

CUDB Dynamic Library for Distribution

PROGRAMMER'S GUIDE

Copyright

© Ericsson AB 2016. All rights reserved. No part of this document may be reproduced in any form without the written permission of the copyright owner.

Disclaimer

The contents of this document are subject to revision without notice due to continued progress in methodology, design and manufacturing. Ericsson shall have no liability for any error or damage of any kind resulting from the use of this document.

Trademark List

All trademarks mentioned herein are the property of their respective owners. These are shown in the document Trademark Information.



Contents

1	Introduction	1
1.1	Scope	1
1.2	Revision Information	1
1.3	Target Groups	1
1.4	Prerequisites	1
1.5	Typographic Conventions	1
2	Overview	3
2.1	Description	3
2.1.1	Configuring the Dynamic Library	3
2.1.2	Compiling the Dynamic Library	9
2.1.3	Installing the Dynamic Library	10
3	Examples	11
3.1	Basic Example Library	11
3.2	Testing the Dynamic Library	12
3.3	Header File Template	14
	Glossary	17
	Reference List	19





1 Introduction

This document describes how to develop customized subscriber data distribution policies in the Ericsson Centralized User Database (CUDB) by an external library.

1.1 Scope

During provisioning time, CUDB is using a default policy to determine the Data Store Unit Group (DSG) for each Distribution Entry (DE) provisioned by the system. The goal of this procedure is to keep a balanced memory occupation level in all the DSGs of the system.

However, to suit specific customer needs, CUDB supports the configuration and execution of custom data distribution policies that override the default policy. This document describes how to configure a custom data distribution policy. Refer to [CUDB LDAP Data Access](#), Reference [1] for more information on data distribution.

1.2 Revision Information

Rev. A	This document is based on 4/19817-HDA 104 03/9, and contains editorial changes only.
---------------	--

1.3 Target Groups

This document is intended for developers who implement custom distribution algorithms for provisioning users in accordance with customer requests.

1.4 Prerequisites

Users of this document must have knowledge and experience in C++ programming.

1.5 Typographic Conventions

Typographic conventions can be found in the following document:

— [Typographic Conventions](#)





2 Overview

The default CUDB subscriber data distribution policy is based on detecting the memory occupation of the DSGs: the system aims to select DSGs with lower memory occupation levels to use as storage when distributing data. Optionally, the multiple geographical areas feature can also be used to configure data distribution: refer to [CUDB Multiple Geographical Areas](#), Reference [2] for more information.

Custom data distribution policies can be configured by using Dynamically Loaded (DL) Linux libraries, implemented with C++ and compiled for 64-bit systems by using a generic Linux C++ compiler, such as gcc. The procedure is described in the next sections in more detail.

2.1 Description

This section provides detailed description for the CUDB Dynamic Library feature, and provides examples for the compilation process and library installation.

A CUDB dynamic library consists of a header file defining the types of the distribution functions running in the system, along with the implementation of the functions.

2.1.1 Configuring the Dynamic Library

The parameters of the external library are provided by using generic structures. The structures are defined in the `cudb_dist_ext_func.h` header file which can be included in the library compilation.

The header file contains two important pieces of information:

- The definition of the used structures.
- The declaration of the required functions.

Both of these must be copied from the header file, as the file is not distributed. The structures and functions are described below in more detail.

2.1.1.1 Configuring Structure Definitions

Example 1 provides an example of the structure definitions declared in `cudb_dist_ext_func.h`. See Section 2.1.1.2 on page 4 for an explanation of the functions included in this example.



```

#include <vector>

typedef struct
{
    long cv_len;
    char *cv_val;
} charVal;

typedef struct
{
    charVal cName;
    vector<charVal> cValue;
} argStruct;

typedef vector<argStruct*> arg_attribs;
// e.g. {"mScId", "1234"} , {"ZoneId", "1"} , ...
typedef vector<argStruct*> arg_dyndata;
// e.g. {"usage", "1 50"}
typedef vector<argStruct*> arg_confdata;
// e.g. {"zone", "1 1 2"} , {"default_zone", "1"} ,
// {"limit", "99"} , ...

```

Example 1 Structure Definitions

2.1.1.2 Declaring External Functions

Example 2 provides an example of the external functions that can be declared in the header file:

```

extern "C"{
    int external_distalloc
    (arg_attribs arg1,
    arg_dyndata arg2,
    arg_confdata arg3,
    char* errorMsg);
}

#define EXTFUNC_NAME "external_distalloc"
#define ERROR_NO_DSG_DEFAULT_DISTRIBUTION -1
#define ERROR_NO_DSG_NO_DISTRIBUTION -9

```

Example 2 Declaration of External Functions

The functions used in Example 2 are as follows:

- arg1 contains details about the DE to be provisioned, for example, the attributes of the mscId or assocId entries.
- arg2 contains dynamic information, for example, the memory occupation of each DSG.



- `arg3` contains the configuration data, for example, the zone directives or the occupancy limit.
- `errorMsg` is a pointer where any error message is returned. The required memory for this string is already allocated in CUDB. The DL can use up to 255 characters including End of String (EOS) characters.

This function returns the selected DSG, or a value lower than 0 in case of error. Two possible error codes are available:

- `ERROR_NO_DSG_DEFAULT_DISTRIBUTION`: This message indicates that the external algorithm can not be executed, therefore CUDB tries to apply the default algorithm.
- `ERROR_NO_DSG_NO_DISTRIBUTION`: This message indicates than an error must be returned in CUDB and the default algorithm must not be applied. The last parameter contains the error message.

In case of errors, the relevant events are logged according to [CUDB Node Logging Events, Reference \[3\]](#).

The external functions can receive the following fixed keywords with their parameters:

- `arg1`: Keywords of this function may provide all the attributes of the DEs (such as `mscId`, `assocId`, `ZoneId`, `objectClass`, and so on). Therefore, its exact contents depend on the object provided for the keyword. Refer to [CUDB LDAP Interwork Description, Reference \[4\]](#) for more information.

Example 3 shows how to implement an `arg1` vector containing `ZoneId 1`:

```
// param 1 : dn attributes
arg_attribs arg1;

argStruct* a1 = new argStruct();
charVal cvname;
cvname.cv_len = 6;
cvname.cv_val = new char[7];
strcpy(cvname.cv_val, "ZoneId\0");
a1->cName = cvname;

charVal cvvalue;
cvvalue.cv_len = 1;
cvvalue.cv_val = new char[2];
strcpy(cvvalue.cv_val, "1\0");
(a1->cValue).push_back(cvvalue);

arg1.push_back(a1);
```

Example 3 Implementing `arg1` containing `ZoneId 1`



— **arg2**: Keywords of this function provide dynamic data. The following types of dynamic data can be defined:

- The memory usage of a specific DSG, configured as follows:

usage <DSG number> <occupancy level>

For example, **usage 1 50** assigns an occupancy level of 50% to DSG 1.

- The list of DSGs not used for provisioning. It can arrive without elements. The function is configured as follows:

non_prov_list <DSG number 1> <DSG number 2>...

For example, **non_prov_list 1 2** disables DSG 1 and DSG 2 for provisioning.

Example 4 below shows how to implement the above non-provisioning list:



```
// param 2 : dsgs list
arg_dyndata arg2;

argStruct* a2 = new argStruct();

charVal cvname2;
cvname2.cv_len = 5;
cvname2.cv_val = new char[6];
strcpy(cvname2.cv_val, "usage\0");
a2->cName = cvname2;

charVal cvDsgId;
cvDsgId.cv_len = 1;
cvDsgId.cv_val = new char[2];
strcpy(cvDsgId.cv_val, "1\0");
(a2->cValue).push_back(cvDsgId);

charVal cvDsgUs;
cvDsgUs.cv_len = 2;
cvDsgUs.cv_val = new char[4];
strcpy(cvDsgUs.cv_val, "50\0");
(a2->cValue).push_back(cvDsgUs);

arg2.push_back(a2);

a2 = new argStruct();

charVal cvname2Prov;
cvname2Prov.cv_len = 13;
cvname2Prov.cv_val = new char[14];
strcpy(cvname2Prov.cv_val, "non_prov_list\0");
a2->cName = cvname2Prov;
charVal cvDsgIdProv;
cvDsgIdProv.cv_len = 1;
cvDsgIdProv.cv_val = new char[2];
strcpy(cvDsgIdProv.cv_val, "1\0");
(a2->cValue).push_back(cvDsgIdProv);

arg2.push_back(a2);
```

Example 4 Implementing Non-Provisioning List

— **arg3:** Keywords of this function provide configuration data. The configurable parameters are as follows:

- **zone:** Zone definition provided by configuration management. The parameter format is the following:

zone <zone ID> <DSG numbers in the zone>



For example, `zone 1 1 2` indicates that Zone 1 consists of DSG 1 and DSG 2.

- `default_zone`: The default zone provided by the configuration management. The parameter format is the following:

`default_zone <zone ID>`

For example, `default_zone 1` sets Zone 1 as the default value.

- `limit`: The memory occupancy level (defined in percentage) where a DSG is considered full. The parameter format is the following:

`limit <occupancy level>`

For example, if set as `limit 99`, the affected DSG is considered full if memory occupancy reaches 99%.

Example 5 below shows how to implement an `arg3` vector containing the above example value for zone:



```
// param 3 : config data
arg_confdata arg3;

argStruct* a3 = new argStruct();
charVal cvname3;
cvname3.cv_len = 4;
cvname3.cv_val = new char[5];
strcpy(cvname3.cv_val, "zone\0");
a3->cName = cvname3;

charVal cvZone;
cvZone.cv_len = 1;
cvZone.cv_val = new char[2];
strcpy(cvZone.cv_val, "1\0");
(a3->cValue).push_back(cvZone);

charVal cvDsg1;
cvDsg1.cv_len = 1;
cvDsg1.cv_val = new char[2];
strcpy(cvDsg1.cv_val, "1\0");
(a3->cValue).push_back(cvDsg1);

charVal cvDsg2;
cvDsg2.cv_len = 1;
cvDsg2.cv_val = new char[2];
strcpy(cvDsg2.cv_val, "2\0");
(a3->cValue).push_back(cvDsg2);

arg3.push_back(a3);
```

Example 5 Implementing arg3 Vector for zone

2.1.2 Compiling the Dynamic Library

The DL must be compiled in 64 bits using a standard C++ compiler, such as Linux gcc.

Example 6 provides generic compiler and linker commands for a DL called testDL.

```
Invoking: GCC C++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -fPIC -MMD -MP
    -MF"testDL.d" -MT"testDL.d" -o"testDL.o"
    "./testDL.cpp"
Invoking: GCC C++ Linker
g++ -shared -o"libcudbDistrDL.so" ./testDL.o
```

Example 6 Compiler and Linker Commands



2.1.3 Installing the Dynamic Library

For the detailed steps of installing or uninstalling a DL, refer to CUDB System Administrator Guide, Reference [5].

Note: After installing the DL, the new distribution policy is applied. Uninstall the custom DL to revert to the default distribution policy.



3 Examples

This section contains additional examples to get a better understanding on how dynamic libraries are created, compiled and deployed. The section contains the following examples:

- A basic example library (see Section 3.1 on page 11).
- An application to test external libraries (see Section 3.2 on page 12).
- A template for the `cudb_dist_ext_func.h` header file (see Section 3.3 on page 14).

3.1 Basic Example Library

The new shared library must be coded using C++. Once implemented, the DL must be installed as described in Section 2.1.3 on page 9.

Example 7 shows a new algorithm that returns the DSG with the lowest occupancy, excluding DSG number 0. The basic CPP file looks as follows:

```
#include "cudb_dist_ext_func.h"

#include <vector>
#include <string.h>
#include <stdlib.h>
extern "C" {

int external_distalloc(arg_attribs entryAttrs,
                      arg_dyndata dynamicData,
                      arg_confdata configData,
                      char* retErrorCode){

    int usage_ret = 0;
    int i_ret = -1;
    vector<argStruct*>::iterator dynamicData_it;

    for (dynamicData_it = dynamicData.begin(); dynamicData_it != dynamicData.end(); dynamicData_it++){
        argStruct * dynData_Ptr;
        dynData_Ptr = *dynamicData_it;
        vector<charVal*>::iterator dynData_cValue_it = (dynData_Ptr->cValue).begin();

        if ( strcmp(dynData_Ptr->cName.cv_val, "usage")==0 ){
            int idDSG = atoi((*dynData_cValue_it).cv_val);
            dynData_cValue_it++;
            if(idDSG!=0){
                if(i_ret == -1){
                    i_ret = idDSG;
                    usage_ret = atoi((*dynData_cValue_it).cv_val);
                }else if(atoi((*dynData_cValue_it).cv_val) < usage_ret){
                    i_ret = idDSG;
                }
            } //idDSG!=0
        }
    } //for
    return (i_ret);
}

} // extern "C"
```

Example 7 New Algorithm Returning DSG with Lowest Occupancy

The above code can be compiled with the following commands:



— Invoking the GCC C++ Compiler:

```
g++ -O0 -g3 -Wall -c -fmessage-length=0 -fPIC -MMD -MP  
-MF"testDL.d" -MT"testDL.d" -o"testDL.o" "testDL.cpp"
```

— Invoking the GCC C++ Linker:

```
g++ -shared -o"libcudbDistr.so" ./testDL.o
```

3.2 Testing the Dynamic Library

Example 8 provides a program that can be used to test external libraries:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <dlfcn.h>  
#include <string.h>  
  
#include "cudb_dist_ext_func.h"  
  
int extFunction(char*);  
  
int main()  
{  
    char retErrorCode[255];  
    memset((char*)retErrorCode, '\0', 255);  
    int ret = extFunction(retErrorCode);  
  
    printf("\n Ret Value = %d\n", ret);  
    printf(" Ret Error Msg = %s\n\n", retErrorCode);  
  
    return 0;  
}  
  
int extFunction(char* retErrorCode)  
{  
    // handling DL  
    void *handle;  
    char *dl_error;  
    int retValue = -1;  
  
    handle = dlopen("./libcudbDistr.so", RTLD_LAZY);  
    if (handle)  
    {  
        // External funct exists. Trying to apply ext algorithm  
        int (*external_distalloc)(arg_attribs,  
                                arg_dyndata,  
                                arg_confdata,  
                                char*);  
        *(void**) (& external_distalloc) =  
            dlsym(handle, "external_distalloc");  
        if ((dl_error = dlerror()) != NULL) {  
            // Error when accessing distribution external library  
            retValue = -2;  
        }else{  
            // Prepare params to external function  
  
            // Param 1 : dn attributes  
            arg_attribs entryAttrs;  
  
            argStruct* zoneIdAttr = new argStruct();  
            charVal cvname;  
            cvname.cv_len = 6;  
            cvname.cv_val = new char[7];  
            strcpy(cvname.cv_val, "ZoneId\0");  
            zoneIdAttr->cName = cvname;  
        }  
    }  
}
```




```

charVal cvvalue;
cvvalue.cv_len = 1;
cvvalue.cv_val = new char[2];
strcpy(cvvalue.cv_val, "1\0");
(zoneIdAttr->cValue).push_back(cvvalue);

entryAttrs.push_back(zoneIdAttr);

// Param 2 : DSG memory occupation
arg_dyndata dynamicData;

//DSG 1 usage : {"usage", {"1", "50"}}
argStruct* dsg1Usage = new argStruct();

charVal dsg1Usage_cvname;
dsg1Usage_cvname.cv_len = 5;
dsg1Usage_cvname.cv_val = new char[6];
strcpy(dsg1Usage_cvname.cv_val, "usage\0");
dsg1Usage->cName = dsg1Usage_cvname;

charVal cvDsg1Id;
cvDsg1Id.cv_len = 1;
cvDsg1Id.cv_val = new char[2];
strcpy(cvDsg1Id.cv_val, "1\0");
(dsg1Usage->cValue).push_back(cvDsg1Id);

charVal cvDsg1Usage;
cvDsg1Usage.cv_len = 2;
cvDsg1Usage.cv_val = new char[3];
strcpy(cvDsg1Usage.cv_val, "50\0");
(dsg1Usage->cValue).push_back(cvDsg1Usage);

dynamicData.push_back(dsg1Usage);

//DSG 2 usage {"usage", {"2", "40"}}
argStruct* dsg2Usage = new argStruct();

charVal dsg2Usage_cvname;
dsg2Usage_cvname.cv_len=5;
dsg2Usage_cvname.cv_val=new char[6];
strcpy(dsg2Usage_cvname.cv_val, "usage\0");
dsg2Usage->cName = dsg2Usage_cvname;

charVal cvDsg2Id;
cvDsg2Id.cv_len = 1;
cvDsg2Id.cv_val = new char[2];
strcpy(cvDsg2Id.cv_val, "2\0");
(dsg2Usage->cValue).push_back(cvDsg2Id);

charVal cvDsg2Us;
cvDsg2Us.cv_len = 2;
cvDsg2Us.cv_val = new char[3];
strcpy(cvDsg2Us.cv_val, "40\0");
(dsg2Usage->cValue).push_back(cvDsg2Us);

dynamicData.push_back(dsg2Usage);

// Param 3 : config data
arg_confdata configData;
// Zone configuration : { "zone", {"1", "1", "2"} }
argStruct* zoneConfig = new argStruct();
charVal cvname3;
cvname3.cv_len = 4;
cvname3.cv_val = new char[5];
strcpy(cvname3.cv_val, "zone\0");
zoneConfig->cName = cvname3;

charVal cvZone;
cvZone.cv_len = 1;
cvZone.cv_val = new char[2];
strcpy(cvZone.cv_val, "1\0");
(zoneConfig->cValue).push_back(cvZone);

```



```

charVal cvDsg1;
cvDsg1.cv_len = 1;
cvDsg1.cv_val = new char[2];
strcpy(cvDsg1.cv_val, "1\0");
(zoneConfig->cValue).push_back(cvDsg1);

charVal cvDsg2;
cvDsg2.cv_len = 1;
cvDsg2.cv_val = new char[2];
strcpy(cvDsg2.cv_val, "2\0");
(zoneConfig->cValue).push_back(cvDsg2);

configData.push_back(zoneConfig);

// external function
retValue = (*external_distalloc)
            (entryAttrs,
             dynamicData,
             configData,
             retErrorCode);

entryAttrs.clear();
dynamicData.clear();
configData.clear();
}
}
dlclose(handle);
}
return (retValue);
}

```

Example 8 Program Using External Library

The above code can be compiled with the following commands:

— Invoking the GCC C++ Compiler:

```
g++ -O0 -g3 -Wall -c -fmessage-length=0 -fPIC -MMD -MP
-MF"main.d" -MT"main.d" -o"main.o" "main.cpp"
```

— Invoking the GCC C++ Linker:

```
g++ -o"testProgram" ./main.o -ldl
```

3.3 Header File Template

Example 9 provides a template for the `cudb_dist_ext_func.h` header file.



```

#ifndef CUDB_DIST_EXT_FUNC_H_
#define CUDB_DIST_EXT_FUNC_H_

#include <vector>

using namespace std;

// Structures used by external function
typedef struct
{
    long cv_len;
    char *cv_val;
} charVal;

typedef struct
{
    charVal cName;
    vector<charVal> cValue;
} argStruct;

typedef vector<argStruct*> arg_attribs; // e.g. {"mScId", 1, "1234"} , {"ZoneId", 1, "1"} , ...
typedef vector<argStruct*> arg_dyndata; // e.g. {"usage", 2, "1 50"} , ...
typedef vector<argStruct*> arg_confdata; // e.g. {"zone", 4, "1 1 2 3"} , {"default_zone", 1, "1"} , {"limit", 1, ...}

// Prototype of external function
// Params:
//   arg_attribs : attributes of the distributed entry to be provisioned
//   arg_dyndata : dynamic data ( memory usage and list of dsg not available for provisioning)
//   arg_confdata : static configuration data ( zones, default zone and occupancy limit )
//   char* [out] : returned error message
// Return values:
//   -1 : external algorithm execution failed; default algorithm must be applied
//   -9 : no DSG must be selected, an error must be returned
//   otherwise, return selected DSG
int (*cudb_external_distalloc) (arg_attribs entryAttris, arg_dyndata dynamicData, arg_confdata configData, char* error)

#define EXTFUNC_NAME "external_distalloc"
#define ERROR_NO_DSG_DEFAULT_DISTRIBUTION -1
#define ERROR_NO_DSG_NO_DISTRIBUTION -9

#endif /*CUDB_DIST_EXT_FUNC_H_*/

```

Example 9 Header File Template





Glossary

For the terms, definitions, acronyms and abbreviations used in this document, refer to [CUDB Glossary of Terms and Acronyms](#), Reference [6].





Reference List

CUDB Documents

- [1] CUDB LDAP Data Access
- [2] CUDB Multiple Geographical Areas
- [3] CUDB Node Logging Events
- [4] CUDB LDAP Interwork Description
- [5] CUDB System Administrator Guide
- [6] CUDB Glossary of Terms and Acronyms