Oracle® Communications DSR Release 8.4

DCA Programmer's Guide

F12305 Revision 01

April 2019

ORACLE®

DSR Release 8.4 DCA Programmer's Guide

Table of Contents

List of Figures

List of Tables

1. Introduction

Diameter Custom Applications (DCA) is a framework that enables a significant reduction of the coding – testing – deployment – maintenance cycle in the development of Diameter applications.

The present document is intended to developers of DCA Apps. It describes how DCA Apps can be created, how their business logic and configuration data can be provisioned, how their lifecycle from development to production can be managed, and the various APIs available.

Following the DCA Framework and DCA Apps activation (Chapter 2), the document is organized around three DCA Apps examples: Blacklist (Chapter 3), CountULR (Chapter 6), and Rate (Chapter 8), which demonstrate the basic features of the DCA Framework. A number of additional chapters, interleaved with the chapters describing the three DCA Apps provide a gradual insight into essential capabilities of the DCA framework, like the DCA App lifecycle management (Chapter 4), stateful DCA Apps development mechanisms (Chapter 5) and tools for monitoring the execution of DCA Apps (Chapter 7).

Chapter 9 provides a complete GUI reference, with the Development Environment described in Chapter 10.

The various APIs available are described in Chapter 11.

1.1 References

[1] CGBU_018429 - DCA Framework and Application Activation and Deactivation

[2] E58954-02, DSR Software Installation and Configuration Procedure

1.2 Glossary

This section lists terms and acronyms specific to this document.

Acronym	Description
API	Application Programming Interface
ART	Application Routing Table
AVP	Attribute Value Pair (in context of Diameter protocol)
ComAgent	Communication Agent
DA-MP	Diameter Agent Message Processor
DAI	DSR Application Infrastructure
DAL	Diameter Application Layer
DBCA	Database Change Agent
DCA	Diameter Custom Applications (framework)
DRL	Diameter Routing Layer
DSR	Diameter Signaling Router
EDL	Encode-Decode Library
I-SBR	Independent SBR (Session Binding Repository)
JSON	Java Script Object Notation
MEAL	Measurement, Event and Alarm
MO	Managed Object
NOAM	Network Operations Administration and Maintenance

Acronym	Description
OAM	Operations, Administration & Maintenance
OID	Object Identifier (SNMP)
Perl	Practical Extraction and Reporting Language – a scripting language
PRT	Peer Routing Table
SNMP	Simple Network Management Protocol
SOAM	Site Operations Administration and Maintenance
TTR	Trace Transaction Record (in context of IDIH)
U-SBR	Universal SBR (Session Binding Repository) – used by DCA apps to store generic application state data

1.3 Terminology

Acronym	Description
A-Level	NOAM –level
Asynchronous Call Symbol	Symbol in the Development Environment that represents a code statement that calls an asynchronous function provided by the DCA Perl API. The code statement occurs within a preceding Execution Block. The symbol displays the name of an asynchronous function that is invoked.
B-Level	SOAM- level
DCE Development Environment	Web application where a custom Diameter application developer can edit, save, check syntax, compile the application code for a Diameter Custom Application, and generate an Interactive Flow Control Chart from the application code.
Execution Block Symbol	Symbol in the Development Environment that corresponds to an application subroutine where the name of the symbol is also the name of the subroutine.
Internal Variable	A storage mechanism that allows persistence during a Diameter transaction lifetime.
Start Symbol	Symbol in the Development Environment that marks the start of execution for the application.
Termination Symbol	Symbol in the Development Environment that represents the end of the application's execution.

1.4 WARNING on Copy and Pasting Code from this Guide

Please note that when copy and pasting code from Microsoft Word or other editors or document viewers into the Development Environment editor, some characters (typically punctuation characters like quotes) may end up having non-ASCII character codes, which leads to compilation errors. For instance:

```
Checking syntax...
Unrecognized character \x<…> in column <…> at script file line <…>.
Check Syntax found errors. Correct the syntax errors and try again
```

The solution is to delete the copy and pasted punctuation character and re-type it in the Development Environment editor.

2. DCA Activation and Deactivation

Activation and deactivation are standard procedures that enable the DSR applications in general and DCA Apps in particular to be installed and uninstalled on a network.

2.1 DCA Activation

To start developing a new DCA App, perform the following two steps:

1. Activate the DCA framework on the NO. See Procedure 5 in [1] CGBU_018429 - DCA Framework and Application Activation and Deactivation for the instructions.

   This step needs to be performed only once for a given network.

2. Activate the new DCA App on the NO. See Procedure 6 in [1] CGBU_018429 - DCA Framework and Application Activation and Deactivation for the instructions.

   Perform this step once per DCA App (similar to native DSR applications). Note, however, that only a limited number of DCA Apps (currently 5) can be simultaneously activated. Therefore, deactivate old DCA Apps to make room for new DCA Apps.

Figure 1 provides an overview of the activation-deactivation lifecycle.

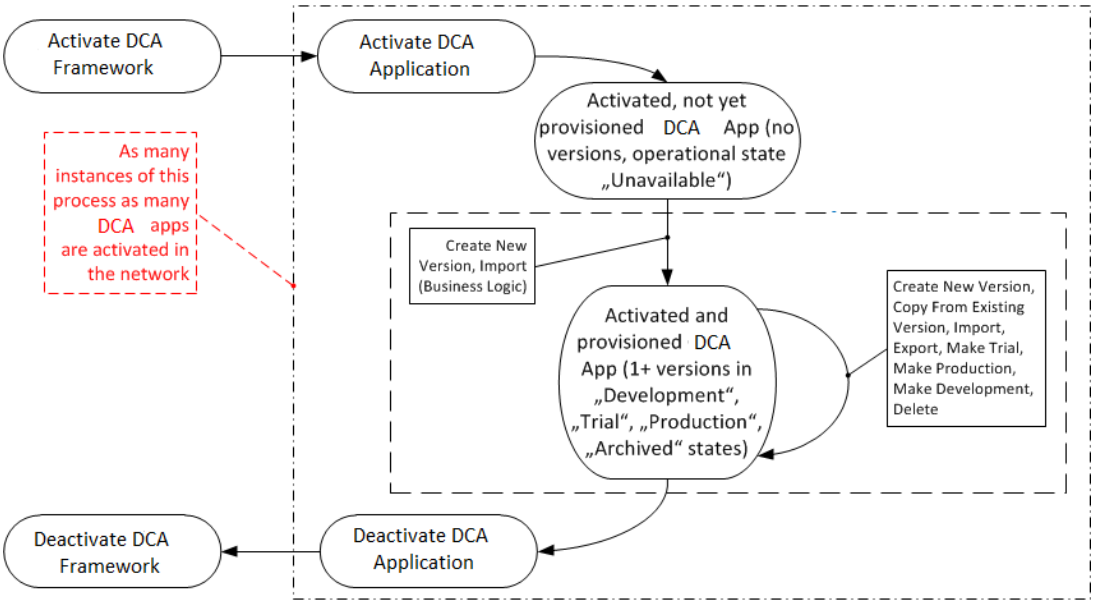


Figure 1: DCA Activation- Deactivation Lifecycle

2.1.1 DCA Framework Activation

When the DCA framework is initialized, the DCA Framework folder with the Configuration file becomes visible in the left side menu (Figure 2).

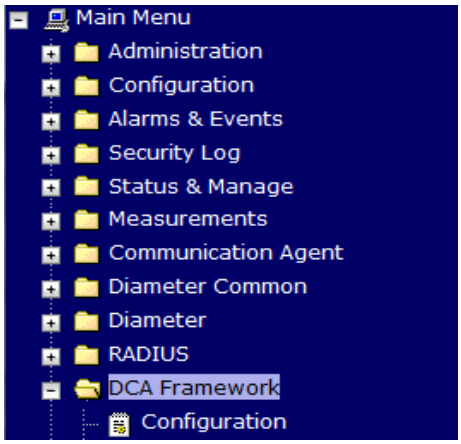


Figure 2: DCA Framework Menu

All the measurements (Figure 3) and KPIs (Figure 4) associated with the DCA Framework become visible as well.

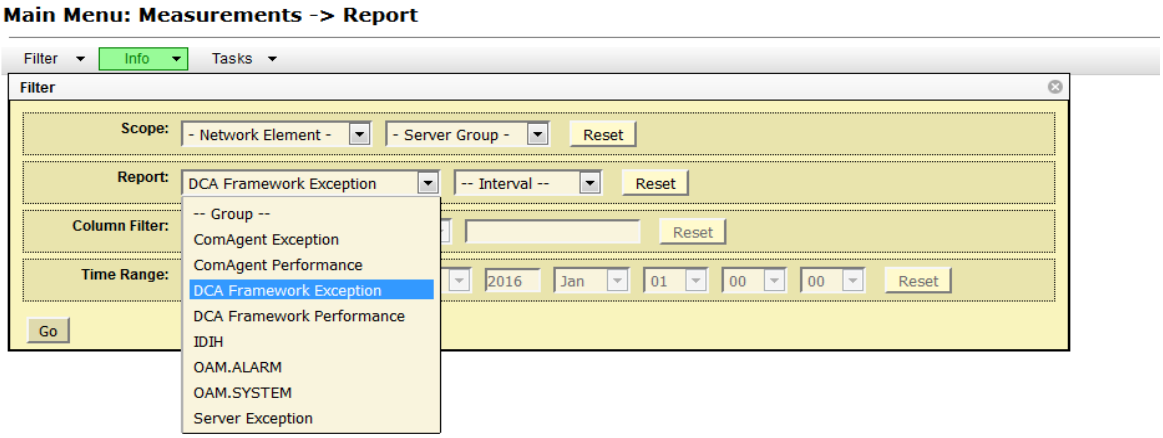


Figure 3: DCA Measurements

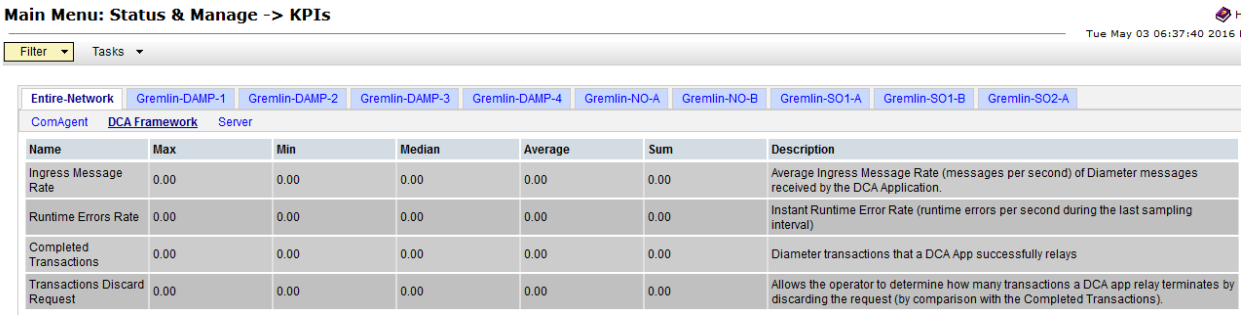


Figure 4: DCA KPIs

2.1.2 DCA App Activation

When the new DCA App is activated, the DCA App subfolder with the name provided by the user during the activation procedure becomes visible in the left side menu (Figure 5). The DCA App subfolder includes the screens for enabling the business logic and provisioning configuration data. The DCA App becomes visible across DSR (ART, maintenance screen, etc.).

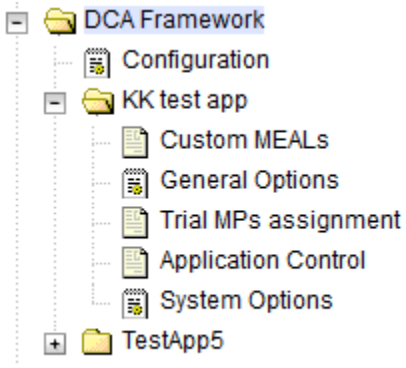


Figure 5: DCA Application Menu

2.1.3 Post-Activation DCA App State

Following the activation procedure, the DCA App is in the disabled state. While in the disabled state, Diameter traffic is not delivered to the DCA App. First, enable the DCA App from the **SO Main Menu: Diameter→Maintenance→Applications**. Note that on this screen the DCA App is identified by the short name configured by the user during the DCA App activation procedure.

Independently from the enabled/disabled state of the DCA App, at this stage no version of the DCA App has been provisioned yet. As a result, there is no version in Production and Trial state. As long as no Production or Trial version is available for a DA-MP to run, the DCA App's operational status is Unavailable (see **Main Menu: Diameter→Maintenance→Applications** on the SO).

The behavior of a DCA App while in Unavailable operational state (provided that the DCA App has been enabled) is configurable on the SO from the **Main Menu: DCA Framework→<DCA App Name>→System Options** (see Section 9.4); possible options are dropping the Diameter request, forwarding the Diameter request, or returning a Diameter answer with a configurable error code.

From this point on the user can provision the configuration and business logic for the DCA App.

2.2 DCA Deactivation

The deactivation procedures enable a DCA App and, respectively, the DCA framework to be removed from a given network.

2.2.1 DCA Application De-Activation

The deactivation of a DCA App is not allowed as long as versions of the respective DCA App are still in Production and/or a Trial state (see Chapter 4).

Following deactivation, the DCA App's GUI folder under the DCA Framework menu item disappears. The DCA App is deregistered from the ART; its KPIs and measurements do not display or report any longer.

2.2.2 DCA Framework De-Activation

DCA framework deactivation is not allowed as long as at least one DCA App is activated in the network.

Following deactivation, the DCA framework GUI folder disappears from the left-hand GUI menu.

3. DCA App Provisioning – The Blacklist DCA App

This section is a tutorial to provision the configuration data and business logic for a simple DCA App.

3.1 The Blacklist DCA App

The Blacklist DCA App checks the Origin-Host AVP of incoming Diameter requests and verifies whether it is blacklisted or not. In case the Origin-Host is blacklisted, the Diameter request is dropped, otherwise, the Diameter request is forwarded unchanged.

3.2 Prerequisites

The DCA Framework must have been previously activated as described in [1] CGBU_018429 - DCA Framework and Application Activation and Deactivation. Also, a DCA App with the name Blacklist is activated as described in [1] CGBU_018429 - DCA Framework and Application Activation and Deactivation.

The Blacklist DCA App has to be enabled on all the DA-MPs in the network from the SO **Main Menu: Diameter→Maintenance→Applications**.

An ART rule is added that enables Diameter messages to be delivered to the Blacklist DCA App.

3.3 The Process

The following step must be followed to provision the Blacklist DCA App:

Step 1: Configure the general options and behavior of the Blacklist DCA App.

Step 2: Create a new development version of the Blacklist DCA App.

Step 3: Define the structure of tables to store the Blacklist configuration data.

Step 4: Provision the Blacklist configuration data.

Step 5: Provision the Blacklist business logic – essentially a Perl script.

Step 6: Render the Flow Control Chart based on the Perl script. Save and perform syntax checks.

Step 7: Test the Blacklist DCA App: configure the Trial DA-MPs and promote Blacklist to Trial state.

Step 8: Compile Blacklist, promote Blacklist to Production state.

3.3.1 Step 1: Configure the DCA App's General Options and Behavior

At this stage, there is no version available for the Blacklist DCA App. As a result, the DCA App is in the Unavailable operational state. No traffic is forwarded to the Blacklist DCA App and, for outside observers, the DCA App behaves as specified in the SO screen **Main Menu: DCA Framework→<DCA App Name>→System Options**, **Application unavailable configuration** section (see also Section 9.4).

The **Run-time error configuration** section of the same screen defines the behavior of the DCA App in case a runtime error occurs during the execution of the event handlers.

Finally, the DCA App programmer must ensure the names specified on the NO screen **Main Menu: DCA Framework→<DCA App Name>→General Options** (see Section 9.2.3) for the Diameter request and answer event handlers (Perl subroutines) are consistently used in the Perl script.

For Blacklist in particular, **Perl Subroutine for Diameter Answer** is left empty because there is no event handler defined to process the Diameter answers.

3.3.2 Step 2: Create New Development Application Version

Go to the **Main Menu: DCA Framework→<DCA App Name>→Application Control** screen on the **NO** and click **Create New Development** (see Figure 6). The Create New Development screen displays. Specify a name for the newly created Blacklist version and optionally provide comments (e.g., author name, brief description of the business logic, etc.). Figure 7 shows the newly created version.

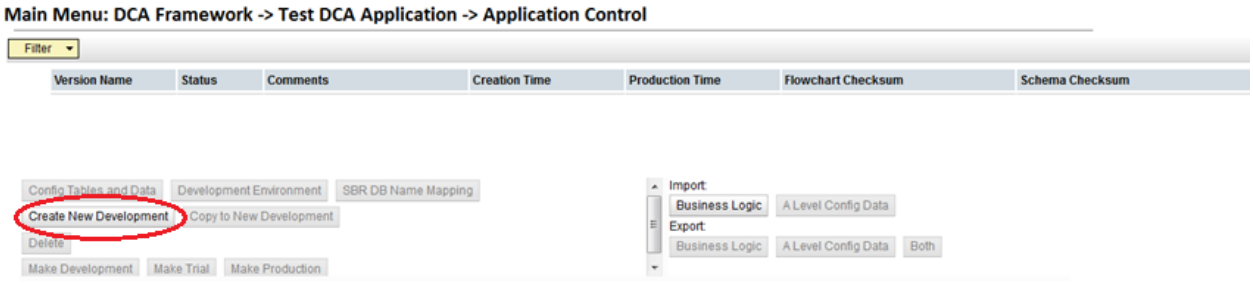


Figure 6: Create a New Application Version

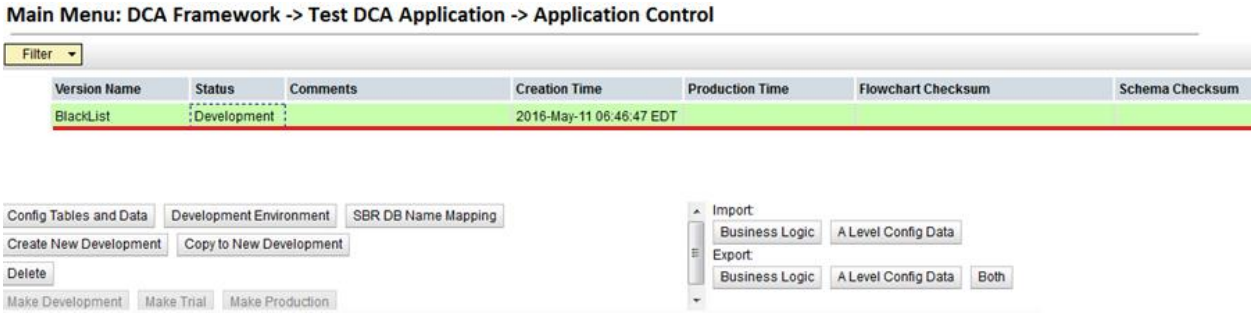


Figure 7: New Application Version Created

3.3.3 Step 3: Define the Configuration Data Structure

Select the newly created development application version on the Application Control screen and click **Config Tables and Data**. The Tables screen (Figure 8) displays. Click **Insert** on the Tables screen and

create a new configuration table for provisioning the blacklist. The Blacklist DCA App configuration table contains only one field: OriginHost, which is of type DiameterIdentity, see Figure 8).

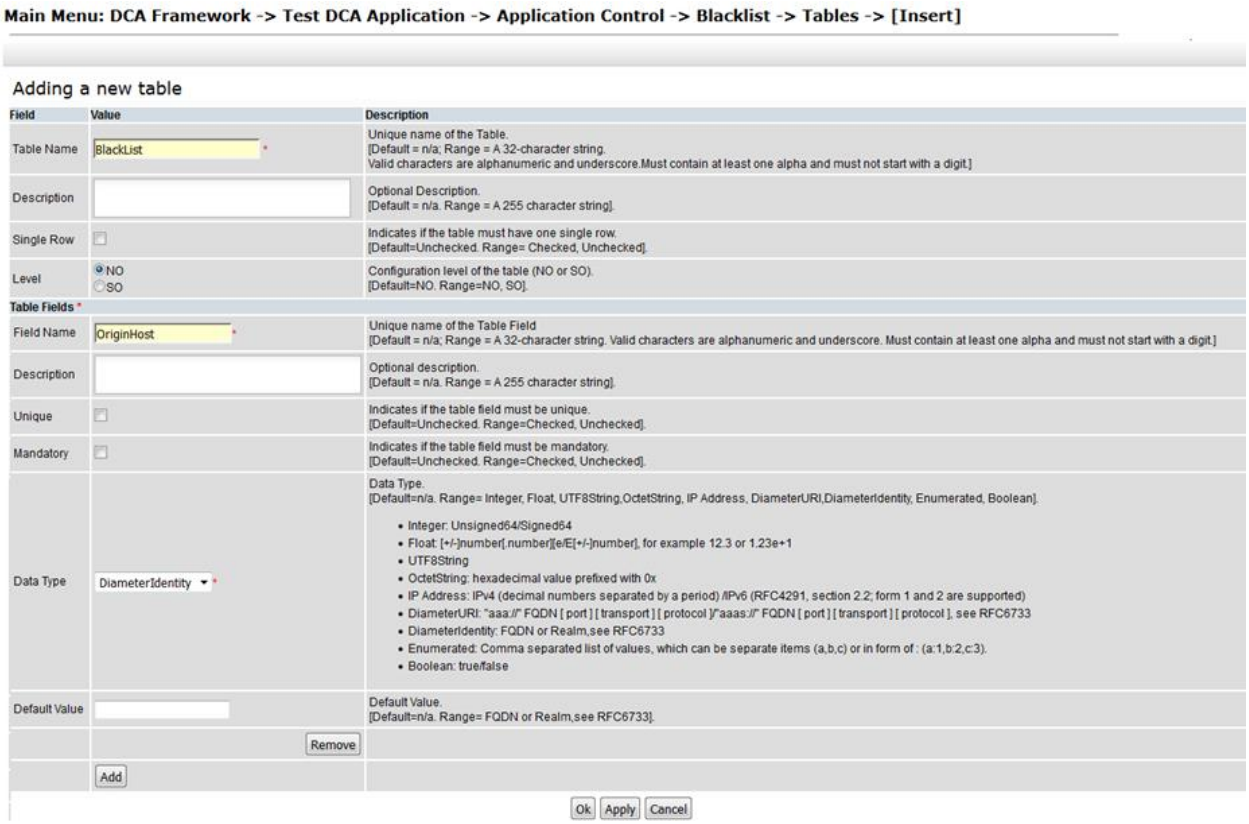


Figure 8: Create a New Database

Note: In this example, the configuration table is defined at the NO level. That means the configuration table is replicated to all the DA-MPs in the network.

Alternatively, a configuration table may be defined at the SO level. That means, while its structure is defined across the entire NO, its content is replicated only to the DA-MPs in each individual SO. In this way distinct SOs may use different configuration data (see Section 9.3.5).

3.3.4 Step 4: Provision the Configuration Data

Once the structure of the Blacklist table is defined, the table displays on the Tables screen. Select it and click **Provision Table**. The Provision Table View screen displays (Figure 9). Click **Insert** on the

Provision Table View screen and insert all the Blacklisted Origin-Hosts to the table one by one (Figure 10).

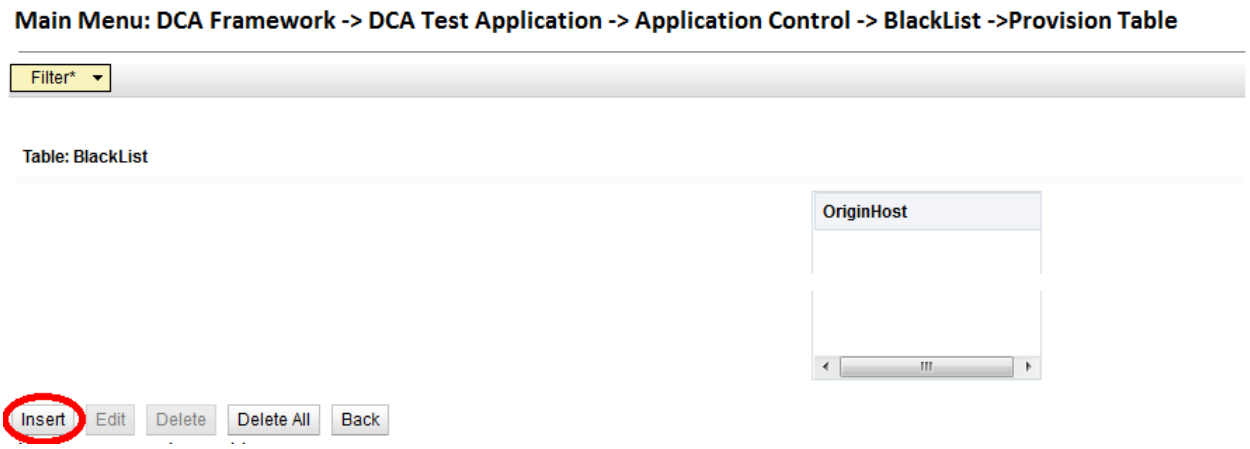


Figure 9: Provision Table BlackList

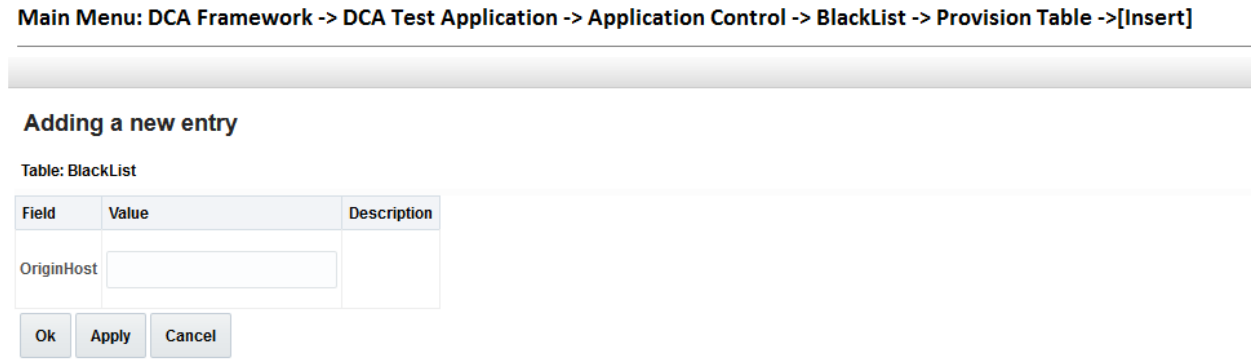


Figure 10: Insert a New Data Row to the BlackList Table

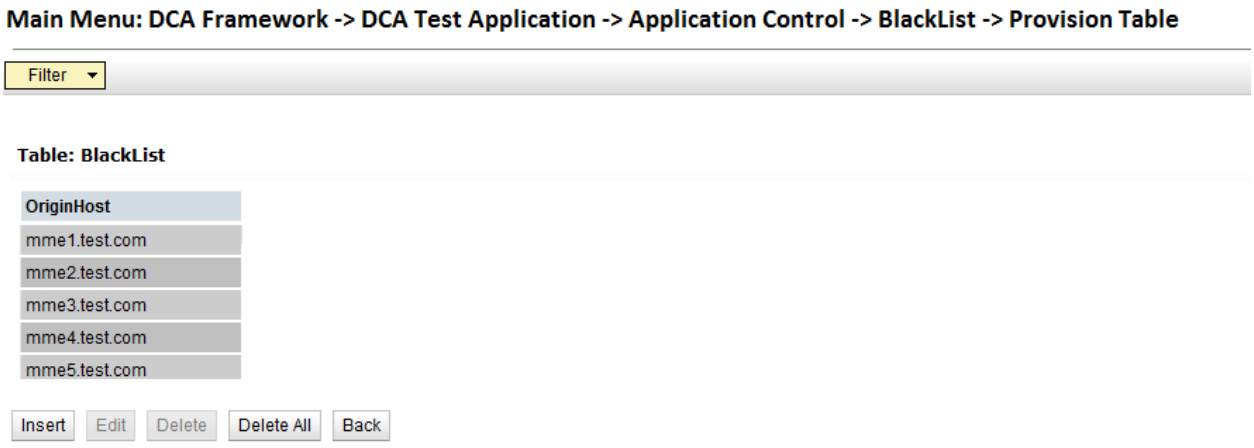


Figure 11: Provision DCA DB Tables

3.3.5 Step 5: Provision the Business Logic

Go back to the Application Control screen, select the application version, and click **Development Environment**.

In the development environment, you can edit, save, check syntax, and compile the DCA App's Perl code, which defines the business logic the DCA App implements. An interactive Flow Control Chart is also rendered based the DCA App's Perl script. The Flow Control Chart provides an overview of the control flow within the DCA App and is useful in following the asynchronous calls and indicating the terminating actions (forward, drop, or return answer). See Chapter 10 for more details on Development Environment.

The development environment of the Blacklist DCA App is illustrated in Figure 12.

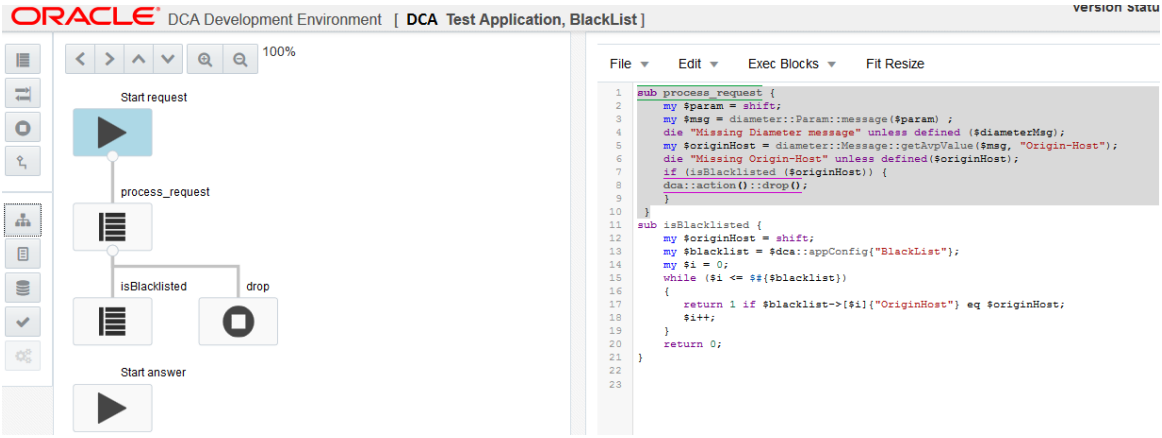


Figure 12: The Blacklist DCA App Development Environment

First, the DCA App programmer has to write in the right-hand panel the Perl code illustrated in Figure 13. The left-hand panel containing the flowchart is empty until the flowchart is rendered in Step 6.

```
sub process_request {
   my $param = shift;
   my $msg = diameter::Param::message($param) ;
   die "Missing Diameter message" unless defined ($msg);
   my $originHost = diameter::Message::getAvpValue($msg, "Origin-Host");
   die "Missing Origin-Host" unless defined($originHost);
   if (isBlacklisted ($originHost)) {
      dca::action::drop();
   } else {
      dca::action::forward();
   }
}
sub isBlacklisted {
   my $originHost = shift;
   my $blacklist = $dca::appConfig{"BlackList"};
   my $i = 0;
   while ($i <= $#{$blacklist}) {
      return 1 if $blacklist->[$i]{"OriginHost"} eq $originHost;
      $i++;
   }
   return 0;
}
```

Figure 13: Blacklist Perl Code

The Perl script (see Figure 13) makes use of the `getAvpValue` function to read the value of an AVP. The `getAvpValue` function is part of the EDL API, which is described in Section 11.1.2. It also uses the

`drop` and `forward` functions to discard and respectively forward the Diameter request. The drop function is part of the basic routing API, which is described in Section 11.4.

3.3.5.1 Where is the Perl Script Being Executed?

Although the Perl script is edited using the NO GUI, the Perl script is replicated to and eventually executed on the DA-MPs. In other words, there is no possibility of making the Perl script process traffic other than running it on the DA-MPs.

3.3.5.2 How Do the Event Handlers Get Invoked?

The business logic of a DCA App consists of a collection of event handlers, which are invoked when a Diameter message is delivered to the respective DCA App. A DCA App may therefore define one event handler for Diameter requests and one event handler for Diameter answers. Subsequent sections introduce another category of event handlers, related to asynchronous database queries, but let's stick to the Blacklist DCA App for now. Blacklist defines only one event handler: `process_request`. Unlike `isBlacklisted`, which is a standard Perl subroutine invoked from `process_request`, `process_request` itself is not explicitly invoked from anywhere in the Perl script. The event handlers are explicitly invoked by the Perl running environment of the DCA framework. Their names are configured in the **NO Main Menu→DCA Framework→< Application Name>→General Options** screen and by default these names are `process_request` and `process_answer`. These names may be changed, but one needs to make sure that the configured event handler names are consistent with the names used in the Perl script. Also, the event handler names are left empty if there is no corresponding event handler defined in the Perl script (see Figure 14).

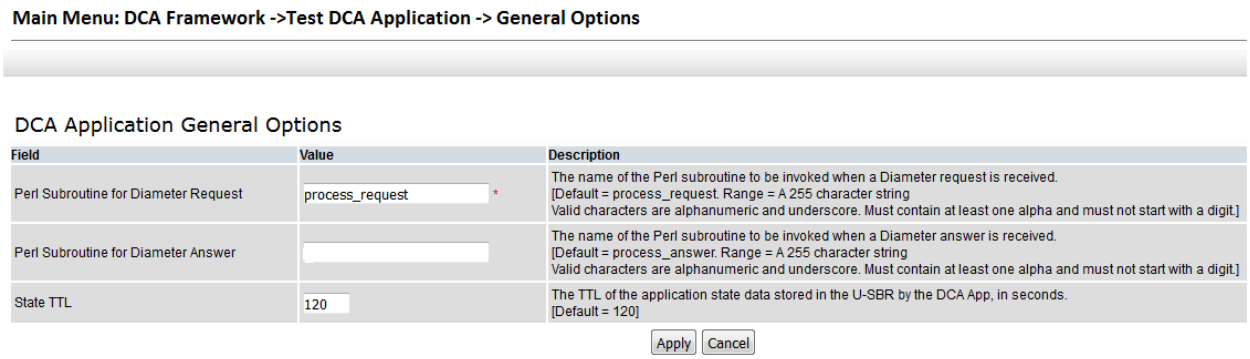


Figure 14: Event Handler Subroutine Name Configuration

3.3.5.3 How Does the DCA App Configuration Data Get Accessed?

The configuration data of a DCA App is accessible to the Perl script through the `$dca::appConfig` variable, which is a complex variable representing a hash of arrays of hashes. One has to dereference it with exactly the same table names and field names specified when the structure of the configuration tables has been defined in step 3.3.3:

```
$dca::appConfig{"<table_name>"}->[<record_number>]{"<field_name>"}
```

in our case:

```
$dca::appConfig{"BlackList"}->[<record_number>]{"OriginHost"}
```

3.3.5.4 What is the Main Part Good For?

Blacklist has an empty Main Part. The Main Part of a Perl script is where the Perl interpreter starts executing instructions. In DCA, the main part is executed **only once** following the successfully compilation of the script.

The Main Part is typically used to perform whatever initializations are necessary (like for instance Custom MEAL objects, as we describe later on).

Another task that fits into the Main Part is DCA App configuration data post-processing. We have seen in Section 3.3.5.3 that the Blacklist configuration data is accessible to the business logic (Perl script) as an array. Blacklist simply loops through the array when looking for each Origin-Host, but a more performance–aware version would certainly convert the array into a more performant data structure, like for instance a hash table keyed by the Origin-Host values.

Other DCA apps may even need to use multiple keys (hence multiple hash tables) or compound keys; the Main Part is the right place to perform this kind of structural optimizations on the DCA App configuration data.

3.3.6 Step 6: Render Flow Control Chart, Save Script, Check Syntax

After editing the script, while in the Development state, the following actions are possible (see Figure 15):

- **Render Chart** (to generate the flowchart from the Perl code);
- **Save** (to save the Perl code and the flowchart);
- **Check Syntax** (to check syntax of Perl script).

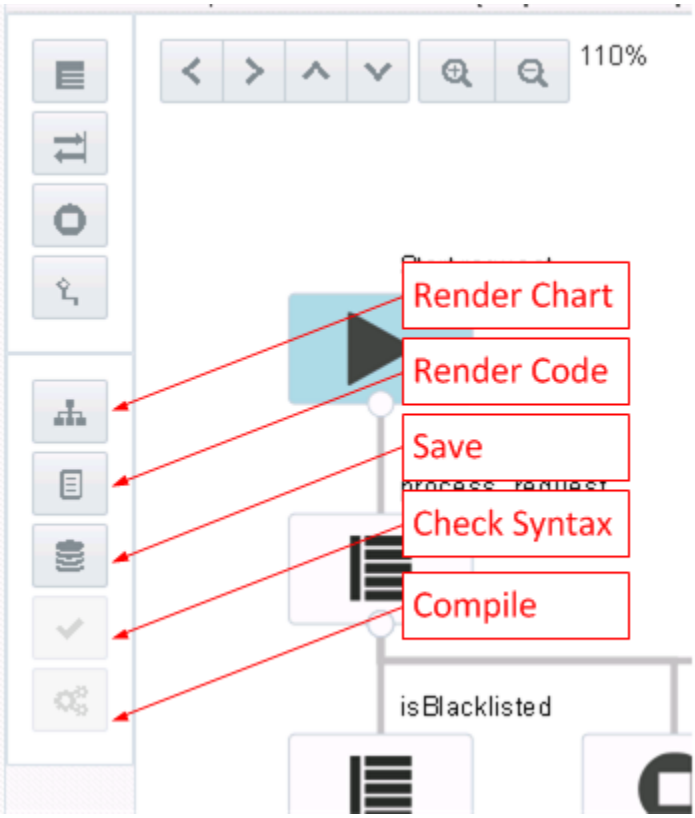


Figure 15: Development Environment Buttons

The **Render Chart** action generates a flowchart based on the Perl code. Note that the flowchart has a Perl subroutine granularity and not a Perl instruction granularity. The flowchart's main purposes are: (i) to describe how the callback subroutines are linked to the event handlers (Diameter message handlers or other callback subroutines) that registers them and (ii) to indicate the terminating actions (drop, forward or return answer).

The flowchart does not illustrate on which condition a Perl subroutine is invoked (i.e., if conditions) or how many times a Perl subroutine is invoked (i.e., loop conditions). Also, the **Render Chart** action is explicitly triggered by clicking the corresponding button after each modification of the Perl script.

Save allows the flowchart and Perl code to be saved, while the DCA App version is in Development or Trial state.

Check Syntax becomes enabled once the **Save** action has been completed, while the DCA App version is in Development or Trial state. It performs a syntax check on the Perl code and displays the errors if the syntax check fails.

3.3.7 Step 7: Test the DCA App Version

Having the configuration data and business logic provisioned, it is now time to test the Blacklist DCA App.

A DCA App version is tested by promoting it to the Trial state, which automatically results in running it on the dedicated Trial DA-MPs.

The first step is, therefore, to configure the Trial DA-MPs, which can be done from the Trial MPs Assignment screen (see Figure 16 and Section 9.2.4).

The Trial DA-MPs assignment is configured per DCA App, that is, it needs not be repeated for each DCA App version.

Note also that our network contains only one DA-MP, which is also a Trial DA-MP. However, in a real life deployment, there would typically be a few Trial DA-MPs and a number of non-Trial DA-MPs.

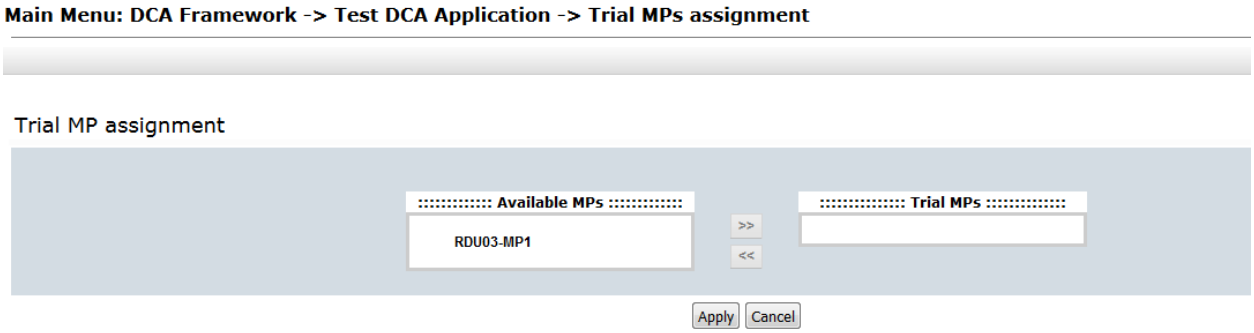


Figure 16: Trial MP Assignment

Next, on the Application Control screen, promote the DCA App version from Development to Trial state by selecting it and clicking **Make Trial**.

While in Trial state the DCA App version can be: modified, saved, have the syntax checked and, in addition to the Development state, it can also be compiled (by clicking **Compile**, see Figure 15), as further described in Chapter 4. During each new cycle starting with the first Perl code modification and lasting until the next successful compilation (with an arbitrarily number of modifications, save and syntax check actions taking place during this time), the Trial DA-MPs execute the previously successfully compiled Perl script of the respective DCA App version.

If successfully compiled, the Blacklist DCA App on the Trial DA-MP switches into the operational state Available (see the **SO Main Menu: Diameter→Maintenance→Applications** screen). On the non-Trial DA-MPs the DCA App operational state remains Unavailable because there is no DCA App version in Production state at this moment.

3.3.8 Step 8: Promote the DCA App Version to Production State

A successfully compiled Trial DCA App version can be promoted to the Production state. For this purpose, on the Application Control screen, select the DCA App version and click **Make Production**.

At this stage the only DCA App version available so far is in Production state. All non-Trial DA-MPs start running it and on these DA-MPs, the DCA App operational state becomes Available. Because there is no DCA App version in the Trial state, the Trial DA-MPs run the Production version as well.

Please note that our network is a very particular case that contains one single DA-MP, which is configured as a Trial DA-MP. This means that the Production version is executed on only this DA-MP if and only if no Trial version exists. As soon as a (new) Development version is promoted to the Trial state, the Trial DA-MP stops executing the Production version and starts executing the (new) Trial version.

While in Production state, the business logic of the DCA App version cannot be changed anymore. It is only the configuration data that can be updated.

We have achieved our initial objective of running the Blacklist DCA App in our network. From this point on a number of alternatives are possible:

- Demote the DCA App version from Production state back to Development to fix bugs, re- test and promote back to Production state.
- Copy the DCA App version into a new version with the purpose to improve its business logic (in terms of efficiency, functionality, or both) and eventually promote the newer version to Production state.
- Export the DCA App version from the current network and import it onto another network.

We are touching on the DCA App lifecycle management topic, which is described in more detail in the next chapter.

4. DCA Application Lifecycle

The DCA Application Lifecycle enables the DCA App programmer to manage the lifecycle of a DCA App.

So far we have developed one single DCA App version, we tested it and promoted to the Production state. The state transitions are illustrated in Figure 17.

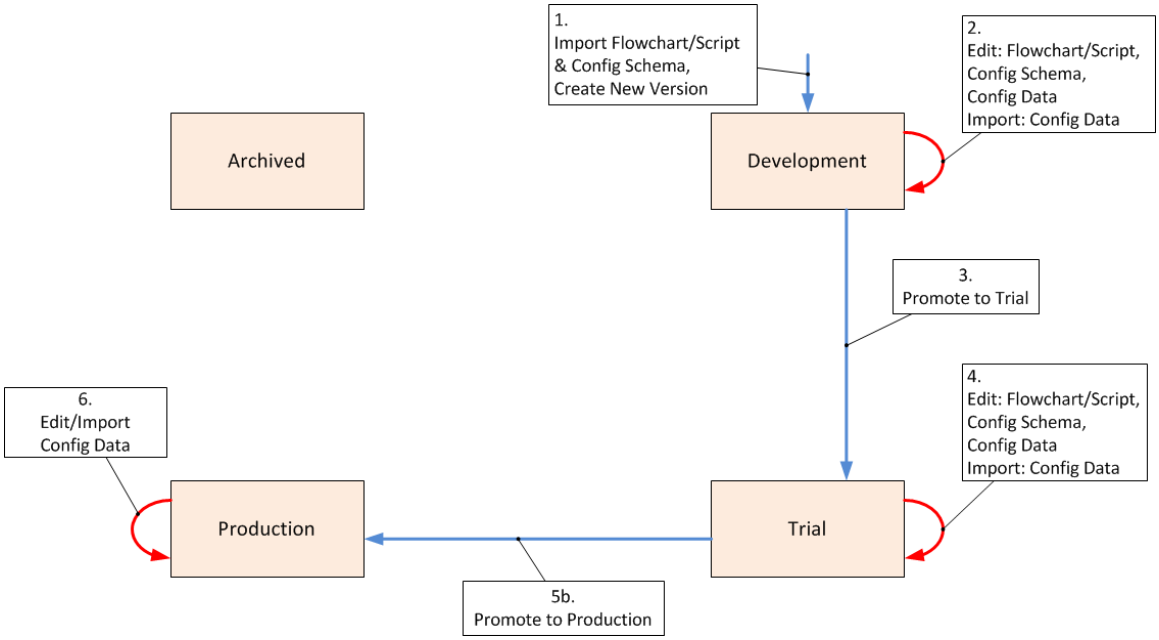


Figure 17: Transitions from Development to Production State

In a real life deployment, a DCA App may need to be continuously enhanced both in terms of efficiency as well as features. A typical approach would be to clone the DCA App version currently in Production state to a new version in Development state, work on the new version (while the old version is processing the Diameter traffic), test the new version and eventually replace the older version in Production state with the newer one. This process is illustrated by the transition path 7→3→5b→9 in Figure 18.

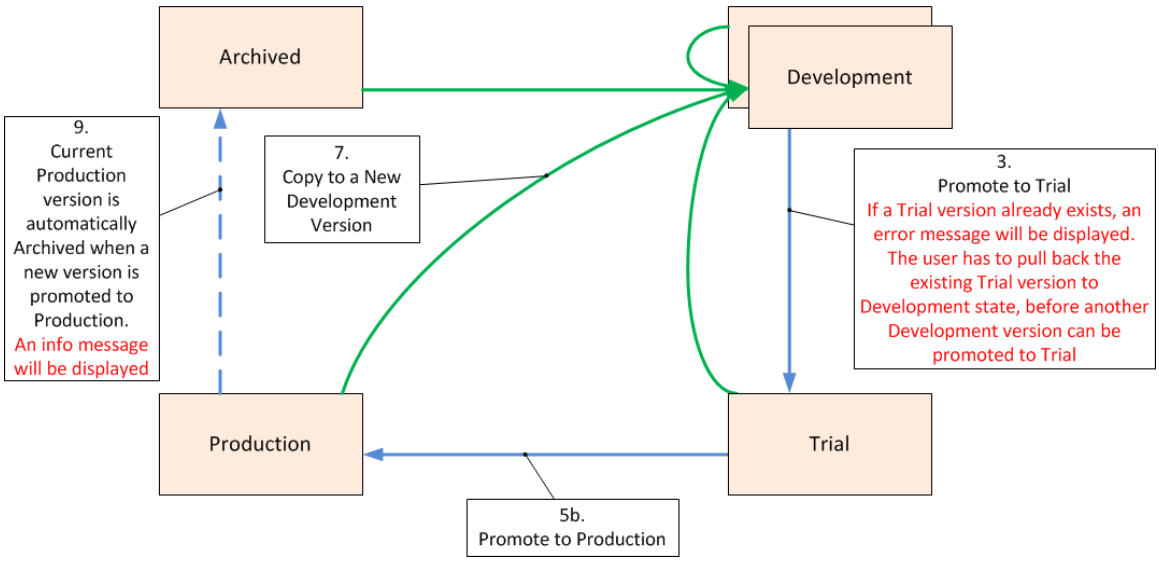


Figure 18: Creating a New DCA App Version

The DCA App Lifecycle management is done via the **Main Menu: DCA Framework→<DCA App Name>→Application Control** screen.

Each DCA app version can be in one of the following states:

- **Development** (initial state)
  - There are zero or more Development versions in the system.
  - Development version is not executed on any MP.
  - Configuration schema (databases), configuration data, flowchart may be updated.
  - A new version in Development state is created in the system when:
    - **Create New Development** is clicked, see Section 9.2.5. In this case, the version has an empty flowchart, empty configuration schema, and empty configuration data.
    - Importing the business logic (with or without configuration data), see Section 9.2.9. In this case the flowchart and the configuration schema (databases) is copied from the imported version. Optionally, configuration data may be imported along with the business logic as well.
    - Copying a new Development version from an existing version in the system, see Section 9.2.7. In this case, the business logic and the configuration data of the selected version are copied into the new version.
- **Trial**
  - There are zero or one Trial versions in the system.
  - Trial version is executed on the DA-MPs assigned to run the Trial version
  - If no Trial version exists, then the Trial MPs runs the Production version (see Figure 19).
  - Configuration schema (databases), configuration data, flowchart may be updated.
- **Production**
  - There is zero or one Production version in the system.
  - When no Production version exists in the system, the operational state of the DCA application on MPs supposed to run the Production version is set to Unavailable (**Main Menu: Diameter→Maintenance→Applications)**. This may happen if the Production version is rolled back to the Development state or deleted.
  - Is executed:
    - On all the DA-MPs, if no Trial version exists, or
    - On all the DA-MPs except the DA-MPs assigned to run the Trial version, if a Trial version exists (see Figure 19).
  - Configuration schema (databases) & Flowchart are read-only.
  - Configuration data may be updated.
- **Archived**
  - There are zero or more Archived versions in the system.
  - Archived versions are the application versions that have previously been in the Production state. They serve as backups for bringing the system back to a previous known state with minimum service interruption.
  - Archived version is not executed on any MP.
  - Configuration schema (databases), Configuration Data and Flowchart are read-only, but can be exported and copied into a new version.

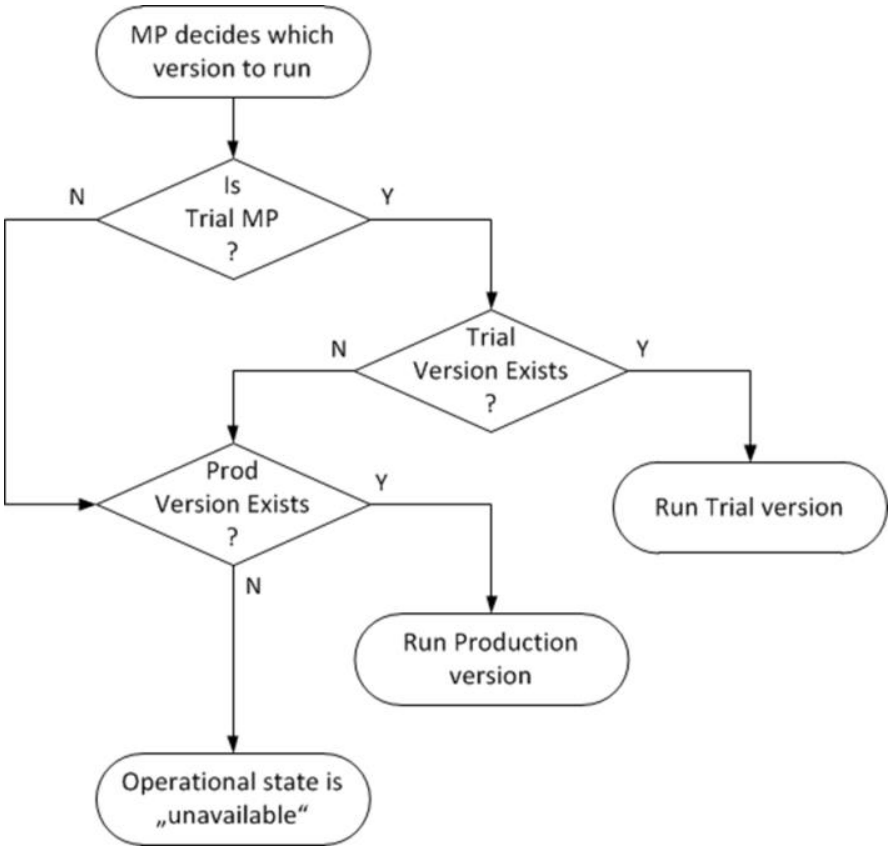


Figure 19: Assignment of the Version to a DA-MP

The following transitions are possible for a given DCA App version:

- Development → Trial (only if syntax was successfully checked and no other version is in Trial state)
- Trial → Production (only if the code/flow control chart was successfully compiled and no other version is in Production state)
- Production → Archived (automatic transition when a new version is promoted to Production)
- Trial → Development
- Production → Development (the operational state of the DCA App becomes Unavailable)
- Archived → Development
- Archived → Trial
- Archived → Production

5. Developing Stateful DCA Apps

The Blacklist DCA App introduced in Chapter 3 was a stateless Diameter application because it was processing each Diameter message individually without maintaining any state between a Diameter request and its corresponding answer (Diameter transaction state) or across Diameter transactions (e.g., Diameter session state) or across Diameter sessions (e.g., user state).

DCA Apps may, however, need to store state:

1. Diameter transaction state – for instance collect some information from the Diameter request and use that information when processing the Diameter answer.

This task can be addressed in two ways:

a. Using the Diameter transaction context variables API documented in Section 11.2.2.

b. Developers familiar with the Internal Variables from the Mediation feature may use Internal Variables for this purpose, as described in Section 11.2.1. However, Internal Variables involve a configuration overhead and therefore unless there is a strong argument in favor of using them (e.g., they need to be set or read from Mediation rules) the Diameter transaction context variables, being a purely programming interface, are preferable

2. Diameter session or user state – for instance collect information across multiple Diameter transactions in the same session or user information across multiple Diameter sessions.

   This task can be addressed using the Universal Session Binding Repository (U-SBR) and is described in Section 11.7.

6. A Stateful DCA App Using the U-SBR Infrastructure

In Chapter 3 we developed a stateless DCA App. Chapter 5 introduces the mechanisms available in DCA to develop stateful DCA Apps.

This chapter describes the additional configuration steps to perform, and introduces the API available to develop a stateful DCA App that uses the U-SBR (Universal Session Binding Repository). The U-SBR provides a generic interface to the I-SBR (Independent Session Binding Repository), which implements a scalable, distributed, and persistent database infrastructure, which DCA Apps and other Oracle applications may use.

6.1 The CountULR DCA App

The CountULR DCA App maintains a per-user count of ULR messages and deletes it when a CLR message from the respective user is received. The user is identified based on the content of the User-Name AVP in the incoming Diameter requests.

6.2 Prerequisites

The DCA framework must have been previously activated as described in [1] CGBU_018429 - DCA Framework and Application Activation and Deactivation. Also, a DCA App with the name CountULR is activated as described in [1] CGBU_018429 - DCA Framework and Application Activation and Deactivation.

The CountULR DCA App has to be enabled on all the DA-MPs in the network from the SO **Main Menu: Diameter→Maintenance→Applications**.

An ART rule is added that enables ULR and CLR Diameter requests to be delivered to the CountULR DCA App.

6.3 The Process

The following steps must be followed to provision the CountULR DCA App:

Business Logic and Configuration Data Provisioning	U-SBR DB Configuration
Step 1: Configure the general options and behavior of the CountULR DCA App. **Step 2**: Create a new development version of the CountULR DCA App.	**Step A**: Configure one or more U-SBR DBs (as required by the DCA App business logic).
Step 3: Define the structure of tables to store the CountULR configuration data. **Step 4**: Provision the CountULR configuration data. **Step 5**: Provision the CountULR business logic – essentially a Perl script. **Step 6**: Render the Flow Control Chart based on the Perl script. Save and perform syntax checks.	**Step B**: Configure a logical-to-physical U-SBR DB mapping
Step 7: Test the CountULR DCA App: configure the Trial DA-MPs and promote CountULR to Trial state. **Step 8**: Compile CountULR, promote CountULR to Production state.	

Steps 1 to 8 are similar to those described in Chapter 3.

Steps A and B are required to create an U-SBR DB and allow the CountULR DCA App to interact with it. U-SBR DB configuration is independent from the DCA App configuration, except that a relative ordering must be followed:

- Step A may be executed in any order relative to steps 1 and 2.
- Step B must follow step 2 because a logical-to-physical mapping is always associated with a DCA App version.
- Step B may be executed in any order relative to steps 3 to 6;
- Step 7 must follow step B.

A valid execution sequence is: Steps 1, 2→A→3, 4, 5, 6→B→7, 8.

6.3.1 Step 1: Configure the DCA App's Global Options and Behavior

In addition to the considerations discussed in Section 3.3.1, for DCA Apps that use U-SBR, the following configuration options may need to be adjusted:

- On the NO screen **Main Menu: DCA Framework→<DCA App Name>→General Options** (see Section 9.2.3):
  - Application State Data TTL, which defines the time interval after which the state data is considered expired and is deleted by the U-SBR audit mechanism. The lifetime of the state data is initialized to TTL when created and is then automatically extended with the TTL value each time the state data is updated. The lifetime of the state data depends on the business logic that the DCA App implements and as a rule of thumb, it is twice the expected validity period of the state data stored. For instance, if a DCA App is supposed to reject, under some specific circumstances, an user's Diameter requests for a certain time interval, then the double of this time interval would be a good value for the state data lifetime

  - Read-Only U-SBR Access as Guest, which may be used to control the access of the DCA App to U-SBR DBs owned by other DCA Apps. This option is not relevant to CountULR because CountULR exclusively uses the U-SBR DB owned by itself (see Section 6.3.3.5)
- It is recommended the state data size (consisting of the size of the lookup key and respectively the size of the state data itself) of any new DCA App to be kept below the default values configured on the **NO Main Menu: DCA Framework→Configuration** screen (see Section 9.2.1). If, for good reasons, a DCA App requires a larger lookup key or more data to store, then these limits are increased.

  Note that these limits apply globally to all active DCA Apps. As a result, decreasing these value may result in existing DCA Apps having their U-SBR queries rejected with a `dca::sbr::ResultCode::MaxStateSize` error, and is, therefore, not recommended.

6.3.2 Step 2: Create a New Development Version

See Section 3.3.2.

6.3.3 Step A: Configure the U-SBR DBs

Configuring a U-SBR DB must be preceded by configuring the underlying I-SBR topology:

- Configure the I-SBR topology
  - **Step A.1**: Servers Configuration
  - **Step A.2**: Server Groups Configuration
  - **Step A.3**: Places Configuration
  - **Step A.4**: Place Associations Configuration
  - **Step A.5**: Resource Domains Configuration
- Configure the U-SBR DB
  - **Step A.6**: U-SBR Database Configuration

The configuration of the I-SBR topology and SBR Databases is also described in more detail in [2] E58954-02, DSR Software Installation and Configuration Procedure.

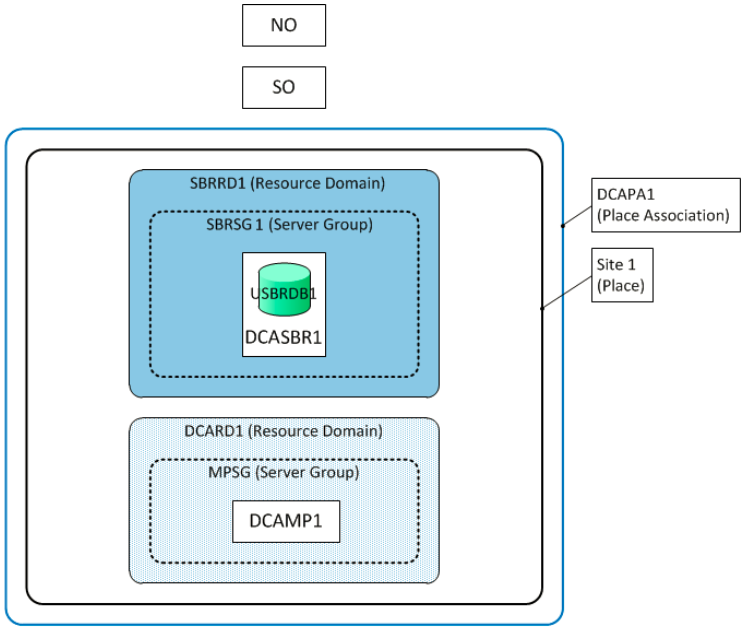


Figure 20: SBR Topology Example

The CountULR DCA App uses a simple I-SBR topology, illustrated in Figure 20.

The topology consists of a DA-MP (DCAMP1), which processes the Diameter messages; a SBR-MP (DCASBR1), which stores the U-SBR DB (USBRDB1); and a NO and a SO.

Next, we have:

- One server per each server group (DCAMP1 in MPSG and DCASBR1 in SBRSG1)
- One server group per resource domain (MPSG in DCARD1 and SBRSG1 in SBRRD1)
- Both resource domains are in the same place (Site1)
- The Place Association (DCAPA1) includes just one Place (Site1)

6.3.3.1 Step A.1: Servers Configuration

Servers are the processing units of the application with various roles within the application: Network OAM&P (NOAMP), System OAM (SOAM), and MP. For our case, we would need to configure two MPs – one for the DA-MP that processes the Diameter messages and one for the SBR-MP.

On the Servers screen, configure the SBR MP DCASBR1 with the MP Role and the DA-MP DCAMP1, see Figure 21.

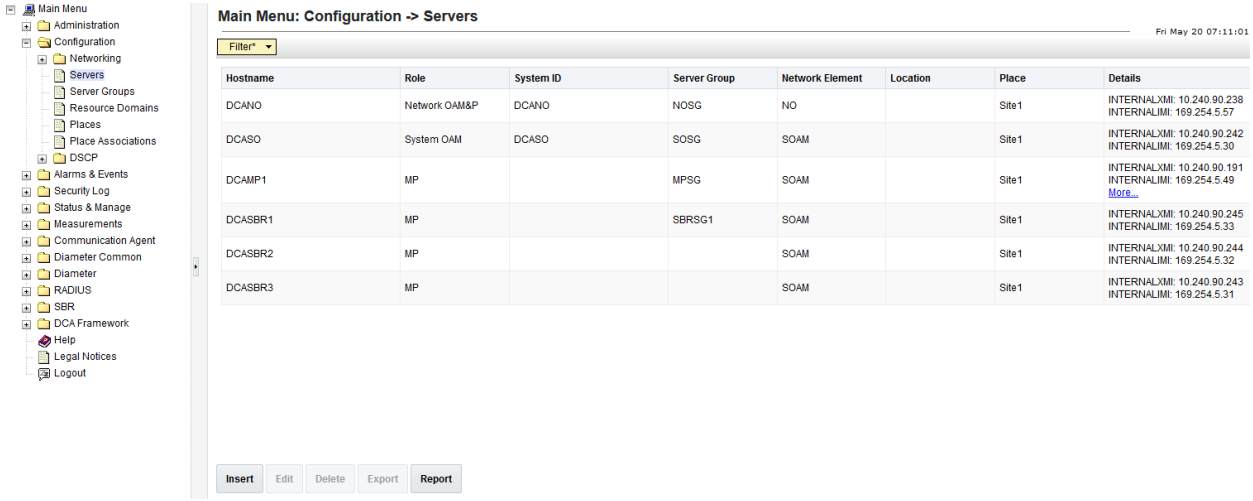


Figure 21: Servers Configuration

6.3.3.2 Step A.2: Server Group Configuration

The Server Groups allow the user to assign a function (DSR, SBR, etc.), parent relationships, and levels to a group of servers that share the same role, such as NOAMP, SOAM, and MP servers.

On the Server Group screen, configure the new SBR Server Group SBRSG1 that includes the DCASBR1 server and has SBR function, see Figure 22. Assign the Parent Relationship and Level C to a group of servers that share the SBR Role. Configure the DA-MP MPSG Server Group that includes DCAMP1 server and has the function of DSR.

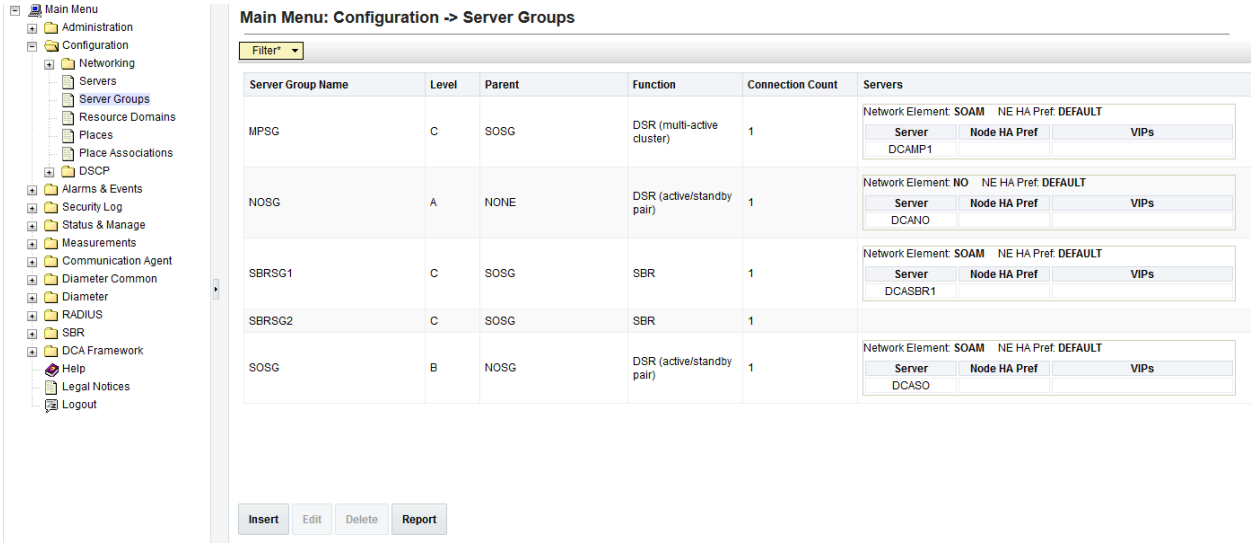


Figure 22: Server Groups Configuration

6.3.3.3 Step A.3: Places Configuration

The Places allow building associations for groups of servers at a single geographic location. These places can then be grouped into place associations, which create relationships between one or more place.

On the Places screen configure a new place Site1. Set a unique instance name, a Place Type Site, and a group of server members belonging to the site. For our example, all available servers are in the same place, see Figure 23 and Figure 24.

A Place Type is always Site.

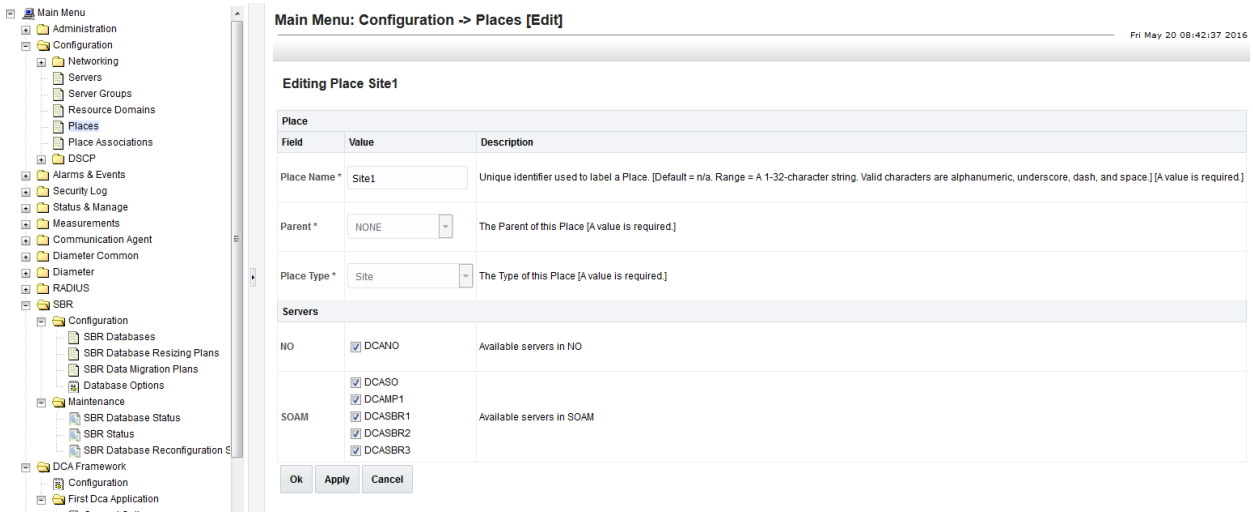


Figure 23: Places Configuration

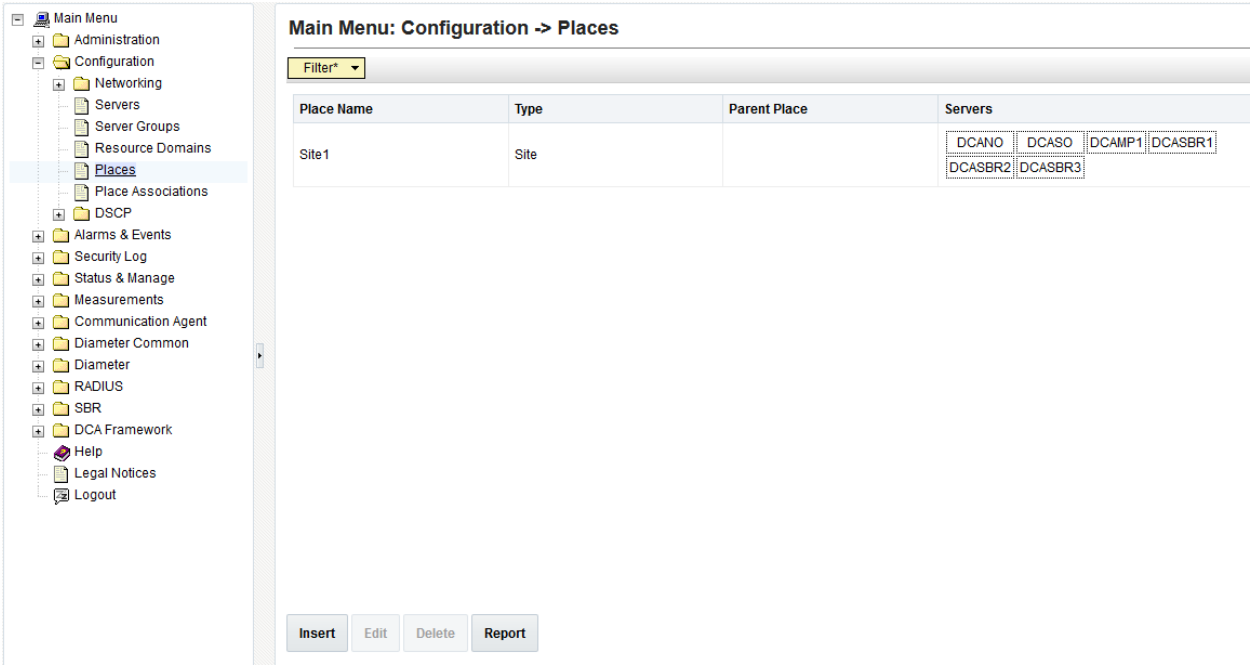


Figure 24: View Places

6.3.3.4 Step A.4: Place Associations Configuration

The Place Association function allows you to create relationships between places. Places are groups of servers at a single geographic location.

On the Place Associations screen, create the new place association DCAPA1 that includes Site1. Select the Place Association Type Applications Region, see Figure 25 and Figure 26.

Always select Applications Region type for the DCA applications and the SBR databases they use.

The Place Association in the SBR Databases configuration defines the scope of Database users. The database in the associated Place Association can only be accessed by the DA-MPs in the same Place Association.

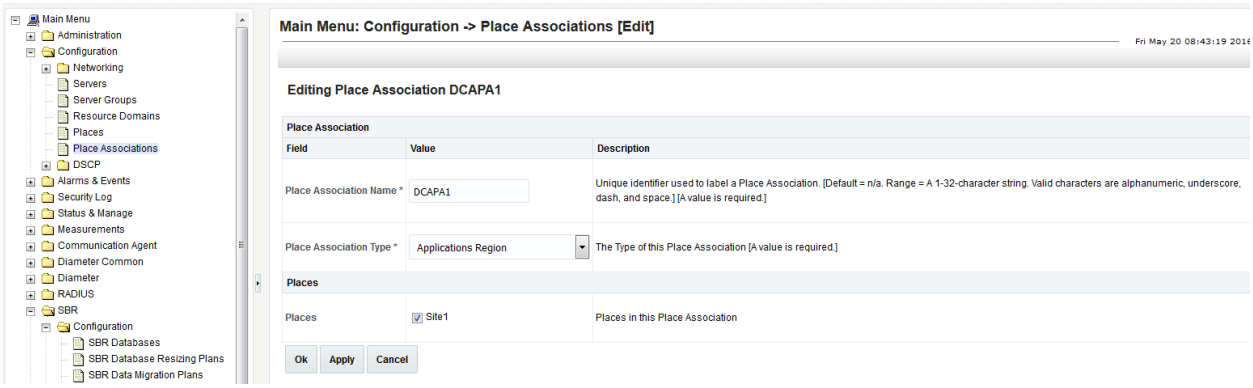


Figure 25: Create Place Association

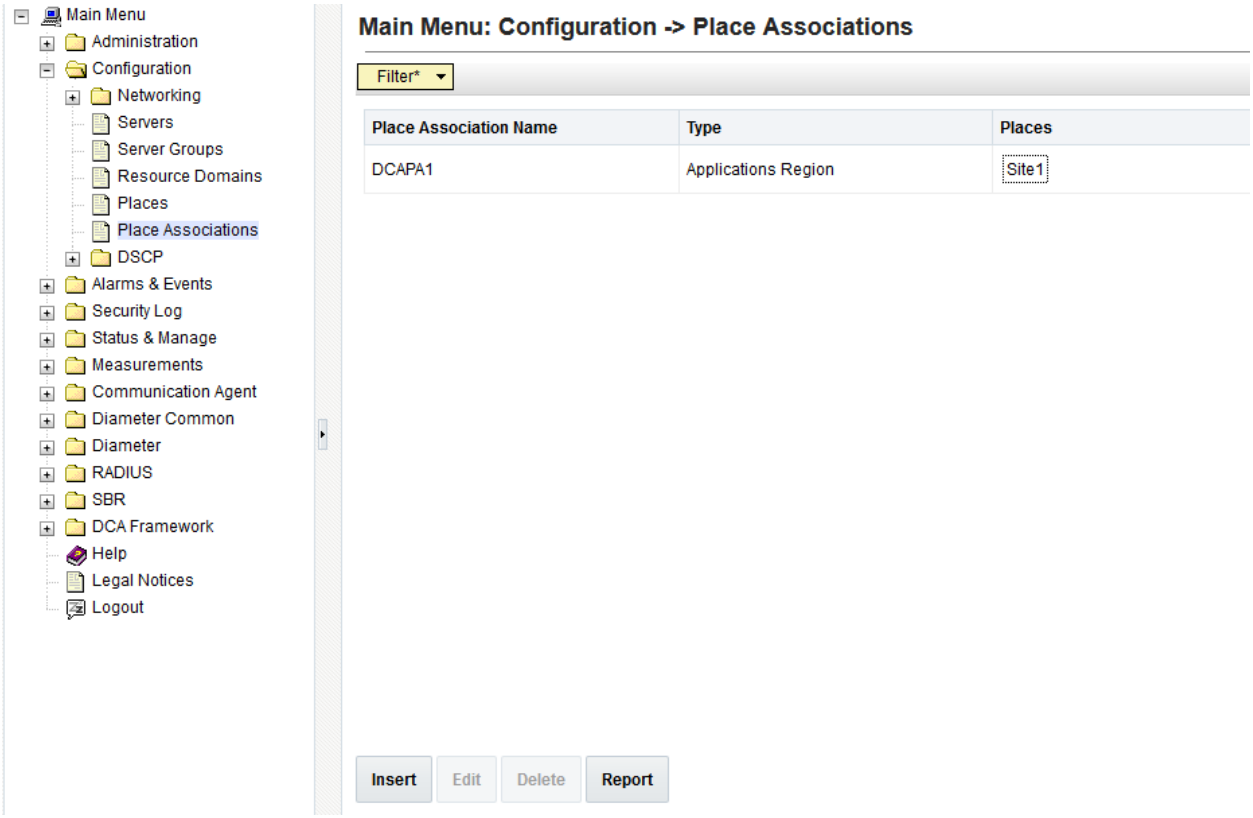


Figure 26: View Place Association

6.3.3.5 Step A.5: Resource Domain Configuration

The Resource Domains (RD) screen enables users to assign a set of Server Groups to a Resource Domain profile, which identified the database type.

On the Resource Domains screen configure the new resource domain SBRRD1. Assign the Server Group SBRSG1, select the Resource Domain Profile Session Binding Repository for the SBR database (see Figure 27 and Figure 29).

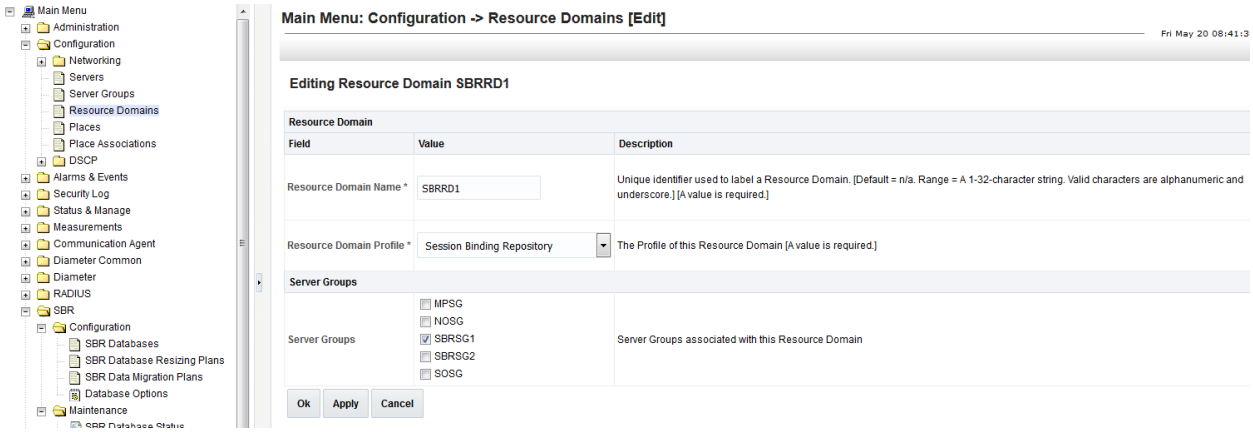


Figure 27: SBR Resource Domain Configuration

On the Resource Domains screen, configure the new resource domain DCARD1. Assign a Server Group MPSG, select the Resource Domain Profile DCA Application MPs (see Figure 28 and Figure 29).

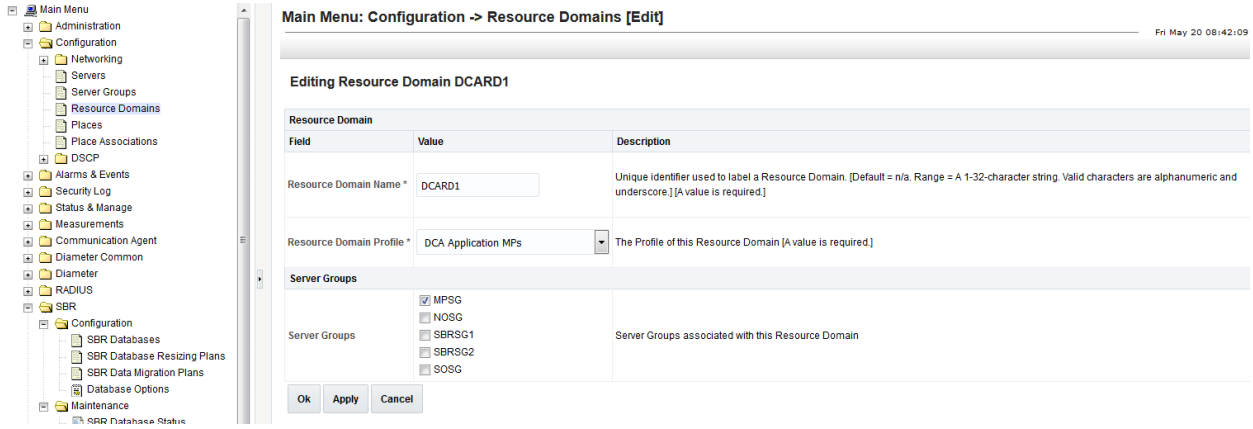


Figure 28: DCA Application MP Resource Domain Configuration

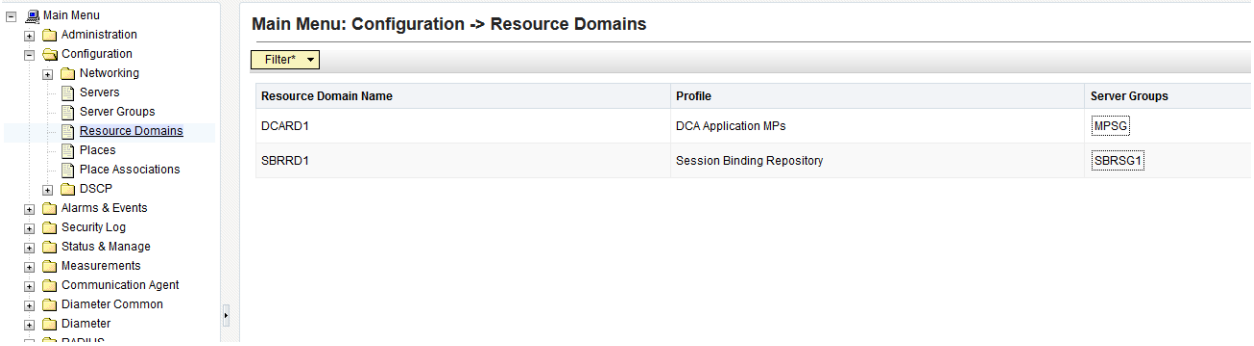


Figure 29: View Resource Domain Configuration

6.3.3.6 Step A.6: SBR Database Configuration

On the SBR Databases screen, create the new database USBR1 (see Figure 30).

Select Database Type: Universal, Resource Domain: SBRRD1, Number of Service Groups: 1, Place Association: DCAPA1, Owner Application: CountULR, as illustrated in Figure 31.

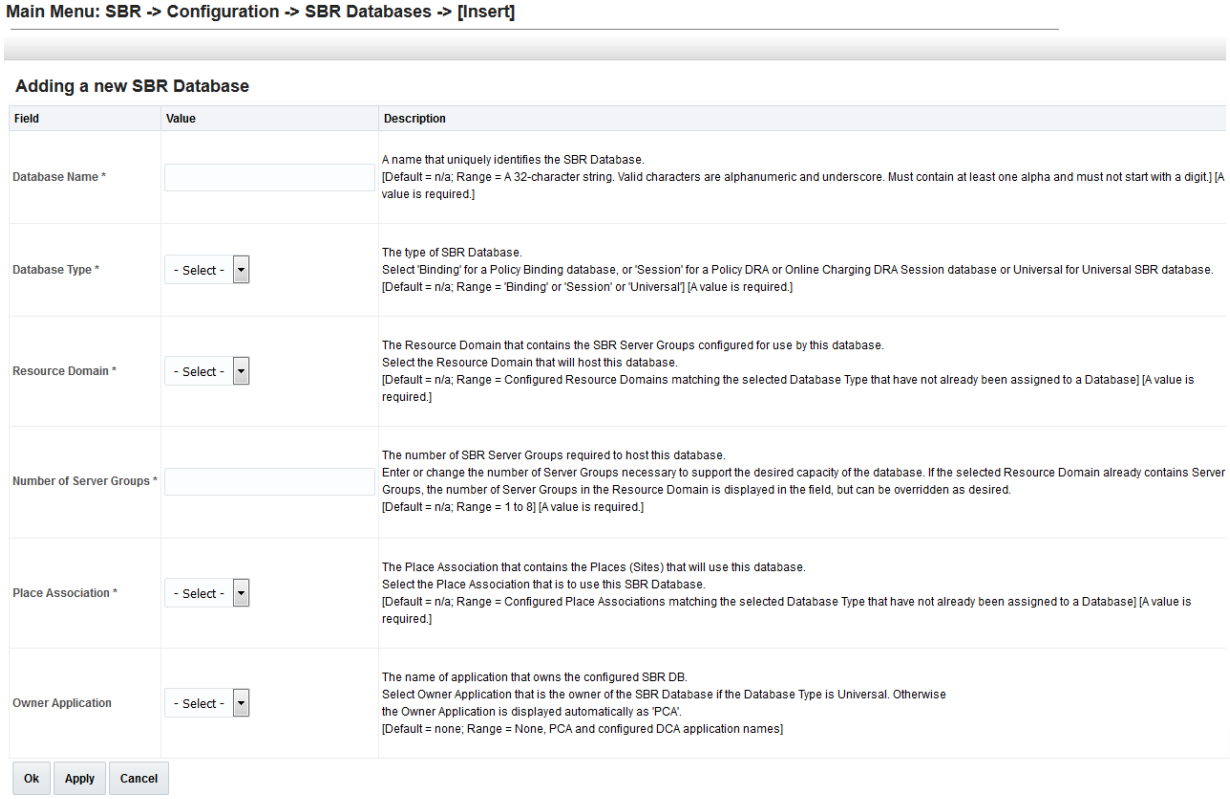


Figure 30: Create SBR Database

Each U-SBR DB is assigned to an owner DCA App. This is necessary for the U-SBR to support multiple DCA Apps (i.e., the owner DCA App and an arbitrary number of guest DCA Apps) to query the same U-SBR DB. The owner DCA App can perform all possible queries on the U-SBR DB. Guest DCA Apps on the other hand can restrict their access to read-only access to the U-SBR DB by checking the Read

Only U-SBR Access as Guest option on the **Main Menu: DCA Framework→<DCA App Name>→General Options** (see Section 9.2.3).

The U-SBR DB configured in a Place Association is only accessed by the DA-MPs in the same Place Association.

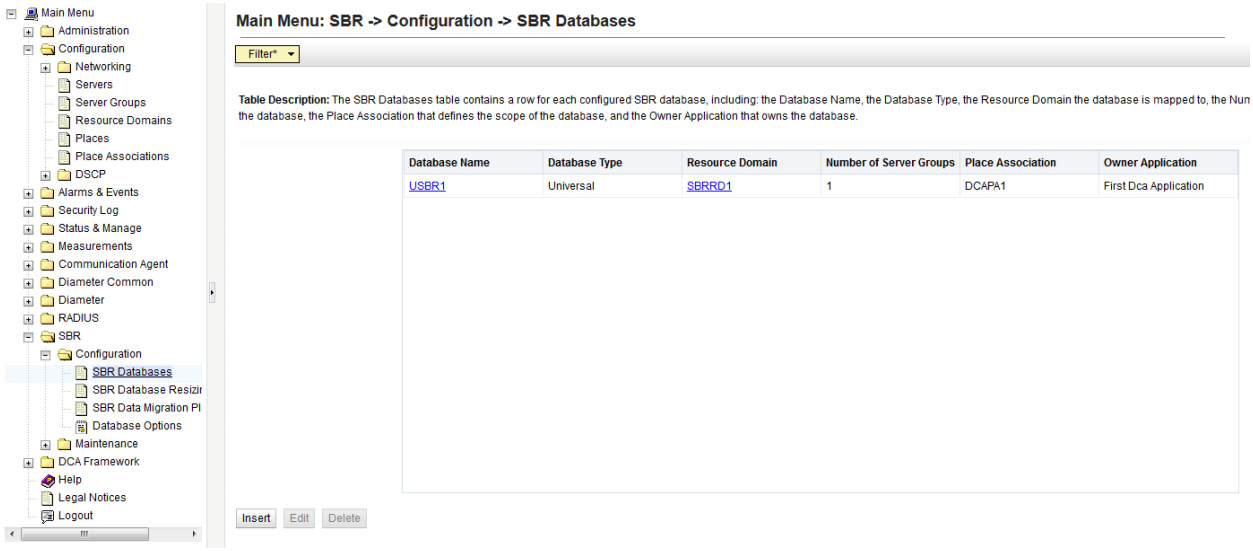


Figure 31: View SBR Database

From the NO Main Menu→SBR→Maintenance→SBR Database Status, prepare and enable the USBR1 database.

6.3.4 Step 3: Define the Configuration Data Schema

CountULR does not use any DCA App configuration data.

6.3.5 Step 4: Provision the Configuration Data

CountULR does not use any DCA App configuration data.

6.3.6 Step 5: Provision the DCA App Business Logic

The CountULR DCA App implements the following business logic:

- When receiving a ULR message, extract the user name from the User-Name AVP and check if a state has been created for the respective user:
  - If the user name is not found, create a state that contains a counter set to 1.
  - If the user name already exists, read the existing state, increment the counter and write the state back to the U-SBR DB.
- When receiving a CLR message, extract the user name from the User-Name AVP and delete the state corresponding to the respective user, if it exists.

Figure 32 illustrates a typical call flow. CountULR uses three U-SBR API calls: createOrRead, concurrentUpdate and delete. The U-SBR API is described in Section 11.7.

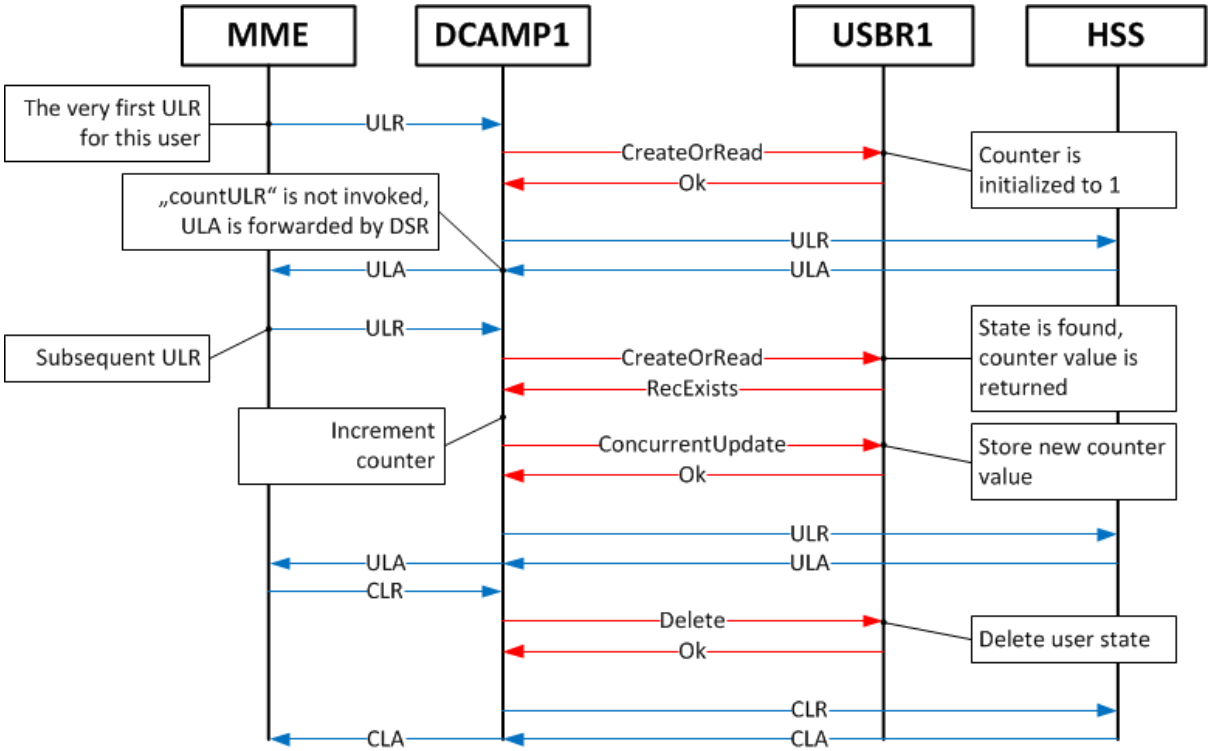


Figure 32: CountULR Call Flow

The Perl code is illustrated in Figure 25.

```
use constant{
      # key types for our app – only NAI is currently used,
      # the others are for exemplification
      IMSI => 0,
      SESSION => 1,
      NAI => 2,
      IPv4 => 3,
      # command codes for S6 commands
      ULR_CMD => 316,
      CLR_CMD => 317,
};

# this function is called when receiving a diameter request
# message
sub process_request{
      # session state to be stored on the sbr
      # the session state stores:
      #  - no of requests for this user-name
      #  - no of success replies for this user-name
      #  - no of error replies for this user-name
      my $sbr_state =
      {
```

```
        requests => 1           # only requests are currently counted
        #ok_replies => 0,
        #err_replies => 0
};

# diameter message is the first parameter
my $param = shift;
# only one key type for this app: NAI
my $key_type = NAI;
# get the diameter message object
my $msg = diameter::Param::message($param);
if(!defined($msg)){
        die "Bad diameter message parameter.";
}

# try to get the the diameter command code from the diameter message
my $cmd = diameter::Message::commandCode($msg);
if(!defined($cmd)){
        die "No command code in diameter message.";
}

# get User-Name from the message
my $user = diameter::Message::getAvpValue($msg,"User-Name");
if(!defined($user)){
        # could not create $user
        die "Could not get the User-Name value from the message"
}

if(ULR_CMD == $cmd){
        # process Update-Location-Request
        # instantiate and send the "CreateOrRead" SBR stack event
        my $result = dca::sbr::sbrInstance("sbr")->createOrRead(
                $key_type,
                dca::sbr::KeyDataType::STRING, $user,
                dca::sbr::StateDataType::STRING, $sbr_state,
                "createOrReadCb");
        # check the "synchronous" error
        if(!defined($result)){
                # could not create the sbr request
                die "could not create the SBR request";
        }
}

elsif(CLR_CMD == $cmd){
        # process Cancel-Location-Request
        # instantiate and send the "Delete" SBR stack event
        my $result = dca::sbr::sbrInstance("sbr")->delete($key_type,
                dca::sbr::KeyDataType::STRING, $user,
                "deleteCb");
```

```
            # check the "synchronous" error
            if(!defined($result)){
                  # could not create the sbr request
                  die "could not create the SBR request";
            }
      }
      else{
            die "unknown diameter command received";
      }
}

# this function is called when receiving a diameter answer
# message
sub process_answer{

}

# this function is called when receiving an DeleteStateResult
# answer from the SBR
sub deleteCb{
      my $sbr_code = dca::sbr::result()->code();
      if(!defined($sbr_code)){
            # could not get the result code of the SBR answer
            die "did not get the result code of SBR answer";
      }

      if(dca::sbr::ResultCode::RecNotFound == $sbr_code){
            die "could not find a record with the given key on the SBR";
      }
      elsif( dca::sbr::ResultCode::Ok != $sbr_code){
            die "SBR error: $sbr_code";
      }
}

# this function is called when receiving an CreateOrReadStateResult
# answer from the SBR
sub createOrReadCb
{
      my $sbr_code = dca::sbr::result()->code();
      # check the result code
      if(dca::sbr::ResultCode::RecExists == $sbr_code){
            my $sbr_state = dca::sbr::result()->data();

            # diameter message is the first parameter
            my $param = shift;
            # only one key type for this app: NAI
            my $key_type = NAI;
            # get the diameter message object
            my $msg = diameter::Param::message($param);
```

```
            if(!defined($msg)){
                  die "Bad diameter message parameter.";
            }

            # get User-Name from the message
            my $user = diameter::Message::getAvpValue($msg,"User-Name");
            if(!defined($user)){
                  # could not create $user
                  die "Could not get the User-Name value from the message"
            }

            # record was already existing on the SBR; update it
            $sbr_state->{requests}++;
            my $result = dca::sbr::sbrInstance("sbr")->concurrentUpdate(
                  $key_type,
                  dca::sbr::KeyDataType::STRING, $user,
                  dca::sbr::StateDataType::STRING, $sbr_state,
                  "concurrentUpdateCb");
            # check the error
            if(!defined($result)){
                  # could not create the sbr request
                  die "could not create the SBR request";
            }
      }
      elsif( dca::sbr::ResultCode::Ok != $sbr_code){
            die "SBR error: $sbr_code";
      }
}

# this function is called when receiving an ConcurrentUpdateStateResult
# answer from the SBR
sub concurrentUpdateCb{
      # use "result" API function to retrieve error code and data:
      my $sbr_code = dca::sbr::result()->code();

      # check the result code
      if(dca::sbr::ResultCode::RecObsoleted == $sbr_code){
            # record was already updated by another MP on the SBR;
            # try to update it once again
            my $sbr_state = dca::sbr::result()->data();
            # diameter message is the first parameter
            my $param = shift;
            # only one key type for this app: NAI
            my $key_type = NAI;
            # get the diameter message object
            my $msg = diameter::Param::message($param);
            if(!defined($msg)){
                  die "Bad diameter message parameter.";
            }
```

```
            # get User-Name from the message
            my $user = diameter::Message::getAvpValue($msg,"User-Name");
            if(!defined($user)){
                  # could not create $user
                  die "Could not get the User-Name value from the message"
            }
            $sbr_state->{requests}++;
            my $result = dca::sbr::sbrInstance("sbr")->concurrentUpdate(
                  $key_type,
                  dca::sbr::KeyDataType::STRING, $user,
                  dca::sbr::KeyDataType::STRING, $sbr_state,
                  "concurrentUpdateCb");
            # check the error
            if(!defined($result)){
                  # could not create the sbr request
                  die "could not create the SBR request";
            }
      }
      elsif( dca::sbr::ResultCode::Ok != $sbr_code){
            die "SBR error: $sbr_code";
      }
}
```

Figure 33: CountULR Perl Code

6.3.6.1 What Does a State Consist Of?

A state is essentially a mapping between a Key and a Value. What exactly the Key and Value are is completely under the DCA App's control. The U-SBR does not attach any semantics to a DCA App state. In CountULR the Key is the user name extracted from the User-Name AVP and the Value is basically a counter that counts the total number of ULR messages.

Even though CountULR uses a single Key (of type NAI), DCA Apps may, in general, use multiple Keys (IMSI, MSISDN, IP addresses, Diameter Session-Id, etc.).

A DCA App may distinguish between the different Keys by declaring their Key Types. The Key Type helps avoid collisions like for instance between NAI key "fred" and IPv4 address key 66.72.65.64, or between IP source address key 1.2.3.4 and destination IP address key 1.2.3.4.

The Value associated to a Key is the value of a Perl variable. For CountULR, the Value is a Perl hash table containing one key requests that store an integer representing the ULR counter. Perl complex data structures like hash tables and arrays are converted to JSON and stored in the U-SBR DB as strings. When retrieved from the U-SBR they are converted back to the original data structure. Scalar Perl variables, on the other hand, need not undergo a JSON conversion.

Finally, the data type of Key and Value need to be specified to one of the pre-configured data types: dca::sbr::KeyDataType::BCD, dca::sbr::KeyDataType::UINT32, dca::sbr::KeyDataType::INT64, dca::sbr::KeyDataType::STRING, dca::sbr::KeyDataType::IPv4, dca::sbr::KeyDataType::IPv6, and respectively: dca::sbr::StateDataType::BCD, dca::sbr::StateDataType::UINT32, dca::sbr::StateDataType::STRING, dca::sbr::StateDataType::IPv4, dca::sbr::StateDataType::IPv6.

This helps the U-SBR DB to optimize the way the Key-Value pair is stored and retrieved.

6.3.6.2 What are Asynchronous API Calls and Callbacks?

The `dca::sbr::sbrInstance("sbr")→createOrRead`, `dca::sbr::sbrInstance("sbr")→concurrentUpdate` and `dca::sbr::sbrInstance("sbr")→delete` API functions initiate, each of them, an U-SBR DB query. They are **asynchronous** functions, in the sense that they do not wait until a response from the U-SBR is received. They construct the U-SBR DB query and return immediately, to allow the other Diameter messages to be processed. The query itself is sent after the event handler execution completes.

How can then the DCA App learn about the outcome of the U-SBR DB query it just sent? It may be observed that all the U-SBR API functions can register, as the last parameter, the name of a **callback** subroutine. The callback subroutine is invoked by the DCA framework when the outcome of the corresponding U-SBR DB query is known. The outcome may be: (i) an error condition that prevented the U-SBR query to even be sent; (ii) the U-SBR DB response itself; or (iii) an error condition indicating that no response has been received within a certain timeout interval.

6.3.6.3 How is the U-SBR State Returned to the Perl Script?

In the callback subroutine the DCA App programmer can use the `dca::sbr::result()` class to retrieve the error code and, if the query was successful, the result.

6.3.6.4 What is Concurrent in a concurrentUpdate?

Incrementing a counter in a distributed system is not as trivial an operation as it may seem because race conditions may occur between different threads, processes or hosts that attempt to increment the same counter at the same time.

In our case, such race condition may occur when ULR messages for the same user name are received at around the same time or in quick succession. This can obviously happen when the network contains multiple DA-MPs, but it can also happen in our simplified topology with one single DA-MP because there are always multiple Perl interpreters running simultaneously that execute the event handlers. There are therefore multiple CountULR execution instances, running in parallel, at any given time.

A CountULR execution instance is basically reading the counter value from the U-SBR DB record that corresponds to the user name, incrementing it and updating the record on the U-SBR, i.e., a read-increment-update sequence. The trick is to check that the record a CountULR execution instance is trying to update is the same record that was previously read. If it is not the same, then one or more other CountULR execution instances have incremented the counter in the meantime and the operation needs to be repeated on the new counter value, otherwise the counter value is corrupted and the counter value is, in the end, less than it should be.

When the concurrentUpdate query detects that the U-SBR DB record has been updated, it automatically returns the new record so that an explicit new read operation is not needed.

This mechanism is called optimistic offline locking and is often encountered in transactional DBs. It is optimistic because the rate of the race conditions is expected to be relative low compared to total number of increment operations. It is offline because the race condition is resolved by re-trying the operation, rather than effectively locking the record. Figure 34 illustrates such a race and how the optimistic offline locking mechanism solves it. For simplicity, we show two competing DA-MPs, however, as already mentioned, it equally applies to one single DA-MP running multiple Perl interpreters.

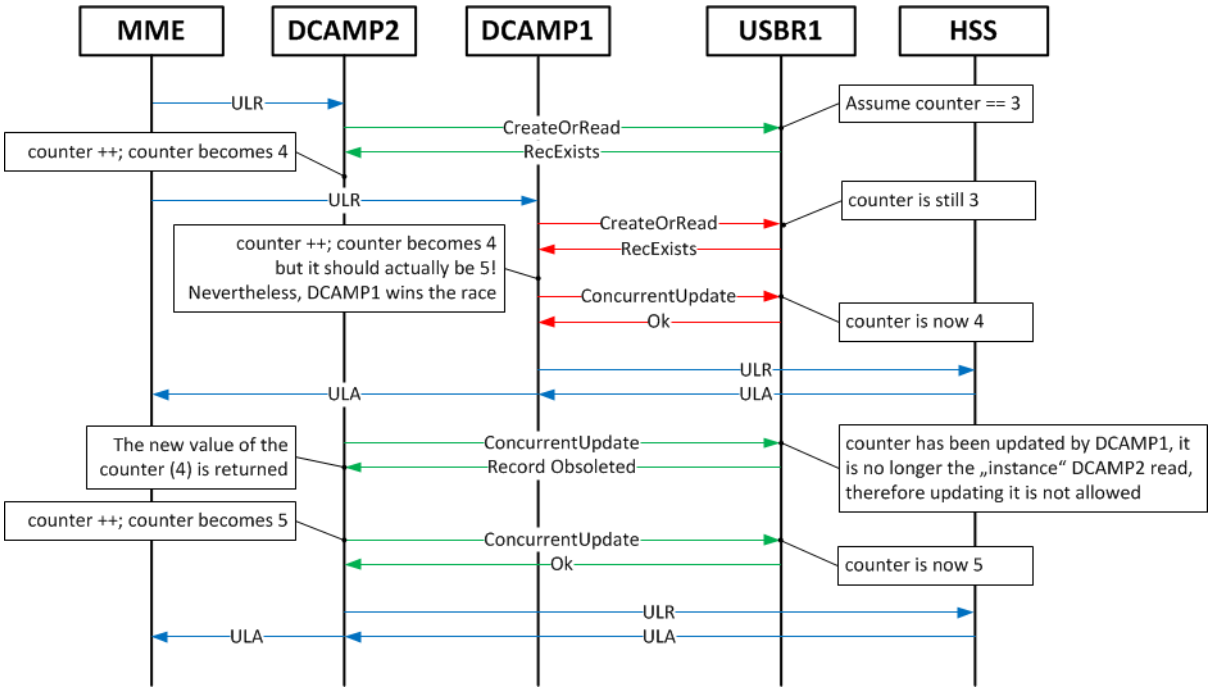


Figure 34: A Counter Increment Race

6.3.7 Step 6: Render the Flow Control Chart

Render the Flow Control Chart based on the Perl script. Save the code and check the syntax.

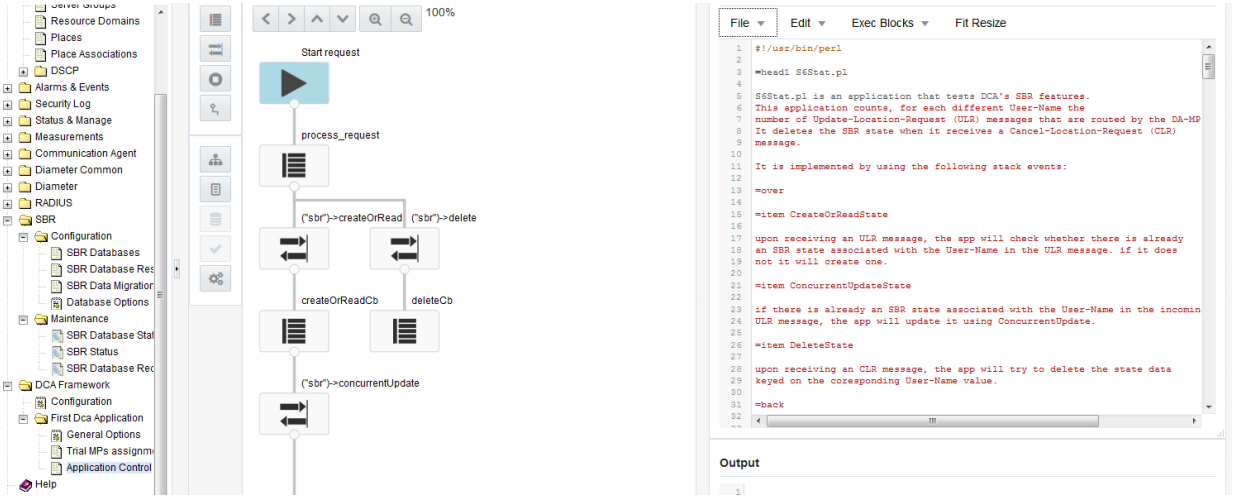


Figure 35: Flow Control Chart

6.3.8 Step B: Logical to Physical U-SBR DB Name Mapping

Logical-to-physical U-SBR name mapping provides the glue between the Perl script and the U-SBR topology. It enables:

- The Perl script to remain unchanged across deployments (Lab→Live, Oracle→Customer) by using the same logical names for the U-SBR DBs, while the topology and names of the physical U-SBR DBs in each particular network may vary.

- The Perl script to remain unchanged across Place Associations inside the same deployment, because the names of physical U-SBR DBs are different in each Place Association. This situation is not apparent in our example, because we are using a network that consists of only one site.
- Different versions of the same DCA App to use different logical names mapping to the same physical U-SBR DBs.
- Different versions of the same DCA App to use the same logical names mapping to different physical U-SBR DBs, because the DB layout (number of U-SBR DBs or their scope site vs. global) has changed in newer versions.
- A DCA App to map a logical U-SBR DB name to a physical U-SBR DBs of another DCA App.

The logical-to-physical U-SBR DB mapping is configured per DCA App version. In the Application Control screen, select a DCA App version and click the **SBR DB Name Mapping**, see Figure 36.

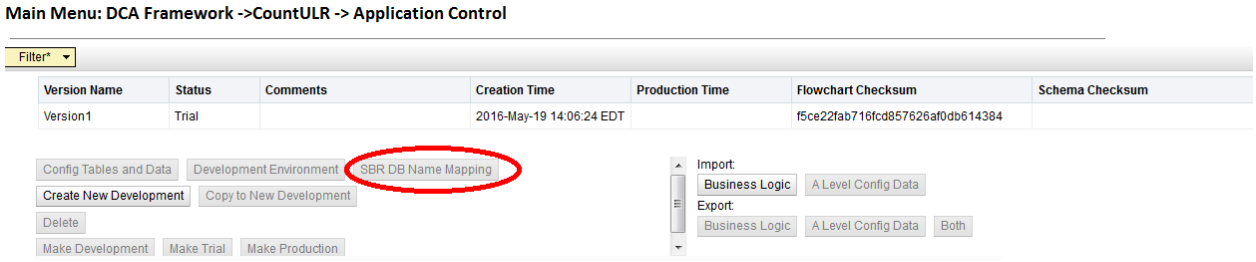


Figure 36: SBR DB Name Mapping

Assign the logical U-SBR name "sbr" to the physical U-SBR Name USBR1, see Figure 37.

The "sbr" name must be consistently used in the Perl script as a parameter to the sbrInstance() each time an U-SBR API function is invoked. As a result, the queries sent from the Perl script to "sbr" is delivered to the USBRDB1.

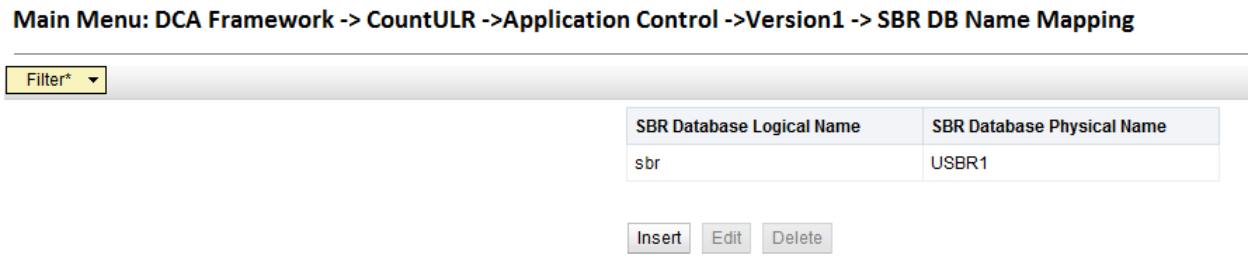


Figure 37: View SBR DB Name Mapping

6.3.9 Step 7: Test the DCA App Version

See Section 3.3.7.

6.3.10 Step 8: Promote the DCA App Version to Production

See Section 3.3.8.

7. Monitoring a DCA App

This chapter provides a general description of Custom MEALs, templates and their purpose. The monitoring of the execution of a DCA App is possible by means of the Custom MEAL feature.

The Custom MEAL feature enables a DCA App programmer to define and use measurements, KPIs, and events, on demand:

- Measurements are used to count specific events or amounts, as required by the DCA App's business logic. Their historical values measured during specific time intervals and/or on specific hosts are available via reports;
- KPIs display real-time statistics of the measured events or amounts, like for instance average values;
- Events may be triggered automatically when the currently measured values exceed the configured thresholds.

  Alternatively, events may be triggered explicitly from the DCA App code.

The Custom MEAL feature hides most of the complexity of the underlying DSR objects that implement the measurements, KPIs, and events by defining a number of four templates, which are designed to implement specific tasks:

- The Counter template – is used to count events. The counter values are available only off-line through the Measurement Reports.
- The Rate template – is most typically used to calculate message rates. It generates KPIs, Measurement Reports and may be used to automatically raise alarms if the configured threshold values are exceeded.
- The Basic template – is used to measure averages or number of elements in a set (e.g., to calculate average size of AVPs, messages or number of users registering/deregistering). It generates KPIs, Measurement Reports and may be used to automatically raise alarms if the configured threshold values are exceeded.
- The Event template – is used to explicitly raise/clear alarms or generate events from the Perl script when specific business logic conditions are detected.

Each of the templates is available in scalar and arrayed format.

We denote by "differentiation" the process of assigning a C-MEAL template instance to a DCA App. We denote by "un-differentiation" the reverse process of removing a C-MEAL from a DCA App and basically returning it to the pool of un-differentiated C-MEAL, from where it can be re-assigned to another (or even the same) DCA App.

8. A DCA App Using Custom MEALs

Chapter 7 introduced the Custom MEAL (C-MEAL) templates and their applicability. This chapter describes a simple DCA App that uses a Rate C-MEAL to monitor the rate of the incoming Diameter requests with just two lines of Perl code.

8.1 The Rate DCA App

The Rate DCA App differentiates a Rate C-MEAL, initializes it, and pegs it every time a Diameter request is received. The operator can monitor the incoming message rate in real time (KPI), check the history of the measured value (measurement report) and get notified when the configured thresholds are exceeded (alarm).

8.2 Prerequisites

The DCA Framework must have been previously activated as described in [1] CGBU_018429 - DCA Framework and Application Activation and Deactivation. Also, a DCA App with the name "Rate" is activated as described in [1] CGBU_018429 - DCA Framework and Application Activation and Deactivation.

The Rate DCA App has to be enabled on all the DA-MPs in the network from the SO **Main Menu: Diameter→Maintenance→Applications**.

An ART rule is added that enables Diameter requests to be delivered to the Rate DCA App.

8.3 The Process

The following steps must be followed to provision the Rate DCA App:

Business Logic and Configuration Data Provisioning	Custom MEAL Configuration
Step 1: Configure the general options and behavior of the Rate DCA App. **Step 2**: Create a new development version of the Rate DCA App. **Step 3**: Define the structure of tables to store the Rate configuration data. **Step 4**: Provision the Rate configuration data. **Step 5**: Provision the Rate business logic – essentially a Perl script. **Step 6**: Render the Flow Control Chart based on the Perl script. Save and perform syntax checks.	**Step I**: Differentiate a scalar Rate C-MEAL.
Step 7: Test the Rate DCA App: configure the Trial DA-MPs and promote Rate to Trial state. **Step 8**: Compile Rate, promote Rate to Production state.	

Steps 1 to 8 are similar to those described in Chapter 3. Step I is required to assign a C-MEAL to the Rate DCA App, which can be then be used via the C-MEAL API, which is described in Section 11.6.

Step I may be executed in any order relative to steps 1 to 5.

8.3.1 Step I: Differentiate a C-MEAL

C-MEALs are differentiated from the **Main Menu→DCA Framework→Rate→Custom MEALs** screen, by clicking **Insert**. For the Rate DCA App in particular, TestRate, a scalar rate C-MEAL, is differentiated (see Figure 38). TestRate raises an alarm when the configured thresholds are exceeded. The threshold values represent percentages from the 100% Threshold Value, which in our example is exactly 100.

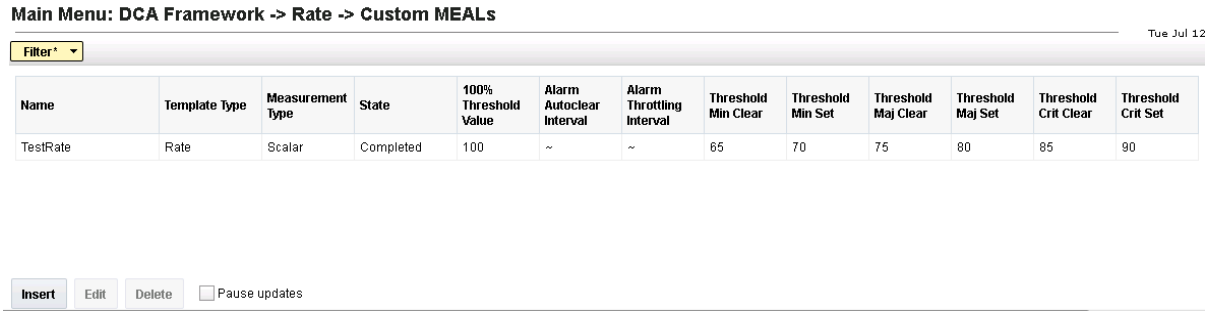


Figure 38: TestRate Differentiation

8.3.2 Step 1: Configure the DCA App's General Options and Behavior

See Section 3.3.1.

8.3.3 Step 2: Create a New Development Version

See Section 3.3.2

8.3.4 Step 3: Define the Configuration Data Schema

Rate does not need any DCA App configuration data.

8.3.5 Step 4: Provision the Configuration Data

Rate does not need any DCA App configuration data.

8.3.6 Step 5: Provision the DCA App Business Logic

The Rate DCA App implements a simple business logic that consists of pegging the Rate C-MEAL each time a Diameter request is received.

The Perl code is illustrated in Figure 39. Note that the C-MEAL name used to initialize the Perl object must be the same as the one configured for the C-MEAL during differentiation (TestRate).

```
my $rateObject = new dca::meal::rate("TestRate");
die "Failed to bind to the rate template" unless $rateObject;
                     # force *compilation* error if
                     # rateObject initialization fails

sub process_request{
    die "Pegging 'TestRate' Failed" unless $rateObject->peg();
                    # a *runtime* error will be generated in the unlikely
                    # event peg() fails
}

# And that's it! Alarms will be automatically raised when the configured
# thresholds are exceeded
```

Figure 39: The "Rate" DCA App Code

8.3.7 Step 6: Render the Flow Control Chart

The same process described in Section 3.3.6 is followed.

8.3.8 Step 7: Test the DCA App Version

The same process described in Section 3.3.7 is followed.

At this stage, we can finally monitor the Rate DCA App in the following ways:

- The DCA:Rate KPI group includes all the KPIs that belong to the Rate DCA App. In the **Main Menu→Status&Manage→KPIs** the DCA:Rate group is included in the KPI filter criteria (see Figure 40). As a result, the exponentially smoothened average of the ingress rate (TestRate) is displayed in real time (see Figure 41).

The history of the measured values can be accessed from the Main **Menu→Measurements→Report** screen. The DCA:Rate measurements group includes all the measurements that belong to the Rate DCA App and is included in the filtering criteria (see Figure 42). As a result, the history of the TestRate measurements is displayed (see Figure 43).

- An alarm with the corresponding severity is raised when the respective threshold values are exceeded. This can be seen for instance in Figure 41. The alarm details can be accessed from **Main Menu→Alarms&Events**. Figure 44 illustrates the alarm history, obtained by progressively increasing the message rate above the critical set threshold and then progressively reducing it below the minor clear threshold.

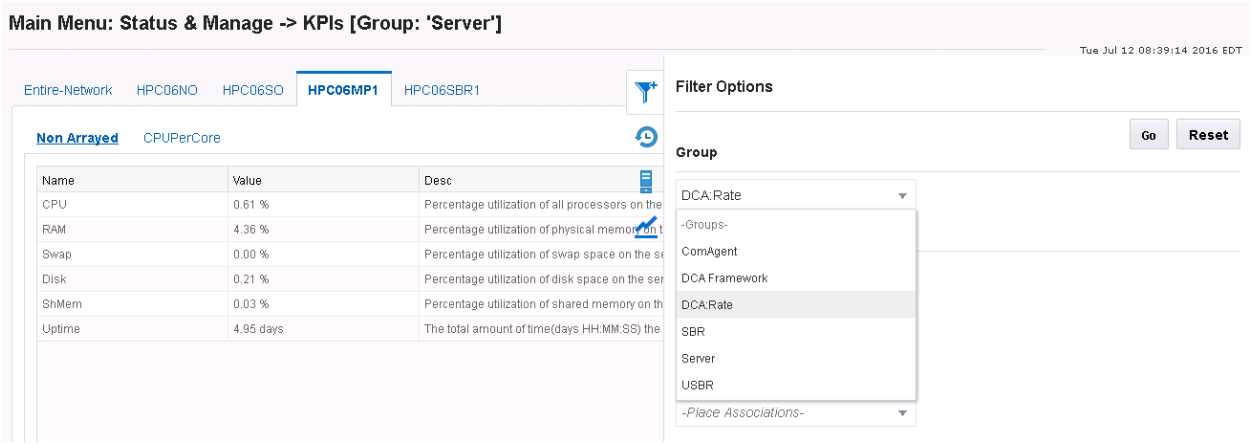


Figure 40: Filter the DCA:Rate KPIs

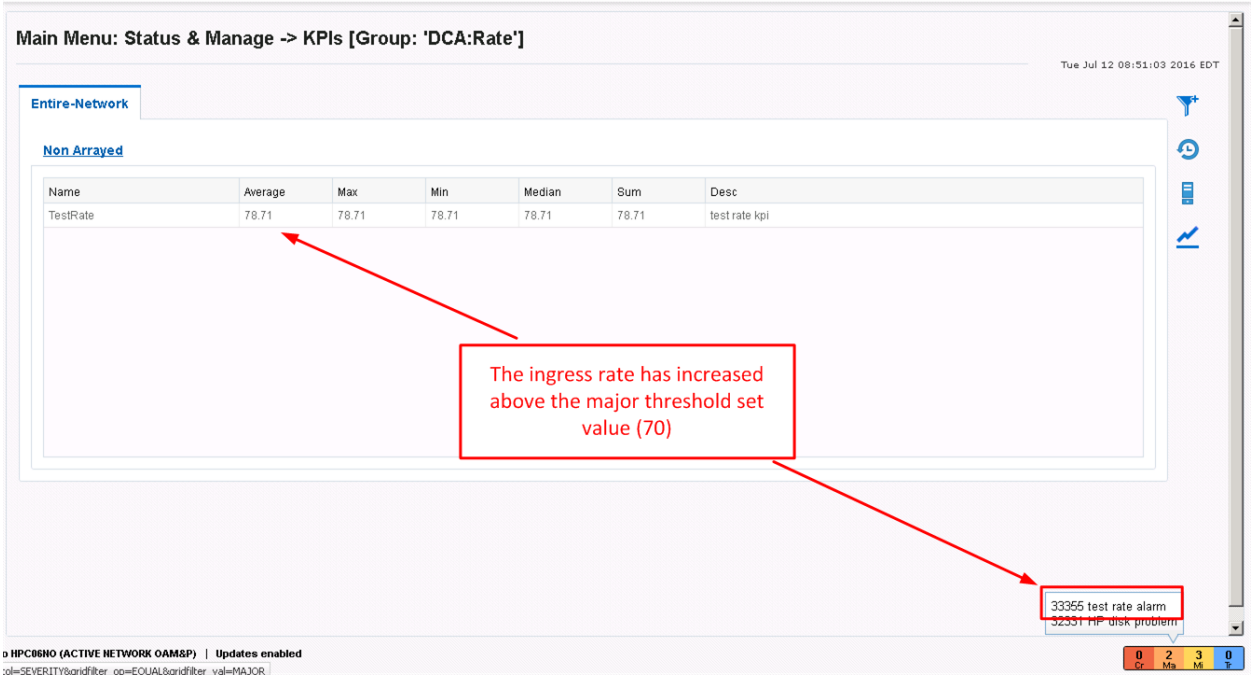


Figure 41: Display TestRate KPI

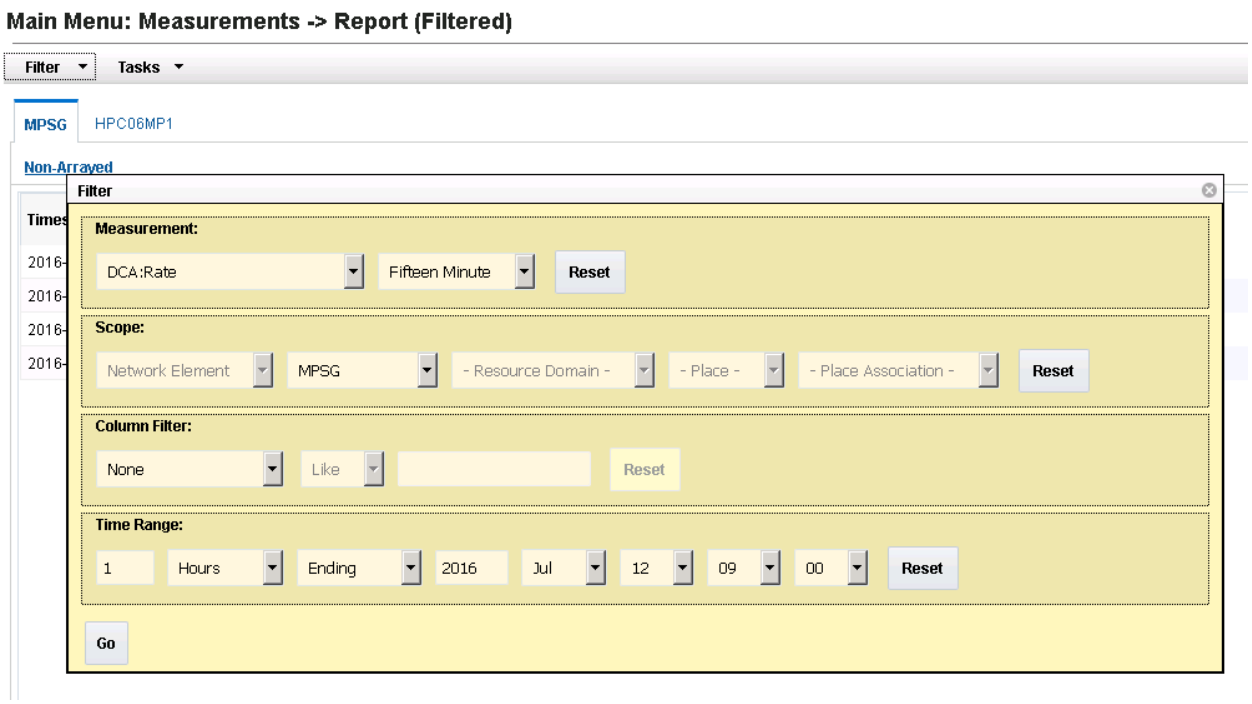


Figure 42: Filter the DCA:Rate Measurements

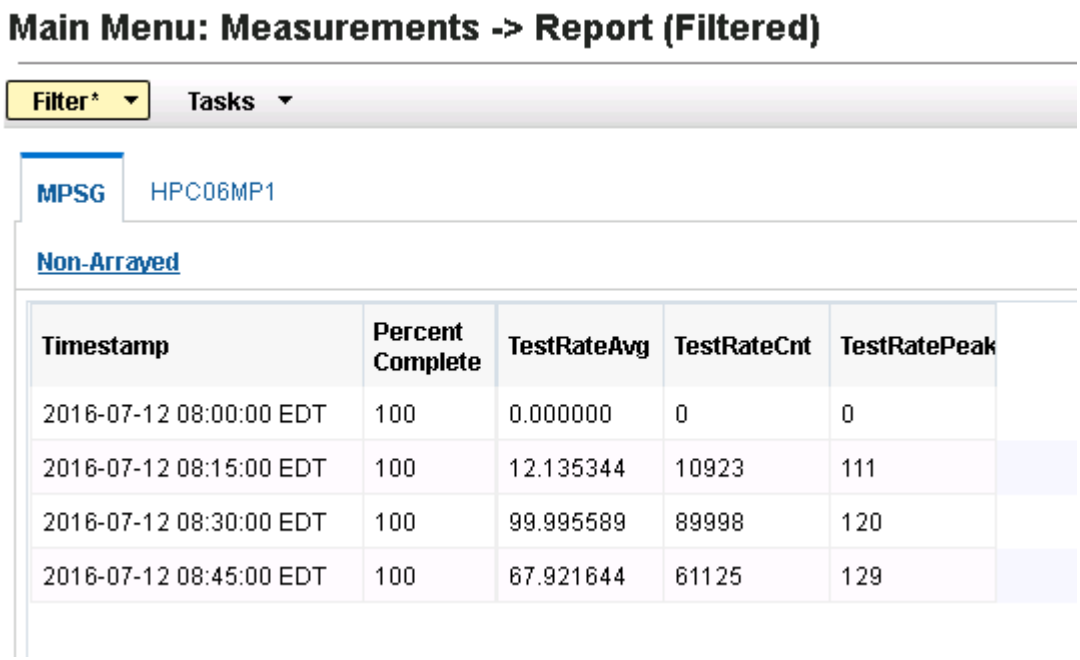


Figure 43: Display the TestRate measurements

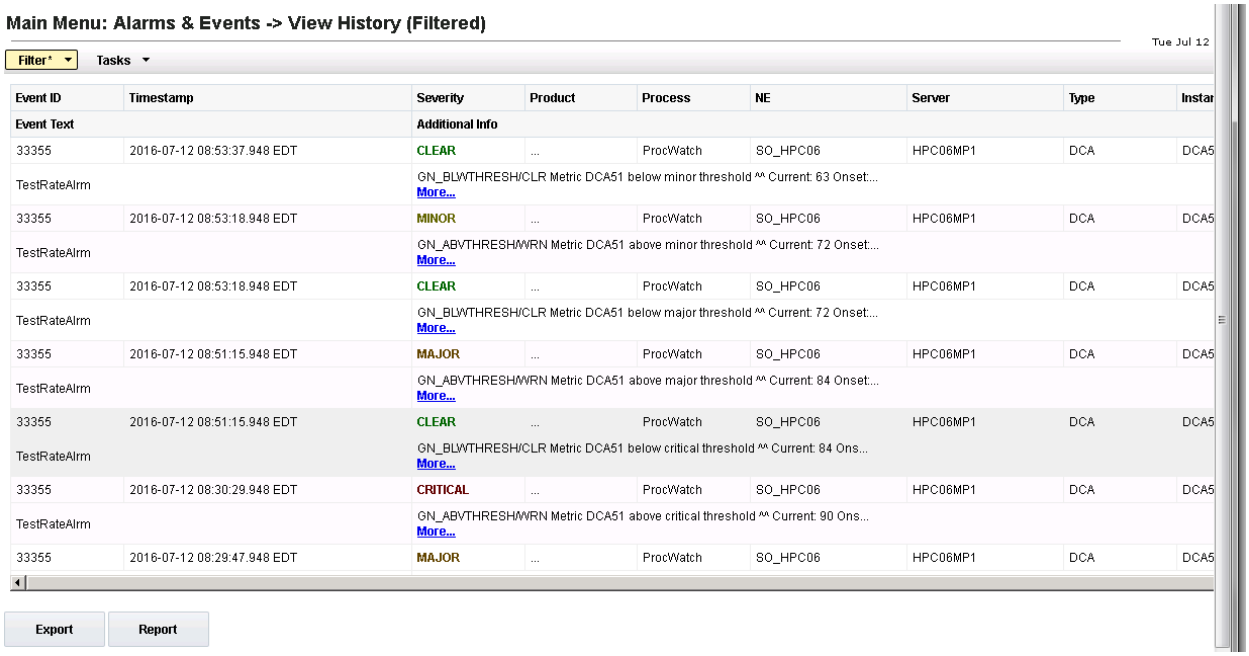


Figure 44: TestRate Alarm History

8.3.9 Step 8: Promote the DCA App Version to Production

The same process described in Section 3.3.8 is followed.

9. GUI Overview

9.1 NO/SO differences

Table 1: NO/SO GUI differences

NO	SO
Framework Configuration	Read-only
General Options	Read-only
Custom MEALs	Read-only
Trial MP Assignment	Read-only
New application versions are created	-
Existing application versions are copied	-
Business Logic and/or NO Config data imported/exported	SO Config data imported/exported
SBR DB Mane Mapping	Read-only
Flowchart and Script Development	Read-only
Application version state transitions	Read-only
Defining the configuration tables (schema)	Read-only
Provisioning NO Configuration Data (table content)	Provisioning SO Configuration Data (table content) NO configuration read-only.
-	System Options

9.2 NO Screens

The DCA Framework left hand menu on the NO includes the following screens:

- Configuration Screen

Each activated application is represented by the separate menu folder with the given application name. The application folder on the NO includes the following screens (Application Control screen contains the buttons that lead to other DCA screens):

- Custom Meals
- General Options Screen
- Trial MPs Assignment Screen
- Application Control Screen
  - Create New Development Screen
  - Copy to New Development Screen
  - Import Pop-Up Window
  - Export Pop-Up Window
  - SBR Database Name Mapping
  - Development Environment
  - Tables Screen
    - Provision Tables Screen

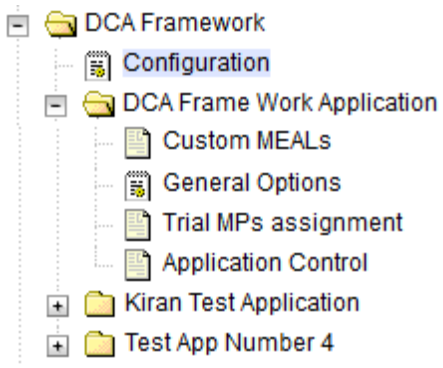


Figure 45: NO Screens

9.2.1 Configuration Screen

The NO **Main Menu→DCA Framework→Configuration** screen allows configuring DCA Framework parameters: Maximum Size of Application State and Maximum Size of the Key. See Figure 46.

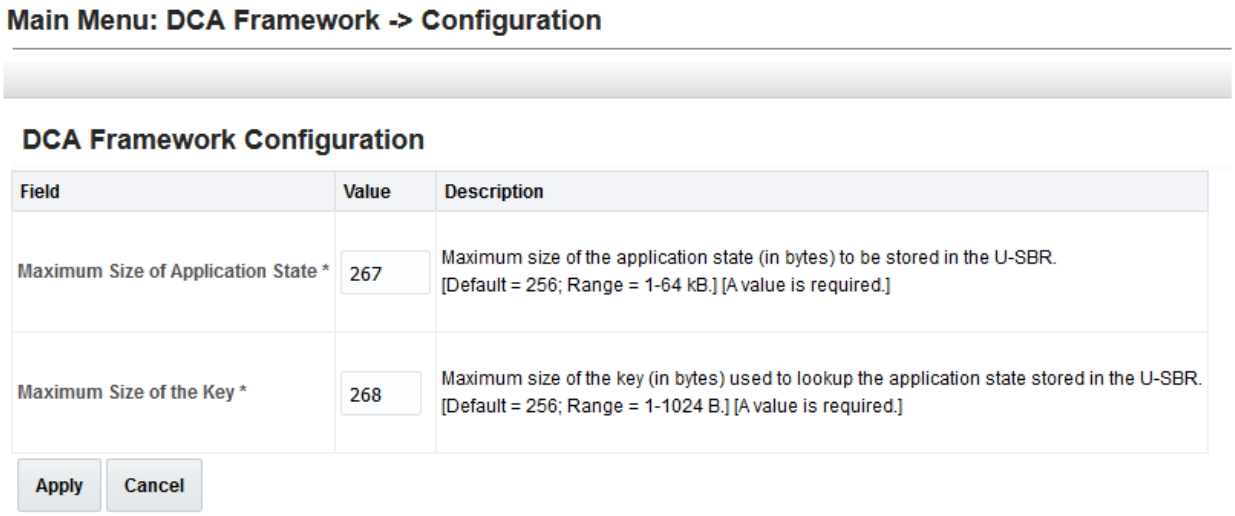


Figure 46: NO Configuration Screen

9.2.2 Custom MEALs

9.2.2.1 View Custom MEALs

The NO **Main Menu: DCA Framework→<DCA App Name>→Custom MEALs** screen (illustrated in Figure 47) lists the Custom MEAL templates differentiated for the current DCA App. It also enables new Custom MEAL templates to be differentiated and differentiated Custom MEAL templates to be modified.

There are a limited number of Custom MEAL templates of each type for all the DCA Apps activated in a network. An error displays if the DCA App programmer attempts to exceed these limits.

It is not possible to modify the counter/basic/rate/event and scalar/arrayed type of a differentiated Custom MEAL template. If the type needs to be modified, then a new Custom MEAL template is created (provided the limits have not been exceeded yet) and the old one is deleted.

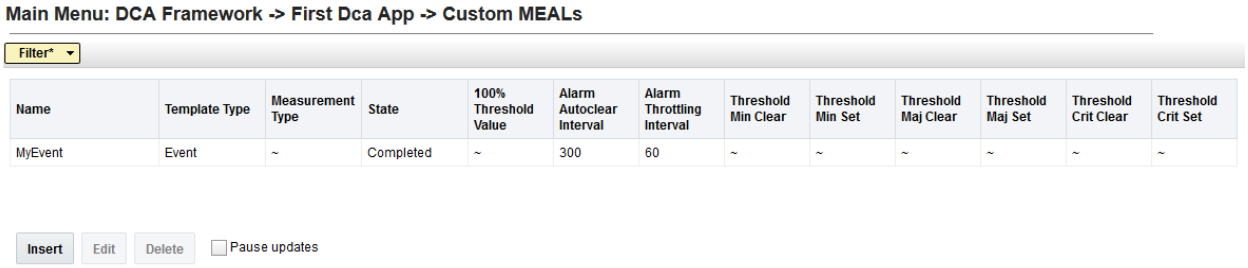


Figure 47: The Custom MEAL View Screen

9.2.2.2 Configure the Counter Custom MEAL Template

Figure 48 illustrates the configuration options for inserting a Counter template.

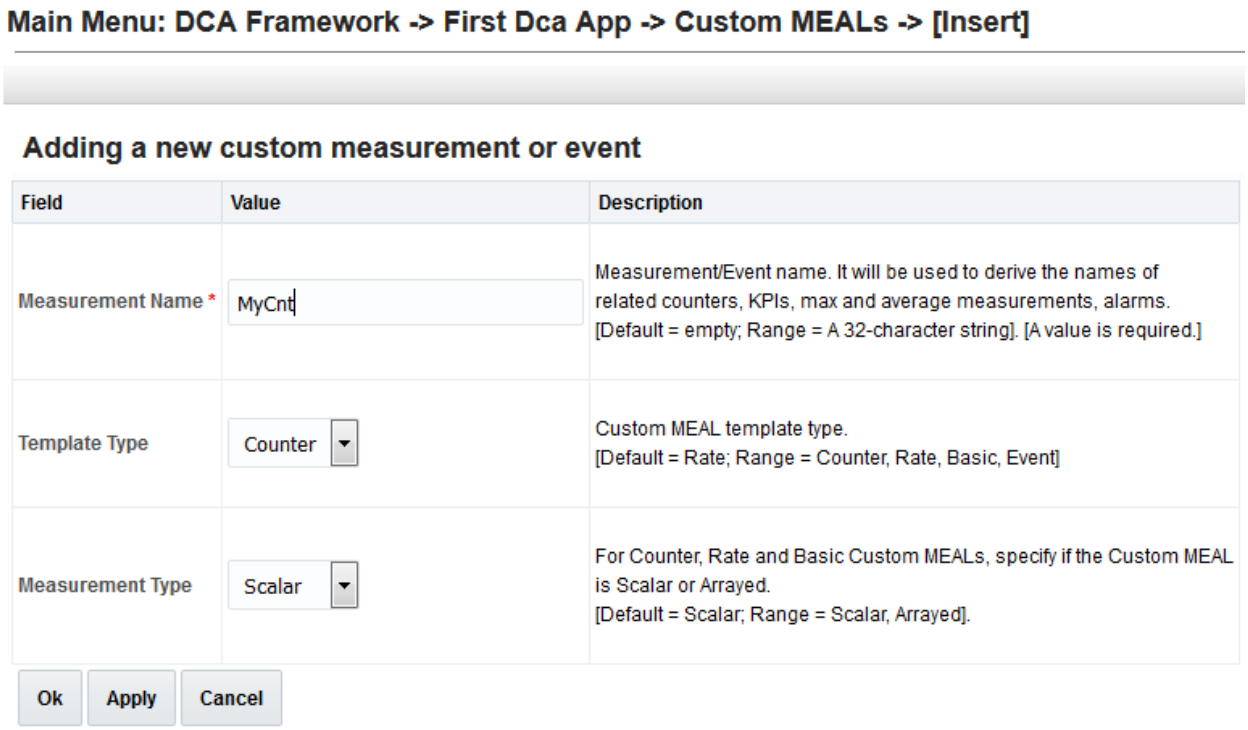


Figure 48: The Counter Template Configuration Screen

9.2.2.3 Configure the Basic Custom MEAL Template

Figure 49 illustrates the configuration options for inserting a Basic template. The Basic template is optionally associated with an alarm, which is automatically raised if the configured thresholds are exceeded.

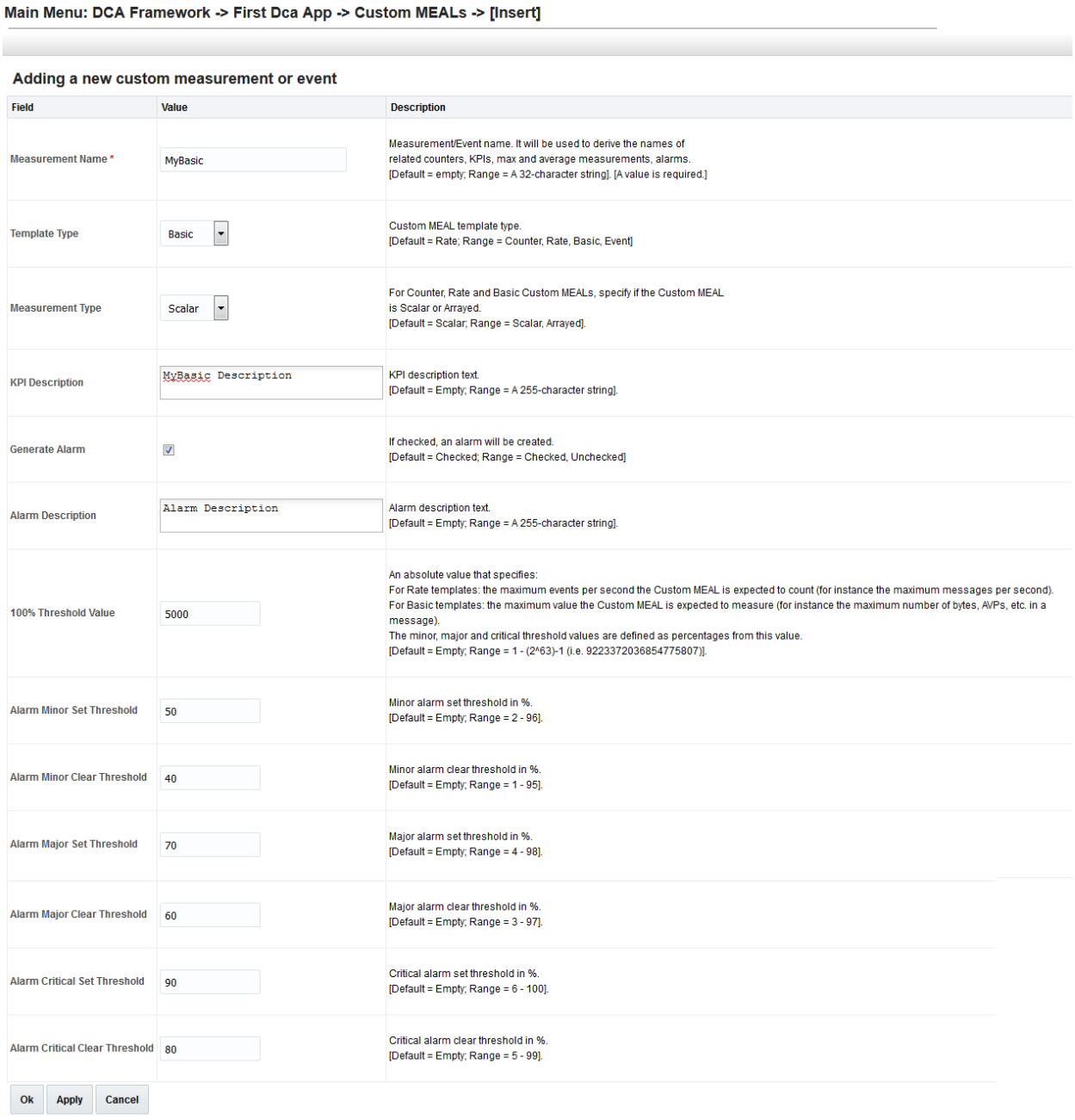


Figure 49: The Basic Template Configuration Screen

9.2.2.4 Configure the Rate Custom MEAL Template

Figure 50 illustrates the configuration options for inserting a Rate template. The Rate template is optionally associated with an alarm, which is automatically raised if the configured thresholds are exceeded.

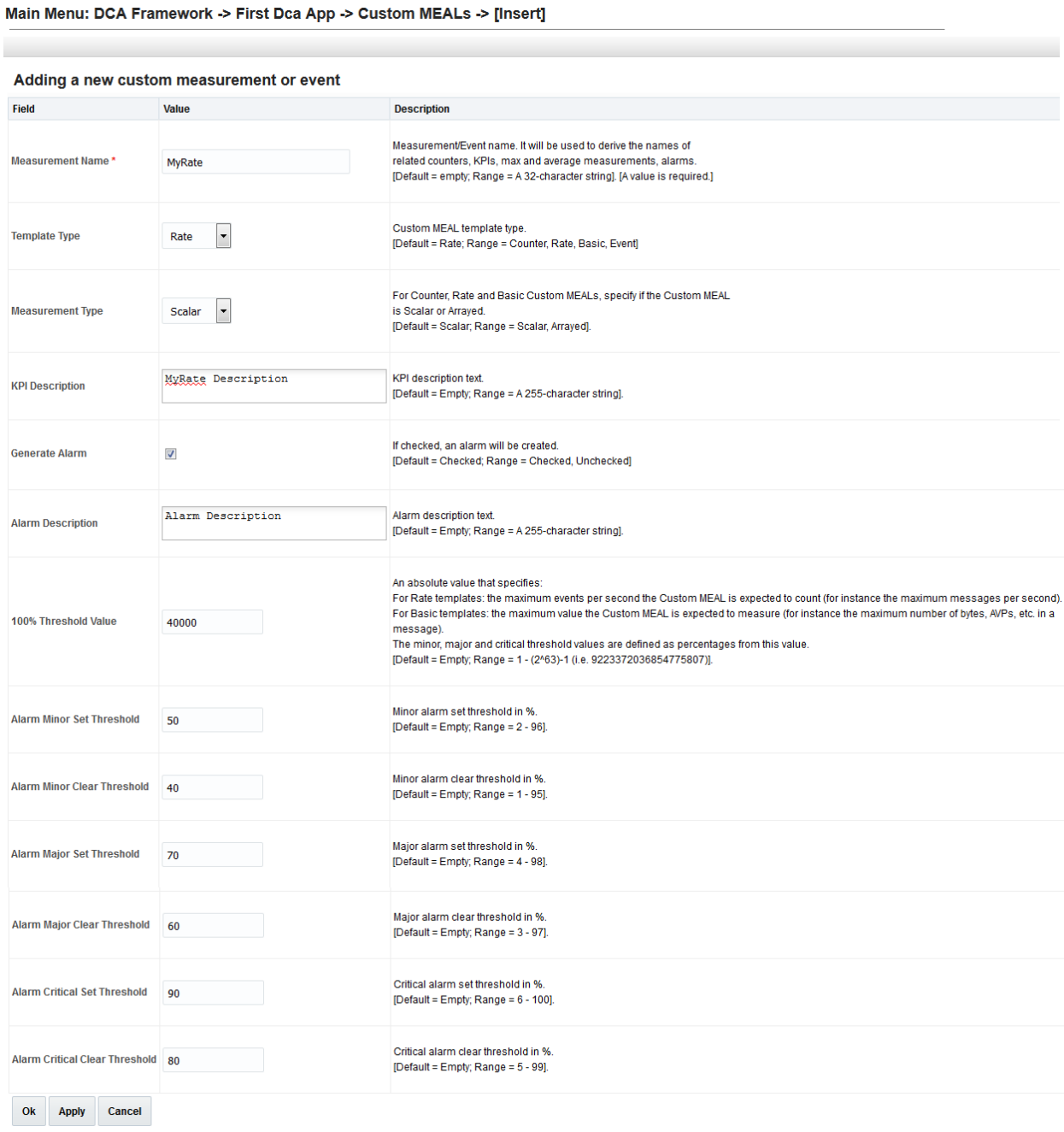


Figure 50: The Rate Template Configuration Screen

9.2.2.5 Configure the Event Custom MEAL Template

Figure 51 illustrates the configuration options for inserting an Event template.

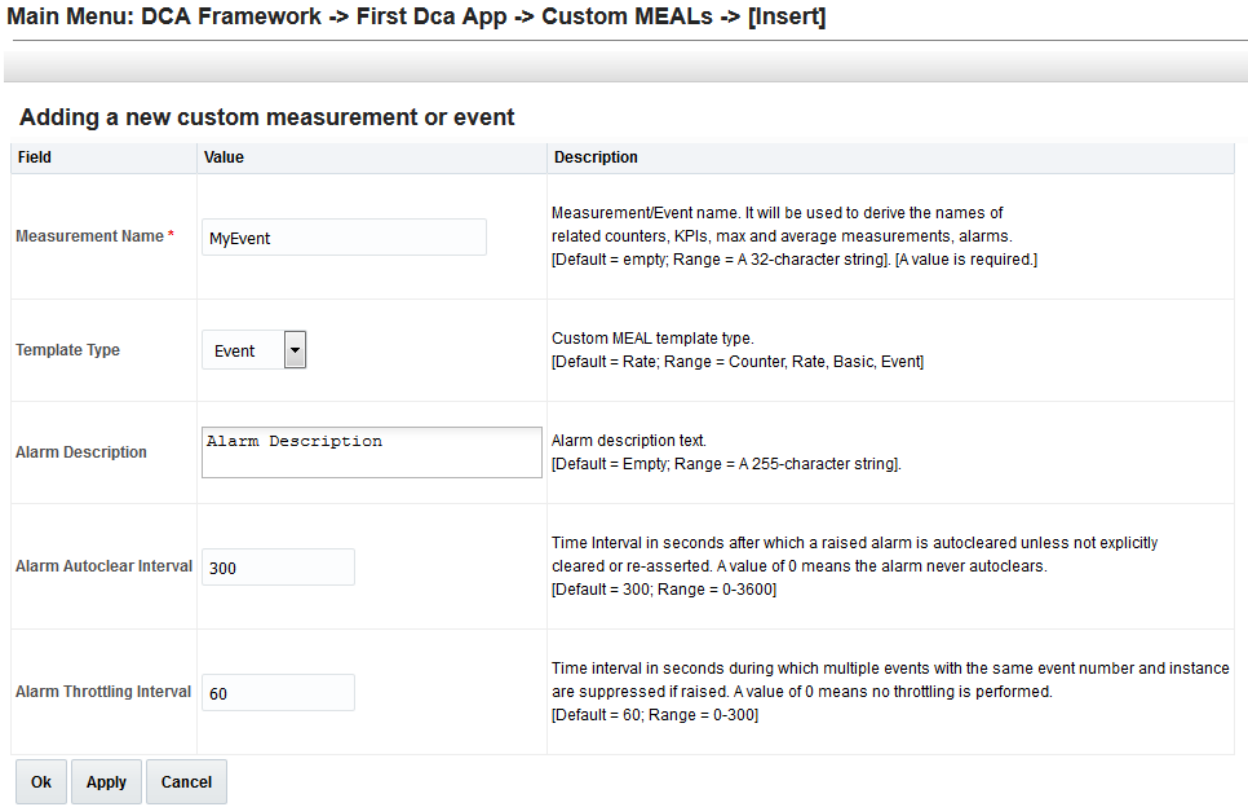


Figure 51: The Event Template Configuration Screen

9.2.3 General Options Screen

The NO **Main Menu→DCA Framework→< Application Name>→General Options** screen enables specifying the Perl Subroutines for Diameter Request and Answer, Application State Data TTL, Read Only U-SBR Access as Guest and Max. U-SBR Queries per Message. See Figure 52.

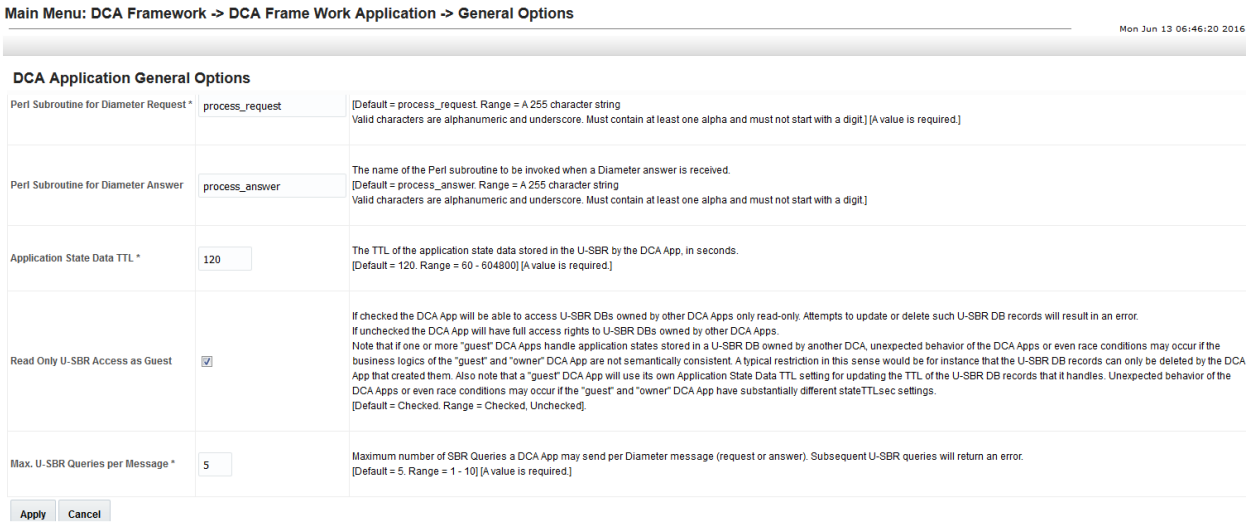


Figure 52: NO General Options

9.2.4 Trial MPs Assignment Screen

The **NO Main Menu→DCA Framework→<DCA App Name>→Trial MPs Assignment** screen allows specifying which DA-MPs run the Trial version of the DCA App (see Figure 53). If there is no Trial version available, the Trial DA-MPs runs the Production version, if there is any available.

If a DCA App version is promoted to the Trial state but no Trial DA-MPs are currently configured assigned, a warning message displays.

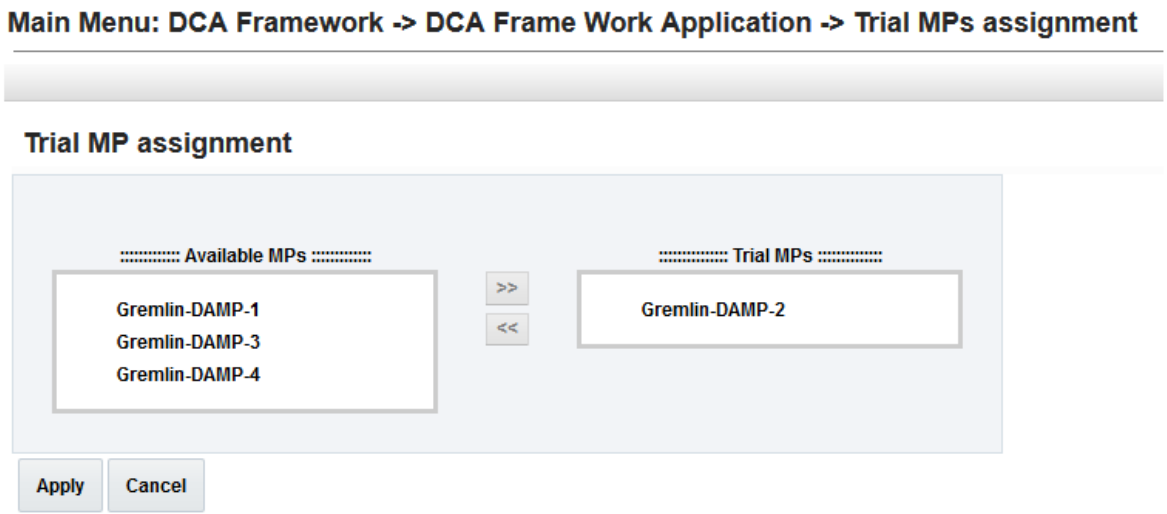


Figure 53: NO Trial MPs Assignment

9.2.5 Application Control Screen

The NO **Main Menu→DCA Framework→< Application Name>→Application Control** screen (see Figure 54) allows:

- Listing all application versions configured in the system
- Inserting a new application version (via NO New Development Insert Screen)
- Copying and modifying an existing application version (via NO New Development Copy Screen)
- Exporting an application version entirely (business logic + provisioned data from the NO)
- Exporting only the NO provisioned data of an application version
- Importing a previously exported application version (business logic + NO provisioned data) (via NO Import Pop-Up Window).
- Importing only the NO provisioned data to an existing application version (via NO Import Pop-Up Window)
- Accessing the application version configuration tables (via NO Tables View Screen)
- Accessing business logic and flowchart of an application version (via NO Development Environment Screen)
- Deleting an existing application version
- Changing the status of an application version (Development, Trial, Production, Archived)

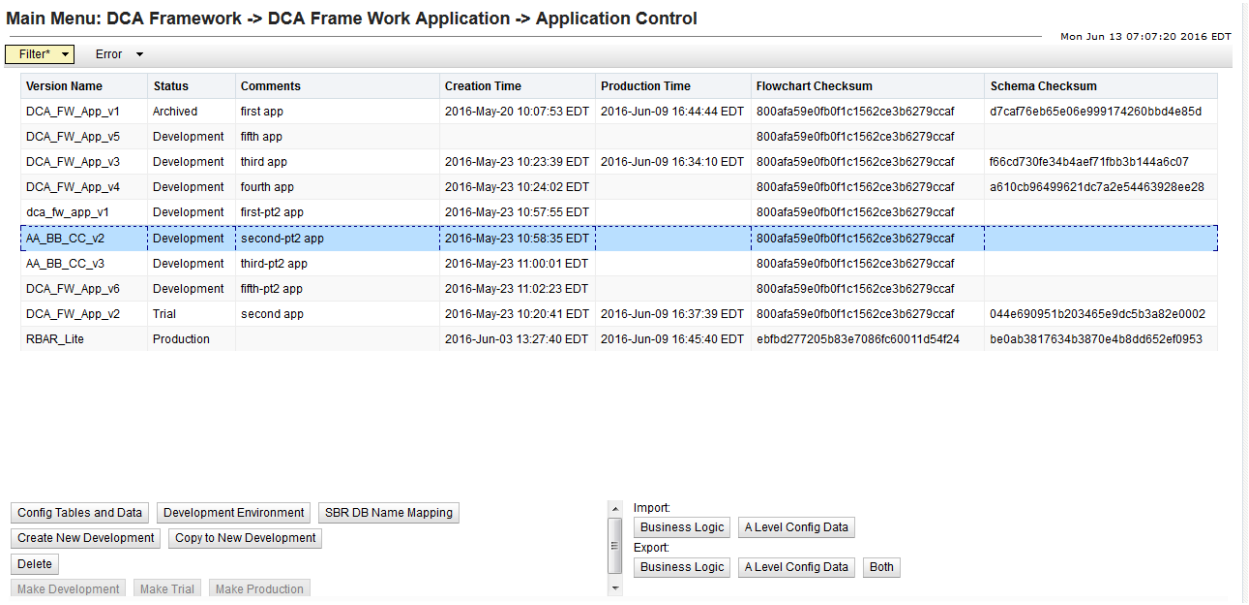


Figure 54: NO Application Control

9.2.6 Create New Development Screen

The NO **Main Menu→DCA Framework→<Application Name>→Application Control→Create New Development** screen allows creating a new DCA App version with a given name and comments. It is accessed by clicking **Create New Development** on the Application Control screen, see Figure 55.

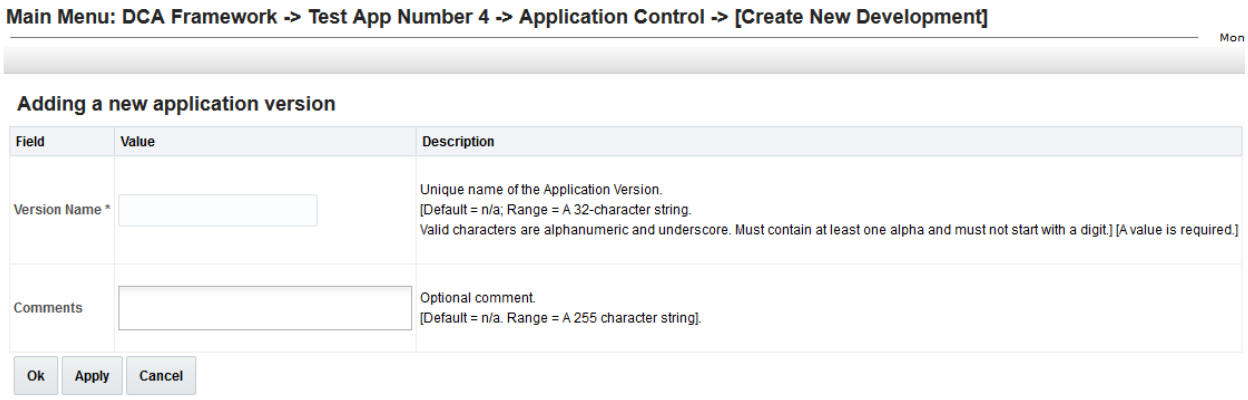


Figure 55: NO Create New Development Screen

Currently, there might be up to 10 application versions at a time.

9.2.7 Copy to New Development Screen

The NO **Main Menu-→DCA Framework→< Application Name>→Application Control→Copy to New Development** screen allows copying an entire DCA App version, consisting of business logic (Perl script, flowchart, and configuration table schemas) and the NO provisioned configuration data, into a new version. It is accessed by selecting the application version and clicking **Copy to New Development** on the Application Control screen, see Figure 56.

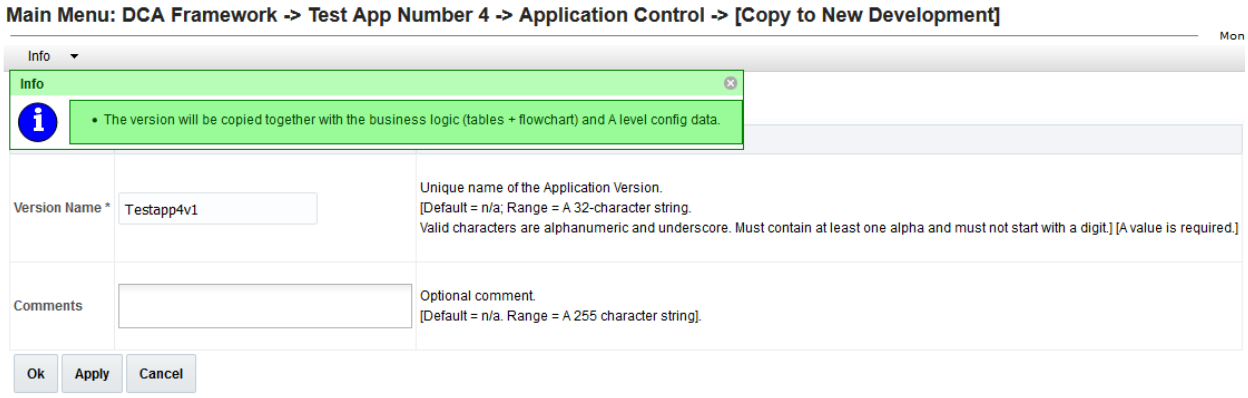


Figure 56: NO Copy to New Development

When the new Application Version is copied, it becomes visible on the Application Control screen displaying the user provisioned name in the Version Name column and comments in the Comments column.

The copied Application also includes the business logic (DB tables + Perl script) and the A level (NO level) configuration data (if any was specified).

9.2.8 Export Pop-Up Window

The exported application version is stored in the form of a JSON file.

DCA Framework GUI offers three export options:

- Export the business logic only (that includes the defined tables, flow control chart, the script, custom Meals, KPIs, Events associated with the application version, logical to physical SBR Mapping. It does not include the provisioned data).
- Export the business logic and the configuration data (in addition to the business logic the provisioned data for the tables is also exported).
- Export the configuration data only.

For the first option, select the application version and click **Export Business Logic** (becomes enabled when the row is selected).

For the second option, select the application version and click **Export Both** (becomes enabled when the row is selected).

For the third option, select the application version and click **Export A Level Config Data** (becomes enabled when the row is selected). The export popup window is illustrated in Figure 57.

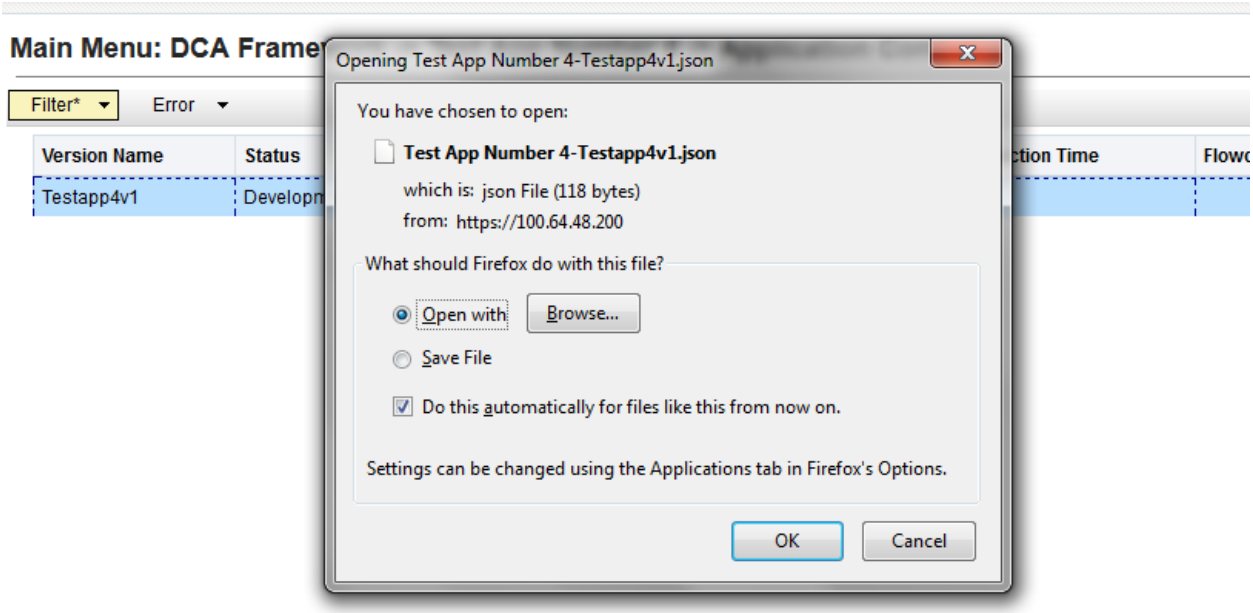


Figure 57: NO Export

When the user tries to export the business logic, there is a validation to check whether the flowchart/script has been compiled. If not, the export is aborted and the error is given.

The A level (NO level) configuration data can be exported from the NO machine, but not from the SO.

9.2.9 Import Pop-Up Window

The NO Import Pop-Up window allows specifying a JSON file from which the business logic (if required) and the NO provisioned data is imported.

Note: The provisioned data imported to the existing business logic is appended to the existing data rows.

If the user wants to overwrite the configuration data, it is recommended to first delete all provisioned rows on the Provision Table screen and then import the new configuration data.

DCA Framework GUI offers three import options:

- Import the business logic only (that includes the defined tables, flow control chart, the script, custom Meals, KPIs, Events associated with the application version, logical to physical SBR Mapping. It does not include the provisioned data import; hence, the defined tables are empty after the import).
- Import the business logic and the configuration data (in addition to the business logic the provisioned data for the tables is also imported).
- Import the configuration data only.

For the first option, click **Import Business Logic** (always enabled) on the NO Application Control screen. Leave the checkbox **Import also Config data** unchecked, see Figure 58. Select the file.

For the second option, click **Import Business Logic** (always enabled) on the NO Application Control screen. Check **Import also Config data** the checkbox. Select the file.

For the third option, select the application version and click **Import A Level Config Data** (becomes enabled when the row is selected), see Figure 59. Select the file.

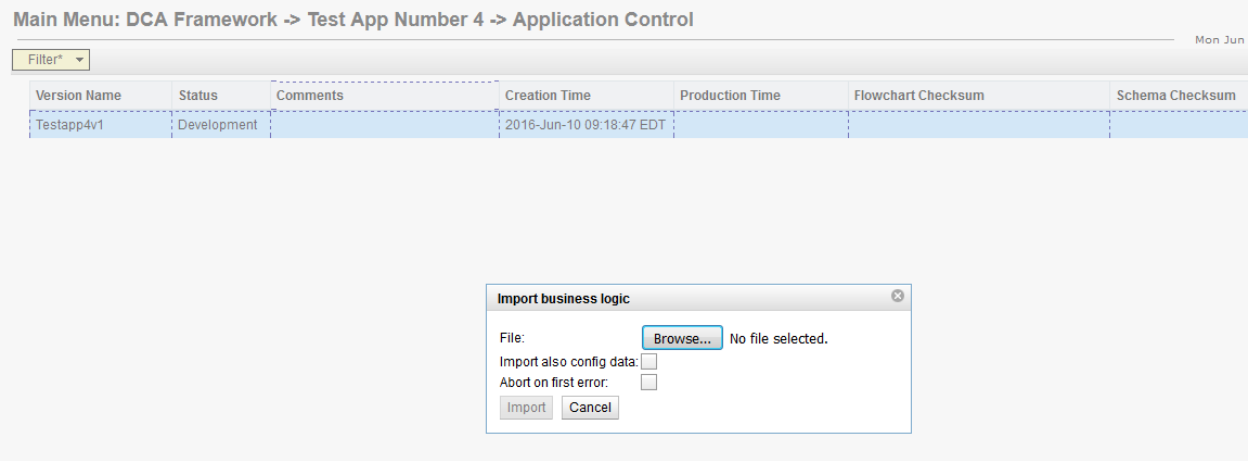


Figure 58: NO Import Business Logic

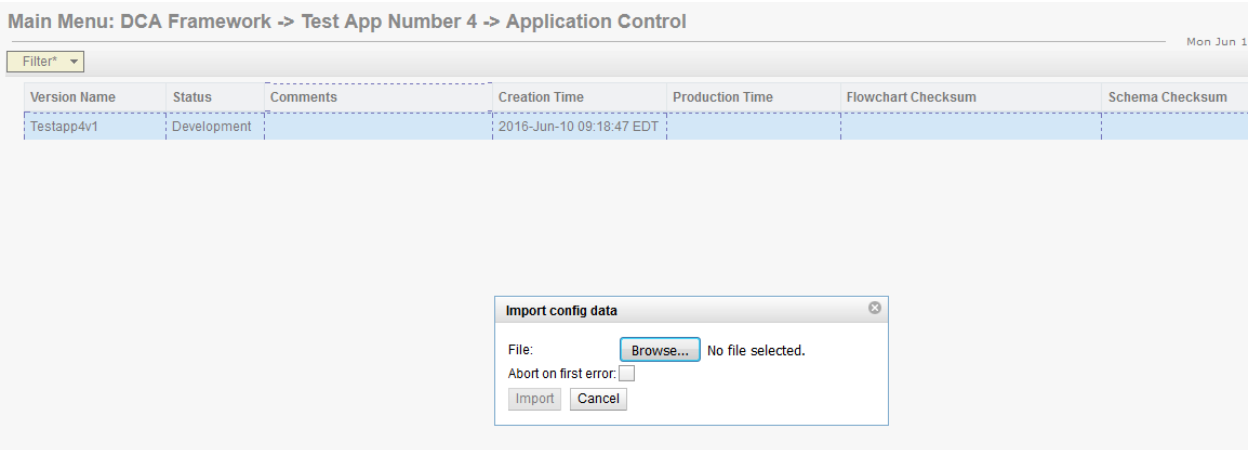


Figure 59: NO Import Configuration Data

During the import, validations are performed in a particular order to ensure the format of the DCA App configuration data to be imported is compatible with that of the target DCA App version.

As a result, a number of fatal errors may occur during the import, which forces the import to be aborted regardless of **Abort on first error** checkbox. Such fatal errors are:

- File larger than 25MB.
- File has wrong structure or missing data.
- All the errors that happen during the business logic import.
- If the user tries to import the config data to an existing application version, but none of the table names from the imported file matches the table names of the selected application.
- If the user tries to import the config data to an existing application version, but none of the field names in the tables from the imported file matches the field names in the tables of the selected application.
- Level mismatch. A-level DCA Application configuration data can be imported only on the A level. The same applies to the B level data.

Non-fatal errors, on the other hand, let the user decide whether to abort the import or not (depending on the value of **Abort on first error** checkbox).

9.2.10 SBR DB Name Mapping Screen

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→SBR Database Name Mapping View** screen (see Figure 60) allows viewing and configuring the mapping between U-SBR DB logical names (as used in Perl script) and U-SBR DB physical names. It is accessed by selecting an application version and clicking **SBR DB Name Mapping** on the Application Control screen.

Note: All the SBR names referred in the application version script are matched to the SBR physical names.

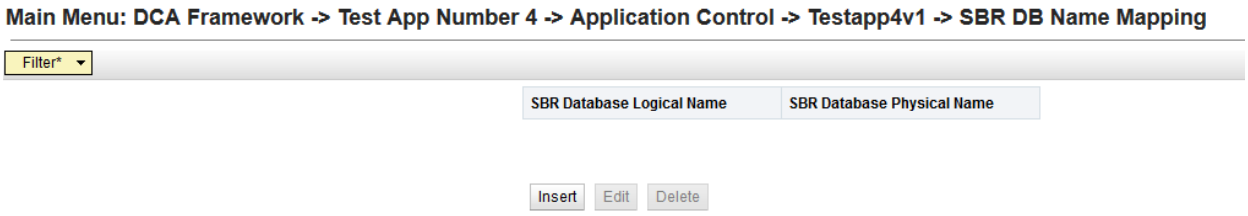


Figure 60: NO SBR DB Name Mapping View

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→SBR DB Name Mapping Insert** screen (see Figure 61) allows creating the new mapping

between U-SBR DB logical names (as used in Perl script) and U-SBR DB physical names. It is accessed by clicking **Insert** on the SBR DB Name Mapping View screen.

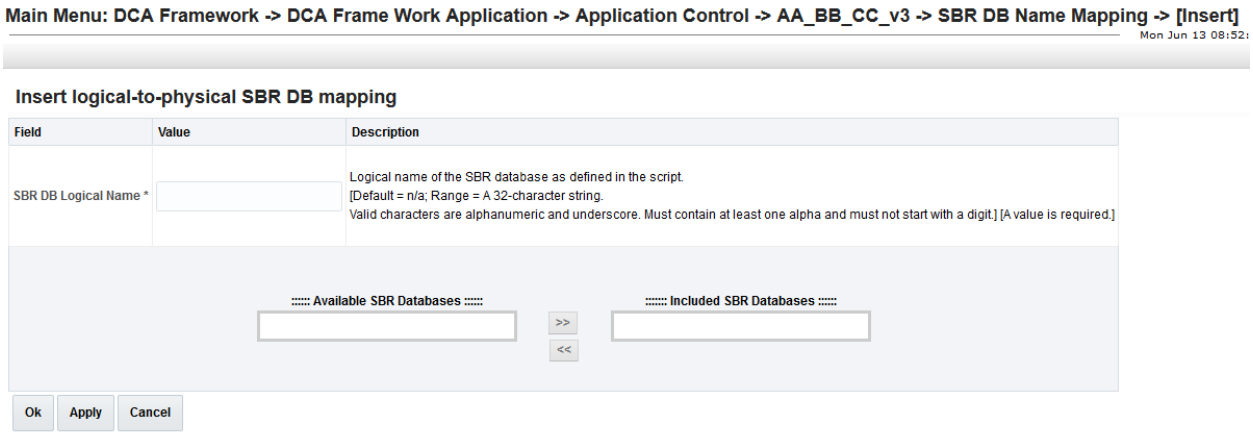


Figure 61: NO SBR DB Name Mapping Insert

Specify the logical name that is used by the application version script and move the corresponding physical SBRs to the right list Included SBR Databases.

Each DCA App running on a particular DA-MP monitors the administrative state of the resolved physical U-SBR DBs and their sub-resource routing state, and updates its own operational state to Unavailable in any of the following cases:

- The U-SBR DB's administrative state is not Enabled.
- The U-SBR DB's administrative state is Enabled but all of its sub-resources are unavailable or are not reporting.

The Alarm ID 33306 is raised if a logical U-SBR DB name cannot be resolved to a physical U-SBR DB name (none of the physical U-SBR DBs mapped to a logical U-SBR DB is located in the same Place Association as the DA-MP performing the resolution). The Alarm ID 33306 is cleared when the logical-to-physical U-SBR DB resolution process is (re-)triggered.

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→SBR Database Name Mapping Edit** screen allows editing the mapping between U-SBR DB logical names (as used in Perl script) and U-SBR DB physical names.

9.2.11 Development Environment

Development Environment is accessed by selecting the application version and clicking **Development Environment** on the Application Control screen. The DCA Development Environment (DCA-DE) is where a custom Diameter application developer can edit, save, check syntax, and compile the application code for a Diameter Custom Application.

See [1] CGBU_018429 - DCA Framework and Application Activation and Deactivation for more details.

9.2.12 Tables Screen

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Tables View** screen (see Figure 62) allows:

- Listing all the config tables (NO+SO) defined for an application version

- Inserting/editing a new config table (NO or SO) for the development or trial application version (via NO Table Insert/Edit Screen).
- Deleting an existing config table (NO or SO) for the development or trial application version
- Viewing an existing config table of an archived or production application version (via NO Table View Screen).
- Accessing the Provision Table View and Insert/Edit screens (via NO Provision Table View Screen, NO Provision Table Insert Screen and NO Provision Table Edit Screen).

The Tables View screen is accessed by selecting the application version and clicking **Config Tables and Data** on the Application Control screen.

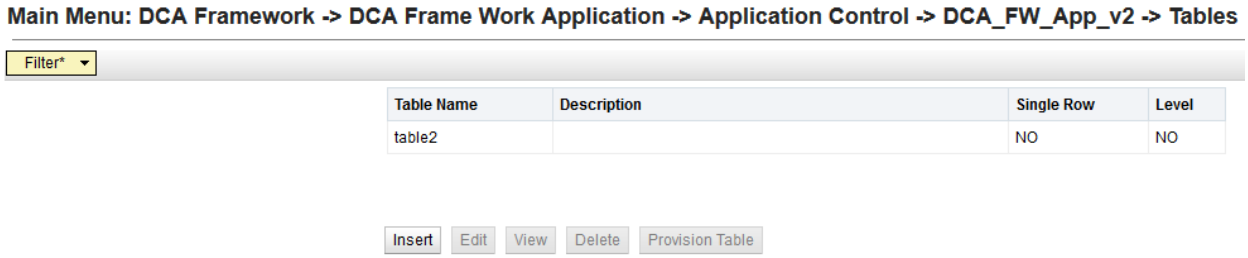


Figure 62: NO Tables View Screen

Insert, **Edit**, and **Delete** are disabled on the Tables View screen for the archived and production application versions.

View is enabled for the archived and production application versions if the table is selected.

View is disabled for the development and trial application version.

Provision Table is always enabled if the NO table is selected (it is disabled for the SO tables from the NO GUI).

Table 2 illustrates the access rights for the DCA App configuration schema and data provisioning tables. The NO/SO DCA database tables (schema) can be created, deleted and modified from the NO GUI for the development and trial application versions; they can be only viewed for the archived and production application version. The NO DCA database tables can be provisioned anytime from the NO GUI. The SO tables cannot be provisioned from the NO GUI.

Table 2: NO GUI tables and configuration data accessibility

The accessibility of level A and level B table schema and content from the NO GUI:

	NO GUI		
	Archived	**Production**	**Development, Trial**
NO tables schema (level A)	ro	ro	rw
NO tables content (level A)	rw	rw	rw
SO tables schema (level A - shares same field as NO tables schema)	ro	ro	rw
SO tables content (level B)	n/a	n/a	n/a

ro: read-only
rw: read-write
n/a: not available

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Table Insert** screen (see Figure 63) allows defining a new configuration table (NO or SO). It is accessed by clicking **Insert** on the Tables View screen for the development and trial application versions.

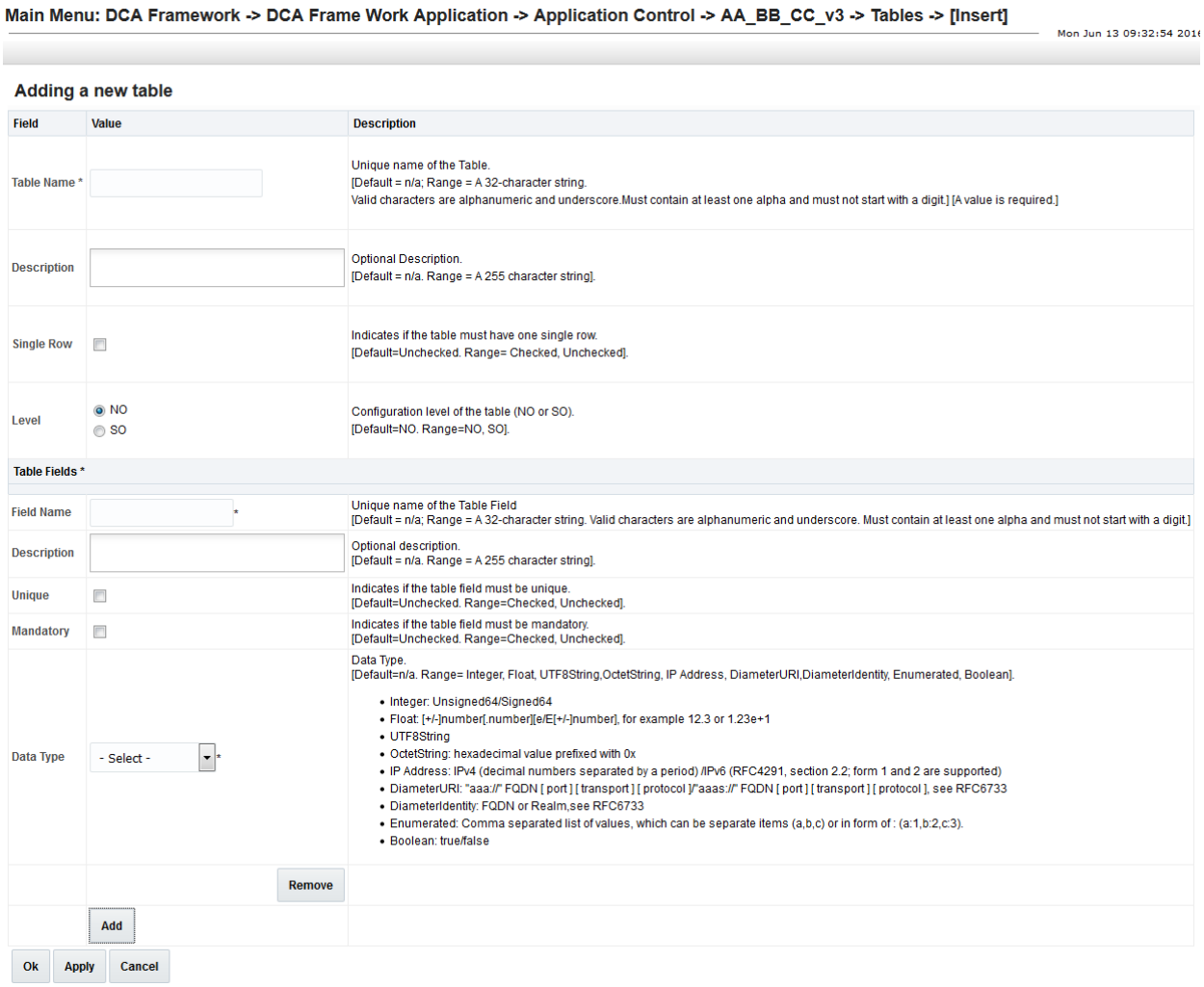


Figure 63: NO Tables Insert Screen

Currently, there might be up to 10 configuration tables per application version (NO+SO).

The configuration table definition includes:

- Table Name and Description
- Number of table rows (single vs multiple up to 2000 rows)
- Table level (whether the table resides on the NO or the SO)
- Table Fields (up to 20 now)
  - Field Name and Description
  - Whether the field is unique
  - Whether the field is mandatory
  - Field Data Type
  - Field Default value

The table fields can be of the following types (depending on the selected data type, ranges must be also defined):

- Integer (Range: Min. and Max. values)
- Float (Range: Min. and Max. values)
- UTF8String (Range: Max. length)
- OctetString (Range: Max. length)
- IPaddress
- DiameterURI
- DiameterIdentity
- Enumerated (The values)
- Boolean

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Table Edit** screen allows editing the schema of an existing DCA App configuration table (NO or SO).

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Table View (Read-only Insert/Edit)** screen allows viewing a DCA App configuration table in read-only mode. It is accessed when the table is selected and **View** is clicked on the NO Tables View screen for the archived and production application version.

9.2.13 Provision Tables Screen

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Provision Table View** screen (Figure 65) allows:

- Listing all the data rows provisioned for the NO configuration table
- Inserting a new data row to the NO configuration table (via NO Provision Table Insert Screen)
- Editing a data row of the NO configuration table (via NO Provision Table Edit Screen)
- Deleting a data row from the NO configuration table
- Deleting all provisioned rows at once

It is accessed by selecting the table and clicking **Provision Table** on the Tables View screen, see Figure 64.

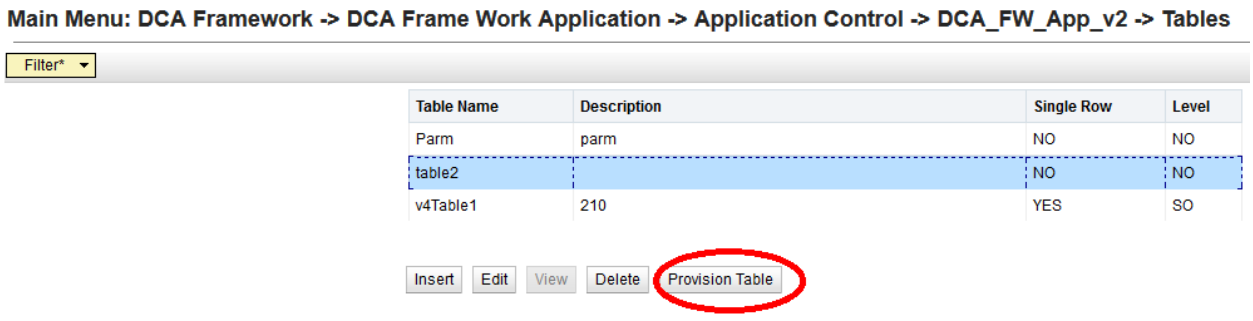


Figure 64: Provision Table Button

Provision Table is disabled for the SO tables from the NO GUI, see Table 2.

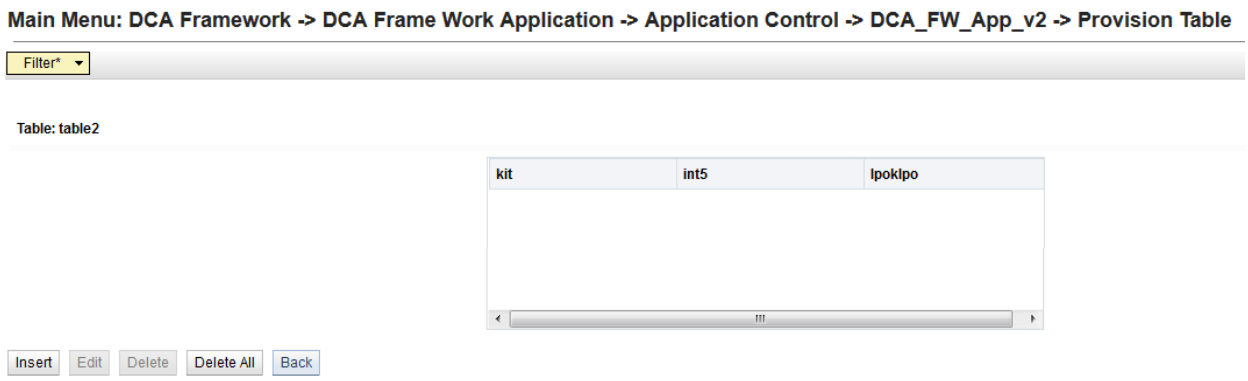


Figure 65: NO Provision Table View Screen

Up to 2000 rows of data can be provisioned per table unless the table has only single row (the **Single row** checkbox has been checked on the Table Insert screen).

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Provision Table Insert** screen (see Figure 66) allows inserting a new data row to the NO configuration table.

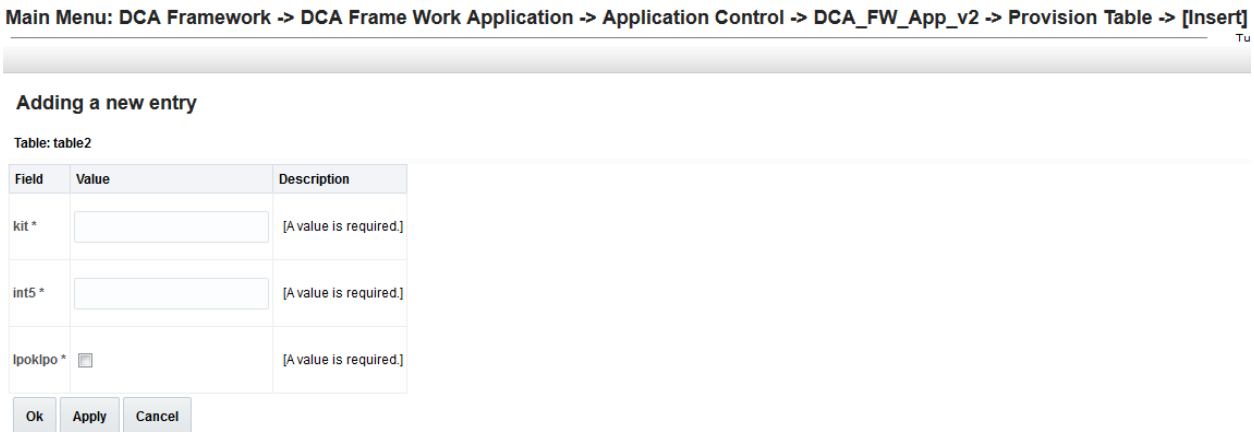


Figure 66: NO Provision Table Insert Screen

During the data insert, the GUI performs the following validations:

- Whether the mandatory value is present
- Whether the unique value is unique
- Whether the maximum of data rows is reached
- Whether the data inserted corresponds to the specified field data type
- Whether the data inserted is between the specified min-max range for the field
- Whether the entered sting value is no longer than the allowed maximum for the field
- Whether the entered enumerated value is within the allowed range of enumerated values for the field
- Etc.

The NO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Provision Table Edit** screen allows editing a data row of the NO configuration table.

9.3 SO Screens

The DCA Framework left hand menu on the SO includes the following screens:

- Configuration Screen (NO screen, read-only on the SO)

Each activated application is represented by the separate menu folder with the given application name. The application folder on the NO includes the following screens (Application Control screen contains the buttons that lead to other DCA screens):

- Custom Meals (NO screen, read-only on the SO)
- General Options Screen (NO screen, read-only on the SO)
- Trial MPs Assignment Screen (NO screen, read-only on the SO)
- Application Control Screen
  - Import Pop-Up Window
  - Export Pop-Up Window
  - SBR Database Name Mapping (NO screen, read-only on the SO)
  - Development Environment (NO screen, read-only on the SO)
  - Tables Screen (NO screen, read-only on the SO, except for **View** and **Provision Table**)
    - Provision Tables Screen
  - System Options Screen

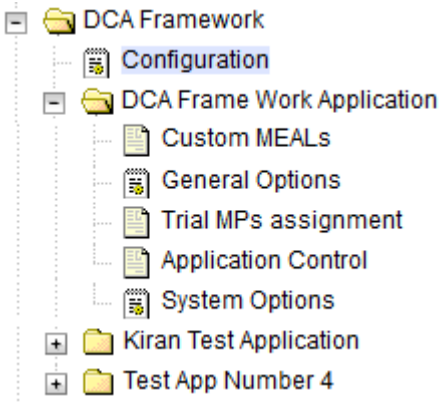


Figure 67: SO Screens

9.3.1 Application Control Screen

The SO **Main Menu→DCA Framework→< Application Name>→Application Control** screen (see Figure 68) allows:

- Listing all application versions configured in the system
- Exporting only the SO provisioned data of an application version (via SO Export Pop-Up Window)
- Importing only the SO provisioned data to an existing application version (via SO Import Pop-Up Window).
- Accessing the application version configuration tables (via SO Tables View Screen)
- Accessing the flowchart and business logic of an application version (via development environment, read-only)

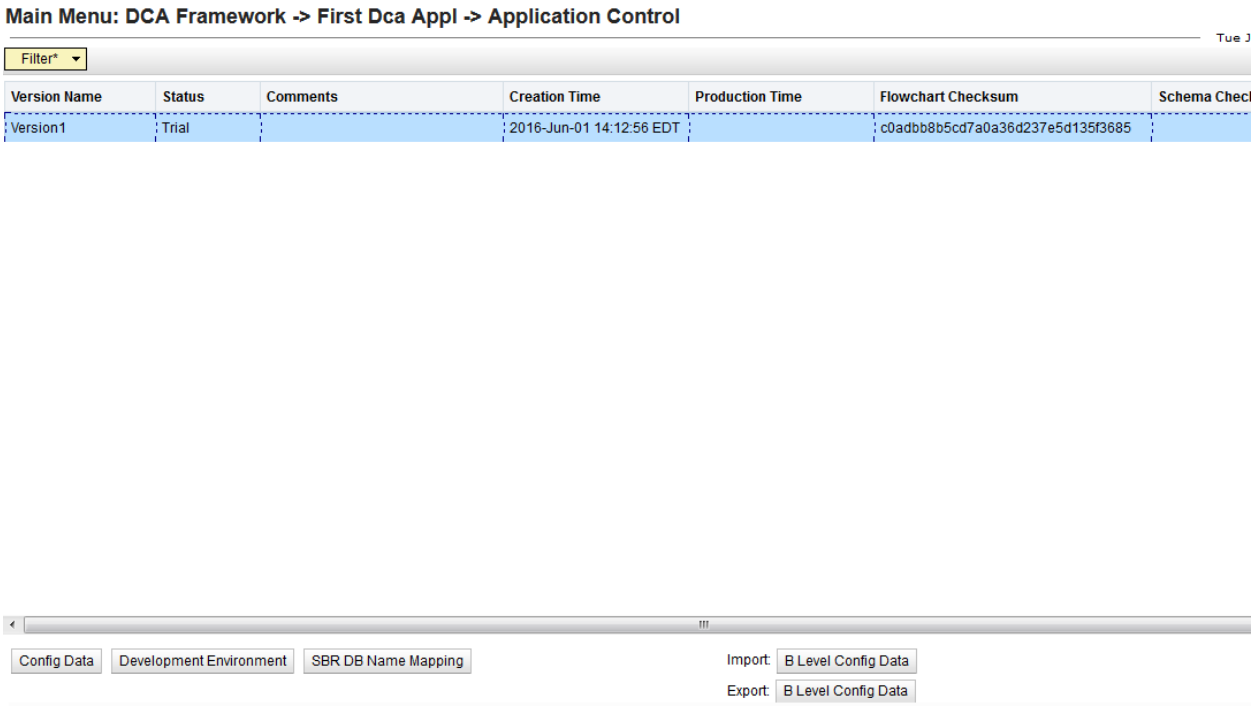


Figure 68: SO Application Control Screen

9.3.2 Export Pop-Up Window

The B level (SO level) configuration data can be exported from the SO machine, but not from the NO.

To export the configuration data to a JSON file, select the application version and click **Export B Level Config Data** (becomes enabled when the row is selected).

9.3.3 Import Pop-Up Window

The SO Import Pop-Up window allows specifying a JSON file from which the SO provisioned data is imported.

Note: The provisioned data imported to the existing business logic is appended to the existing data rows.

If the user wants to overwrite the configuration data, it is recommended to first delete all provisioned rows on the Provision Table screen and then import the new configuration data.

The B level (SO level) configuration data can be imported only to the SO machine.

To import the configuration data from the JSON file, select the application version and click **Import B Level Config Data** (becomes enabled when the row is selected). Select the file.

9.3.4 Tables Screen

The SO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Tables View** screen (see Figure 69) allows:

- Listing all the config tables (NO+SO) defined for an application version
- Viewing an existing config table (via NO/SO Table View Screen)

- Accessing the Provision Table View and Insert/Edit screens (via SO Provision Table View Screen, SO Provision Table Insert screen and SO Provision Table Edit Screen).

The SO Tables View screen is accessed by selecting the application version and clicking **Config Data** on the SO Application Control screen.

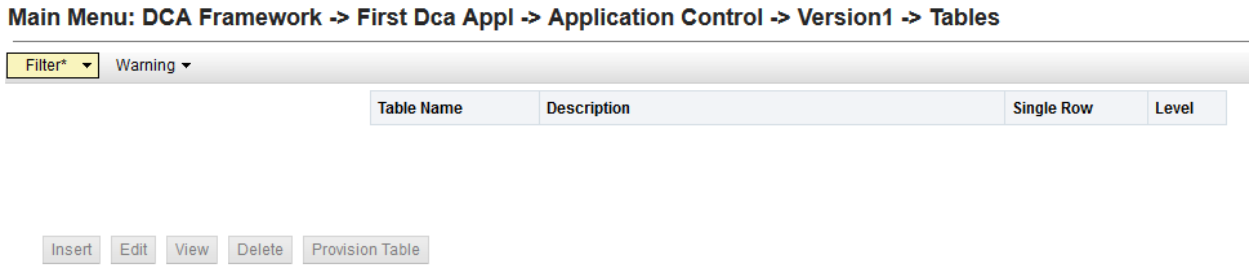


Figure 69: SO Tables View Screen (empty)

Insert, **Edit**, and **Delete** are disabled on the SO Tables View screen.

View is enabled if the table is selected.

Provision Table is always enabled if the NO/SO table is selected.

Table 3 illustrates the access rights for the DCA App configuration schema and data provisioning tables on the SO. The NO/SO DCA App table schemas can only be viewed. The level A DCA App configuration tables content can only be view from the SO GUI. The level B DCA App configuration tables can be provisioned.

Table 3: SO GUI tables and Configuration Data Accessibility

The accessibility of level A and level B table schema and content from the SO GUI:

	SO GUI		
	Archived	**Production**	**Development, Trial**
NO tables schema (level A)	ro (replicated)	ro (replicated)	ro (replicated)
NO tables content (level A)	ro (replicated)	ro (replicated)	ro (replicated)
SO tables schema (level A - shares same field as NO tables schema)	ro (replicated)	ro (replicated)	ro (replicated)
SO tables content (level B)	rw	rw	rw

ro: read-only
rw: read-write
n/a: not available

The SO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Table View (Read-only Insert/Edit)** screen allows viewing a configuration table in read-only mode. It is accessed when the table is selected and **View** is clicked on the SO Tables View screen.

9.3.5 Provision Tables Screen

The SO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Provision Table, View** screen allows:

- Listing all the data rows provisioned for the SO-rooted DCA App configuration table.

- Inserting a new data row to the SO-rooted DCA App configuration table (via SO Provision Table Insert Screen).
- Editing a data row of the SO-rooted DCA App configuration table (via SO Provision table Edit Screen).
- Deleting a data row from the SO-rooted DCA App configuration table.
- Deleting all provisioned rows at once.

Note: The NO-rooted DCA App configuration tables, as well as the schema definitions of both the NO-rooted and SO-rooted DCA App configuration tables are accessible on the SO only in read-only mode.

The SO Provision Table View screen is accessed by selecting the table and clicking **Provision Table** on the SO Tables View screen.

The SO **Main Menu→DCA Framework→<Application Name>→Application Control→<Version Name>→Provision Table**, Insert screen allows inserting a new data row to the SO-rooted DCA App configuration table.

The SO **Main Menu→DCA Framework→< Application Name>→Application Control→<Version Name>→Provision Table, Edit** screen allows editing a data row of the SO-rooted DCA App configuration table.

9.4 System Options

System Options screen is present on the SO only. See Figure 70- Figure 74.

System Options screen enables the configuration of the DSR application parameters that are:

- Relevant to the operational status Unavailable. These options allow you to specify the behavior when the application state is Unavailable (**Main Menu: Diameter→Maintenance→Applications)**. The possible behavior is:
  - Continue Routing
  - Use default route + specify application unavailable route list
  - Send Answer with Result-Code AVP + specify Result-Code and Error Message
  - Send Answer with Experimental-Result AVO + specify Result-Code, Error Message, and Vendor-Id.

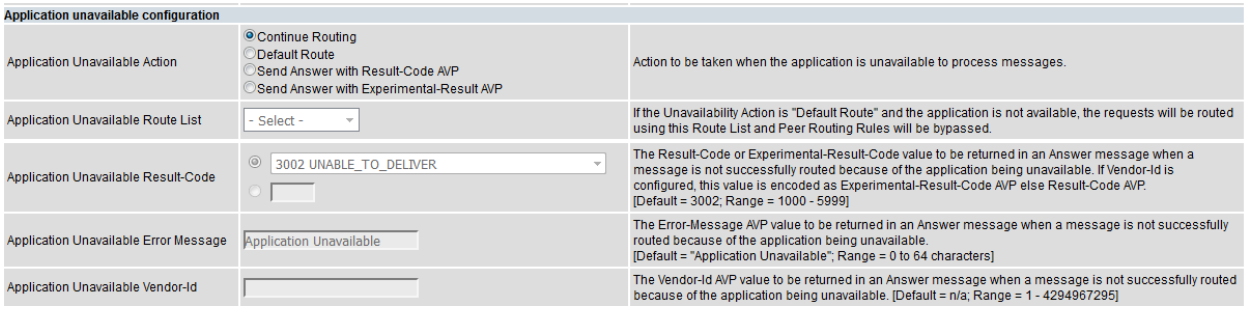
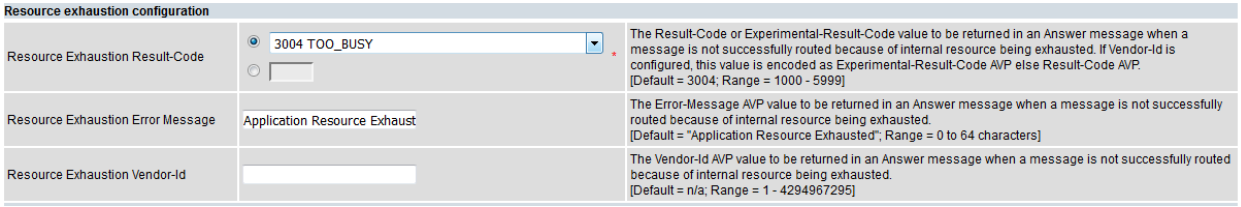


Figure 70: System Options for the Unavailable Operation Status

- Relevant to the case when the DRL resources are exhausted. The behavior is to send an error message with the specified Result-Code, Error Message, and Vendor-Id.

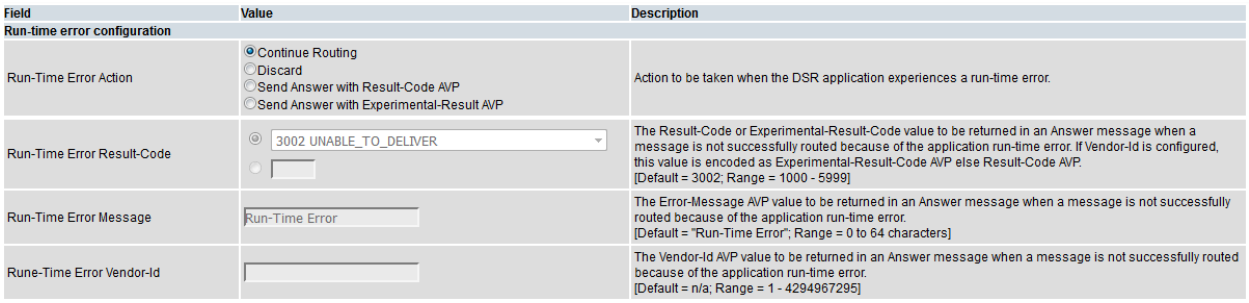


Resource exhaustion configuration		
Resource Exhaustion Result-Code	3004 TOO_BUSY	The Result-Code or Experimental-Result-Code value to be returned in an Answer message when a message is not successfully routed because of internal resource being exhausted. If Vendor-Id is configured, this value is encoded as Experimental-Result-Code AVP else Result-Code AVP. [Default = 3004; Range = 1000 - 5999]
Resource Exhaustion Error Message	Application Resource Exhaust	The Error-Message AVP value to be returned in an Answer message when a message is not successfully routed because of internal resource being exhausted. [Default = "Application Resource Exhausted"; Range = 0 to 64 characters]
Resource Exhaustion Vendor-Id		The Vendor-Id AVP value to be returned in an Answer message when a message is not successfully routed because of internal resource being exhausted. [Default = n/a; Range = 1 - 4294967295]

Figure 71: System Options for the Exhausted DRL Resources

- Relevant to the run-time error. These options allow to specify the behavior in case of a run-time error. Runtime errors fall into two categories:
  - Perl specific runtime errors – e.g., division by zero, a "die" statement, calling an undefined (utility, not event handler) subroutine etc.
  - Runtime errors triggered by the DCA framework – e.g., invoking an event handler that does not exist or exceeding the maximum configured number of executed opcodes.

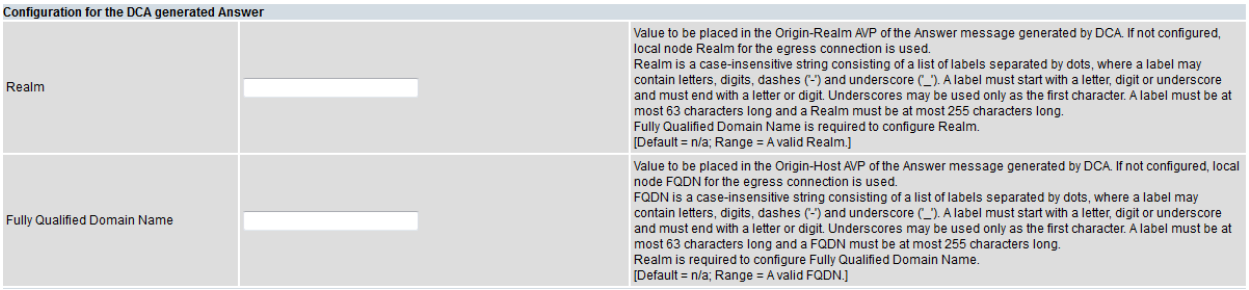
  The possible behavior is:
  - Continue Routing
  - Discard
  - Send Answer with Result-Code AVP + specify Result-Code and Error Message
  - Send Answer with Experimental-Result AVO + specify Result-Code, Error Message, and Vendor-Id.



Field	Value	Description
Run-time error configuration		
Run-Time Error Action	Continue Routing Discard Send Answer with Result-Code AVP Send Answer with Experimental-Result AVP	Action to be taken when the DSR application experiences a run-time error.
Run-Time Error Result-Code	3002 UNABLE_TO_DELIVER	The Result-Code or Experimental-Result-Code value to be returned in an Answer message when a message is not successfully routed because of the application run-time error. If Vendor-Id is configured, this value is encoded as Experimental-Result-Code AVP else Result-Code AVP. [Default = 3002; Range = 1000 - 5999]
Run-Time Error Message	Run-Time Error	The Error-Message AVP value to be returned in an Answer message when a message is not successfully routed because of the application run-time error. [Default = "Run-Time Error"; Range = 0 to 64 characters]
Rune-Time Error Vendor-Id		The Vendor-Id AVP value to be returned in an Answer message when a message is not successfully routed because of the application run-time error. [Default = n/a; Range = 1 - 4294967295]

Figure 72: System Options for the Run-Time Error

- Realm and FQDN that are placed in Answer message generated by the DCA. These are the values that are placed in the Origin-Realm and Origin-Host AVPs of the Answer message generated by DCA. If they are not configured, local node Realm and FQDN for the egress connection are used.



Configuration for the DCA generated Answer		
Realm		Value to be placed in the Origin-Realm AVP of the Answer message generated by DCA. If not configured, local node Realm for the egress connection is used. Realm is a case-insensitive string consisting of a list of labels separated by dots, where a label may contain letters, digits, dashes ('-') and underscore ('_'). A label must start with a letter, digit or underscore and must end with a letter or digit. Underscores may be used only as the first character. A label must be at most 63 characters long and a Realm must be at most 255 characters long. Fully Qualified Domain Name is required to configure Realm. [Default = n/a; Range = A valid Realm.]
Fully Qualified Domain Name		Value to be placed in the Origin-Host AVP of the Answer message generated by DCA. If not configured, local node FQDN for the egress connection is used. FQDN is a case-insensitive string consisting of a list of labels separated by dots, where a label may contain letters, digits, dashes ('-') and underscore ('_'). A label must start with a letter, digit or underscore and must end with a letter or digit. Underscores may be used only as the first character. A label must be at most 63 characters long and a FQDN must be at most 255 characters long. Realm is required to configure Fully Qualified Domain Name. [Default = n/a; Range = A valid FQDN.]

Figure 73: System Options for the Realm and FQDN

- Application invocation. This option is needed to indicate if the subsequent invocation of application on a different node in the network is allowed or not.

  If unchecked, the DSR-Application-Invoked AVP is inserted, preventing the same DSR application on another DSR node from receiving the Diameter message.

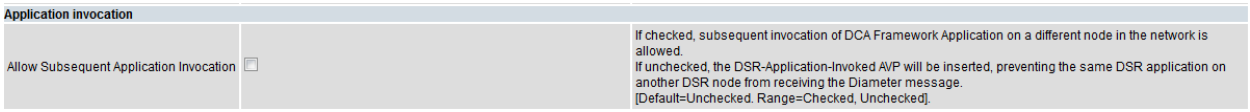


Figure 74: System Options for the Application Invocation

10. Development Environment Overview

10.1 Development Environment Modes

DCA Development Environment opens if the user clicks **Development Environment** on the **Main Menu→[Application Name]→Application Control** screen. **Development Environment** is disabled if the **Application Control: Script and Flow Control Chart** DCA Framework View Permission is unchecked.

The DCA Development Environment can be accessible in two modes of operation:

- Edit Mode (any change is possible and can be saved)
- View Mode (the Code Text Editor is read-only, Toolbox and Action commands are disabled, the Flow Control Chart interactions are disabled)

The DCA DE can be accessible in the View only mode for the following cases:

- If the selected application version is either in Production or Archived status.
- If the **Application Control: Script and Flow Control Chart** DCA Framework Edit Permission is unchecked while the View Permission is checked.

DCA DE starts, but does not retrieve the Perl code and Flow Control Chart data if the **View Permission** is unchecked.

The DCA DE can be accessible in the Edit mode for the following cases:

- If the selected application version is either in Development or Trial status and the **Application Control: Script and Flow Control Chart** DCA Framework Edit Permission is checked.

10.2 Layout

The DCA Development Environment GUI Layout contains a top banner and the following sections, see Figure 75 and Figure 76:

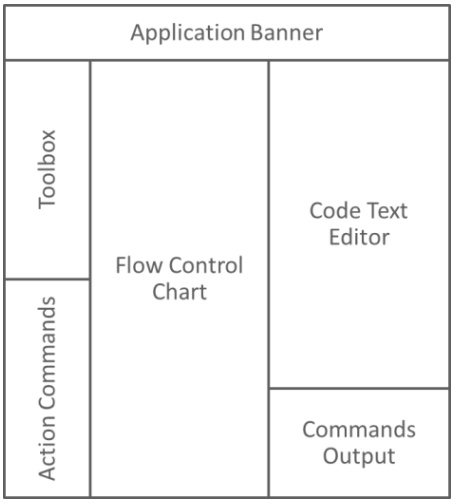


Figure 75: Layout Structure

Figure 76: Layout Print Screen

The **Application Banner** displays the application version name and status.

The **Toolbox** displays the available commands for creating Flow Control Chart symbols: Create Exec Block, Create Async Call, Create Termination, Create Connection, see Figure 77.

The **Action Commands** display the available commands for managing the application code and Flow Control Chart: Render Chart, Save, Check Syntax, Compile, see Figure 77.

The **Flow Control Chart** displays the Flow Control Chart illustration of the application code structure.

The **Code Text Editor** displays the application code.

The **Commands Output** displays output messages from Action Commands.

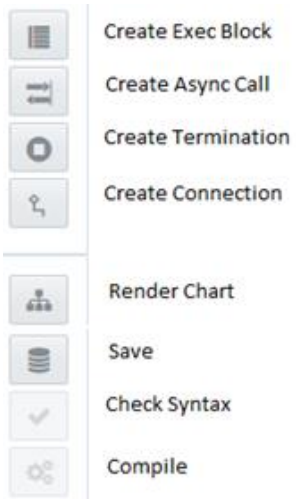


Figure 77: Toolbox and Actions

10.3 Code Text Editor

The Code Text Editor includes the drop-down menus File, Edit, Exec Blocks and a **Fit Resize** button.

Figure 78: Code Text Editor

The File drop-down menu contains the following commands:

- Open (for uploading the Perl script from the selected file)
- Save (for saving the Perl script as a file)

The Exec Blocks drop-down menu contains all executions blocks present on the Flow Control Chart, providing navigation of code subroutines.

The Edit drop-down menu contains the following commands:

- Undo (the command erases the last change in the code, revert it to an older state)
- Redo (the command reverses the undo)
- Find/ Replace (the command searches for a specified text and when found, replace it with another specified text).

The application code consists of:

- Internal variable declarations
- The main subroutine
- Other referenced subroutines (if any)

The Code Text Editor provides automatic text markers that cause parts of the Perl code to be distinctively highlighted to the user for the following code elements:

- Perl subroutines
- DCA API asynchronous calls
- DCA API termination calls
- DCA API call

10.4 Flow Control Chart

- The Toolbox commands are used to add symbols and connections to the Flow Control Chart area to illustrate the flow of control between executions blocks and asynchronous calls for the application.
- The symbol can be added to the Flow Control Chart area or adjusted by dragging the symbol to the desired location. The connection is maintained during dragging.
- The Render Chart action command renders a new Flow Control Chart from the application code in the Code Text Editor.
- The Save action command saves the chart data along with the application code.
- There is a pan and zoom control panel above the Flow Control Chart for adjusting the location and scale of the visible chart boundaries to reveal hidden chart content.
- The symbol selected in the Flow Control Chart area becomes highlighted. Selecting the symbol in the Flow Control Chart area causes the Code Text Editor to select the code associated with the symbol and to scroll the selected code into view.
- The start of the connection line is marked by the circle representing an exit from a previous symbol, and the other end of the line marks an entry to a symbol.

10.4.1 Start Symbol

- Start symbol is automatically created by the Render Chart action command. Start symbols are connected to the Execution Block symbols for the request and answer subroutines specified on the Application General Options screen (Section 9.2.3).
- If the subroutine names are not yet configured, or the configured names do not match any subroutines in the code, Start symbols are rendered in the chart with no connections and are marked with validation errors.
- Selecting the Start symbol causes the Code Text Editor to scroll to the first line of code for the connected Execution block.

10.4.2 Execution Block Symbol

- The name of an Execution Block symbol is also the name of the subroutine.
- An Execution Block has one entry connection, which can proceed from one of the following symbols: Start, execution Block, or Async Call.
- An Execution Block has one or more exit connections to any of the following symbols: Execution Block, Termination, or Async Call.
- Selecting the Execution Block symbol causes the Code Text Editor to select the code for the associated subroutine and scroll it into view.

10.4.3 Asynchronous Call Symbol

- The Asynchronous Call symbol displays the name of the asynchronous function that is invoked.
- An Async Call symbol always has an entry connection from an Execution Block Symbol.
- ***Note***: A symbol inserted by the user with no connections is ignored.
- An Async Call symbol always has an exit connection to an Execution Block that has a name, which matches the callback subroutine name input parameter to the asynchronous call.
- Selecting the Async Call symbol displays the asynchronous function call statement in the Code Text Editor.

10.4.4 Termination Symbol

- The Termination symbol displays a final action name that corresponds to an occurrence of a termination call in the Execution Block connected to the Termination symbol.
- The allowed final action names are Forward, Drop, Answer.
- A Termination symbol can only have an entry connection and no exit connection.
- A Termination symbol can only be connected to the exit of an Execution Block symbol.
- Selecting the Termination symbol displays the code statement for the final action in the Code Text Editor.

10.4.5 Delete symbol from the Flow Control Chart

- Right-clicking the symbol or connection in the Flow Control Chart area causes a hidden menu to be displayed.
- The menu associated with every chart symbol and connection displays the following commands:
  - Delete (selecting this command deletes the symbol or connection from the Flow Control Chart)

    Delete is enabled only for symbols/connections created using the Toolbox commands and that are not yet associated to the code.

    Note: Symbols and connections created using the Render Chart cannot be deleted because they are associated with code in the Code Text Editor. The associated code must be deleted first, and then **Render Chart** can update the Flow Control Chart to reflect the deletion.
  - Rename (selecting this command enables renaming the symbol).

    Note: **Rename** displays only for the exec, async, and termination blocks and not for connections.

10.4.6 Flow Control Chart Validation

- The Flow Control Chart validation (validation errors show up and/or clear) is triggered by the following events:
  - A new chart is rendered by the Render Chart command
  - Right as a change is made
- The Flow Control Chart Validation finds errors in the structure of the application code before the Compile action command is clicked.
- An error icon is displayed beside each chart symbol name that has a validation error.
- Hovering over the error icon of a symbol displays the validation error message for the symbol.
- Existing validation errors are cleared at the start of validation.
- The Flow Control Chart validates each Start symbol is connected to a single Execution Block. Otherwise, the **Start symbol has no Execution Block connection** error displays.
- The Flow Control Chart validates the Execution Block connected to a Start symbol has an exit connection that directly or indirectly leads to a Termination symbol. Otherwise, the **Starting Execution Block exit connection does not lead to a direct or indirect Termination** error displays.
- The Flow Control Chart validates each Async Call connects to a post-processing Execution Block. Otherwise, the **Async Call has no connection to a post-processing Execution Block** error displays.

  Note: This error message occurs when the Async Call statement in the code references a post-processing subroutine that does not exist.

10.4.7 Command Output Area

- The output messages from the action commands are displayed in the Command Output text area.
- When the Command Output text area gets full, the oldest text lines are removed to make room for the new lines.
- The Command Output text area is able to display a maximum of 500 lines of text.
- The Save, Check Syntax, and Compile action commands produce log events in the system log where DSR stores all GUI log events.
- Each log event includes the user, app ID, app name, app version, and action command name.

10.4.8 Render Chart

- The Render Chart action command analyzes the current application code and create a new Flow Control Chart to depict the code.
- The command output of the Render Chart action command reads "Rendering Chart…" and "Render Chart done".
- The Render Chart action command renders chart symbols on detecting subroutines, asynchronous calls and termination calls in the application code.
- The detection of a subroutine declaration works on the following coding convention:

  The declaration of a subroutine name and opening brace appear on the same line. The closing brace for the subroutine appears alone on a separate line. Example:

```
# Perl language example
sub process_request {
     # Code statements here . . .
}
```

- The detection of an asynchronous call declaration works on the following coding convention:

  For an asynchronous call statement the logical database name input argument must be passed as a literal string (quoted text) – not a variable or expression.

- For an asynchronous call statement, the callback input argument is the name of the post-processing subroutine where execution continues after the asynchronous call. The callback input argument must be passed as a literal string (quoted text) – not as a subroutine reference. For example:

```
# Perl language example
sub process_request {
     # Asynchronous call
     dca::sbr::sbrInstance("sbrDB")->read (
                  $subscription_type,  # key type
                  $subscription_key,   # key value
                  "read_result"  # callback to process result
       );
}
```

10.4.9 Save

- The Save action command is enabled in Edit Mode if the current known Application State is Development or Trial.
- The Save action command saves the application code and Flow Control Chart to the system database.
- The command output of the Save action command reads "Saving…" and "Save done".

10.4.10 Check Syntax

- The Check Syntax action command makes the Perl interpreter check the syntax of the last saved code and report any syntax errors.
- The Check Syntax action command is enabled in Edit Mode if the current known Application State is Development or Trial and the application code and chart have been saved.
- Click **Check Syntax** to retrieve the latest application data and compare current application data.

10.4.11 Compile

- The Compile action command compiles the application code and is enabled in Edit mode if the current known Application State is Trial, and this action command has not been run since the last Check Syntax Action command was executed.
- The command output of the Compile action command reads "Compiling…" and "Compile done".

10.5 Race Conditions

- If multiple users are changing the application version code/flowchart simultaneously, only the first one is able to submit the changes (commands Save, Check Syntax, Compile). If the rest are trying to submit the changes, the flowchart checksum validation fails and they would not be able to overwrite the code/flowchart in the database.
- If the Save action command is clicked while the current application state is Development or Trial, and the last-saved checksum has changed, saving is aborted and the error message displays:

  ```
  "Action command 'Check Save' aborted.  A newer version of the application
  code and Flow Control Chart has been saved in the system.
  Select OK to overwrite the latest saved data.
  Select Cancel to close without overwriting."
  ```

  If the user confirms, overwrite the latest version of the code and Flow Control chart with the current application data.
- If the Check Syntax action command is clicked while the current application state is Development or Trial, and the last-saved checksum has changed, checking syntax is aborted and the error message displays:

  ```
  "Action command 'Check Syntax' aborted.  A newer version of the
  application code and Flow Control Chart has been saved in the system.
  Select OK to overwrite the latest saved data.
  Select Cancel to close without overwriting."
  ```

  If the user confirms, overwrite the latest version of the code and Flow Control chart with the current application data.
- If the Compile action command is clicked while the current application state is Trial, and the last-saved checksum has changed, compiling is aborted and the error message displays:

  ```
  "Action command 'compile' aborted.  A more recent version of the code and
  Flow Control Chart exists.  Do you want to overwrite the current code and
  Flow Control chart with the latest data?"
  ```

  If the user confirms, overwrite the latest version of the code and Flow Control chart with the current application data.

- If multiple users are working with the application version, and there is an attempt to submit the code/flowchart changes (commands Save, Check Syntax, Compile) while the application state has changed and now is inappropriate for the code/flowchart update, the error occurs. (The web application running in the browser polls the web server every 10 seconds to get the latest application data and check for an application state change. The enabled/disabled state of **Save/Check Syntax/Compile** is only accurate within a 10 seconds time window).
- If **Save** is clicked while the current application state is Production or Archived, saving is aborted and the error message displays:

  ```
  "Action command 'Save' aborted.  The Application Version Status has
  changed from '<state>' to '<state>' which is invalid for the action
  command."
  ```

  A confirmation dialog box displays with **Ok** and **Cancel** and the following text:

  ```
  "The Application Version Status has changed from '<state>' to '<state>'.

  You will now be switched to View Mode and will not be able to save
  changes.

  Select OK to load and view the latest saved data.

  Select Cancel to continue viewing the current data."
  ```

  If the user confirms, the current code and Flow Control chart is overwritten with the latest data.
- If the Check Syntax action command is clicked while the current application state is Production or Archived, checking syntax is aborted and the error message displays:

  ```
  "Action command 'Check Syntax' aborted.  The Application Version Status
  has changed from '<state>' to '<state>' which is invalid for the action
  command."
  ```

  A confirmation dialog box displays **Ok** and **Cancel** and the following text:

  ```
  "The Application Version Status has changed from '<state>' to '<state>'.

  You will now be switched to View Mode and will not be able to save
  changes.

  Select OK to load and view the latest saved data.

  Select Cancel to continue viewing the current data."
  ```

  If the user confirms, overwrite the current code and Flow Control chart with the latest data.
- If the Compile action command is clicked while the current application state is Development, Production or Archived, compiling is aborted and the error message displays:

  ```
  "Action command 'Compile' aborted.  The Application Version Status has
  changed from '<state>' to '<state>' which is invalid for the action
  command."
  ```

11. APIs

This chapter documents the various APIs available to a DCA App programmer.

11.1 The EDL API

11.1.1 API to Manipulate the Diameter Header

Purpose: Retrieve the Diameter message object needed for subsequent operations on the Diameter message header and body.

Prototype:

```
my $msg = diameter::Param::message($param);
```

where `$param` is a default parameter provided by all the event handlers and may be retrieved with:

```
my $param = shift;
```

Purpose: Read the Diameter version number in the Diameter header.

Prototype:

```
my $ver = diameter::Message::version($msg);
```

where `$ver` is `undef` in case of failure (e.g., wrong object passed in `$msg`) or the Diameter version number if success.

Purpose: Set the Diameter version number in the Diameter header.

Prototype:

```
$err = diameter::Message::setVersion($msg, $ver);
```

where `$err` is `undef` in case of failure (e.g., wrong object passed in `$msg`) or a non-zero value in case of success.

Purpose: Return the length (as number of bytes) of the Diameter message.

Prototype:

```
my $len = diameter::Message::messageLength($msg);
```

where `$len` is `undef` in case of failure (e.g., wrong object passed in `$msg`) or the length of the Diameter message if success

Purpose: Read the Command Flags of the Diameter message.

Prototype:

```
my $cmdFlags = diameter::Message::commandFlags($msg);
```

where `$cmdFlags` is `undef` in case of failure (e.g., wrong object passed in `$msg`) or the Command Flags if success.

Purpose: Read the Request flag of the Diameter message.

Prototype:

```
my $r = diameter::Message::isRequest($msg);
```

where `$r` is 1 if the Request flag is set, 0 if the Request flag is not set, or `undef` if error (e.g., wrong object passed in `$msg`).

Purpose: Read the Diameter Proxiable flag in the Diameter header.

Prototype:

```
my $p = diameter::Message::isProxiable($msg);
```

where `$p` is 1 if the Proxiable flag is set, 0 if the Proxiable flag is not set or `undef` if error (e.g., wrong object passed in `$msg`).

Purpose: Set (set to 1) the Diameter Proxiable flag in the Diameter header.

Prototype:

```
$err = diameter::Message::setProxiable($msg);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Clear (set to 0) the Diameter Proxiable flag in the Diameter header.

Prototype:

```
$err = diameter::Message::clearProxiable($msg);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Read the Diameter Error flag in the Diameter header.

Prototype:

```
my $e = diameter::Message::isError($msg)
```

where `$e` is 1 if the Error flag is set, 0 if the Error flag is not set or `undef` if error (e.g., wrong object passed in `$msg`).

Purpose: Set (set to 1) the Diameter Error flag in the Diameter header.

Prototype:

```
$err = diameter::Message::setError($msg);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Clear (set to 0) the Diameter Error flag in the Diameter header.

Prototype:

```
$err = diameter::Message::clearError($msg);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Read the Diameter Retransmission flag in the Diameter header.

Prototype:

```
my $t = diameter::Message::isRetransmission($msg);
```

where `$t` is 1 if the Retransmission flag is set, 0 if the Retransmission flag is not set or `undef` if error (e.g., wrong object passed in `$msg`).

Purpose: Set (set to 1) the Diameter Retransmission flag in the Diameter header.

Prototype:

```
$err = diameter::Message::setRetransmission($msg);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Clear (set to 0) the Diameter Retransmission flag in the Diameter header.

Prototype:

```
$err = diameter::Message::clearRetransmission($msg);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Read the Diameter 4th reserved bit of the Command Flags in the Diameter header.

Prototype:

```
my $r4 = diameter::Message::isReservedBit4($msg);
```

where `$t` is 1 if the 4th bit in the Command Flags flag is set, 0 if the bit is not set or `undef` if error (e.g., wrong object passed in `$msg`).

Purpose: Set (set to 1) the Diameter 4th reserved bit of the Command Flags in the Diameter header.

Prototype:

```
$err = diameter::Message::setReservedBit4($msg);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Clear (set to 0) the Diameter 4th reserved bit of the Command Flags in the Diameter header.

Prototype:

```
$err = diameter::Message::clearReservedBit4($msg);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Read/Set/Clear the Diameter 5th, 6th, and 7th reserved bit in the Command Flags in the Diameter header.

Prototype:

See three examples above where the Bit4 suffix in the function names is accordingly replaced by Bit5, Bit6, and Bit7, respectively.

Purpose: Read the Diameter Command Code in the Diameter header.

Prototype:

```
my $cmd = diameter::Message::commandCode($msg);
```

where `$cmd` is `undef` if error (e.g., wrong object passed in `$msg`) or contains the Command Code if success.

Purpose: Set the Diameter Command Code in the Diameter header.

Prototype:

```
$err = diameter::Message::setCommandCode($msg, $cmd);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Read the Diameter Application-ID in the Diameter header.

Prototype:

```
my $appId = diameter::Message::applicationId($msg);
```

where `$appId` is `undef` if error (e.g., wrong object passed in `$msg`) or contains the Application-ID if success.

Purpose: Set the Diameter Application-ID in the Diameter header.

Prototype:

```
$err = diameter::Message::setApplicationId($msg, $appId);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Read the Diameter Hop-by-Hop Identifier in the Diameter header.

Prototype:

```
my $hbh = diameter::Message::hopByHopId($msg);
```

where `$hbh` is `undef` if error (e.g., wrong object passed in `$msg`) or contains the Hop-by-Hop Identifier if success.

Purpose: Set the Diameter Hop-by-Hop Identifier in the Diameter header.

Prototype:

```
$err = diameter::Message::setHopByHopId($msg, $hbh);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or a non-zero value if success.

Purpose: Read the Diameter End-to-End Identifier in the Diameter header.

Prototype:

```
my $err = diameter::Message::endToEndId($msg);
```

where `$err` is `undef` if error (e.g., wrong object passed in `$msg`) or contains the End-to-End Identifier if success.

Purpose: Set the Diameter End-to-End Identifier in the Diameter header.

Prototype:

```
$err = diameter::Message::setEndToEndId($msg, $e2e);
```

where `$err` is `undef` if error (e.g wrong object passed in `$msg`) or a non-zero value if success.

11.1.2 API to Manipulate the Diameter AVPs

Purpose: Read from a Diameter message the value of an AVP identified by name and instance number.

Prototype:

```
my $val = diameter::Message::getAvpValue($msg, $avp_name [,
$instance]);
```

The return values are:

- `undef` if `$instance` is 0.
- `undef` if there are less instances of the AVPin the Diameter message than the `$instance` value or an AVP with the specified name does not exist in the Diameter message or the AVP name is not specified in the AVP Dictionary.
- The value of the `$instance`-th instance of the AVP (starting from 1).
- The value of the first instance of the AVP if `$instance` has been omitted.

- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Purpose: Add at the end of the Diameter message an AVP identified by name and value.

Prototype:

```
my $err = diameter::Message::addAvpValue($msg, $avp_name, $avp_val);
```

The return values are:

- Non-zero in case of success.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if the AVP name exists in the AVP Dictionary.
- `undef` if the AVP value cannot be converted to the AVP data type specified in the AVP Dictionary.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Purpose: Set the value of an AVP in a Diameter message.

Prototype:

```
my $err = diameter::Message::setAvpValue($msg, $avp_name, $avp_val [,
$instance]);
```

If `$instance` has been omitted, the first instance of the AVP is set.

The return values are:

- Non-zero in case of success.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if the AVP name exists in the AVP Dictionary.
- `undef` if the AVP name is valid but no such AVP exists in the Diameter message.
- `undef` if `$instance` is 0.
- `undef` if the AVP exists in the Diameter message but `$instance` value is greater than the number of AVP instances in the Diameter message.
- `undef` if the AVP value cannot be converted to the AVP data type specified in the AVP Dictionary.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Purpose: Set the value of an existing AVP in a Diameter message or add that AVP at the end of the Diameter message if the message already contains exactly `$instance` – 1 AVPs.

Prototype:

```
my $err = diameter::Message::setAddAvpValue($msg, $avp_name, $avp_val
[, $instance]);
```

If `$instance` has been omitted, it defaults to 1.

The return values are:

- 1 in case an AVP with the specified instance number exists and its value has been successfully set.

- 2 if the Diameter messages contains exactly `$instance` – 1 AVPs of the specified type, in which case the `$instance's` AVP is added to the end of the message.
- `undef` if the Diameter messages contains strictly less than `$instance` – 1 AVPs of the specified type.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if the AVP name exists in the AVP Dictionary.
- `undef` if the AVP name is valid but the Diameter messages already contains `$instance` or more AVPs of the specified type.
- `undef` if `$instance` is 0.
- `undef` if the AVP value cannot be converted to the AVP data type specified in the AVP Dictionary.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Purpose: Read the value of an AVP's flag octet.

Prototype:

```
my $flags = diameter::Message::getAvpFlags($msg, $avp_name [,
$instance]);
```

The return values are:

- The value of flags octet of the `$instance`-th instance of the AVP (starting from 1).
- The value of the first instance of the AVP if `$instance` has been omitted.
- `undef` if there are less instances of the AVP in the Diameter message than the `$instance` value.
- `undef` if `$instance` is 0.
- `undef` if an AVP with the specified name does not exist in the Diameter message.
- `undef` if the AVP name is not specified in the AVP Dictionary.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Purpose: Set the value of an AVP's flag octet.

Prototype:

```
my $err = diameter::Message::setAvpFlags($msg, $avp_name, $mask [,
$instance]);
```

A 1 bit in `$mask` indicates a bit to set, while a 0 bit in `$mask` preserves the original bit value.

If `$instance` has been omitted, the flags of the first instance of the AVP is set.

The return values are:

- Non-zero in case of success.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if the AVP name is valid but no such AVP exists in the Diameter message.
- `undef` if the AVP exists in the Diameter message but `$instance` value is greater than the number of AVP instances in the Diameter message.

- `undef` if `$instance` is 0.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Note: The V bit preserves the original value regardless the `$mask` value.

Purpose: Clear specific bits in an AVP's flag.

Prototype:

```
my $err = diameter::Message::clearAvpFlags($msg, $avp_name, $mask [,
$instance]);
```

A 1 bit in `$mask` indicates a bit to clear, while a 0 bit in `$mask` preserves the original bit value.

If `$instance` has been omitted, the flags first instance of the AVP is cleared.

The return values are:

- Non-zero in case of success.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if the AVP name is valid but no such AVP exists in the Diameter message.
- `undef` if the AVP exists in the Diameter message but `$instance` value is greater than the number of AVP instances in the Diameter message.
- `undef` if `$instance` is 0.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Note: The V bit preserves the original value regardless the `$mask` value.

Purpose: Delete an AVP identified by name, from a Diameter message.

Prototype:

```
my $err = diameter::Message::delAvp($msg, $avp_name [, $instance]);
```

If `$instance` has been omitted, the first instance of the AVP is deleted.

The return values are:

- 1 in case AVP is deleted.
- 0 if AVP does not exist in message.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if the AVP exists in the Diameter message but `$instance` value is greater than the number of AVP instances in the Diameter message.
- `undef` if `$instance` is 0.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Purpose: Delete all the instances of an AVP from a Diameter message.

Prototype:

```
my $err = diameter::Message::delAvpAll($msg, $avp_name);
```

The return values are:

- 1 in case AVP is deleted.
- 0 if AVP does not exist in message.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Note: The AVPs on the same nesting level are deleted, i.e., the un-grouped AVPs in a Diameter message, if the function is called with a Diameter message parameter or the AVPs in a specific grouped AVP that are not deeper nested in a further grouped AVP, if the function is called with a Grouped AVP parameter.

Purpose: Return the number of instances of an AVP from a Diameter message.

Prototype:

```
my $cnt = diameter::Message::countAvp($msg, $avp_name);
```

The return values are:

- 0 if the AVP does not exist in the Diameter message.
- A strictly positive number indicating the number of occurrences of the respective AVP in the Diameter message.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Note: The AVPs on the same nesting level are counted, i.e., the un-grouped AVPs in a Diameter message, if the function is called with a Diameter message parameter or the AVPs in a specific grouped AVP that are not deeper nested in a further grouped AVP, if the function is called with a Grouped AVP parameter.

Purpose: Check whether a specific AVP (instance) exists in a Diameter message.

Prototype:

```
my $exists = diameter::Message::avpExists($msg, $avp_name [,
$instance]);
```

The return values are:

- True if `$instance` is omitted and at least one AVP with the specified name exists.
- True if `$instance` is specified and an AVP with the specified name and instance number exists.
- False if no AVP with the specified name exists in the Diameter message.
- False if `$instance` is specified, at least one AVP with the specified name exists, but the number of instances of the respective AVP is strictly less than the specified `$instance`.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

Note: The AVPs on the same nesting level are checked, i.e., the un-grouped AVPs in a Diameter message, if the function is called with a Diameter message parameter or the AVPs in a specific grouped AVP that are not deeper nested in a further grouped AVP, if the function is called with a Grouped AVP parameter.

Purpose: Return the length of the payload of an AVP from a Diameter message.

Prototype:

```
my $len = diameter::Message::avpDataLength($msg, $avp_name [,
$instance]);
```

If `$instance` has been omitted, the length of the first instance of the AVP is returned.

The return values are:

- `undef` if no AVP with that name exists in the Diameter message.
- `undef` if `$instance` is specified but less than `$instance` AVPs exists in the Diameter message.
- A strictly positive number or 0, indicating the length of the payload of the indicated AVP instance.
- `undef` if the AVP name does not exist in the AVP Dictionary.
- `undef` if `$msg` does not contain a `diameter::Message` or `diameter::GroupedAvp` object or the other parameters (if any) are `undef`.

11.1.3 API to Manipulate the Diameter Grouped AVPs

All the API functions introduced in the previous section, work on grouped AVPs as well. For instance, the value of the Subscription-Id grouped AVP may be read with:

```
my $gVal = diameter::Message::getAvpValue($msg, "Subscription-Id");
```

and the Subscription-Id grouped AVP may be added to a Diameter message with:

```
my $err = diameter::Message::addAvpValue($msg, "Subscription-Id",
$gVal);
```

Note that in this case, `$gVal` is an OctetString that contains both the Subscription-Id-Type and the Subscription-Id-Data AVPs.

This approach is particularly handy when the Subscriber-Id grouped AVP needs to be copied from one Diameter message to another, without having to look into the individual AVPs included in it.

However, if accessing the individual AVPs included into a grouped AVP is desired, then the `getGroupedAvp` and `addGroupedAvp` API calls provide the necessary support:

Purpose: Access a Grouped AVP in a Diameter message.

Prototype:

```
my $gAvp = diameter::Message::getGroupedAvp($msg, $avp_name [,
$instance]);
```

The return values are:

- `undef` if the AVP name does not exist in the AVP dictionary.
- `undef` if AVP name exists in the AVP dictionary but it is not defined as a Grouped AVP.

- `undef` if the AVP name is valid but the Diameter message does not contain a Grouped AVP with that name.
- `undef` if the AVP name is valid but the Diameter message contains less Grouped AVPs with that name than specified in `$instance`.
- A `diameter::GroupedAvp` Grouped AVP object that corresponds to the respective instance of the Grouped AVP (or to the first instance if `$instance` is omitted).

The `$gAvp diameter::Grouped` AVP object can be used to manipulate the AVPs that it contains using any of the API functions introduced so far:

```
$result = diameter::GroupedAvp::<API_function>($gAVP,
<API_function_params>);
```

where the `$gAVP` object of type `diameter::GroupedAvp` replaces the `$msg` object of type `$diameter`::`Message` and `$result` represents the return parameter of the respective API function..

Note: `getGroupedAvp` works recursively to get a grouped AVP (`$nested`_gAVP) contained in another grouped AVP (`$gAvp`):

```
my $nested_gAvp = diameter::Message::getGroupedAvp($gAvp,
$avp_name);
```

where `$gAvp` is a `diameter::GroupedAvp` object

Purpose: Add a Grouped AVP to the end of a Diameter message.

Prototype:

```
my $gAvp = diameter::Message::addGroupedAvp($msg, $avp_name);
```

where $gAvp is a `diameter::GroupedAvp` object.

The return values are:

- `undef` if the AVP name does not exist in the AVP dictionary.
- `undef` if AVP name exists in the AVP dictionary but it is not defined as a Grouped AVP.

A `diameter::GroupedAvp` Grouped AVP object can be further used to manipulate the AVPs that it contains:

```
my $subscription_id = diameter::Message::addGroupedAvp($msg,
"Subscription-Id");

diameter:GroupedAvp::addAvpValue($subscription_id, "Subscription-Id-
Type", $avp_val);

diameter::GroupedAvp::addAvpValue($subscription_id, "Subscription-Id-
Data", $avp_val);
```

Note: `addGroupedAvp` works recursively to add a grouped AVP (`$nested`_gAVP) within another grouped AVP (`$gAvp`):

```
my $nested_gAvp = diameter::Message::addGroupedAvp($gAvp,
$avp_name);
```

where `$gAvp` is a `diameter::GroupedAvp` object.

11.2 Diameter Transaction Stateful APIs

11.2.1 Internal Variables

This API is primary intended to enable a DCA App to interact with Mediation Rules through Internal Variables. Internal Variables have been introduced by the Mediation feature and can be configured from **Main Menu: Diameter→Mediation→Internal Variables**. Internal Variables are persistent throughout the lifetime of a Diameter transaction.

Purpose: Access Internal Variables.

Prototype:

```
my $iv_ref = new diameter::InternalVarDef("<IV_Name>");

my $internalVarMap = diameter::Param::internalVarMap($param);
```

where `$param` is the opaque parameter passed to every event handler and `<IV_Name>` is the name assigned to the Internal Variable in **Main Menu: Diameter→Mediation→Internal Variables**.

Note: The Internal Variables are configurable at the B level, therefore the `<IV_Name>` must be configured on all the sites. Otherwise, the initialization fails when invoked on those DA-MP located in sites where `<IV_Name>` does not exist.

Purpose: Set and Get Internal Variables.

Prototype:

```
diameter::InternalVarMap::set($internalVarMap, $iv_ref, $val);

$val = diameter::InternalVarMap::get($internalVarMap, $iv_ref);
```

Enables setting values to and retrieving values from an internal variable, where `$iv_ref` and `$internalVarMap` are initialized as shown before.

11.2.2 Diameter Transaction Context Variables

The Diameter transaction context variables offer Diameter transaction persistent storage, similar to Internal Variables. Unlike Internal Variables, Diameter transaction context variables are not configurable via the GUI (which provides for a much simpler API) and cannot be shared with other features.

Purpose: Store Diameter transaction context variables

Prototype:

```
$err = dca::transctx::store("<var_id>", $var)
```

The function returns `undef` if `$var` is `undef` or any error occurs (e.g., `$var` is a Perl hash or array that cannot be successfully encoded into JSON or DSR cannot allocate more memory space for the Diameter context variable) and 1 if the operation is successful.

Purpose: Retrieve Diameter transaction context variables

Prototype:

```
$var = dca::transctx::fetch("<var_id>");
```

`undef` is returned in case of failure (e.g., `<var_id>` is not found because no variable with that name has been previously stored).

11.3 Read DCA App Configuration Data

This API enables a DCA App to access its configuration data, which was specified and provisioned as described in Sections 3.3.3 and 3.3.4.

When the Perl script is generated, the DCA App configuration data is converted into a Perl variable. The Perl variable name is `%dca::appConfig` and is a hash (one key for each table) of arrays (one index for each record) of hashes (one key for each field in the table).

Read-only access on the DCA App configuration data is enforced using the Const::Fast CPAN module and applies to the data included in the `%dca::appConfig` definition (which is automatically generated from the DCA App configuration data).

Note that there are semantical differences from one Const::Fast version to another, which affect the way `%dca::appConfig` can be subsequently manipulated in the Perl script with regard to adding new records to `%dca::appConfig` or accessing records that are not defined in `%dca::appConfig`.

For instance, in version 0.006, which is the one currently used, an attempt to read or assign a value to an inexistent table (outermost hash key) `%dca::appConfig` results in a runtime error.

On the other hand, assigning values to inexistent indexes (table records) and/or inexistent fields (innermost hash key) succeeds and can be subsequently successful read, while reading from inexistent indexes and/or inexistent fields return `undef`. These indexes and fields are not written back to the DCA App configuration data.

Purpose: Read the DCA App configuration data.

Prototype:

```
$dca::appConfig{"<config_table_name>"}[<row_index>]{"<field_name>"}
```

for non-"single row" configuration tables.

```
$dca::appConfig{"<config_table_name>"}{"<field_name>"}
```

for "single row" configuration tables.

Example: Assuming a DCA App defines a configuration table called "MyTable" with two fields "FieldA" and "FieldB" and provisions a few rows, it is possible to retrieve the NOAM and SOAM provisioned data from the DCA app in the following way:

```
for $i (0 .. $#dca::appConfig{"MyTable"} ) {
dca::application::logInfo($dca::appConfig{"MyTable"}[$i]{"Field1"});
dca::application::logInfo($dca::appConfig{"MyTable"}[$i]{"Field2"});
}
```

11.4 Routing API

The routing API enables a DCA App to perform some basic routing functions.

The `dca::action::forward()`, `dca::action::answer($ans)` and `dca::action::drop()` API functions terminate the execution of the event handler. This means that the statements that follow them in the Perl code are not executed. This also has a side effect on the U-SBR queries initiated before invoking any of `dca::action::forward()`, `dca::action::answer($ans)` and `dca::action::drop()` because, as mentioned in Section 6.3.6.2, the U-SBR queries are actually sent after the execution of the

event handler completes: the side effect is the U-SBR queries are also not executed (i.e., sent to the U-SBR).

Besides `dca::action::forward()`, `dca::action::answer($ans)` and `dca::action::drop()`, an event handler's execution flow also terminates (as any other Perl subroutine) when a `return` statement is encountered or when the enclosing curly bracket is reached. In this case, the implicit routing decision that the DCA framework takes depends on the Perl subroutine return value:

- If the return value is greater or equal to 0, then the Diameter message is forwarded.
- If the return value is negative, then the runtime error behavior (Section 3.3.1) is executed.

Purpose: Complete the processing and drop the message.

Prototype:

```
dca::action::drop();
```

Note: Invoking `dca::action::drop()` causes the event handler to immediately terminate execution.

Purpose: Build a Diameter Answer.

Prototype:

```
$ans = new dca::application::answer(<error_code>, <error_text>,
<vendor_id>);
```

The function returns `undef` in case of failure or a `diameter::Message` object.

When receiving a Diameter request or answer this API function enables a DCA App to construct a Diameter answer and either return it to the originator of the corresponding Diameter request or, respectively, substitute the original Diameter answer message.

The EDL API (see Section 11.1) may be used to further process the `$ans` Diameter answer (e.g., add more AVPs).

Purpose: Send a Diameter Answer Created by the DCA App.

Prototype:

```
dca::action::answer($ans);
```

Note: Invoking `dca::action::answer($ans)` causes the event handler to immediately terminate execution.

Purpose: Complete the processing and pass the message.

Prototype:

```
dca::action::forward();
```

Enables a DCA App to pass a Diameter message to the Diameter Routing Layer for routing.

Note: Invoking `dca::action::forward()` causes the event handler to immediately terminate execution.

Purpose: Specify an ART based on which a Diameter request is routed.

Prototype:

```
$err = dca::route::setART(<ART_table_name>);
```

The function returns `undef` if the name of the ART does not exist (failure) or 1 if success.

Before invoking `dca::action::forward()` on a Diameter request, this routing API function enables a DCA App to specify which ART to be used for routing the respective Diameter request.

Note: The ART is configurable at the B level; therefore, the `<ART_table_name>` must be configured on all the sites. Otherwise, the API function fails when invoked on those DA-MP located in sites where `<ART_table_name>` does not exist.

Purpose: Specify a PRT based on which a Diameter request is routed.

Prototype:

$err = dca::route::setPRT(<PRT_table_name>);

The function returns `undef` if the name of the PRT does not exist (failure) or 1 if success.

Before invoking `dca::action::forward()` on a Diameter request, this routing API function enables a DCA App to specify which PRT to be used for routing the respective Diameter request.

Note: The PRT is configurable at the B level, therefore the `<PRT_table_name>` must be configured on all the sites. Otherwise, the API function fails when invoked on those DA-MP located in sites where `<PRT_table_name>` does not exist.

11.5 Debugging API

The Debugging API allows tracking the execution of the event handlers by supporting the equivalent of printf, log, echo, etc., functions in other programming/scripting languages.

The messages are logged in the **dsr.DCA** trace file (use **tr.tail** dsr.DCA). The following masks may be applied using the **tr.set** command to filter the ERROR, INFO, and WARNING error messages: 0x00000001 (error), 0x00000002 (info) and respectively 0x00000004 (warning).

All the traces generated by a DCA app using the API calls is prefixed with the DCA application name (to allow for further filtering, e.g., using the grep utility).

The `log[Info|Warn|Error]` API functions also generate an IDIH trace (see Section 11.8).

Note, however, that in a production network DSR logs only the vital traces are therefore the main debugging tool for DCA Apps in production networks is the IDIH feature.

Purpose: Retrieve the application name.

Prototype:

```
$appname = dca::application::getAppName();
```

Purpose: Retrieve the version name

Prototype:

```
$vername = dca::application::getVersionName();
```

Note: Besides debugging, another possible use case for reading the version name is including it in the DCA app state stored on the U-SBR. This supports backward compatibility in case the DCA app frequently changes the format of the DCA app across DCA app versions.

Purpose: Retrieve the current state.

Prototype:

```
$verstate = dca::application::getState();
```

Note: The states returned can be either Trial or Production, since these are the only states when the DCA App is executed.

Purpose: Generate a trace containing user-defined messages and having a severity of INFO.

Prototype:

```
dca::application::logInfo(<message>);
```

The user-defined messages is logged into dsr.DCA (tr.tail dsr.DCA).

Purpose: Generate a trace containing user-defined messages and having a severity of WARNING.

Prototype:

```
dca::application::logWarn(<message>);
```

Purpose: Generate a trace containing user-defined messages and having a severity of ERROR

Prototype:

```
dca::application::logErr(<message>);
```

11.6 Custom MEAL API

Once the Custom MEAL objects are differentiated from the **Main Menu: DCA Framework→<DCA App Name>→Custom MEALs** screen (see Section 9.2.2), they can be initialized and used from DCA Apps.

11.6.1 Counter Template API

Purpose: A DCA App is able to bind to a Scalar Counter Custom MEAL by referring to it by the Custom MEAL configured name.

Prototype:

```
my $all_Cnt = new dca::meal::counter("MyCnt");
```

where `"MyCnt"` is the name specified when differentiating a Custom MEAL template of type **Counter** and measurement type **Scalar**.

The API call returns a valid Custom MEAL object in case of success. The Custom MEAL object may be used in subsequent API calls to perform specific operations on the Scalar Counter.

In case of failure, `undef` is returned.

Possible failure cases are:

- No Custom MEAL with the specified name is currently defined.
- A Custom MEAL with that name exists, but either the differentiation process is not yet completed, or the un-differentiation process was initiated.
- A Custom MEAL with that name exists, but it is not a Scalar Counter.

Note: As a matter of best practice, the initialization of the Custom MEAL objects is performed in the main body of the Perl script, which is executed once right after a successful compilation (rather than in an event handler):

```
Die "Custom MEAL differentiation failure"
   unless $obj = new dca::meal::<TemplateType>("MyCustomMeal");
```

This ensures a compilation error is triggered if the binding process has failed, for instance because there is a name mismatch between the Perl script and the differentiation GUI screen. Using an undefined `$obj` later in the event handlers triggers run-time errors.

Purpose: A DCA App is able to peg a Scalar Counter Custom MEAL.

Prototype:

```
$err = $all_Cnt->peg();
```

where `$all_Cnt` is a valid Scalar Counter Custom MEAL object.

The API call returns 1 if success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to bind to an Arrayed Counter Custom MEAL by referring to it by the Custom MEAL configured name.

Prototype:

```
my $per_Cnt = new dca::meal::arrayedCounter("MyArrayedCnt");
```

where `"MyArrayedCnt"` is the name specified when differentiating a Custom MEAL template of type **Counter** and measurement type **Arrayed**.

The API call returns a valid Custom MEAL object in case of success. The Custom MEAL object may be used in subsequent API calls to perform specific operations on the Arrayed Counter.

In case of failure, `undef` is returned.

Possible failure cases are:

- No Custom MEAL with the specified name is currently defined.
- A Custom MEAL with that name exists, but either the differentiation process is not yet completed, or the un-differentiation process was initiated.
- A Custom MEAL with that name exists, but it is not an Arrayed Counter.

Purpose: A DCA App is able to peg a specific index of an Arrayed Counter Custom MEAL.

Prototype:

```
$err = $per_Cnt->peg($index);
```

where `$per_Cnt` is a valid Arrayed Counter Custom MEAL object and `$index` is the index to be pegged.

The API call returns 1 if success and `undef` if the either operation on the underlying Comcol object has failed or the index value is negative.

11.6.2 Rate Template

Purpose: A DCA App is able to bind to a Scalar Rate Custom MEAL by referring to it by the Custom MEAL configured name.

Prototype:

```
my $all_Rate = new dca::meal::rate("MyRate");
```

where `"MyRate"` is the name specified when differentiating a Custom MEAL template of type **Rate** and measurement type **Scalar**.

The API call returns a valid Custom MEAL object in case of success. The Custom MEAL object may be used in subsequent API calls to perform specific operations on the Scalar Rate.

In case of failure, `undef` is returned.

Possible failure cases are:

- No Custom MEAL with the specified name is currently defined.
- A Custom MEAL with that name exists, but either the differentiation process is not yet completed, or the un-differentiation process was initiated.
- A Custom MEAL with that name exists, but it is not a Scalar Rate.

Purpose: A DCA App is able to peg a Scalar Rate Custom MEAL.

Prototype:

```
$err = $all_Rate->peg();
```

where `$all_Rate` is a valid Scalar Rate Custom MEAL object.

The API call returns 1 if success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to read the current value of a Scalar Rate Custom MEAL.

Prototype:

```
$val = $all_Rate->readRate();
```

where `$all_Rate` is a valid Scalar Rate Custom MEAL object.

The API call returns an integer representing the current value in case of success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to read the average value of a Scalar Rate Custom MEAL.

Prototype:

```
$val = $all_Rate->readAvgRate();
```

where `$all_Rate` is a valid Scalar Rate Custom MEAL object.

The API call returns an integer representing the average value in case of success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to determine the current severity of the alarm associated to an Scalar Rate template.

Prototype:

```
$err = $all_Rate->getSeverity();
```

where `$all_Rate` is a valid Scalar Rate Custom MEAL object.

The API call returns:

```
dca::meal::Critical, dca::meal::Major, dca::meal::Minor,
dca::meal::Cleared
```

`undef` if the operation on the underlying Comcol object has failed.

Note: The severity values are defined as:

```
use constant {
    Cleared   => 0,
    Info      => 1,
    Minor     => 2,
    Major     => 3,
    Critical  => 4,
};
```

which enables comparing them. For instance:

```
if ($all_Rate->getSeverity() >= dca::meal::Major)
```

is true if the severity is Major or Critical and is false if the severity if Minor. This also applies to Basic as well as arrayed templates.

Purpose: A DCA App is able to bind to an Arrayed Rate Custom MEAL by referring to it by the Custom MEAL configured name.

Prototype:

```
my $per_Rate = new dca::meal::arrayedRate("MyArrayedRate");
```

where `"MyArrayedRate"` is the name specified when differentiating a Custom MEAL template of type **Rate** and measurement type **Arrayed**.

The API call returns a valid Custom MEAL object in case of success. The Custom MEAL object may be used in subsequent API calls to perform specific operations on the Arrayed Rate.

In case of failure, `undef` is returned.

Possible failure cases are:

- No Custom MEAL with the specified name is currently defined.
- A Custom MEAL with that name exists, but either the differentiation process is not yet completed, or the un-differentiation process was initiated.
- A Custom MEAL with that name exists, but it is not an Arrayed Rate.

Purpose: A DCA App is able to peg a specific index of an Arrayed Rate Custom MEAL.

Prototype:

```
$err = $per_Rate->peg($index);
```

where `$per_Rate` is a valid Arrayed Rate Custom MEAL object and `$index` is the index to be pegged.

The API call returns 1 if success and `undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

Purpose: A DCA App is able to read the current value of a specific index of an Arrayed Rate Custom MEAL.

Prototype:

```
$val = $per_Rate->readRate($index);
```

where `$per_Rate` is a valid Arrayed Rate Custom MEAL object and `$index` is the index the current value of which is read.

The API call returns an integer representing the current value of the specified index in case of success and `undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

Purpose: A DCA App is able to read the average value of a specific index of an Arrayed Rate Custom MEAL.

Prototype:

```
$val = $per_Rate->readAvgRate($index);
```

where `$per_Rate` is a valid Arrayed Rate Custom MEAL object and `$index` is the index the average value of which is pegged.

The API call returns an integer representing the average value of the specified index in case of success and `undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

Purpose: A DCA App is able to determine the current severity of the alarm associated to an Arrayed Rate template:

Prototype:

```
$err = $per_Rate->getSeverity($index);
```

where `$per_Rate` is a valid Arrayed Rate Custom MEAL object and `$index` identifies the particular index the alarm status of which is read.

The API call returns:

```
dca::meal::Critical, dca::meal::Major, dca::meal::Minor,
dca::meal::Cleared
```

`undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

11.6.3 Basic Template

Purpose: A DCA App is able to bind to a Scalar Basic Custom MEAL by referring to it by the Custom MEAL configured name.

Prototype:

```
my $all_Size = new dca::meal::basic("MyBasic");
```

where `"MyBasic"` is the name specified when differentiating a Custom MEAL template of type **Basic** and measurement type **Scalar**.

The API call returns a valid Custom MEAL object in case of success. The Custom MEAL object may be used in subsequent API calls to perform specific operations on the Scalar Basic template.

In case of failure, `undef` is returned.

Possible failure cases are:

- No Custom MEAL with the specified name is currently defined.
- A Custom MEAL with that name exists, but either the differentiation process is not yet completed, or the un-differentiation process was initiated.
- A Custom MEAL with that name exists, but it is not a Scalar Basic.

Purpose: A DCA App is able to set the value of a Scalar Basic Custom MEAL.

Prototype:

```
$err = $all_Size->setValue($value);
```

where `$all_Size` is a valid Scalar Basic Custom MEAL object and `$value` is the value the Scalar Basic Custom MEAL is set to.

The API call returns 1 if success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to increment the value of a Scalar Basic Custom MEAL.

Prototype:

```
$err = $all_Size->increment($count);
```

where `$all_Size` is a valid Scalar Basic Custom MEAL object and `$count` is the value the Scalar Basic Custom MEAL is incremented with.

The API call returns 1 if success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to decrement the value of a Scalar Basic Custom MEAL.

Prototype:

```
$err = $all_Size->decrement($count);
```

where `$all_Size` is a valid Scalar Basic Custom MEAL object and `$count` is the value the Scalar Basic Custom MEAL is decremented with.

The API call returns 1 if success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to read the current value of a Scalar Basic Custom MEAL.

Prototype:

```
$val = $all_Size->getValue();
```

where `$all_Size` is a valid Scalar Basic Custom MEAL object.

The API call returns an integer representing the current value in case of success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to read the average value of a Scalar Basic Custom MEAL.

Prototype:

```
$val = $all_Size->getAvgValue();
```

where `$all_Size` is a valid Scalar Basic Custom MEAL object.

The API call returns an integer representing the average value in case of success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to determine the current severity of the alarm associated to an Scalar Basic template.

Prototype:

```
$err = $all_Size->getSeverity();
```

where `$all_Size` is a valid Scalar Basic Custom MEAL object.

The API call returns:

```
dca::meal::Critical, dca::meal::Major, dca::meal::Minor,
dca::meal::Cleared
```

`undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to bind to an Arrayed Basic Custom MEAL by referring to it by the Custom MEAL configured name.

Prototype:

```
my $per_Size = new dca::meal::arrayedBasic("MyArrayedBasic");
```

where `"MyArrayedBasic"` is the name specified when differentiating a Custom MEAL template of type **Basic** and measurement type **Arrayed**.

The API call returns a valid Custom MEAL object in case of success. The Custom MEAL object may be used in subsequent API calls to perform specific operations on the Arrayed Basic template.

In case of failure, `undef` is returned.

Possible failure cases are:

- No Custom MEAL with the specified name is currently defined.
- A Custom MEAL with that name exists, but either the differentiation process is not yet completed, or the un-differentiation process was initiated.
- A Custom MEAL with that name exists, but it is not an Arrayed Basic.

Purpose: A DCA App is able to set the value of an Arrayed Basic Custom MEAL.

Prototype:

```
$err = $per_Size->setValue($value, $index);
```

where `$per_Size` is a valid Arrayed Basic Custom MEAL object, `$index` is the index the value of which is set and `$value` is the value it is set to.

The API call returns 1 if success and `undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

Purpose: A DCA App is able to increment the value of an Arrayed Basic Custom MEAL.

Prototype:

```
$err = $per_Size->increment($count, $index);
```

where `$per_Size` is a valid Arrayed Basic Custom MEAL object, `$index` is the index the value of which is incremented and `$count` is the value it is incremented with.

The API call returns 1 if success and `undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

Purpose: A DCA App is able to decrement the value of an Arrayed Basic Custom MEAL.

Prototype:

```
$err = $per_Size->decrement($count, $index);
```

where `$per_Size` is a valid Arrayed Basic Custom MEAL object, `$index` is the index the value of which is decremented and `$count` is the value it is decremented with.

The API call returns 1 if success and `undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

Purpose: A DCA App is able to read the current value of an Arrayed Basic Custom MEAL.

Prototype:

```
$val = $per_Size->getValue($index);
```

where `$per_Size` is a valid Arrayed Basic Custom MEAL object and `$index` is the index the value of which is read.

The API call returns an integer representing the current value of the specified index in case of success and `undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

Purpose: A DCA App is able to read the average value of an Arrayed Basic Custom MEAL.

Prototype:

```
$val = $per_Size->getAvgValue($index);
```

where `$per_Size` is a valid Arrayed Basic Custom MEAL object and `$index` is the index the average value of which is read.

The API call returns an integer representing the average value of the specified index in case of success and `undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

Purpose: A DCA App is able to determine the current severity of the alarm associated to an Arrayed Basic template.

Prototype:

```
$err = $per_Size->getSeverity($index);
```

where `$per_Size` is a valid Arrayed Basic Custom MEAL object and `$index` identifies the particular index the alarm status of which is read.

The API call returns:

```
dca::meal::Critical, dca::meal::Major, dca::meal::Minor,
dca::meal::Cleared
```

`undef` if either the operation on the underlying Comcol object has failed or the index value is negative.

11.6.4 Event Template

Purpose: DCA App is able to bind to an Event Custom MEAL by referring to it by the Custom MEAL configured name.

Prototype:

```
my $errorEvent = new dca::meal::event("MyEvent");
```

where `"MyEvent"` is the name specified when differentiating a Custom MEAL template of type **Event**.

The API call returns a valid Custom MEAL object in case of success. The Custom MEAL object may be used in subsequent API calls to perform specific operations on the Event.

In case of failure, `undef` is returned.

Possible failure cases are:

- No Custom MEAL with the specified name is currently defined.
- A Custom MEAL with that name exists, but either the differentiation process is not yet completed, or the un-differentiation process was initiated.
- A Custom MEAL with that name exists, but it is not a Event.

Purpose: A DCA App is able to generate an event (Info severity), raise an alarm (Minor, Major, Critical severity) and clear an alarm (Clear severity).

Prototype:

```
$err = $errorEvent->log($severity, $addInfoText);
```

where `$errorEvent` is a valid Event Custom MEAL object, `$severity` is one of the possible values (`dca::meal::Critical`, `dca::meal::Major`, `dca::meal::Minor`, `dca::meal::Cleared`) and `$addInfoText` is the text that should be included in the alarm's additional information field.

The API call returns 1 if success and `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to determine whether an event or alarm is throttled before trying to raise it (again).

Prototype:

```
$err = $errorEvent->isThrottled($severity);
```

where `$errorEvent` is a valid Event Custom MEAL object, `$severity` is one of the possible values (`dca::meal::Critical`, `dca::meal::Major`, `dca::meal::Minor`, `dca::meal::Info`).

The API call returns:

- 1 if the event/alarm is throttled.
- 0 if the event/alarm is not throttled.
- `undef` if the operation on the underlying Comcol object has failed.

Purpose: A DCA App is able to determine the current severity of an event or alarm:

Prototype:

```
$err = $errorEvent->getSeverity();
```

where `$errorEvent` is a valid Event Custom MEAL object.

The API call returns:

```
dca::meal::Critical, dca::meal::Major, dca::meal::Minor,
dca::meal::Info, dca::meal::Cleared
```

`undef` if the operation on the underlying Comcol object has failed.

11.7 U-SBR API

The U-SBR API enables a DCA App to create, read, update, and delete data in a U-SBR DB. As described in Section 6.3.6.2 the U-SBR API calls work asynchronously and a callback subroutine is necessary to fetch the result of the query.

11.7.1 The Prototype of Queries and Query Results

This section describes the common structure of the U-SBR API functions and how the results of a U-SBR query can be retrieved in the Perl script.

Section 11.7.2 further describes the particularities of each individual U-SBR API function.

11.7.1.1 Specifying the Query

All the U-SBR API functions share a common prototype:

```
$err = dca::sbr::sbrInstance(<usbr_logical_name>)-><API_function>(
                  <key_type>, <key_data_type>, $key,
                  <value_data_type>, $value,
                  <callback_subroutine>,
                  [<flags>]);
```

where:

- `<usbr_logical_name>` is a string (a constant value or a scalar variable) containing the logical name of the U-SBR DB the query is sent to. The logical names for the physical U-SBR DBs are

configured from **Main Menu: DCA Framework→<DCA App Name>→Application Control**, by selecting the DCA App version and clicking on **SBR DB Name Mapping**.

- `<API_function>` is one of: create, createOrRead, read, update, concurrentUpdate, and delete, respectively.
- `<key_type>` is typically a constant value defined by the DCA App. It distinguishes between different key types that a DCA App may use (e.g., IMSI, NAI, IP, IP_SRC, etc.). For example, the key value "fred" of type **NAI** is a different key from 66.72.65.64 of type **IP**, even though they have the same binary representation.
- `<key_data_type>` is pre-defined constant that describes the data type of the key and must be one of:
  - dca::sbr::KeyDataType::BCD – the key is a scalar.
  - dca::sbr::KeyDataType::UINT32 – the key is a scalar.
  - dca::sbr::KeyDataType::INT64 – the key is a scalar.
  - dca::sbr::KeyDataType::STRING – the key is a scalar.
  - dca::sbr::KeyDataType::IPv4 – the key is a NetAddr::IP object.
  - dca::sbr::KeyDataType::IPv6 – the key is a NetAddr::IP object.

  Note: There is no explicit data type for float numbers; float numbers are converted to strings.
- `$key` is a Perl variable that holds the key part of the key-value pair to be created, read, updated or deleted.
- `<value_data_type>` is pre-defined constant that describes the data type of the key and must be one of:
  - dca::sbr::StateDataType::BCD – the key is a scalar.
  - dca::sbr::StateDataType::UINT32 – the key is a scalar.
  - dca::sbr::StateDataType::STRING – the key is a scalar, an array reference or a hash reference.

    Note: Arrays and hashes are encoded into JSON and stored in the U-SBR DB in string format.
  - `dca::sbr::StateDataType::IPv4` – the key is a NetAddr::IP object.
  - `dca::sbr::StateDataType::IPv6` – the key is a NetAddr::IP object.

  Note: There is no explicit data type for float numbers; float numbers are converted to strings.
- `$value` is a Perl variable that holds the value part of the key-value pair to be written into the U-SBR (via create or update operations). Note, therefore, that read and delete do not specify a `$value` parameter and as a result no `<value_data_type>` parameter.
- `<callback_subroutine>` is a string representing the name of the Perl subroutine that are invoked by the DCA framework to deliver the query result,
- `<flags>` is an optional OR-mask of predefined flags that may apply to certain API functions.

The API call returns:

- 1 if the parameters are successfully parsed and encoding into a Stack Event.

  Note that, because the API call works asynchronously, at this stage the query has not been sent yet, its outcome cannot be known, `$err` merely tells whether a query could be successfully built.
- `undef` if parsing or encoding the parameters fails.

11.7.1.2 Retrieving the Query Result

The result of a U-SBR query can be retrieved in the callback function by using the `dca::sbr::result()` class. An error code is always returned and some queries also return data (consisting of the data type along with the data itself):

- $err_code = dca::sbr::result()->code();

Retrieves the error code. If the error codes indicates success (`dca::sbr::ResultCode::Ok`) then some API functions also return data, which can be retrieved using the `dataType()` and `data()` methods described below.

A number of error codes are common to all U-SBR API functions:

- `dca::sbr::ResultCode::Ok` – indicates the query has successfully executed the intended operation;
- `dca::sbr::ResultCode::DBError` – an error occurred on the SBR side that prevented the query to be executed;
- `dca::sbr::ResultCode::SendError` – an error occurred when attempting to send the query, typically because of ComAgent overload (ComAgent related alarms are raised in this case);
- `dca::sbr::ResultCode::LogicalNameMismatch` – indicates that no mappings to physical U-SBR DBs have been configured for the logical name used in the `<usbr_logical_name>` parameter. Alarm 33313 are raised;
- `dca::sbr::ResultCode::AccessError` – occurs when (i) the physical U-SBR DBs, to which the `<usbr_logical_name>` parameter is mapped to, are owned by another DCA App (see **Main Menu: SBR→Configuration→SBR Databases**, Owner Application column) and (ii) the current DCA App is configured to access the physical U-SBR DBs owned by other DCA Apps only in read-only mode (see **Main Menu: DCA Framework→<DCA App Name>→General Options**, **Read-Only U-SBR Access as Guest** option);
- `dca::sbr::ResultCode::MaxStateSize` – the size of either the key or the data, the DCA App attempts to look up or respectively store in the U-SBR DB, exceeds the configured maximum sizes (**Main Menu: DCA Framework→Configuration**, **Maximum Size of Application State** and respectively **Maximum Size of the Key** options)
- `dca::sbr::ResultCode::MaxEventReached` – the maximum number of U-SBR queries that a Diameter message event handler is allowed to send has been exceeded (see **Main Menu: DCA Framework→<DCA App Name>→General Options**, **Max. U-SBE Queries per Message** option).

A few error codes (dca::sbr::ResultCode::GenericErrRecExists, dca::sbr::ResultCode::GenericErrRecNotFound and dca::sbr::ResultCode::GenericErrRecObsoleted) are specific to certain U-SBR API functions.

- $data_type = dca::sbr::result()->dataType();

If the result contains data, then datatype() returns the data type of the stored data, i.e., one of: dca::sbr::StateDataType::BCD, dca::sbr::StateDataType::UINT32, dca::sbr::StateDataType::INT64, dca::sbr::StateDataType::STRING, dca::sbr::StateDataType::IPv4, dca::sbr::StateDataType::IPv6;

If the result contains no data, then `datatype()` returns `undef`.

- $data = dca::sbr::result()->data();

If the result contains data, then `data()` returns the stored data.

If the result contains no data, then `data()` returns `undef`.

11.7.2 The U-SBR API Functions

Purpose: Attempts to create a key-value record in a U-SBR DB or fails if a record with the same key already exists.

Prototype: (see also Section 11.7.1.1)

```
$err = dca::sbr::sbrInstance(<usbr_logical_name>)->create(
                  <key_type>, <key_data_type>, $key,
                  <value_data_type>, $value,
                  <callback_subroutine>);
```

Query Results: The possible result of the create API function are described in the table below (see also Section 11.7.1.2):

dca::sbr::result()->code()	**dca::sbr::result() ->dataType()**	**dca::sbr::result() ->data()**
dca::sbr::ResultCode::Ok (The record does not exist and was created)	N/A[1]	N/A
dca::sbr::ResultCode::DBError, dca::sbr::ResultCode::SendError dca::sbr::ResultCode::LogicalNameMismatch dca::sbr::ResultCode::AccessError dca::sbr::ResultCode::MaxStateSize dca::sbr::ResultCode::MaxEventReached	N/A	N/A
dca::sbr::ResultCode::ErrRecExists	N/A	N/A

Purpose: Creates a key-value record in a U-SBR DB or returns the record, if a record with the same key already exists.

Prototype: (see also Section 11.7.1.1)

```
$err = dca::sbr::sbrInstance(<usbr_logical_name>)->createOrRead(
                  <key_type>, <key_data_type>, $key,
                  <value_data_type>, $value,
                  <callback_subroutine>);
```

Query Results: The possible result of the create API function are described in the table below (see also Section 11.7.1.2):

dca::sbr::result()->code()	**dca::sbr::result() ->dataType()**	**dca::sbr::result() ->data()**
dca::sbr::ResultCode::Ok (The record does not exist and was created)	N/A	N/A
dca::sbr::ResultCode::DBError, dca::sbr::ResultCode::SendError dca::sbr::ResultCode::LogicalNameMismatch dca::sbr::ResultCode::AccessError dca::sbr::ResultCode::MaxStateSize dca::sbr::ResultCode::MaxEventReached	N/A	N/A

[1] The programmer does not rely on the returned variable being defined, undefined, or having any particular value.

dca::sbr::result()->code()	dca::sbr::result() ->dataType()	dca::sbr::result() ->data()
dca::sbr::ResultCode::ErrRecExists	The data type of the existing record	The existing record

Purpose: Reads the value associated to a key from the U-SBR DB, or fails if the key is not found.

Prototype: (see also Section 11.7.1.1)

```
$err = dca::sbr::sbrInstance(<usbr_logical_name>)->read(
                  <key_type>, <key_data_type>, $key,
                  <callback_subroutine>);
```

Note that no `$value` parameter is present since no value is supposed to be written into the U-SBR DB.

Query Results: The possible result of the create API function are described in the table below (see also Section 11.7.1.2):

dca::sbr::result()->code()	dca::sbr::result() ->dataType()	dca::sbr::result() ->data()
dca::sbr::ResultCode::Ok (The record exists and was read)	The data type of the existing record	The existing record
dca::sbr::ResultCode::DBError, dca::sbr::ResultCode::SendError dca::sbr::ResultCode::LogicalNameMismatch dca::sbr::ResultCode::AccessError dca::sbr::ResultCode::MaxStateSize dca::sbr::ResultCode::MaxEventReached	N/A	N/A
dca::sbr::ResultCode::ErrRecNotFound	N/A	N/A

Purpose: Attempts to update the value associated with a key in the U-SBR DB or fails if a record with the key could not be found.

Prototype: (see also Section 11.7.1.1)

```
$err = dca::sbr::sbrInstance(<usbr_logical_name>)->update(
                  <key_type>, <key_data_type>, $key,
                  <value_data_type>, $value,
                  <callback_subroutine>);
```

Query Results: The possible result of the create API function are described in the table below (see also Section 11.7.1.2):

dca::sbr::result()->code()	dca::sbr::result() ->dataType()	dca::sbr::result() ->data()
dca::sbr::ResultCode::Ok (The record exists and was updated)	N/A	N/A

dca::sbr::result()->code()	dca::sbr::result() ->dataType()	dca::sbr::result() ->data()
dca::sbr::ResultCode::DBError, dca::sbr::ResultCode::SendError dca::sbr::ResultCode::LogicalNameMismatch dca::sbr::ResultCode::AccessError dca::sbr::ResultCode::MaxStateSize dca::sbr::ResultCode::MaxEventReached	N/A	N/A
dca::sbr::ResultCode::ErrRecNotFound	N/A	N/A

Purpose: Attempts to update the value associated with a key that was previously retrieved (typically using a read or a createOrRead operation) from the U-SBR DB. It fails if the key-value record has been updated in the meantime by a concurrent update query.

Prototype: (see also Section 11.7.1.1)

```
$err = dca::sbr::sbrInstance(<usbr_logical_name>)->concurrentUpdate(
                  <key_type>, <key_data_type>, $key,
                  <value_data_type>, $value,
                  <callback_subroutine>);
```

Query Results: The possible result of the create API function are described in the table below (see also Section 11.7.1.2):

dca::sbr::result()->code()	dca::sbr::result() ->dataType()	dca::sbr::result() ->data()
dca::sbr::ResultCode::Ok (The record exists and was successfully updated)	N/A	N/A
dca::sbr::ResultCode::DBError, dca::sbr::ResultCode::SendError dca::sbr::ResultCode::LogicalNameMismatch dca::sbr::ResultCode::AccessError dca::sbr::ResultCode::MaxStateSize dca::sbr::ResultCode::MaxEventReached	N/A	N/A
dca::sbr::ResultCode::ErrRecNotFound	N/A	N/A
dca::sbr::ResultCode::ErrRecObsoleted (The record exists but was updated by a concurrent update query. The DCA App re-processes the returned value and retries the operation)	The data type of the updated record	The updated record

Purpose: Deletes a key-value record from the U-SBR DB, or fails if the key is not found.

Prototype: (see also Section 11.7.1.1)

```
$err = dca::sbr::sbrInstance(<usbr_logical_name>)->delete(
                  <key_type>, <key_data_type>, $key,
                  <callback_subroutine>);
```

Note that no `$value` parameter is present since no value is supposed to be written into the U-SBR DB.

Query Results: The possible result of the create API function are described in the table below (see also Section 11.7.1.2):

dca::sbr::result()->code()	dca::sbr::result() ->dataType()	dca::sbr::result() ->data()
dca::sbr::ResultCode::Ok (The record exists and was deleted)	N/A	N/A
dca::sbr::ResultCode::DBError, dca::sbr::ResultCode::SendError dca::sbr::ResultCode::LogicalNameMismatch dca::sbr::ResultCode::AccessError dca::sbr::ResultCode::MaxStateSize dca::sbr::ResultCode::MaxEventReached	N/A	N/A
dca::sbr::ResultCode::ErrRecNotFound	N/A	N/A

11.8 Peer Information

This section describes the APIs to fetch the Peer Information.

11.8.1 Check for Configured Peer.

A DCA app shall be able to check if a given Peer Name is configured in the system [SO] or not.

Prototype:

$status = dca::peerInfo::isPeerExists(<Peer name>);

Where <Peer Name> is the name of configured Peer in the system [SO] and $status is 1 if given <Peer Name> is configured in the SO GUI or 0 otherwise.

11.8.2 Fetch the Originator Peer.

When receiving a Diameter Message, a DCA app shall be able to fetch the Originator of the Diameter Message.

Prototype:

$peerName = dca::peerInfo::getOriginPeerName();

Where $peerName is the Peer Node name as configured in the SO GUI or undef in case of failure while fetching the Peer Node detail.

12. Interaction with IDIH

Table 4 illustrates the IDIH events generated by a DCA App.

Table 4: IDIH Events

Event ID	Event	Type	Scope	Instance Data	When Recorded
2300	Diameter Request processing routine invoked	Routine Invocation	App Data	• DCA application short name • Subroutine name	The Diameter Request processing routine of a DCA application is invoked by the DCA framework.

Event ID	Event	Type	Scope	Instance Data	When Recorded
2301	Diameter Answer processing routine invoked	Routine Invocation	App Data	• DCA application short name • Subroutine name	The Diameter Answer processing routine of a DCA application is invoked by the DCA framework.
2302	U-SBR Query send	U-SBR Query	App Data	• DCA application short name • Stack Event ID (create, read, update, …) • DAL ID of the application owning the U-SBR DB • Key value • Key type	An U-SBR query is prepared by a DCA application.
2303	Callback invoked	Routine Invocation	App Data	• DCA application short name • Callback name	An U-SBR query returns a result and a callback subroutine is invoked.
2304	Subroutine name not found	Execution Exception	App Data	• DCA application short name • Subroutine name	The scripting language interpreter returns an error indicating the subroutine doesn't exist.
2305	Runtime error	Execution Exception	App Data	• DCA application short name • Subroutine name • Error message returned by the interpreter	The scripting language interpreter returns an error indicating the a runtime error occurred.
2306	Debug message	Debug	App Data	• A debug message	A module included by default by the DCA framework enables DCA Apps to generate debug messages
2307	U-SBR Query Result received	U-SBR Query	App Data	• DCA application short name • Stack Event ID (create, read, update, …) • Data value • Data type • Query Result return code	An U-SBR query result is received by a DCA application.

Event ID	Event	Type	Scope	Instance Data	When Recorded
2308	U-SBR Query send failed	U-SBR Query	App Data	• DCA application short name • Error code	Sending the U-SBR query has failed because business logic related issues (e.g., max. limit of queries has been reached, L2P mapping error, etc.), due to ComAgent related issues (e.g., routing) or due to transport issues (e.g., timeout)

Except for event 2306, which is explicitly generated by the debug API functions, and event 2305, which is generated when a runtime error is encountered, all other events are generated automatically by the DCA framework when a specific point in the control flow is reached. For instance, Figure 79 illustrates the IDIH event trace of a U-SBR query from the moment the query is initiated until the callback is invoked.

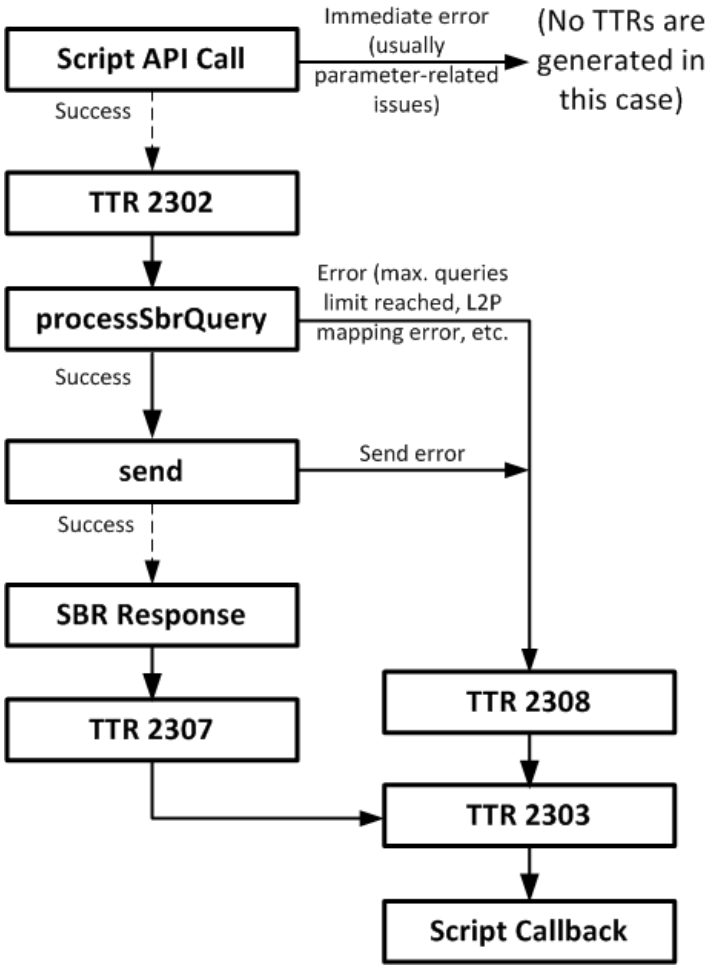


Figure 79: IDIH Event Trace of an U-SBR Query

The event 2302 is preceded by event 2300, 2301, or 2303, depending on whether the U-SBR query has been initiated from a Diameter request event handler, from a Diameter answer event handler or,

respectively, from the callback subroutine of a previous U-SBR query like for instance when a concurrent update is retried.

13. Best Practices

This chapter summarizes the basic rules to follow when writing a DCA App.

13.1 The Main Part of the Perl Script

The main part of the Perl script is executed only once when the Perl script is compiled. It is therefore the right place to perform sanity checks and post-process the DCA App configuration data. For instance the code below:

```
if(! defined($dca::appConfig{XYZ_Table}) ||
                                ("ARRAY" ne ref($dca::appConfig{XYZ_Table})))
{
    # dca::application::logInfo("Missing XYZ configuration table");
    die "Missing XYZ configuration table";
}
```

checks that the XYZ_Table exists and that it is a valid reference to an array. Note that single-row configuration tables are references to hash tables (i.e., use HASH instead of ARRAY).

If the validation fails, then Alarm 33309 Script Compilation Error is raised when the Perl script is compiled.

The DCA App configuration data for multi-row (i.e., not single-row) tables is stored in a Perl variable of type array; in our example `$dca::appConfig{XYZ_Table}` is a reference to such an array. Arrays are however very inefficient data structures to perform real-time lookups because they require looping through them each time. For this reason, depending on which fields a DCA App uses to lookup the configuration data in real time, the DCA App would typically post-process the configuration data (in the Perl script main part) by going record-by-record through it and copying each record into a separate hash table where the tags are the values of the lookup field. If necessary multiple such hash table may be prepared for real-time use, or other data structured (e.g., trees) may be created from the initial configuration data.

Post-processing increases the memory usage (with the new data structures that are created) and increases the Perl script compilation time (because the execution of the main part is always triggered by a successful Perl script compilation and therefore it may be regarded as a side effect to compiling the Perl script). However, the real-time performance gain is likely to be significant – probably orders of magnitude depending on the DCA App configuration data size.

13.2 Perl Global Variables

Do not use Perl global variables (defined in the main part of the script) to pass data between the various event handlers and callbacks in a DCA App. This is because the event handlers and callbacks are executed on demand by a pool of Perl interpreter threads, which means:

1. Event handlers and callbacks that process the same Diameter message may be executed by different Perl interpreters;
2. The same Perl interpreter executes event handlers and callbacks that process many different Diameter messages.

Use instead the transaction context variables defined in Section 11.2.2.

13.3 Returning Control from a Perl Subroutine

The control flow paths of an event handler or callback end in one of the following ways:

1. A `dca::action` API call.
2. An U-SBR query.

   The example below provides an example of how an event handler or callback ends with an U-SBR query:

```
my $result = dca::sbr::sbrInstance("sbr")->createOrRead ($key_type,
                    dca::sbr::KeyDataType::INT64, $imsi,
                    dca::sbr::StateDataType::STRING, $sbr_state,
                    "create_or_read_nonpref_cb");
    # check for "synchronous" error
    if(!defined($result)) {
        # could not create the sbr request - depending on the business
logic,
        # log an error message and fall back, or raise runtime error
alarm:
        die "could not create the SBR request";
    } else {
        # the processing continues asynchronously
        # in the "create_or_read_nonpref_cb"
        exit;
    }
```

   Note that no dca::action API call follows an U-SBR query because the U-SBR query is not going to be sent at all in this case: a dca::action API call ends the processing of the Diameter message, no query and no callback are executed any longer for the respective Diameter message.

3. A `die` statement.

   Alarm 33304 DCA Runtime Errors is raised and the Diameter message is routed as indicated by the configuration (see Section 9.4). The text specified as a parameter to the `die` statement is included in the alarm's additional information.

4. The control flow reaches (i) a `return` statement or (ii) the closing bracket that ends the Perl subroutine scope. In this case, the action taken by the DCA framework depends on the value returned from the respective Perl subroutine; in the latter case the return value is the result of the last executed evaluation before the ending bracket is reached:

   a. If the return value is grater or equal to zero, the Diameter message is forwarded.

   b. If the return value is less than zero, Alarm 33304 DCA Runtime Errors is raised and the Diameter message is routed as indicated by the configuration (see Section 9.4).

13.4 Callbacks

The first thing to do in a callback is to check the result code. Section 11.7.1.2 describes the error codes that apply to all U-SBR API functions. For most DCA Apps is enough to check whether the U-SBR query was successful (`dca::sbr::ResultCode::Ok` was returned) and continue processing and respectively to abort execution (by invoking `die`) in all other cases.

Note however that individual U-SBR queries have specific error codes – these are highlighted with a red background throughout Section 11.7.2. Some of these specific error codes do not necessary involve that the processing is aborted; for instance: `dca::sbr::ResultCode::ErrRecExists` in case of a `createOrRead` query indicates that the query performed "read" rather than a "create", `dca::sbr::ResultCode::ErrRecObsoleted` in case of a `concurrentUpdate` indicates that the query is repeated.

13.5 Sending multiple U-SBR Queries

There might be situations when processing a Diameter message requires a sequence of U-SBR queries (e.g., a read and, based on the state data returned, also an update). It is not possible to send concurrent U-SBR queries – i.e., more than one U-SBR query from the same Perl subroutine (event handler or callback); if multiple U-SBR queries are initiated from a Perl subroutine, only the last one is sent.

Multiple U-SBR queries (during the processing of the same Diameter message) are sent sequentially, i.e., the event handler initiates the first U-SBR query, the callback of the first U-SBR query initiates the second U-SBR query, the callback of the second U-SBR query initiates the third U-SBR query, etc., the callback of the last U-SBR query routes the Diameter message (e.g., by using a `dca::action` API call).

13.6 Accessing Lower Layer Data from Mediation

The DCA EDL API (see Section 11.1) as well as part of the Routing API (see Section 11.4 `setART` and `setPRT`) is similar to the API offered by the Mediation feature. However, there are a couple of API functions, supported by Mediation not available in DCA. This is because Mediation is invoked in the context of Diameter Routing Layer (DRL), whereas the DCA operates at the application layer (i.e., one layer above DRL). These API calls are:

```
$param->ingressConnectionName( );
    Return the name of the ingress connection if available.

$param->ingressPeerName( );
    Return the name of the ingress peer if available.
```

(where `$param` is retrieved as described in Section 11.1.1) and may be needed if the DCA business logic depends on the connections or peers the Diameter messages are received from. This information may be made available to DCA through Internal Variables, using the following procedure: (i) a Mediation script uses the above API calls to retrieve the ingress connection/peer and (ii) writes it into an Internal Variable, from where (iii) DCA can read when the event handler is invoked.

Note that the following API functions:

```
$param->egressConnectionName( );
    Return the name of the egress connection if available.

$param->egressPeerName( );
    Return the name of the egress peer if available.
```

are not usable in DCA because the routing decision is made after the DCA event handlers complete the execution (and the message is returned back to DRL).

13.7 Performance Tuning

DCA Applications may require performance tuning depending upon the complexity of business logic and need for MPS.

Users can determine the need to do performance tuning based on:

- The number of DcaRequestTaskThr and DcaAnswerTaskThr threads needed on a DA-MP, which depends on whether the DCA application performs computation on:
  - Diameter request leg of the diameter transaction.
  - Diameter answer leg of the diameter transaction.
  - Both diameter request and diameter answer leg of the diameter transaction.
- The number of DcaSbrEventTaskThr threads, which must be directly proportional with number of SBR transactions per diameter transactions performed by the DCA application.
- The number of SBR SGs in a U-SBR DB (resource domain) is based on how many SBR transactions per second are planned to be processed by the respective U-SBR DB, which depends on the DCA application requirements.

Note: Please contact the customer support team for tuning the performance parameters.