based on the size and type of the deployment.

The SM is a Java-based server application, and can be installed on supported Solaris platforms (CD-ROM installable version). Configuration and management of the SM is performed using command line utilities (CLU) and configuration files that are installed on the target platform as part of the module's installation.

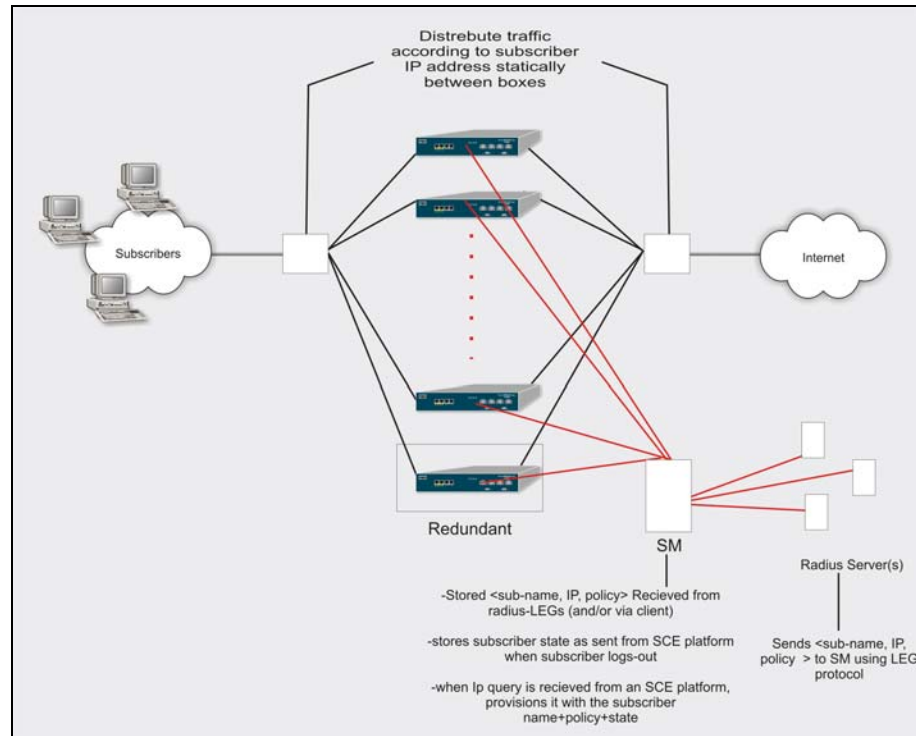
The SM makes use of a third-party database called TimesTen (an embedded, in-memory commercial database), as its high performance back end.

Pull-mode

By making use of SE-SM *pull mode*, the SM can dynamically populate SCE Platforms with subscriber information, when network activity from the subscriber address is detected in a particular device. This is useful for the following:

- More than the supported number (40,000) of subscribers are using the same link, but with no more than 40,000 concurrently active. In this case, the pull-mode functionality is used to cache-in and cache-out subscribers from an SCE Platform, as their activity is detected.
- In topologies in which the actual device through which a subscriber will flow cannot be deduced from the IP-address assignment process in a static fashion. For example, when multiple SCE Platforms are deployed in parallel (using a L3-switch to ensure that the traffic to a single IP address always uses the same path), as in the following diagram.

Figure 6-1: Dynamic Subscriber Aware in Pull Mode



Each time an SCE Platform detects traffic from an IP address for which it does not know the subscriber, it queries the SM for information (note that a platform can be instructed to perform pull operations from specific incoming IP address ranges).

When a subscriber is removed from an SCE Platform, its long-term state (used-quotas) is stored in the SM for future use.

Note that in this topology, the same subscriber can be served by different SCE devices at different points in time (depending on the IP address assigned).

Also, note that a redundant SCE Platform can be set up to take over if one of the primary ones fails (N+1 redundancy). The pull mechanism ensures that the redundant SCE Platform will be populated with the relevant subscribers.

Subscriber State

The SM and its SCE Platform also share subscriber state information. When a subscriber logs out (or is cached out of an SCE Platform), part of the process involves sending its state information (for example, quotas) to the SM for long-term storage. When the subscriber logs in again (or traffic from the subscriber's assigned IP is once again detected), this information is then provisioned to the SCE Platform currently being used (which could be a different device).

Subscriber-Integration: PRPC Protocol

The SM supports a subscriber-integration protocol (PRPC), and additional tools and ready-made components for simplifying subscriber integration in various environments.

The PRPC protocol is used for communicating subscriber information to the SM. Integration toolkits are available for both C/C++ and Java.

The PRPC protocol is used for communicating subscriber information to the SM. Integration toolkits are available for both C/C++ and Java.

Generic SM APIs are provided for Java and C (see the **smartSUB Manager User Guide**). When using these APIs with the SCAS BB, the user has to specify the (SCAS-specific) name for the **packageId** property, which holds the subscriber's package-id.

Following is an example of using the SM Java API with SCAS BB:

```
// subscriber-id
String subscriberName = "JerryS";

// mappings
String[] mappings      = new String[]{ "80.179.153.29" };
short[] mappingTypes  = SMApiConstants.ALL_IP_MAPPINGS;

// properties
String[] propertyKeys  = new String[]{ "packageId" };
String[] propertyValues = new String[]{ "0" };

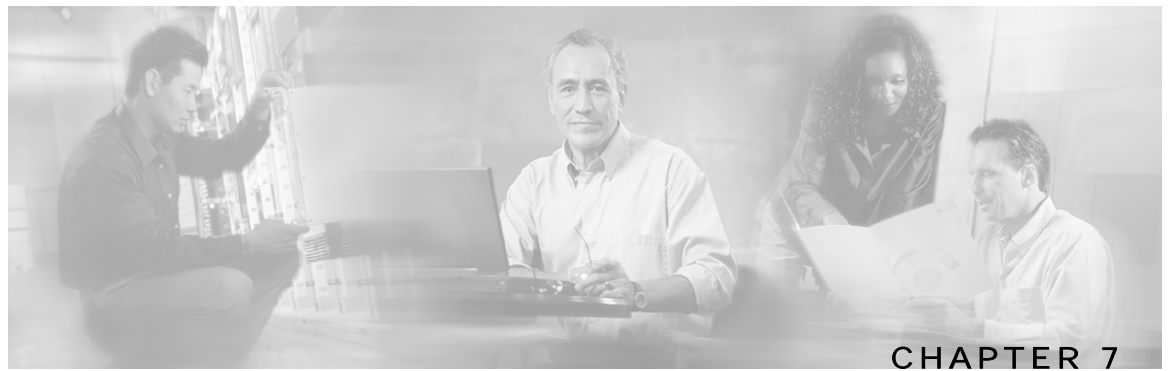
// other settings
String domain          = "subscribers";
boolean isMappingAdditive = false;
int    autoLogoutTime  = -1; // never

// login
smApi.login(subscriberName, mappings, mappingTypes, propertyKeys,
            propertyValues, domain, isMappingAdditive, autoLogoutTime);
```

Subscriber-Integration: CNR (DHCP) Plug-in

To facilitate and simplify subscriber integration in cable environments that use Cisco's Network Registrar DHCP server, Cisco provides an out-of-the-box CNR plug-in. By using the PRPC protocol, this plug-in communicates IP address lease information to the Service Control SM, and synchronizes with the SM the IP addresses assigned to the CPEs by the CNR DHCP server.

CNR versions are supported for both Windows and Solaris platforms.



Quota Provisioning API

This chapter describes the External Quota Provisioning (QP) API.

This chapter contains the following sections:

- [External Quota Provisioning](#) 7-2
- [Quota Provisioning Life Cycle](#) 7-3
- [Limitations](#) 7-4
- [Installing the External Quota Provisioning APIs](#) 7-4
- [QP API \(Java\) Methods](#) 7-5
- [QP API \(Java\) Code Examples](#) 7-8
- [QP API \(C\) Methods](#) 7-9
- [QP API \(C\) Code Examples](#) 7-11
- [Error Codes and Exception Handling](#) 7-14

External Quota Provisioning

External Quota Provisioning is a quota enforcement mechanism to be used by Service Control partners for creating application-aware quota and volume-based services. This mechanism allows subscriber-level quota provisioning by an external entity, and is meant to be used in integration with vendors of subscriber management platforms.

For further description of external quota provisioning, see the *Service Control Application Suite for Broadband User Guide*.

The Quota Provisioning API (QP API) is an extension to Service Control SM API, and comes in both C/C++ and Java versions. It relies on the SM API's ability to connect to an SM, and to add and configure a subscriber. The QP API adds the ability to set quota (`setSubscriberQuota`), to add additional quota (`addSubscriberQuota`), and to read the remaining quota to/from a subscriber's quota-buckets (`getRemainingSubscriberQuota`).

For further description of the SM API, see the *C/C++ API for SM Guide* and the *Java API for SM Guide*.

Using these capabilities, OSS-systems can develop logic to control subscribers usage of traffic according to applications/services using such service models as prepaid, quota-based consumption, etc.

Service Configuration for External Quota Provisioning

There are several guidelines that must be followed when creating the SCAS BB service configuration (PQB) to applied to the system in order for the QP API to be effective:

- **Packages:**
 - The Package Quota Management Mode should be set to "External".
 - When configuring buckets, the appropriate bucket type should be set. Available types are "Volume (L3 KBytes)" or "Number of Sessions".
 - In the usage limits definitions for the appropriate service rules, the appropriate buckets should be selected. Service traffic consumes quota from the selected buckets. The rule's breach handling action can be used to configure the level of service to assign to this traffic while the bucket is depleted.
- **RDRs:** To activate the generation of related RDRs, the following RDRs must be enabled in the RDR Settings:
 - Quota Breach RDR
 - Remaining Quota RDR
 - Quota Threshold RDR

For more information about Packages, Rules and RDR configuration, see the *Service Control Application Suite for Broadband User Guide*.

Quota Bucket States

This section describes the quota buckets states possible while a subscriber is consuming quota from it and while additional quota is provisioned to it through the QP API.

There are three possible bucket states:

- **Above threshold:** While the bucket is **above threshold**, quota is consumed, and remaining quota is reported in Remaining Quota RDRs.
- **Below threshold:** When remaining quota goes **below threshold**, a Quota Threshold RDR is generated. Performing add/set quota operations in response to this RDR may put the bucket above threshold again, before it is depleted. Remaining quota is reported in Remaining Quota RDRs as quota is consumed.
- **Depleted :** The bucket is **depleted** when quota goes below 0. In this case the bucket will maintain quota deficit, or "negative" remaining quota, which is reported in Remaining Quota RDRs. Entering "depleted" state causes the generation of Quota Breach RDR, and the rule's breach handling action, if defined, is applied to this traffic. Performing add/set quota operations at this point may put the bucket out of depleted state.

Note that the quota amount in un-initialized buckets is 0, which means that the first time quota is consumed, the bucket would be depleted.

Quota Provisioning Life Cycle

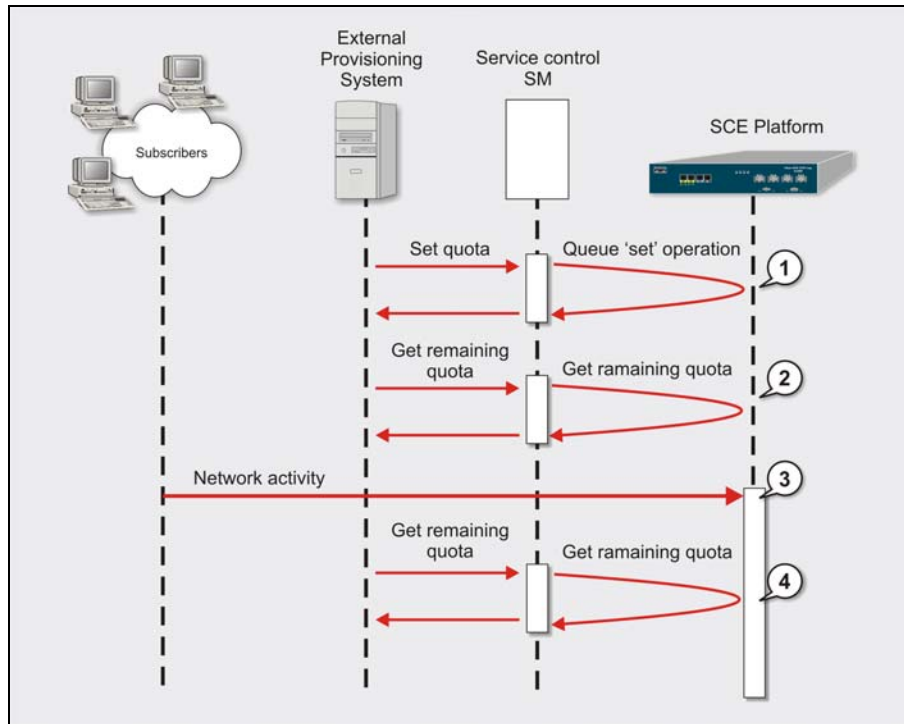
This section describes the life cycle of quota provisioning operations and of a subscriber's quota state, as the subscriber logs in and out, and generates network traffic.

The important thing to understand about quota provisioning operations is that although the relevant QP API methods queue the requested Add or Set operation and return immediately, the actual quota modification takes place *only* when the subscriber generates traffic; that is, when the subscriber performs some network activity. Therefore, during the time from when the external provisioning system requests a quota modification until the time the modification actually takes place, the subscriber's state does not reflect the quota modification. Thus, reading the remaining quota of the subscriber during that in-between time will show values that do *not* take into account the most recent modification.

The following sequence diagram shows an example of how quota modifications take place as a result of quota provisioning operations and subscriber network activity.

- Step 1** When the external quota provisioning system performs a quota modification operation (in this case, a set-quota operation), the modification is queued but not yet handled (1 in diagram).
- Step 2** Therefore, when the external quota provisioning system reads the remaining quota, it still sees the old value before the modification (2 in diagram).
- Step 3** The modification takes place only when the subscriber generates traffic (3 in diagram).
- Step 4** Now, when the external quota provisioning system reads the remaining quota, it sees the updated value (4 in diagram).

Figure 7-1: Quota Provisioning Life Cycle



Limitations

- **Positive quota balance per bucket is limited to 256 GBytes.**

Attempting to grant a subscriber additional quota, exceeding the 256 GBytes limit, will not generate an error, but the resulting quota balance will turn negative! Therefore, as a user you should be aware to the manner you use the add-quota command.

Similarly, subscribers can have a quota deficit of up to 256 GBytes per bucket. When a subscriber is in deficit in one of his buckets that bucket has a negative balance value, which can be as low as -256 GBytes. Beyond this negative value the system stops charging the over-consumed bucket, and the balance will remain -256 GBytes.

- **Issue with 256 successive set-quota operations.**

Similar to the previous bullet, setting quota balance via the set-quota command should not be invoked excessively if a subscriber is logged-off or inactive. To be precise, 256 successive updates of a quota bucket via the set-quota command while the subscriber is inactive will not generate an error but the bucket's balance will not be correct.

Installing the External Quota Provisioning APIs

The C and Java QP APIs are packaged in separate files, whose contents are as follows:

- qp-c-api-dist.tar.gz
 - Documentation

- Include files
- Solaris SO file
- WinNT DLL and LIB files
- qpapi.jar
- qpapi-javadoc.zip

To install, first install the SM API, and then repeat the same installation steps with the QP API.

To compile and run, follow the steps of compiling and running the SM API, and add the QP API files to the *PATH/Class-path* as well. If you are using the Java API include *um_core.jar* in your classpath. The *um_core.jar* is contained within the SCAS BB package.

QP API (Java) Methods

This section lists the methods of the blocking QP API for Java. The signature of each method is followed by a description of its input parameters and its return values.

addSubscriberQuota

Syntax

```
void addSubscriberQuota(String subscriberName,  
                        int[] quota)  
throws QPApiException, ConnectionDownException
```

Description

Adds the specified quota to the current quota in the quota buckets of the specified subscriber.

Parameters

subscriberName: Subscriber ID.

quota: An array of 16 quota values (L3 kilobytes or number of sessions) to be added to the current quota of the specified subscriber's quota buckets.

Return Value

None.

addSubscriberQuota

Syntax

```
void addSubscriberQuota(String subscriberName,  
                        int bucketNum  
                        int[] quota)  
  
throws QPApiException, ConnectionDownException
```

Description

Adds the specified quota to the current quota in the specified quota bucket of the specified subscriber

Parameters

subscriberName: Subscriber ID.

bucketNum: Bucket Number

quota: Quota value (L3 kilobytes or number of sessions) to be added to the current quota of the specified subscriber's quota bucket.

Return Value

None.

getSubscriberQuota

Syntax

```
int[] getSubscriberQuota(String subscriberName)  
  
throws QPApiException, ConnectionDownException
```

Description

Gets the remaining quota in each quota bucket of the specified subscriber.

Parameters

subscriberName: Subscriber ID.

Return Value

An array of the remaining quota values (L3 kilobytes or number of sessions) in each quota-bucket of the specified subscriber.

setSubscriberQuota

Syntax

```
void setSubscriberQuota(String subscriberName,  
                        int[] quota)  
  
throws QPApiException, ConnectionDownException
```

Description

Sets the specified quota in the quota buckets of the specified subscriber.

Parameters

`subscriberName`: Subscriber ID.

`quota`: An array of 16 quota values (L3 kilobytes or number of sessions) to which the specified subscriber's quota buckets should be set.

Return Value

None.

QP API (Java) Code Examples

This section presents the following code examples using the QP API (Java):

```
import java.util.Arrays;
import com.cisco.apps.scas.api.QPApiException;
import com.cisco.apps.scas.api.QPBlockingApi;
import com.cisco.management.framework.client.ConnectionDownException;
/**
 * External Quota Provisioning Example
 **/
public class ExternalQPExample {

    public static final String SM_ADDRESS = "10.1.12.65";

    static public void main(String[] args) throws Exception{

        QPBlockingApi qpBlockingApi = null;
        try{
            //instantiate api and create a connection
            qpBlockingApi = new QPBlockingApi();
            qpBlockingApi.connect(SM_ADDRESS);

            int[] defaultQuota = new int[16];

            //provision each bucket of subscriber "sub1"
            //with 1000 KBytes or 1000 Sessions.
            Arrays.fill(defaultQuota,1000);
            qpBlockingApi.setSubscriberQuota( "sub1",
defaultQuota);

            //dd to bucket 2 of subscriber "sub1"
            //with 500 KBytes or 500 Sessions.
            qpBlockingApi.addSubscriberQuota( "sub1", 0,
500);

            //get "sub1" remaining quota - this method
```

```

        //invocation will work only if the
        //subscriber is logged in.
        //The remaining quota will reflect
        //the last two modifications if
        //subscriber has generated traffic
        int[] remainingQuota =

qpBlockingApi.getRemainingSubscriberQuota("sub1");
        printRemainingQuota( remainingQuota);

        }catch(QPApiException e){
            System.out.println("Error Code is: " +
e.getCode() );
            System.out.println("Error Message is: " +
e.getMessage());
        }catch(ConnectionDownException e){
            System.out.println("Error due to connection
failure: " + e.getMessage());
        }catch( IllegalStateException e ){
            System.out.println("Error due to connection
failure: " + e.getMessage());
        }finally{
            if( qpBlockingApi != null &&
qpBlockingApi.isConnected() )
                qpBlockingApi.disconnect();
        }
        System.exit(0);
    }

    private static void printRemainingQuota(int[] remainingQuota)
    {
        for (int bucketIdx = 0;
            bucketIdx < remainingQuota.length;
            bucketIdx++) {
            System.out.println(
                "bucketIdx="
                + bucketIdx
                + ",remainingQuota="
                +
remainingQuota[bucketIdx]);
        }
    }
}

```

QP API (C) Methods

This section lists the methods of the blocking QP API for C. The signature of each method is followed by a description of its input parameters and its return values.



Note

The QP API C functions have the prefix "QPB_" added to their names. The first parameter in all QP API C functions is `argApiHandle`, which is an API handle created by the calling the `init` function.

addQuota

Syntax

```
ReturnCode* QPB_addQuota (QPB_HANDLE argApiHandle, char* argName, int* argQuotas)
```

Description

Adds the specified quota to the current quota in the quota buckets of the specified subscriber.

Parameters

`argName`: Subscriber ID.

`argQuotas`: An array of 16 quota values (L3 kilobytes or number of sessions) to be added to the current quota of the specified subscriber's quota buckets.

Return Value

A pointer to a `ReturnCode` structure.

getRemainingQuota

Syntax

```
ReturnCode* QPB_getRemainingQuota (QPB_HANDLE argApiHandle, char* argName)
```

Description

Gets the remaining quota in each quota bucket of the specified subscriber.

Parameters

`argName`: Subscriber ID.

Return Value

A pointer to a `ReturnCode` structure holding an integer array of the remaining quota values (L3 kilobytes or number of sessions) in each quota-bucket of the specified subscriber.

setQuota

Syntax

```
ReturnCode* QPB_setQuota (QPB_HANDLE argApiHandle, char* argName, int* argQuotas)
```

Description

Sets the specified quota in the quota buckets of the specified subscriber.

Parameters

`argName`: Subscriber ID.

`argQuotas`: An array of 16 quota values (L3 kilobytes or number of sessions) to which the specified subscriber's quota buckets should be set.

Return Value

A pointer to a `ReturnCode` structure.

QP API (C) Code Examples

This section presents the following code examples using the QP API (C):

- Setting a subscriber's quota, adding quota, and getting the remaining quota.

```

#include <stdio.h>
#include "QpApiBlocking_c.h"

void onExampleDisconnect()
{
    printf("*** DISCONNECTED ***\n");
}

int example(char* argSmAddress)
{
    // init
    printf("initializing\n");
    QPB_HANDLE api;
    api = QPB_init(10,0,20000,10,30);
    if (api == NULL) {
        printf("init failed\n");
        return -1;
    }
    QPB_setName(api, "qp-example");

    // set a disconnect-listener
    QPB_setDisconnectListener(api, onExampleDisconnect);

    // connect
    printf("connecting\n");
    int cnt = 0;
    while (QPB_connect(api, argSmAddress, 14374) == false) {
        if (cnt++ > 10) {
            printf("connect failed, too many reconnects, aborting\n");
            return -1;
        }
    }

    // quota operations

    // prepare a quota bucket array [100, 200, 300,...]
    int quotas[16];
    for (int i = 0; i < 16; ++i) {
        quotas[i] = (i+1)*100;
    }

    // set the quota to subscriber "subs1"
    printf("setting quota\n");
    ReturnCode* rt;
    rt = QPB_setQuota(api, (char*)"subs1", quotas);
    if (isReturnCodeError(rt)) {
        printf((char*)"set-quota failed\n");
        printReturnCode(rt);
        return -1;
    }
    freeReturnCode(rt);

    // add quota to subscriber "subs2"
    printf("adding quota\n");
    rt = QPB_addQuota(api, (char*)"subs2", quotas);
    if (isReturnCodeError(rt)) {
        printf((char*)"add-quota failed\n");
        printReturnCode(rt);
        return -1;
    }
}

```



```
freeReturnCode(rt);

// getting remaining quota of subscriber "subs3"
// this method invocation will work only if the subscriber is logged in.
// the remaining quota will reflect recent modifications if the
// subscriber has generated traffic

printf("getting remaining quota\n");
rt = QPB_getRemainingQuota(api, (char*)"subs3");
printReturnCode(rt);
if (isReturnCodeError(rt)) {
    printf("get-remaining-quota failed\n");
    return -1;
}
freeReturnCode(rt);

// disconnect
if (QPB_disconnect(api) == false) {
    printf("disconnect failed\n");
    return -1;
}
QPB_release(api);

return 0;
}

int main(int argc, char* argv[])
{
    char* smAddress = (char*)"10.1.12.82";
    printf("SM address: %s\n", smAddress);

    if (example(smAddress) < 0) {
        printf("example failed\n");
        return -1;
    }
    return 0;
}
```

Error Codes and Exception Handling

Quota Provisioning API Error Codes

The following table summarizes the list of error codes that the Quota Provisioning API can generate. Note that when using the Quota Provisioning API in conjunction with the SM API, you may also receive error codes from the latter. Please refer to *Appendix A* of the *smartSUB Programmer's Guide* for a full list of those error codes

Table 7-1 Quota Provisioning API Error Codes

Error Code	Description	Recommended Operation / Comments
40,000	Illegal Argument Exception: The value of the quota provided is illegal (for example: too large).	Verify that all quota values passed as arguments are valid
40,002	Subscriber Not Logged in Exception: At attempt to execute an operation on a subscriber that is not logged in has occurred.	Retry after the subscriber is logged into a SCE device.
40,003	Unknown Exception: An unexpected error has occurred.	Consult customer-support.
40,004	Time Out Exception: Happens if the SM database is locked for too long (Internal Error).	Consult customer-support.
40,030	Inactive Subscriber Exception: Happens if a subscriber context can not be located unexpectedly (Internal Error)	Consult customer-support.

Note that not all errors are applicable to all QP API methods.

Managing Exceptions in the Java API

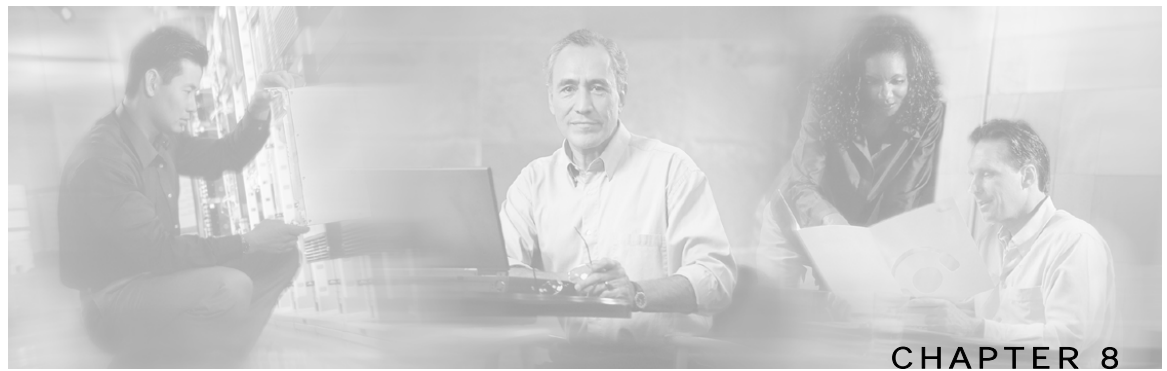
The Java API throws three types of exceptions:

- **ConnectionDownException:** Occurs whenever an established connection to the SM goes down in the middle of an operation.
- **IllegalStateException:** May occur when the connection to the SM fails unexpectedly.
- **QpApiException:** For any other error.

When catching an `QpApiException`, use `qpApiException.getCode()` and compare to the Error Code column in the table above (or from *Appendix A* of the SM documentation). Use `qpApiException.getMessage()` to receive a string error message.

Managing Error Codes in the C/C++ API

Any function call that fails returns an error code. Use that error code and compare to the Error Code column in the table above (or from *Appendix A* of the SM documentation).



Reporter Command Line interface

This chapter describes the various ways of using the Service Control Reporter Command Line Interface for executing the Service Control `reporter` application, including the syntax, switches, and options.

This chapter contains the following sections:

- [Overview of Reporter Command Line Interface](#) 8-1
- [Syntax and Usage](#) 8-1

Overview of Reporter Command Line Interface

The Reporter Command Line Interface is based on a command line application that complements the functionality of the SCAS Reporter GUI. It can be run either as an *executable*, or as a *CGI script*. The Service Control Reporter Command Line Interface provides capabilities and flexibility beyond that of its SCAS Reporter GUI counterpart, and can be integrated into third-party applications to generate usage and statistical based output.

Syntax and Usage

The SCAS Reporter is an application that can be executed in the following ways:

- **Command-Line** - The command-line version accepts input passed as parameters.
- **Command-File** - The command-file version reads its input from a file containing command-line parameters and can be executed as a batch file.
- **Spawned-Executable** - The program is embedded in an application from which it is spawned. The application takes care of dynamically building the string to be passed as a parameter to the SCAS Reporter.

In addition, the SCAS Reporter can be called from:

- **CGI BIN Directory, ISAPI Browser, and NSAPI Browser** - The SCAS Reporter can be called from a CGI BIN directory, or from an ISAPI- or NSAPI-compliant browser. The application returns its output in the form of HTML, which is directed to the browser for viewing. If an error is detected an appropriate error message is displayed.

Command-Line Usage

Following is the syntax of the command line for invoking the SCAS Reporter application:

```
reporter { [-r report-id] | [-n report-name] | [-i report-index] }
         -f [report-format] -k key=value -e [CON]
         -l user-name\password@host-machine
         [drive:][path] <report-filename>
```

Command-Line Syntax

Following are switches in the command line for invoking the SCAS Reporter application:

- Switches `-f`, `-k`, and `-e` are optional.
- One, and only one, of the `-r`, `-n`, or `-i` switches must be specified.
- The other parameters are required.

Command-Line Options

The following table describes the options of the command line for invoking the SCAS Reporter application:

Table 8-1 Command Line Options for SCAS Reporter Application

Option	Action by SCAS Reporter	Comments
<code>-r <report-id></code>	Generates report using report-id.	Use this parameter when generating a report from one of the predefined templates.
<code>-n <report-name></code>	Generates report using report-name.	Use this parameter when generating a report from one of the personalized templates. The name is matched against the saved report names. Report names are case insensitive.
<code>-i <report-index></code>	Generates report using report-index.	Use this parameter when generating a report from one of the personalized templates.
<code>-f [report-format]</code>	Generates report as a chart or table in the specified report-format. If report-format is omitted, the default format is used. Default formats are noted by an asterisk.	
	Table Format:	
	JPG	
	GIF *	
	HTM	
	Chart Format:	
	CSV*	
	HTM	
	XLS	

Option	Action by SCAS Reporter	Comments
-k key=value	Override predefined key with value specified.	
-l user-name\password@hostname	(Required) To log in using the user-name\password@hostname machine.	Use the user-name and password of the workstation hosting the reporter executable.
<report-filename>	(Required) Name of file to which to direct output.	If the filename already exists, the original file is overwritten.
-e [CON]	Redirects output (messages and errors) to an error message file. If the optional CON switch used is a console window and is open, error messages are redirected there.	Since reporter is a Win32 application, stderr is redirected to an error message file.

Command-File Usage

Following is the command for invoking the SCAS Reporter application from a command file:

```
reporter@[drive:][path][command-file]
```

Command-File Syntax

Following is the syntax for invoking the SCAS Reporter application from a command file:

- The executable name is followed by an “at” sign (@), an optional drive and path, and a required name of a command file to be used for input.
- Each line in the command file contains a different command for generating a report.
- Lines beginning with a semicolon (;) are regarded as comment lines and ignored.
- Backslashes (\) can be used to split long lines. Place the backslash at the end of each line to be concatenated, except the last line.



Glossary of Terms

A

Anonymous subscriber mode

A mode of the solution in which the system monitors traffic and assigns service configuration automatically based on to the individual's IP address used on its subscriber-side. This mode can be used to control a subscriber's traffic anonymously, without integrating the system with an OSS system. In this mode, the subscribers defined in the system are anonymous and are distinguished only by their IP address or VLAN ID.

C

Collection Manager (CM)

A software application that is responsible for collecting RDRs from the SCE Platforms, processing them, and preparing them for reports.

Command Line Interface (CLI)

One of the management interfaces to the SCE Platform. It is accessed through a Telnet session or directly via the console port on the front panel of the SCE Platform.

D

Downstream traffic

Traffic entering the SCE Platform from the network side (that is, toward the subscribers).

Dynamic Signature

A dynamic signature is a signature that can be loaded to a running application, and once loaded the application knows to identify the protocol associated with this signature.

Dynamic Subscriber-aware mode

A mode in which the actual subscriber ID is associated with an IP address when the subscriber logs onto the network and is assigned an IP address. To operate in this mode, the system must be integrated with the OSS system that assigns IP addresses to subscribers (typically based on RADIUS or DHCP).

F

Filter Rules

The part of the Service Configuration that lets you direct the SCE Platform to ignore some types of transactions based on Layer 3 and Layer 4 properties, and transmit them unchanged, bypassing the solution service.

G

Global Controllers

Global Controllers are used for controlling the total bandwidth percentage for a selected protocol or package for all subscribers. See also Subscriber BW Controllers.

I**Inline connection mode**

The SCE Platform physically resides on the data link between the subscriber side and the network side, and can both receive and transmit traffic, permitting traffic control as well as monitoring.

L**List**

An IP address range or list of web addresses used to define a service.

M**Monitoring Reports**

Bandwidth, volume and session usage reports generated by the SCAS Reporter at subscriber, package, and global granularity.

N**Network-initiated transactions**

Transactions that were initiated by a host, on the network side, toward a subscriber.

P**Package**

A collection of business policy rules, defining access levels to various services, charging parameters, and traffic control actions to be taken upon certain events. Subscribers are assigned packages (plans) that determine how their network transactions are controlled and charged.

PQI (Service Control Installation) File

An application package file that is installed on the SCE Platform or associated software modules.

Q**Quota**

A (subscriber's) limit for a specific metric, such as bandwidth or volume.

R**RDR (Raw Data Record)**

A data record produced by the SCE Platform that reports on events in the traffic. RDRs produced by the SCE Platform are sent to the Collection Manager and then stored in the Collection Manager database or forwarded to third-party systems. The RDR typically contains quota (see Quota) requests or reports service usage.

Real-time subscriber usage monitoring

Subscribers which are monitored in detail and usage information is frequently reported by the SCE device to facilitate detailed reports.

Receive-only connection mode

The SCE Platform does not reside physically on the data link, and therefore can only receive data and not transmit.

This mode has traffic monitoring capability only.

S**SCAS BB Console**

The user interface used for controlling the SCAS system, used to create, modify, and apply the service configuration.

SCE (Service Engine) Platform

The Service Control purpose-built network element for service control. This hardware device is capable of performing deep packet analysis at wire speed, and control subscribers' traffic based on business policy.

Service

A value-added offering given by the service provider to its subscribers on top of its access network.

For each such commercial service the providers offer to their subscribers, a corresponding service is defined in the Encharge solution for classifying and identifying network transaction associated with the service, reporting on its usage, and controlling its traffic according to the business policy.

Service Configuration

The definition of services within the Encharge solution, the mapping of network transactions to their corresponding services, and the behavior of the SCE Platform on them. The service configuration includes the definition of Services, Packages, Bandwidth Controllers, Filter Rules, etc.

Service Control Application

An SML program that determines how the SCE Platform operates.

Service Rule

A Service is assigned to a Package by defining a Service Rule for the Package.

Session (also called Transaction)

An instance of communication between network hosts. A precise definition of a session is application protocol (Layer 7) dependent.

Signature

A set of parameters that uniquely identify a protocol.

SLI (SML Loadable Image) File

An SLI file is a software package (part of an SCAS solution) that contains the SML application that is loaded onto a SCE Platform. The SML application determines the behavior of the SCE Platform. Different SCE Platforms can have different SML applications, even when they are within the same POP. (Operators do not need to access the SLI file.)

smartSUB Manager (SM)

A middleware software component used in cases where dynamic binding of subscriber information and service configurations is required. The SM manages subscriber information and provisions it in real time to multiple SCE Platforms. The SM can store subscriber service configurations information internally, and act as a state-full bridge between the AAA system (for example, RADIUS and DHCP) and the SCE Platforms.

Static Subscriber-aware mode

A mode in which a specific IP address is bound to each subscriber. This mode is useful when controlling enterprise customers, or when controlling subscribers in groups of predefined subnets (such as users of a specific CMTS/BRAS).

Subscriber

The generic term used to refer to the managed entity for which a service configuration is enforced, and usage is monitored, by an SCAS solution. A subscriber can be defined as an individual IP address, or ranges of IP addresses or VLANs. The system supports different modes of operations including: subscriber-less mode (all control is performed globally), anonymous subscribers mode, and dynamic and static subscriber-aware modes.

**Subscriber BW Controllers
(Bandwidth Controllers)**

Subscriber Bandwidth Controllers (BW Controllers) controls traffic bandwidth for an individual subscriber. See also *Global Controllers* (on page 4-8).

Subscriber-initiated transactions

Transactions that are initiated by a host of a subscriber.

Subscriber-less mode

A mode of the solution that requires no integration, so that the SM component is not required. This mode is not influenced by the number of subscribers or inbound IP addresses, therefore the total amount of subscribers utilizing the monitored link is unlimited from the perspective of the SCE Platform. It is the choice for sites where control and level analysis functions are required only at a global device resolution.

T**Time Based Rule**

An added-value Service Rule that can be attached to either a Total Traffic Rule or to a Service Rule. It is listed as a sub-rule in the Service Rule table. A time based rule is applied for one of the user-defined Weekly Time Frames.

Traffic-Discovery Reports

Statistics reports on network activity based on transaction usage records.

Transaction (also called Session)

An event in traffic that is recognized by the application and is distinguished according to its L3, L4, or L7 characteristics. Different protocols may have different transaction types.

U**Upstream traffic**

Traffic entering the SCE Platform from the subscriber side.



Index

A

- Adding a Service to Service Configuration • 5-13
- Adding Elements to a List Array • 5-8
- Adding Ports to Protocols • 5-10
- addQuota • 7-10
- addSubscriberQuota • 7-5, 7-6
- Aggregation • 5-18
- Anonymous subscriber mode • 2-4, 1
- Anonymous Subscriber Mode • 6-3
- Audience • v

B

- Bandwidth Controller • 5-19
- Block and Redirect • 5-21
- Breach Reports • 5-19
- Breaches • 5-16

C

- Cisco TAC Website • viii
- Collection Manager • 3-4
- Collection Manager (CM) • 1
- Command Line Interface (CLI) • 1
- Command-File Syntax • 8-3
- Command-File Usage • 8-3
- Command-Line Options • 8-2
- Command-Line Syntax • 8-2
- Command-Line Usage • 8-2
- Connecting to the SCE Platform • 5-3
- Creating a Service • 5-12
- Creating and Naming Packages • 5-16

D

- Data Collector • 3-4
- Default and Non-Default Time Frames • 5-20
- Defining Protocols • 5-9

Defining Service Elements • 5-13

Defining Service Rules • 5-16

Description • 7-5, 7-6, 7-7, 7-10

Determining the Type of a List • 5-8

Document

content • vii

Document Content • vii

Document Conventions • vii

Downstream traffic • 1

Dynamic Signature • 1

Dynamic Signatures • 4-5

Dynamic Subscriber-aware mode • 1

Dynamic Subscriber-Aware Mode and smartSUB Manager (SM) • 6-4

E

- Error Codes and Exception Handling • 7-14
- Essential Components • 3-2
- Example - Adding a Service and Applying a Service Configuration • 5-22
- Example - FTP Service Rule • 5-17
- External Quota Provisioning • 7-2

F

- Filter Rules • 1
- Flat Files • 3-6
- Flow of Information • 3-7

G

- getRemainingQuota • 7-10
- getSubscriberQuota • 7-6
- Global Controllers • 4-8, 1

I

- Importing, Exporting, and Creating Service Configurations • 5-6
- Including the SCAS Libraries • 5-3
- Initiating Side • 4-5

Inline connection mode • 2
 Installing the External Quota Provisioning APIs • 7-4
 Integration Factors and Motivation • 3-1
 Integration Points • 3-5
 Introduction • v

J

JAR files • 5-3

L

Limitations • 7-4
 List • 2
 Lists • 4-6, 5-7
 adding elements to • 5-8
 determining type of • 5-8
 examples of • 4-3
 navigating • 5-8
 retrieving • 3-7
 Logical Entities • 4-2

M

Managing Error Codes in the C/C++ API • 7-15
 Managing Exceptions in the Java API • 7-14
 Managing Service Configurations with SCAS Service Configuration API • 5-4
 Monitoring Reports • 2

N

Navigating a List Array • 5-8
 Network-initiated transactions • 2

O

Obtaining Technical Assistance • viii
 Opening a TAC Case • viii
 Overview • 1-1
 Overview of Reporter Command Line Interface • 8-1
 Overview of Service Configuration API • 5-1

P

Package • 2
 Packages • 4-7, 5-13
 examples of • 4-3
 Parameters • 7-5, 7-6, 7-7, 7-10, 7-11
 PQI (Service Control Installation) File • 2
 Preface • v
 Protocols • 4-4, 5-8

 defining • 5-13
 main aspects of • 4-4

Pull-mode • 6-4
 Purpose • vi

Q

QP API (C) Code Examples • 7-11
 QP API (C) Methods • 7-9
 QP API (Java) Code Examples • 7-8
 QP API (Java) Methods • 7-5
 Quota • 2
 Quota Bucket States • 7-3
 Quota Provisioning API • 7-1
 Quota Provisioning API Error Codes • 7-14
 Quota Provisioning Life Cycle • 7-3

R

RDR (Raw Data Record) • 2
 RDRs
 triggered by breach • 5-16, 5-19
 Real-time subscriber usage monitoring • 2
 Receive-only connection mode • 2
 Related Publications • viii
 Reporter application
 command file syntax • 8-3
 command file usage • 8-3
 command line options • 8-2
 command line syntax • 8-2
 command line usage • 8-2
 syntax and usage • 8-1
 Reporter Command Line interface • 8-1
 Retrieving a List Array • 5-7
 Retrieving and Applying Service Configurations • 5-5
 Return Value • 7-5, 7-6, 7-7, 7-10, 7-11
 Rules • 4-6

S

SCAS API Base Classes • 5-2
 SCAS BB Licenses • 3-4
 SCAS BB Service Configuration APIs • 2-7
 SCAS BB Console • 2-6, 2
 SCAS Client/Server Connectivity • 5-2
 SCAS Reporter Command Line Interface • 3-6
 SCE (Service Engine) Platform • 2
 SCE Platform • 3-3
 SCE Platforms • 2-1, 3-3
 triggering RDRs • 3-7
 Service • 3

- Service and Service Elements • 5-11
 - Service Configuration • 2-5, 4-2, 3
 - applying • 3-7
 - components of • 4-3
 - creating • 5-12
 - Global Controllers • 4-8
 - retrieving • 3-7
 - Service Configuration API • 3-6, 5-1
 - programming steps for • 5-1
 - Service Configuration Entities • 4-1
 - Service Configuration for External Quota Provisioning • 7-2
 - Service Configuration Utility • 2-6
 - Service Configurations • 4-2
 - Service Control Application • 3
 - Service Control Application Suite for Broadband
 - components • 2-1
 - console • 2-6
 - functionality • 3-2
 - logical entities • 4-2
 - reporterSee Reporter... • 3-7
 - solution • 2-1
 - Service Control Capabilities • 1-2
 - Service elements
 - components of • 4-3
 - defining • 5-13
 - Service Rule • 3
 - Services
 - creating • 5-12
 - examples of • 4-3
 - Services and Service Elements • 4-3
 - Session (also called Transaction) • 3
 - setQuota • 7-10
 - setSubscriberQuota • 7-7
 - Signature • 3
 - SLI (SML Loadable Image) File • 3
 - SM • 3-3
 - general functions • 6-4
 - SM General Functions • 6-4
 - smartSUB Manager • 3-3
 - smartSUB Manager (SM) • 3
 - Static Subscriber Mode • 2-4
 - Static Subscriber-aware mode • 3
 - Static Subscriber-Aware Mode • 6-3
 - Subscriber • 3
 - Subscriber BW Controllers • 4-8
 - Subscriber BW Controllers (Bandwidth Controllers) • 4
 - Subscriber integration
 - CNR (DHCP) plug-in • 6-6
 - PRPC protocol • 6-6
 - Subscriber Integration • 6-1
 - Subscriber modes • 6-1
 - summary • 2-5, 6-1
 - Subscriber Modes • 6-1
 - Subscriber Modes – Summary • 2-5
 - Subscriber Quota Buckets • 4-9
 - Subscriber State • 6-5
 - Subscriber-aware mode – Dynamic Subscribers • 2-4
 - Subscriber-initiated transactions • 4
 - Subscriber-Integration
 - CNR (DHCP) Plug-in • 6-6
 - PRPC Protocol • 6-6
 - Subscriber-less mode • 2-3, 4
 - Subscriber-less Mode • 6-2
 - Subscribers and Subscriber Modes • 2-3
 - Syntax • 7-5, 7-6, 7-7, 7-10
 - Syntax and Usage • 8-1
 - System
 - components • 2-1
 - System Architecture for Developers • 3-1
 - System Components • 2-1
- T**
- TAC Case Priority Definitions • ix
 - The Cisco Service Control Concept • 1-2
 - The SCE Platform • 1-3
 - The Service Control Solution • 2-1
 - Time Based Rule • 4
 - Traffic-Discovery Reports • 4
 - Transaction (also called Session) • 4
- U**
- Upstream traffic • 4