CHAPTER 5

Service Configuration API

この章では、SCAS Service Configuration API を使用して SCAS BB のサービス設定作業を自動的に行う方法を説明します。この章の目的は次のとおりです。

- SCAS Service Configuration API の主な要素を紹介します。
- SCAS Service Configuration API の使用法、サービス コンフィギュレーションの作成、編集、メンテナンス、管理、および配布機能について説明します。
- SCAS Service Configuration API の使用法を示すコード例を示します。

この章の内容は次のとおりです。

Service Configuration API の概要

*サービス コンフィギュレーション*は、プロバイダーのネットワーク上で Service Control Application Suite for Broadband システムによって実施されるプロバイダーのビジネス ルールを表します。*Service Configuration API* は、*サービス コンフィギュレーション*のプログラミング実装を担うプログラミング エンドです。

サービス コンフィギュレーションを取り扱う作業に Service Configuration API を使用します。SCAS BB Console は、Service Configuration API のプログラミング ツールで作成されています。この API を使用すると、GUI では手作業で行うことになる定常的なサービス設定作業を、自動的に行うことができます。

サービス コンフィギュレーションの定義から SCE プラットフォームでコンフィギュレーションをアクティブ化するまでの、Service Configuration API による主なプログラミング ステップは次のとおりです。

ステップ 1　新しいサービス コンフィギュレーションを定義するか、または SCE から既存のサービス コンフィギュレーションを取得します。

ステップ 2　サービスを定義して、サービス名を割り当てます。

ステップ 3　プロトコルおよびリストを定義します。

ステップ 4　サービス エレメントを定義し、それらのエレメントにプロトコル、方向、およびリスト（オプション）を割り当てます。

ステップ 5　サービス エレメントをサービスに追加します。

ステップ 6　パッケージを定義して、パッケージ名を割り当てます。

ステップ 7　パッケージのサービス ルール（違反前および違反後における帯域幅の制限、サービスのイネーブル化またはディセーブル化など）を定義します。

ステップ 8　サービス コンフィギュレーションを SCE に伝播してアクティブ化します。

SCAS Service Configuration API を使用して行う主な作業は、サービス コンフィギュレーションの作成および処理です。これには、オフラインで行うことができるサービスとルールの定義が含まれます。サービス コンフィギュレーションが完成すると、SCE プラットフォームにそのサービス コンフィギュレーションを伝播できます。

上記の各ステップについて、以下の各項で詳しく説明します。この章を『*Subscriber Management API Reference Manual*』と併せて参照することで、SCAS BB サービス コンフィギュレーションを作成、編集、管理、および配布できるようになります。

SCAS API のベース クラス

アプリケーションの作成および取り扱いに使用する、主な SCAS API クラスを次の表に示します。

表 5-1　主な SCAS API クラス

クラス	目的
EngageAPI	クライアント プログラムがシステムのハードウェアおよびソフトウェアに接続できるようにします。ネットワーク接続機能を実行する、エレメント管理作業の出発点です。
SCAS PolicyAPI	サービス コンフィギュレーションを管理します。

SCAS クライアント / サーバの接続

SCAS Java プログラムなどのクライアント アプリケーションが、SCE プラットフォームに接続できるようにするための SCAS API Java クラスは、`Engage` です。

- SCAS ライブラリの組み込み
- SCE プラットフォームへの接続

SCAS ライブラリの組み込み

SCAS Java API クラスを取り扱うには、Java JDK バージョン 1.3 以降が必要です。

次の JAR ファイルが Java class-path に含まれている必要があります。

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

これらのファイルは、SCAS クライアントのインストレーション プロセスの一環としてワークステーションにインストールされます。このインストレーションは、SCAS BB 製品に付属する SCAS BB *クライアント インストレーション CD-ROM* を使用して行います。

SCAS クライアントを `C:\Program Files\Cisco\SCAS BB x.x.x\` フォルダにインストールした場合、これらの JAR ファイルはすべて `C:\Program Files\Cisco\SCAS BB x.x.x\lib` フォルダにあります。

（注）　上記の JAR ファイルのバージョンは、SCAS BB ソリューションの他のコンポーネントのバージョンと統一されている必要があります。ソリューションの新しいバージョンをインストールまたはアップグレードした場合には、API 開発者向け JAR ファイルも必ずアップグレードしてください。

SCAS Java API を使用するには、事前に SCAS Java パッケージをインポートする必要があります。プログラムの先頭に次のコード行を追加して、インポートします。

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

SCE プラットフォームへの接続

SCE プラットフォームからサービス コンフィギュレーションを取得したり、SCE プラットフォームにサービス コンフィギュレーションを適用したりするには、プラットフォームに接続する必要があります。この接続を可能にする SCAS クラスが、通信およびネットワーク アクティビティの調整を行います。

このクラスの `login` メソッドは、永続的な接続を提供します。このメソッドでは、ユーザ名、パスワード、および接続する SCE のホスト名が必要です。メソッドのいずれかのパラメータが不正な場合、例外が発生します。サービス コンフィギュレーションの取得または適用が終了した時点で、接続を閉じる必要があります。

（注）　**作業の終了時に、必ず `logout` メソッドをコールしてください。SCE への接続をオープンのままにしないでください。また、取得や適用のたびに新しい接続を作成するのではなく、Connection オブジェクトをできるだけ再利用してください。**

`login` メソッドは、永続的な `login` 接続を確立した後で行う *Service Configuration* API コールのための基本的なパラメータ値を確立します。このメソッドは、`Connection` タイプのオブジェクトを返します。このオブジェクトは、*Service Configuration* API の大部分で使用される SCE のハンドルです。

次に、SCE との接続を確立する Java コードを示します。

```
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
}
```

SCE との接続は、終了しないかぎり開いた状態が継続します。SCE との接続を終了するには、次のように `logout` メソッドを使用します。

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

SCAS *Service Configuration* API はサービス コンフィギュレーションのさまざまな編集タスクを実行しますが、サービス コンフィギュレーションを編集している間は SCE に接続する必要はありません。つまり、サービス コンフィギュレーションはオフラインで準備することができます。サービス コンフィギュレーションの編集が完了したら、SCE に接続してサービス コンフィギュレーションを伝播できます。

SCAS Service Configuration API によるサービス コンフィギュレーションの管理

前項では、SCAS API を使用して SCE に接続（ログイン）および接続を終了（ログアウト）する方法を説明しました。そのほかに実行できる操作としては、サービス コンフィギュレーションの新規作成、サービス コンフィギュレーションの適用などがあります。適用とは、SCE プラットフォームにサービス コンフィギュレーションを伝播して、ネットワーク上でアクティブ化できるようにすることです。

Java の `PolicyAPI` クラスが、SCAS Service Configuration API のサービスの出発点となります。

以下の各項で、パッケージ、サービス、サービス エレメント、プロトコル、リスト、およびルールに対して実行できるさまざまな操作の方法を説明し、擬似コードの例を示します。サービス コンフィギュレーションの一連のコンポーネントは、最終的にパッケージにアセンブルして SCE に伝播します。ここで説明する内容は次のとおりです。

- サービス コンフィギュレーションの取得および適用
- サービス コンフィギュレーションのインポート、エクスポート、および作成
- リスト
- プロトコル
- サービスおよびサービス エレメント
- パッケージ

サービス コンフィギュレーションの取得および適用

プロバイダーのネットワークで、サービス コンフィギュレーションに変更を加える場合には、現在使用しているサービス コンフィギュレーションを変更するのが一般的です。変更を行うには、SCE からサービス コンフィギュレーションを *取得* して、編集作業を行う必要があります。*サービス コンフィギュレーション* の編集はオフラインで行い、SCE に伝播できる状態にします。*サービス コンフィギュレーション* を伝播することを、サービス コンフィギュレーションの *適用* という場合もあります。これは、サービス コンフィギュレーションの元の内容を上書きして、SCE プラットフォーム上で新しいビジネス ルールをアクティブ化することを意味します。

サービス コンフィギュレーションの取得および SCE への適用を行うための、2 つの主なサービス コンフィギュレーション メソッドである `retrievePolicy` および `applyPolicy` を、次のコード例で示します。

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

`retrievePolicy()` メンバー関数の 2 番めのパラメータ *SCE* には、1 台または複数の Service Control プラットフォームを定義できます。`SCAS.Policy` を取得したあと、サービス コンフィギュレーションの内容の検査、変更、保存など、さまざまな機能を実行できます。

SCAS アプリケーションを開発するとき、いくつかの便利な機能を利用して SCAS Service Configuration API アプリケーションを確認できます。たとえば、Java アプリケーションでサービス コンフィギュレーションを変更する場合、`savePolicy` メソッドを使用して、サービス コンフィギュレーションの内容をファイルに保存できます。その後、SCAS Service Configuration Manager GUI アプリケーションを使用してこのファイルを開き、アプリケーションで変更したサービス コンフィギュレーション エレメントを検証できます。

SCAS サービス コンフィギュレーションを適用するには、applyPolicy メソッドをコールします。

サービス コンフィギュレーションのインポート、エクスポート、および作成

SCE プラットフォームから既存のサービス コンフィギュレーションを取得するだけでなく、テキスト ファイルを通じてサービス コンフィギュレーションを取得することもできます。次に、サービス コンフィギュレーションをインポートする例を示します。

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

サービス コンフィギュレーションのエクスポートは、アーカイブの目的で、またはサービス コンフィギュレーションを後で編集する場合に行います。次に、サービス コンフィギュレーションをエクスポートする例を示します。

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

一般的には既存のサービス コンフィギュレーションを修正して新しいサービス コンフィギュレーションを作成しますが、必要に応じてサービス コンフィギュレーションを最初から新規作成することもできます。Policy クラスのコンストラクタで、new 演算子を使用してサービス コンフィギュレーションを作成します。次に、このようなサービス コンフィギュレーションの作成例を示します。

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

リスト

ここでは、リストの使用例を示します。有害な内容の Web サイトのリストを作成し、既存のサービス コンフィギュレーションのリスト アレイに追加します。この例で作成するリスト名は *Offensive Content Website List* であり、リスト要素を追加しています。最後にこの新しいリストを、サービス コンフィギュレーションのリスト アレイに追加します。後の段階で、このリストに含まれるホスト名に関しては *HTTP Browsing Protocol* でサービスをアクセスできないようにするルールを実施できます。

Hostlist クラスのコンストラクタでは、パラメータ リスト name およびオプションの description を指定します。リストに新しい要素を追加するには、HostList クラスの add メソッドを使用します。次のように、getListArray メソッドを使用して listArray を取得し、新しく作成したリスト offensiveList をリスト アレイに追加します。

```
. . .
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
}
```

上記のコード例で使用されている各要素について、次の各項で説明します。内容は次のとおりです。

- リスト アレイの取得
- リスト アレイのナビゲート
- リストのタイプの判別
- リスト アレイへの要素の追加

リスト アレイの取得

次の行で、リスト アレイを取得しています。

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

getListArray メソッドにより、タイプ ListArray のリスト アレイを取得します。リスト アレイを取得したあと、リスト サイズを調べたり、リストの個々の要素を取得したりすることができます。

リスト アレイのナビゲート

次の行で、listArray の要素を反復的に取得しています。

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

上記のループによって、listArray の各要素が返されます。getSize メソッドは、ListArray のサイズを返します。リストの個々の要素の getElementAt メソッドにより、Object が割り当てられます。

リストのタイプの判別

getElementAt メソッドによって返されるオブジェクトは、HostList クラスまたは IPRangeList クラスのインスタンスである可能性があります。IPRangeList は、IP 範囲のリストです。次に、このメソッドを使用してリストのタイプを判別する例を示します。

```
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");

}
```

リスト アレイへの要素の追加

次に、リストに新しい要素を追加する例を示します。

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

リストのグループの最初のリストを取得したと仮定すると、*www.cnn.com*（http://www.cnn.com）がリストに追加されます。

プロトコル

SCE プラットフォームは、サービスに対応付けられたネットワーク トランザクションのプロトコルに反応します。サービス コンフィギュレーションにはサービスのリストが含まれます。サービスのプロトコルを作成する方法としては、プロトコルを新しく定義する方法と、既存のプロトコルを拡張してポートを追加する方法の 2 通りがあります。また、プロトコルを定義するダイナミック シグニチャ スクリプトをインポートまたは削除することもできます。以下の各項では、次の事項について説明します。

- プロトコルの定義
- プロトコルへのポートの追加
- ダイナミック シグニチャ スクリプト

プロトコルの定義

プロトコルの認識は、一般的なネットワーキング慣例に基づいて行われます。たとえば、サーバがポート 666 でトランザクションを待ち受ける場合、ポート 666 はタイプ DOOM のプロトコルを待ち受けるのが慣例となっています。提供するサービスに関して慣例に従うと、双方にとって便利です。

次に、quake という名前のプロトコルを、ポート 26000、トランスポート タイプ TCP で定義する例を示します。

```
// 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
}
```

上記の例では、myPolicy という名前のサービス コンフィギュレーションの参照が存在し、Protocol クラスのコンストラクタのパラメータとして渡されていることを前提としています。このコンストラクタによって返される quakeProtocol の参照を使用して、プロトコルの名前を設定します。

次の行では、プロトコルのポート番号 26000、トランスポート タイプ TCP を指定しています。

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

次の行では、このプロトコルをサービス コンフィギュレーションのプロトコル リストに追加しています。

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

以上でプロトコルの定義は終了です。次項では、既存のプロトコルにポートを追加する方法を説明します。

プロトコルへのポートの追加

使用されていない*ポート*を、既存のプロトコルに*追加*できます。基本的にそのポートはそのプロトコルに対応付けられ、プロトコルの範囲が拡張されます。

次に、サービス コンフィギュレーションのプロトコル リストを取得し、HTTP Browsing という名前のシステム定義プロトコルを検索する例を示します。このプロトコルを見つけたあと、ポートを追加してプロトコル定義を拡張します。

```
    . . .
// 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
}
    . . .
```

この例では、HTTP Browsing という名前のプロトコルが存在することを前提としています。次の行で、Protocol クラスの getProtocol メソッドにより、プロトコル アレイを表すハンドルが返されます。

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

次の行で、特定のプロトコルを名前によって検索します。

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

クラス Protocol の add メソッドでは、ポート番号およびトランスポート タイプを指定する必要があります。上記の例では、ポート 8082 をトランスポート タイプ TCP で、既存のサービス HTTP Browsing に追加しています。次の行で、プロトコルにポートを追加します。

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

サービス HTTP Browsing が拡張されました。ポート 8082 で配信される HTTP トランザクションをサービス ルールによって処理するには、この拡張が必要です。

サービスおよびサービス エレメント

次に、2 つのサービス エレメントのあるサービスを作成する例を示します。この例は、1 つの *サービス*に複数の *サービス エレメント*の対応付けが可能であることを示しています。

作成するサービス エレメントの 1 つは Doom、もう 1 つは Quake というゲームです。どちらも有名なゲームで、これらを Gaming Service というサービスに追加します。

```
. . .
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
}

. . .
```

上記のコード例で使用されている各要素について、次の各項で説明します。

- サービスの作成
- サービス エレメントの定義
- サービス コンフィギュレーションへのサービスの追加

サービスの作成

次の行で、Service を含むクラスをインスタンシエートしています。

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

myPolicy という名前の Policy 参照が、Service コンストラクタのパラメータとして渡されていることを前提としています。

次の行で、サービス名を定義しています。

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

サービス エレメントの定義

次の行で、新しいサービス エレメントを作成しています。

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

このサービス エレメントは、サブスクライバ起動による doom プロトコルのすべてのネットワーク トランザクションに適用されます。ただし、特定のネットワーク アドレスに対して実行されたネットワーク トランザクションに限り、このサービス エレメントを適用するように指定することができます。それには、次のようにサーバ アドレスのリストをサービス エレメント付きで指定します。

```
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
}
```

サービス コンフィギュレーションへのサービスの追加

最後のステップとして、サービス コンフィギュレーションにサービスを追加します。次の行で、サービス コンフィギュレーションに gamingService を追加しています。

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

パッケージ

（注）　パッケージの定義は、ライセンスに依存します。ライセンスの容量制限により、そのライセンスで規定されているパッケージ数より多く定義することはできません。

ルールおよびパッケージの定義を始める前に、前述の手順に従って、サービスを定義し、サービス コンフィギュレーションにサービスを追加する必要があります。

次に、パッケージおよびルールを定義し、これらを使用するためのコード例を示します。1 つのサービス コンフィギュレーションで、Family Package および Bachelor Package という 2 つのパッケージを作成しています。

この例では、Parental Watch List を含む Parental Watch Service を作成済みであることを前提としています。Family Package では、Parental Watch Service をイネーブルにします。このパッケージに加入するサブスクライバは、Parental Watch List で定義されている検閲済みのコンテンツにアクセスできません。Bachelor Package では、Parental Watch Service をイネーブルにしないので、これらのサイトへのアクセスが認められます。

```
/* 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
}
```

上記のコード例で使用されている各要素について、次の各項で説明します。内容は次のとおりです。

- パッケージの作成およびパッケージ名の設定
- サービス ルールの定義
- 違反
- 例 — FTP サービス ルール
- アグリゲーション
- 帯域幅コントローラ
- 違反レポート
- デフォルトの時間枠と非デフォルトの時間枠
- ブロックおよびリダイレクト

パッケージの作成およびパッケージ名の設定

パッケージを作成するには、new 演算子を使用します。次の行で、Package を含むクラスをインスタンシエートしています。

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

新しいパッケージを作成したあと、次の行でパッケージに名前を付けています。

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

サービス ルールの定義

次の行で、レジデンシャルな getRule メソッドのパラメータで指定したルールの参照が返されます。

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

違反

ネットワーク トランザクションに指定されている*制限*を超過すると、*違反*が発生します。サービスのルールを宣言するときに、違反を定義します。帯域幅の容量制限またはキロビット / 秒（Kb/s）転送速度の制限を超過した場合や、サービスであらかじめ定義されているその他の制限を超過した場合を、違反とみなすことができます。違反を定義する目的は、このような制限に違反した場合に実行すべきアクションを指定することです。

ルールには、*違反前（pre-breach）*および*違反後（post-breach）*の 2 つのモードがあります。サービス ルールの制限に達しないときは *pre-breach アクション*が実行され、制限に達して超過したときには *post-breach アクション*が実行されます。

これらのアクションの例としては、帯域幅容量の制限、サービスの拒否、RDR レポートのトリガー、および Web サイト リダイレクションがあります。すべての pre-breach ファンクション コール（メソッド）に、対応する post-breach ファンクション コールがあります。

違反はデフォルトの状態では非アクティブなので、違反をアクティブにしてから適用する必要があります。次の行で、setPreBreachAccessMode メソッドのフラグを true に設定することにより、違反前の条件をアクティブにしています。

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

例 — FTP サービス ルール

次に、FTP サービス用のルールを設定する例を示します。この例では、FTP File Downloading という名前のサービスが存在することを前提としています。これらのルールにより、帯域幅は 5 Kb/s、1 日あたり 100 MB に制限されます。どちらかの制限に達すると、サービスを禁止し、RDR をトリガーします。この例では、RDR は 1 日あたりの合計をレポートで使用すると指定しています。最後にサービス コンフィギュレーションを適用し、そのあとで接続を終了します。

```
// 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);
```

上記の例でわかるように、総容量はサービス トランザクションで使用されたキロバイト数または使用されたネットワーク セッション数のどちらでも測定できます。どちらかの制限に達した時点で、サブスクライバの使用帯域幅レートを変更したり、レポート（RDR）を送信したり、サブスクライバをサービスから除外したりできます。

以下の各項で、このプログラムの各部分について個々の要素を説明します。

アグリゲーション

*アグリゲーション*期間は、パッケージごとにグローバルで定義します。アグリゲーションは、RDR が作成されたときレポート生成に使用されます。アグリゲーションは、月、週、日、または時間ごとに測定できます。

同じサービスで、パッケージごとに異なるアグリゲーション期間を使用できます。つまり、1 つのパッケージで FTP Download Service の 1 日あたりのトータルを指定し、同じ FTP Download Service を使用するもう 1 つのパッケージでは 1 時間あたりのトータルを指定することが可能です。このアグリゲーション（トータルでの使用状況に関する情報）は、サブスクライバ宛の項目別の請求書または取引明細書に利用できます。

次の行で、1 日あたりのトータルを使用して RDR を生成することを指定しています。

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

クォータの配分はパッケージごとに定義します。したがって、パッケージが 2 つのサービス（たとえば FTP Download Service および Video Conferencing Service）で構成される場合、アグリゲーションには両方のサービスの合計を反映させます。これはパッケージを設計する際に考慮すべき事項の 1 つです。

上記の例では、サービス コンフィギュレーションをオフラインで編集しているので、一連のコマンドは順不同です。したがって、変更後のサービス コンフィギュレーションを SCE の SCE プラットフォームに伝播するまで、コマンドは実行されません。

帯域幅コントローラ

帯域幅コントローラ（別名メーター）は、特定のサービス トランザクションが使用している容量（Kbps）を測定するためのフロー ルール ベースの機能です。1 フローのトランザクションでも、いくつかのフローを同時に使用するトランザクションでも測定可能です。帯域幅コントローラのルールは、フロー ベース ルール、時間ベース ルール、または両方の組み合わせにすることができます。

帯域幅コントローラには、方向性があります。帯域幅コントローラはそれぞれ、アップストリーム方向またはダウンストリーム方向のどちらかで容量を制御します。この方向は API で設定できます。

フロー数（同時セッション数）およびそれらの合計の使用帯域幅レートがわかれば、プロバイダーがシステム サービスをより強力に制御して、ネットワーク リソースの利用方法に関してより影響を及ぼすことができるので便利です。たとえば、帯域幅コントローラの機能によって、サブスクライバが特定の帯域幅レートを超過しないかぎり、使用できるフロー数を無制限にするという指定が可能です。

ネットワーク トランザクションで使用されるフロー数を制御すれば、たとえば RTPS プロトコルを使用するインターネット ビデオ アプリケーションのデジタル ブロードキャストなどに役立ちます。RTSP は、ビデオ トラフィックのいくつかの同時フローを維持できます。帯域幅コントローラは、サブスクライバが特定のレート制限を超えないように指定するためのツールをプロバイダーに提供します。

次に、フロー レベルでの帯域幅コントローラ機能の使用例を示します。

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

違反レポート

*違反*は、システムで指定された制限を超過した場合に発生します。違反が検出された場合に実行できるアクションは、2 通りあります。それは、(a) サービスを禁止（ブロック）する、または (b) サブスクライバのトランザクションまたはサービスの配信方法に何らかの変更を加えることです。

したがって、システムには 2 タイプのレポート生成機能があります。それは、(a) ブロック レポート、または (b) 違反レポートです。サービスで違反が発生するたびに、違反のタイプに応じてどちらかのレポートをトリガーするように、システムを設定できます。

違反が発生したときに、関数のパラメータを true に設定することによって RDR の作成をトリガーする例を次に示します。

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

違反が発生するたびに RDR を生成するのではなく、サブスクライバに電子メールで通知するように設定することもできます。

デフォルトの時間枠と非デフォルトの時間枠

ここでは、時間枠、デフォルトのルール、非デフォルトのルール、およびデフォルトのルール関数を使用する場合について説明します。

時間枠を使用すると、時間によって異なった動作を実行するルールを設定できます。時間枠の例としては、夕方の時間枠、夜間の時間枠、および週末の時間枠があります。SCAS システムでは、最大 4 つの時間枠を定義できます。

API を使用して、時間枠の例外に基づいて情報を処理するアプリケーションを作成できます。たとえば、通常の曜日に基づくスケジュールでは週末料金が金曜日の午後 4 時 54 分から始まるのに対し、31 日水曜日の午後 3 時から翌日の午後 6 時まで、無制限の帯域幅を使用できる週末料金のビデオ会議サービスのために時間枠を変更するアプリケーションなどです。

`getDefaultRule` メソッドを使用すると、そのルールはすべての時間枠に対してグローバルに適用されます。次に、`getDefaultRule` メソッドを使用してグローバルな時間枠ルールを設定する例を示します。

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

非デフォルト ルールの例は次のとおりです。サブスクライバがサービスへのアクセスを拒否されるようにデフォルトのルールが定義されていると仮定します。さらに、2 つの時間枠を作成したと仮定します。1 つは平日の 2200 ～ 0600 であり、もう 1 つは金曜日の同じ時間帯です。平日の時間枠ではサービスへのアクセスを許可し、金曜日にはサービスへのアクセスを禁止するように、例外を作成できます。

次のコード例では、T1 という 1 つの時間枠を作成しています。デフォルトのルールはすべての時間枠に適用されるので、時間枠 T1 を使用してその例外を設定し、定義した期間中はサービスへのアクセスを許可しています。

```
    . . .
// 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);
    . . .
```

ブロックおよびリダイレクト

block-and-redirect 関数は、HTTP、RTSP など、リダイレクトが可能なプロトコルを使用するサービスにのみ関係します。この関数を使用すると、サブスクライバがサービス パッケージに含まれていないサービスへのアクセスを試みた場合に、サブスクライバのトランザクションを別のネットワーク アドレスにリダイレクトできます。そのネットワーク アドレスまたは Web サイトで、サービスへのアクセス方法についての有益な情報を提供できます。

```
// 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);
```

例 ― サービスの追加とサービス コンフィギュレーションの適用

次の例は、ネットワーク プロバイダーがサブスクライバの内部ネットワーク内でホスティングされるゲーム アプリケーション サーバ上でサービスを設定する方法を示しています。この例では、テンプレートとして Doom ゲーム アプリケーションを使用しています。この Java アプリケーションの機能は次のとおりです。

ステップ 1　新しいサービス コンフィギュレーションを作成します。

ステップ 2　*Local Doom* というサービス エレメントを定義します。

ステップ 3　Doom サービスをサブスクライバ起動にします。

ステップ 4　この Java アプリケーションは、サービスをホスティングするサブスクライバ ネットワーク内のローカル サーバの IP アドレスのリストをマッピングします。

ステップ 5　Java アプリケーションは、サブスクライバによるサービスへの接続時間に基づいて、課金レコードをトリガーします。

```
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) {
```

```
                    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 ñ 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);

        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() {}
}
```