# TRANSPORT API 8.0
# C EDITION

## DACS LOCK LIBRARY
### REFERENCE MANUAL

THOMSON REUTERS

# Contents

# Chapter 1    Introduction

## 1.1        Product Description

The goal of permissioning is to control access to data by users. Using an entitlement system such as DACS (Data Access Control System), permission profiles can be defined identifying what each user is allowed to access. DACS is the entitlement system for the Thomson Reuters Enterprise Platform (TREP). DACS permission checks take place in the Market Data Client application. In order to perform a permission check for an item, DACS must have 'requirements' information for the item and a profile for the user (a.k.a. content based permissioning).

DACS requirements information is organized as numeric expressions. Each subservice of a service, e.g. all the data from an exchange, is assigned a (series of) numeric **entitlement codes (PEs)**. The numeric entitlement codes are transported with items as they move from source servers to Market Data Client applications on the TREP. In order to accommodate the most general case in which information from multiple sources is combined to form new items (such as in compound servers), the **requirements** information for an item is a Boolean expression containing these entitlement codes. For an item obtained directly from a source, this expression is usually a single term. When a compound server combines two items to form a new (compound) item, it must also combine the requirements information to form a new (compound) requirement.

The name of the subservice associated with an entitlement code is maintained in tables within the DACS database and operational permission checking subsystem. The table that relates the entitlement codes for a service to the subservice names for that service is called the **map** for the service.

Requirements are transported on TREP in protocol messages called **locks**. The DACS Lock API provides functions to manipulate locks in a manner such that the source application need not know any of the details of the encoding scheme or message structure. For a source server to be DACS compliant, based on content,  it must publish locks for the items it publishes; i.e., the source server application must produce lock events. Any item published without a lock or with a null lock is available to everybody permissioned for that service, even those without subservice permissions.

If a source server introduces (new) data to TREP that originates outside the network on which the application is running, then the application developer is also responsible for providing the map information for the service.

In addition to source servers that publish new data directly from a vendor, there are also compound servers that gather market information from other sources, manipulate that information, and then re-publish it on TREP. These servers must read locks from the Transport API C Edition to combine them before publishing compound items. Again, the DACS Lock API provides functions to assist with this process. Compound sources must also use a special form of user ID when connecting to the TREP network.

**Note:**        Subject-based sources do not require locks.

## 1.2        Audience

This guide is intended for software programmers who wish to incorporate the DACS Locks into the development of their source applications.

## 1.3 Organization of Manual

The material presented in this guide is divided into the following sections:

| CHAPTER | TITLE | CONTENTS |
| --- | --- | --- |
| Chapter 1 | Introduction | Document description. |
| Chapter 2 | DACS-Compliant Source Server Applications | For subservice-level permissioning, a DACS-compliant source server must satisfy a series of requirements. |
| Chapter 3 | DACS Lock API | The data flow of a DACS Lock and its operation are depicted |
| Chapter 4 | DACS_CsLock() | Description and structural definitions for `DACS_CsLock()`. |
| Chapter 5 | DACS_GetLock() | Description and structural definitions for `DACS_GetLock()`. |
| Chapter 6 | DACS_CmpLock() | Description and structural definitions for `DACS_CmpLock()`. |
| Chapter 7 | DACS_perror() | Description and structural definitions for `DACS_perror()`. |

**Table 1: Manual Overview**

## 1.4 References

[1] *Transport API C Edition Developers Guide*

[2] *DACS Lock API Reference Manual*

## 1.5 Conventions

### 1.5.1 Typographic

- Functions, arguments, and data structures are shown in orange, Lucida Console font.
- Parameters, filenames, tools, utilities, and directories are shown in **Bold font**.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.
- Longer code examples (one or more lines of code) are shown in Lucida Console font against an orange background. For example:

```
/* calculate the length of the new lock */
int lock = dacsInterface.calculateLength(serviceId, operation,
        productEntityList, productEntityListLength, error);
```

## 1.5.2 Programming

Transport API C Edition Standard conventions were followed.

# 1.6 Glossary

| TERM | DESCRIPTION |
|---|---|
| API | Application Programming Interface |
| Application | A program that accesses data from and/or publishes data to the system. |
| Compound Item | A data item prepared from data items retrieved from the system. |
| Concrete service | A set of real-time data items published by a source server. Each concrete service is identified on DACS by a unique name (known as a network). |
| Data Access Control System (DACS) | A tool that allows customers to automatically control who is permitted to use which sets of data in their TREP. |
| Entitlement Code | If a vendor service is permissioned down to the subservice level, an entitlement code must be provided with each item. Based on mapping tables provided by the vendor, DACS uses this code to determine the information provider and/or vendor product associated with an item. |
| Exchange | A commercial establishment at which or through which trading of financial instruments takes place. Exchanges are information providers. |
| Exchange Map Logic | The logic used to construct requirements when an item is supplied by more than one exchange. If OR logic is used, the user only needs permission to access one of the exchanges that supplies the item. If AND logic is used, the user must have permission to access all of the exchanges that supply the item. |
| IDN | Integrated Data Network |
| Map Program | An application associated with a particular vendor service which requests permissioning data from the vendor host and then uses that data to construct various mapping tables required by DACS. |
| Mapping Tables | These tables are used by DACS to derive the requirement for an item. They map entitlement codes to vendor products and, if applicable, to information providers (exchanges and specialist services). |
| Network Service | See concrete service. |
| PE | Permissionable Entity. A number used to designate the permissioning basis of a data item on IDN. (Same as entitlement code.) |
| Permissioning | The control of access to and publication of data items by users. |
| Product | A subset of the data items delivered by an information vendor for which there is a single charge (based on vendor criteria). |
| Product Map Logic | The logic used to construct requirements when an item is supplied by more than one product. If OR logic is used, the user only needs permission to access one of the products that supplies the item. If AND logic is used, the user must have permission to access all of the products that supply the item. |
| Profile | Information, including a list of subservices, that is used during permission checking. There is a profile associated with each user. |
| RFA | Robust Foundation API |

| TERM | DESCRIPTION |
|---|---|
| Service | This term is used in two ways by DACS in a market data system environment. See concrete service and vendor service. |
| Service ID | A unique numeric ID assigned to each network service. On a TREP network, all valid service IDs for a particular system are listed in the global configuration file **rmds.cnf**. |
| Source | An application or server capable of supplying or transmitting information. |
| Source Server | An application program which provides a concrete service. More than one source server can provide the same concrete service. |
| Specialist Data Service | A set of data items provided by a third party (i.e. not from an exchange and not from the vendor delivering the data items to the site). |
| Subservice | A named set of items delivered by a vendor which are authorized as a group; e.g. all instruments traded on NYSE or all items that make up the product Securities 2000. |
| Subservice Type | There are three types of subservices:<br>• Products<br>• Exchanges<br>• Specialist Service |
| TREP | Thomson Reuters Enterprise Platform. |
| User | A person with a unique, system-wide name. |
| Vendor Service | A vendor may offer more than one type of data delivery service to a customer. For example, Reuters provides the IDN service. Each vendor service is identified on DACS by a unique name.<br><br>Each vendor service is associated with one or more concrete services. For example the IDN service may be published on the network by any combination of these concrete services: IDN Selectserver and Reuters Data Feed. |

**Table 2: Glossary and Acronyms**

# Chapter 2     DACS-Compliant Source Server Applications

## 2.1          Overview

For permissioning at the subservice level to work, it is necessary to have information about the permissioning requirements for an item at locations other than just the source. In order for DACS to permission a source service below the service level, the source must (1) contribute to the process by creating locks containing permissioning information and (2) provide data mapping the entitlement codes in the locks to subservice names.

For subservice level permissioning, a DACS-compliant source server must satisfy the series of requirements included in this chapter.

## 2.2          Establish Subservice Names

If a source server provides a service that is subdivided into subservices, the first requirement is that each subservice have a symbolic name. For IDN such names are the names of Reuters products, exchanges, and specialist data services. These subservice names represent subsets of the service being provided by the vendor. An item may be in more than one subservice, and an item might not be in any named subservice. At the time a source publishes an item on the TREP, the source must be able to associate with the item the identities of any subservices to which the item belongs. For example, an IDN source might identify that an item belongs to the subset of items from the New York Stock Exchange and is part of the Equities 2000 product.

The reason for the symbolic names is that these names are used by the system administrator to grant or deny access by users to items in the various subservices. The system administrator performing permissioning will not deal with arbitrary numeric encodings such as the Reuters PE (Permissionable Entity).

## 2.3          Define Entitlement Codes

The second requirement is that a number, called the entitlement code, must be associated with each subservice name. (A subservice can have one or more associated entitlement codes.) At the time an item is published, the source designates the subservice(s) to which the item belongs as a Boolean expression in entitlement codes. The reason for the entitlement codes is that it may be necessary to combine permissioning information from more than one source in order to permission compound items made up of constituents from more than one source.

The PE used for permissioning on IDN (FID 1, PROD_PERM) is an example of an entitlement code.

The list of subservice names and associated entitlement codes is called the *map* for the source.

## 2.4          Create and Write Locks

At the time a source server opens a data stream, it must write a lock containing the entitlement code expression associated with the item. The source server must use an existing Reuters API to publish permissionable data on the TREP.

One of the capabilities of such an API is to create the lock. The arguments to the API function include a list of entitlement codes and the Boolean operator (AND or OR) that indicates how they are to be logically combined.

Once the lock has been created, it needs to be posted to the TREP.

A source application can post a revised lock at any time.

## 2.5　　　Publish the Map

The fourth requirement is that the source must publish (or use *map_generic* to load map through the use of a file), as data items, the map of its entitlement codes and subservice names. As an example, the map for the IDN service is published as a series of RICs referred to as the Reuters Product Definition Pages.[1]

Within the map data there must be a readily available data item that contains a date-time stamp. The value of this date-time stamp must be the date and time when the last change was made to the map. The objective is to permit the application that reads the map to read a single item and determine if the rest of the data has changed and thus needs to be reread.

The map items (records or pages) must be available to a user and application that have no subservice permissions. This is necessary so that the map can be retrieved at a new site that does not yet have permissions distributed.

| Note: | For the IDN services, template files containing preliminary mapping information are provided with the DACS software so that subservice permissioning can be set up before the latest map is actually retrieved from the source. If such a template file is not provided with a third-party source application, then it is important that no subservice permissioning be required in order to retrieve the map from the source. |
|---|---|

## 2.6　　　Provide Mechanism to Read Map and Supply it to DACS Station

There are two ways to load the map information:

- Use of the Generic map collect program (*map_generic*).
- Download of the map.

In order for the DACS administrator to be able to assign permissions to the subservices for a service and for the DACS operational subsystem to perform permission checks, the DACS database must contain the map information for the source. As indicated previously, it is expected that the source will publish this map. Further it is expected that the source application developer will provide an application (known as the map collection program) to:

- Request the map items from the source.
- Determine if there were any changes since the previous map was received.
- Convert any revised information into a file that can be loaded into the DACS database.

The source application developer may also want to provide a map monitor program which can be scheduled to run periodically to retrieve the latest map and see if any changes have occurred since the last map collection (by checking the date-time stamp). Based on the status reported by the monitor program, the administrator knows when the map collection program needs to be run.

DACS does not care about the format of the map published by the source as long as the map collection program for that source produces an appropriately formatted file as described below. (The DACS software package already includes a map collection program for the IDN service.)

For further details on map collect, refer to the *DACS Lock API Reference Manual* specific to the version of DACS that you run.

---

[1]　Refer to the *Reuters Product Definition Pages User Guide* to see how Reuters communicates the permissioning information for its products.

# Chapter 3    DACS Lock API

## 3.1        DACS Lock Operation

The DACS Lock contains requirements for an item that a vendor source deems necessary. On the Market Data Client LAN, users are entitled to specific capabilities. Therefore, when a Lock, which contains requirements, is tested against the capabilities of the user, enough information is available to permission the following:

| PERMISSIONING OPERATION |
| --- |
| USER -> APPLICATION |
| USER -> SERVICE |
| USER -> SUB-SERVICE (entitlement  codes) |

**Table 3: DACS Permissioning Capabilities**

DACS Locks are critical to the operation of the DACS system for content based sources. Subject based sources do not require locks. The DACS Lock contains the requirements for the requested item. The data flow of a DACS Lock and its operation are depicted in the remaining sections of this chapter.

## 3.2        Forming a DACS Lock

The Source Server is responsible for creating a DACS Lock. What information is encoded within the DACS Lock is vendor specific. Two examples are presented to clarify the formation of DACS Locks with respect to a Source Server. Figure 1, "Forming a DACS Lock for RDF," demonstrates the input requirements, the DACS Lock API to be called, and the transport mechanism of the DACS Lock from the Source Server to the Market Data Client application.
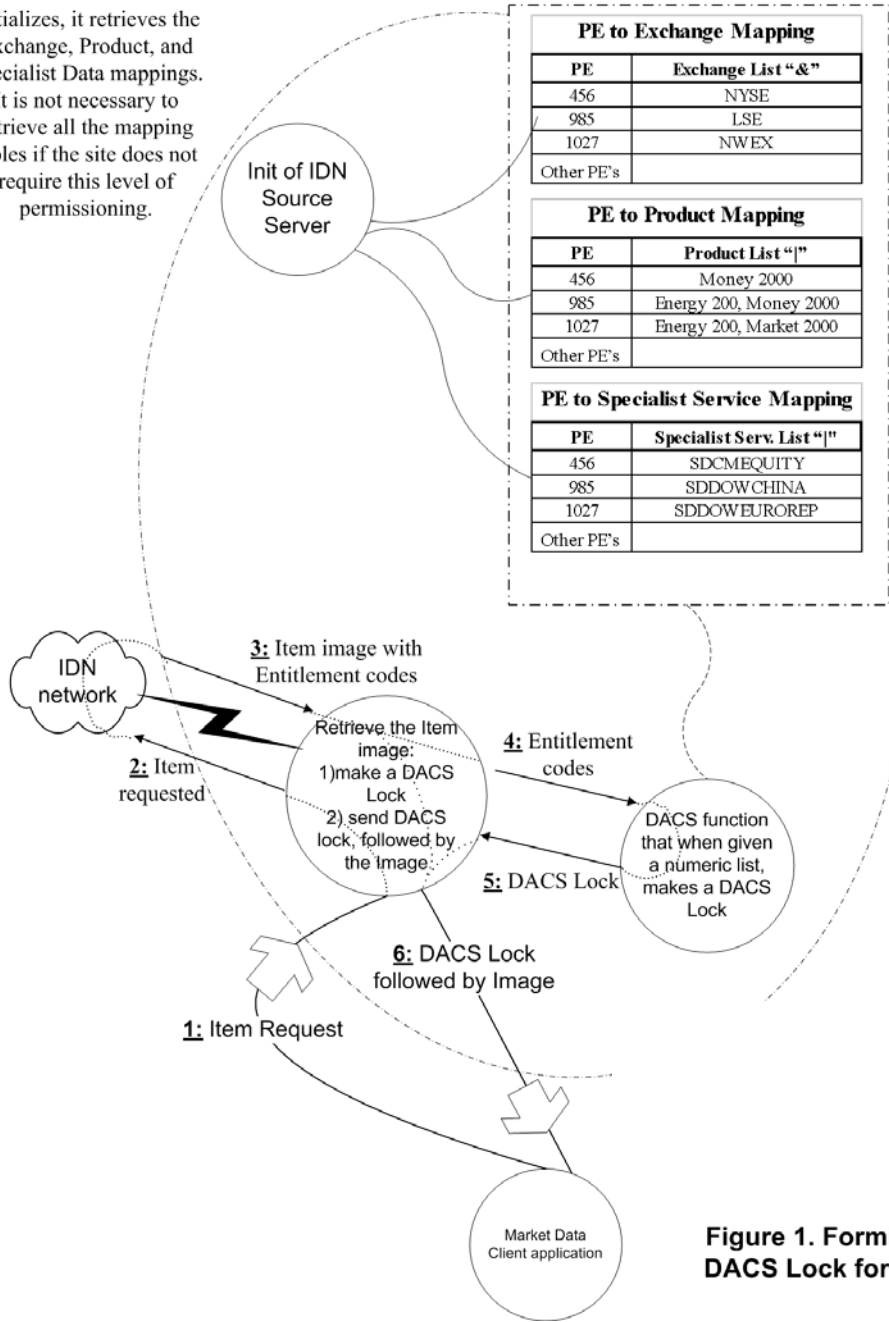


Figure 1. Forming a DACS Lock for RDF

**Figure 1: Forming a DACS Lock for RDF**

Figure 1 presents the following information:

- The DACS station via use of its map collect utility is responsible for retrieving the Exchange, Product, and Subservice mappings. For this mapping to be dynamic, it is necessary for the mapping tables to be retrieved from the datafeed line by the IDN Server.
- It is up to the IDN Server if it retrieves all the mapping tables based on the customer site requirements and on the capabilities of the Server's datafeed line.
- The DACS Lock API function that makes the DACS Lock is supplied the list of entitlement codes that corresponds to the Item. For most Items this will be only one PE. However, in the case of a NEWS2000 Item, the PE list could be up to 256 entitlement codes. A second parameter to this DACS Lock API function is a single operator to be assigned to this entitlement codes list. This is required in the case of NEWS2000 so that the PE list can be assigned an "OR" operator. The other possible operator that can be assigned to the entitlement codes list is the "AND" operator that may be required by other Source Servers.
- The DACS Lock must be sent before the Image. This is required so that the Market Data Client application (via the Transport API) can permission the Item as soon as the Image arrives, instead of having to hold the Image and await the Lock.

## 3.3      What a DACS Lock Contains

The DACS Lock needs to contain that information relevant to the requirements for the Item requested. Since a DACS Lock needs to be the minimal size it can to minimize communication costs, a Source Server encodes the requirements into a single numeric number. In Figure 1, a mapping table is created by the Source Server mapping the textual requirements to a numeric value. By supplying a operator to the DACS Lock API function that creates DACS Locks, complex requirements can be created by the Source Server. Referring to Figure 1, and examining the mapping tables, the following requirements can be formulated:

| ENTITLEMENT CODES | REQUIREMENTS |
|---|---|
| 456 | NYSE & Money 2000 & (NY | LONDON) |
| 985 | LSE & (Energy 2000 | Money 2000) & NY |
| 1027 | NWEX & (Energy 2000 | Markets 2000) & LONDON |

**Table 4: Item Requirement Formulations**

The Item contains the PE and therefore, by using the mapping tables, also contains the permissioning requirements. The Source Server can also add extra requirements to the DACS Lock for an Item by adding extra entitlement codes to the PE list supplied to the appropriate DACS Lock API class. For example, if the Source Server imposes that only a Page_Call application can use this Item, then the Source Server makes a new PE with that requirement and AND's it to the PE for that Item.

| ENTITLEMENT CODES | REQUIREMENTS |
|---|---|
| 5000 | Page_Call |

**Table 5: PE Example that can Add Extra Requirements to an Item**

So if an Item has a PE = 456 and the Source Server requires only Page_Call access, then the Source Server supplies entitlement codes 456 and 5000 as the parameters passed to the appropriate DACS Lock API function that builds DACS Locks from PE lists.

## 3.4 Compression of DACS Locks

To minimize the physical size of a DACS Lock it is compressed. The type of compression is based on Binary Coded Decimal (BCD) algorithm. For example, the DACS Lock API is required to make a lock that has the following PE requirements:

| IDN | BRIDGE |
|---|---|
| 1027 & (456 \| 985) & 5000 | 1 & 2 |

**Table 6: PE Requirements for a Compound Item**

What the compression algorithm performs is an interpretation of operator functions and the BCD conversion of the numeric PE values using the following table:

| OPERATION | REPLACED NIBBLE VALUE |
|---|---|
| numeric 0 – 9 | 0 - 9 |
| "&" and operation | A |
| "\|" or operation | B |
| "EOF" End of DACS Lock | C |
| "EOS" End of Source Server PE List | D |

**Table 7: Operator and BCD Interpretation Table**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| 0x03 | 0x00 | 0x01 | 0x10 | 0x27 | 0xA4 | 0x56 | 0xB9 | 0x85 | 0xD0 | 0x00 | 0x02 | 0x50 | 0x00 | 0xCX |

Version of DACS Lock

0x10 = the first two digits of the 1027 PE.

The "A" flags the "&" operator for the following operand. The "4" is the first digit of the 456 PE.

The "B" flags the "|" operator for the following operand. The "9" is the first digit of the 985 PE.

The "D" flags the End of the IDN Source Servers PC list. The 0 is the first nibble of the next Source Service.

The "1" is the first and only digit of the 1 PE.

The "C" flags the end of the DACS Lock, implying that no more Services and PE follow. The "X" is a don't care value.

NULL

This is the service number for IDN.

0x27 = the last two digits of the 1027 PE.

0x56 = the last two digits of the 456 PE.

0x85 = the last two digits of the 985 PE.

This is the service number for Bridge.

### Representation of a DACS Lock Array after Compression*

*: This diagram is only a representation for purposes of understanding. The actual DACS Lock array and its compression are proprietary to Thomson Reuters. As such, building a DACSLock from this diagram will result in an error.
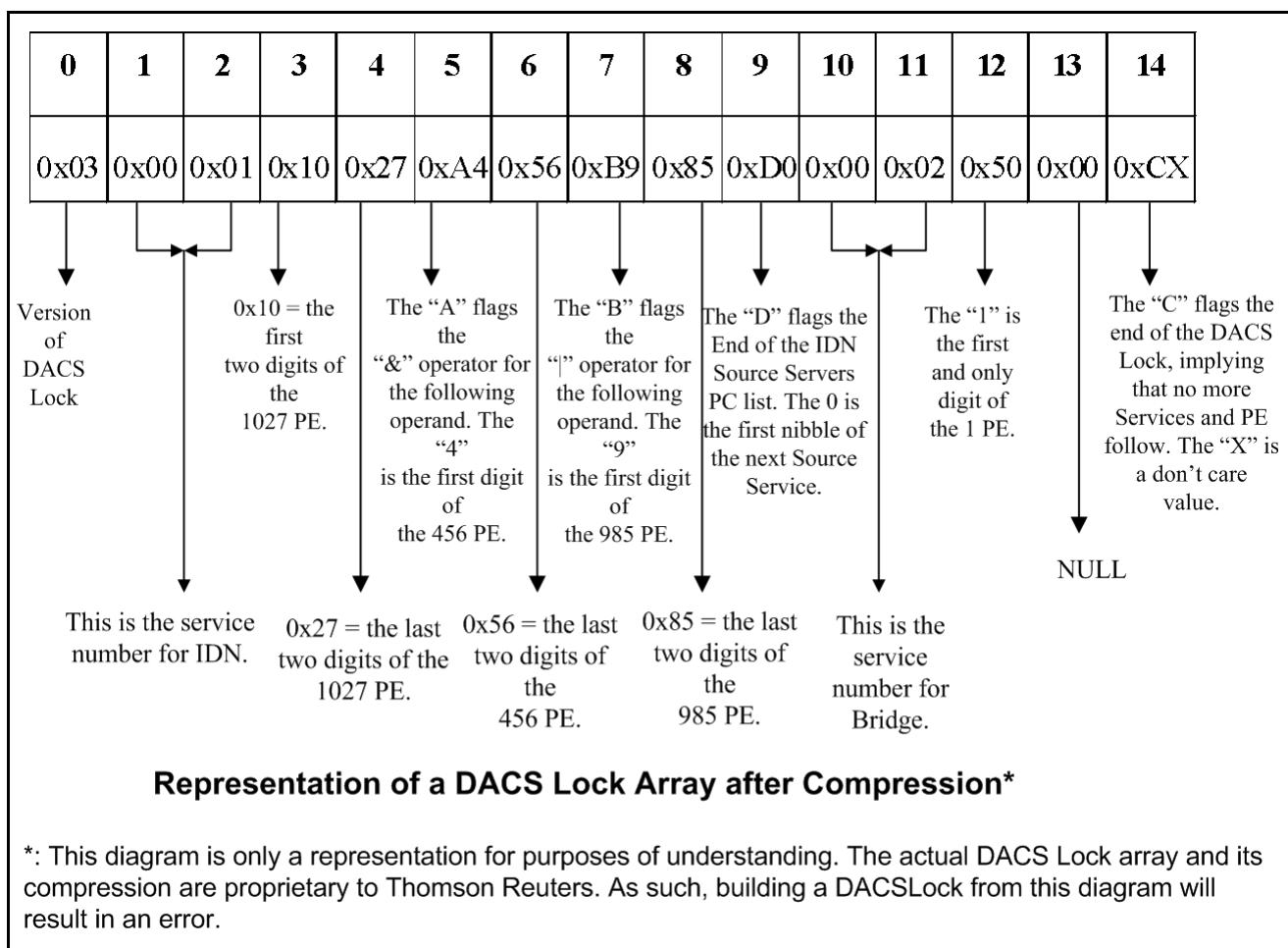
**Figure 2: DACS Lock Array after Compression**

By using the Operator and BCD Interpretation Table, the above unsigned character array is created. This array shows some of the advantages of this compression. First, it makes no stipulation to the maximum value of a PE, thus not imposing a limitation of PE values, other than a possible machine maximum that can be manipulated easily with the software language used, i.e. (2^32)-1 for an unsigned long. The second advantage is the compression is simple and not computational intensive for fast compression and decompression. Third, the compressed DACS Lock is smaller then the actual ASCII string if it were sent.

## 3.5 Compounding DACS Locks

The previous example, in Figure 2, had two Source Server PE lists within the DACS Lock. This situation is possible when a compound server combines the Items from more then one Source Server type. To compound a DACS Lock, the compound server calls the DACS Lock API function that, when given the constituent Item DACS Locks, will create a DACS Locks that contains all of the constituent Item requirements. To minimize the size of a DACS Lock, the DACS Lock API uses Boolean algebra rules to minimize the entitlement codes within the PE list wherever possible. Some of the rules are:

```
(A | B) & A = A
A & A & B = A & B
A | A | B = A | B
```

The Compound Server stores all constituent Item DACS Locks until the Item is no longer required. This functionality is required in the event a new DACS Lock is received by the compound server for an Item that it has open, thus requiring the Compound Server to update the compound item DACS Lock which the received DACS Lock is part of. Then the Compound Server is required to forward its new compound Item DACS Lock to those servers that have the compound Item open.

## 3.6 Transport of DACS Locks

DACS Locks for Compound Servers can grow larger then the maximum size of a single message. If the DACS Lock should be larger then this maximum, the Server is required to split the DACS Lock into multiple messages.

# 3.7 Compound DACS Lock in Relation to Permissioning

The previous sections explain the rules for constructing and transporting the DACS Locks. Figure 3, "Compound Locks", shows what the data flow functionality of a DACS Lock is in an operating environment with two Source Servers, one Compound Server, and a Market Data Client application. In this example the Market Data Client application is requesting an item from a Compound Server. This item is made from the constituent items from the IDN and Bridge Servers.
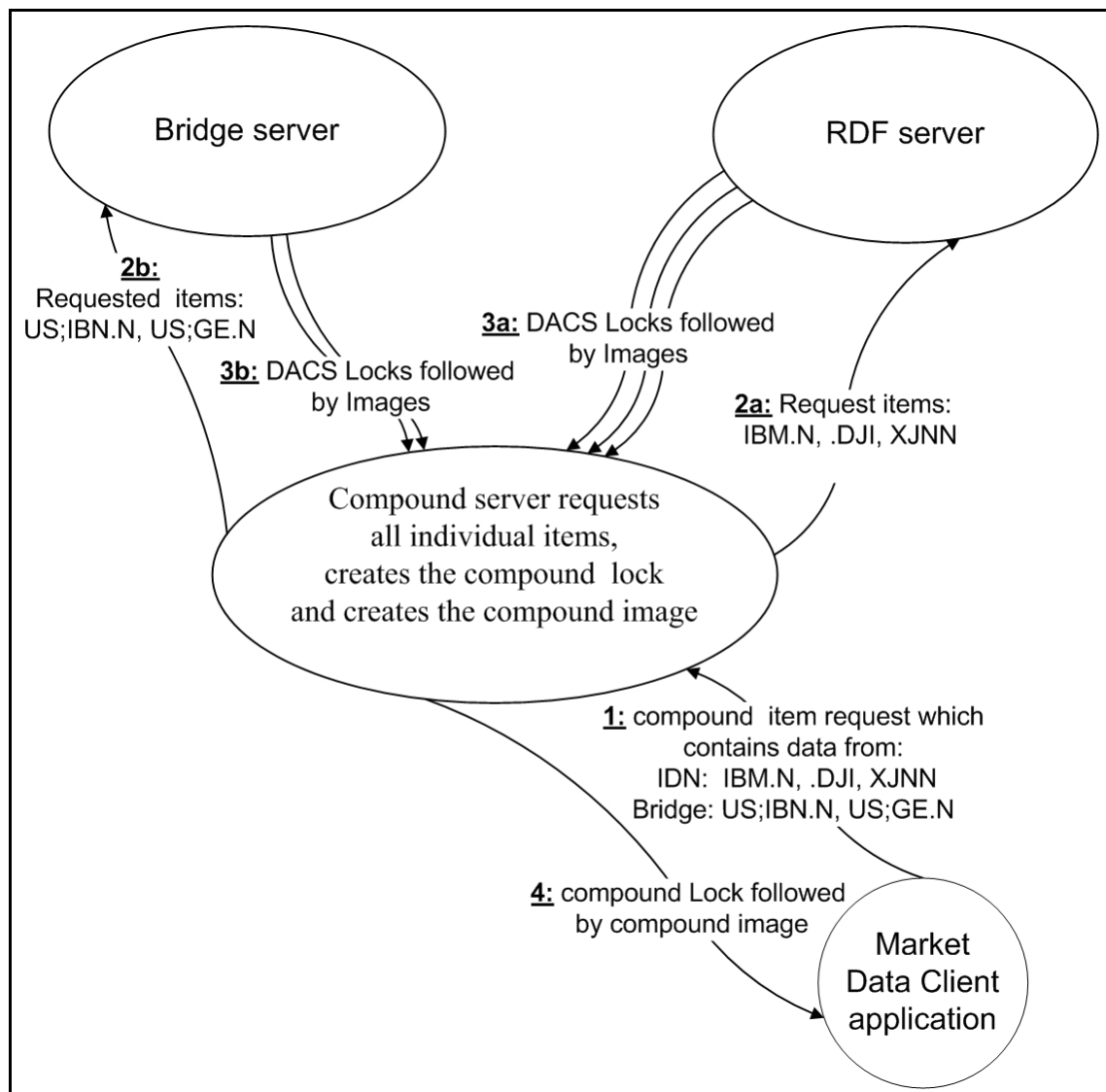
Figure 3: Compound Locks

# Chapter 4   DACS_CsLock()

## 4.1      Synopsis

The DACS_GetLock() function combines a list of access locks into a single composite DACS access lock. You use this function to form complex access locks from constituent access locks that were created by a previous call to the DACS_CsLock() or DACS_GetLock() functions.

```
DACS_CsLock (nm_channel, item_name, newLock, newLockLen, lockList, Dacs_Error)
    int                 nm_channel
    char                *item_name
    unsigned char       **newLock
    int                 *newLockLen;
    COMB_LOCK_TYPE      lockList[];
    DACS_ERROR_TYPE     *Dacs_Error
```

## 4.2      Function Arguments

| ARGUMENT | DESCRIPTION |
|---|---|
| nm_channel | The nm_channel parameter is no longer used but it must be 0 (zero) for backward compatibility. |
| item_name | The item_name parameter must be an empty string (""). (This is also for backward compatibility.) |
| newLock | The newLock parameter is a pointer to an unsigned char pointer that shall point to the generated access lock for the "item" built by the source/compound server. If the pointer is NULL on entry to the DACS_CsLock() function, space for the new access lock will be dynamically allocated using malloc(). It is the responsibility for the caller of DACS_CsLock() to free() the memory allocated when the newly generated access lock is no longer required. |
| newLockLen | The newLockLen parameter is updated to reflect the length of the newly generated lock. If the newLock parameter does not point to a NULL pointer, then the source/compound server application must supply the *newLockLen with the maximum size of the user-supplied access lock pointer. If the *newLockLen parameter supplied by the source/compound server application is less then the length required to fit the access lock, a PERM_FAILURE error is returned. |
| lockList | The lockList parameter is a pointer to an array of pointers to the access locks of associated component items to be combined by the source/compound server. A NULL pointer terminates the list. |
| Dacs_Error | The Dacs_Error parameter points to a data structure of type DACS_ERROR_TYPE in which returned errors will be placed. |

  **Table 8: DACS_CsLock() Functional Arguments**

## 4.3    Data Structure: COMB_LOCK_TYPE

The `COMB_LOCK_TYPE` data structure is defined to be as follows:

```
typedef struct {
    int              server_type;
    char             *item_name;
    unsigned char    *access_lock;
    int              lockLen;
    } COMB_LOCK_TYPE;
```

Where:

- *server_type* must be set to 0 (zero).
- *item_name* is a NULL pointer (for backward compatibility).
- *access_lock* is a pointer to the user-supplied access lock that is to be used as one of the constituent item access locks.
- *lockLen* is the length of the access lock. This value was previously returned from a `DACS_GetLock()` function.

## 4.4    Data Structure: DACS_ERROR_TYPE

The `DACS_ERROR_TYPE` data structure is defined to be as follows:

```
typedef struct {
    short      dacs_error;
    short      dacs_suberr;
    } DACS_ERROR_TYPE;
```

## 4.5    Return Values

This function returns **DACS_SUCCESS** if the function did not encounter a fatal error, and the `newLock` and `newLockLen` parameters shall be populated.

**DACS_FAILURE** is returned if a fatal unrecoverable error was encountered. An ASCII explanation of the error can be further determined by passing the `Dacs_Error` data structure to the `DACS_perror()` function.

# Chapter 5   DACS_GetLock()

## 5.1      Synopsis

The `DACS_GetLock()` function creates a DACS access lock from a list of entitlement codes (i.e., codes specified by the data vendor on which all permissioning is based).

```
DACS_GetLock    (service_type, ProductCodeList, LockPtr, LockLen, Dacs_Error)
    int                     service_type;
    PRODUCT_CODE_TYPE       *ProductCodeList;
    unsigned                char **LockPtr;
    int                     *LockLen;
    DACS_ERROR_TYPE         *Dacs_Error;
```

## 5.2      Function Arguments

| ARGUMENT | DESCRIPTION |
|---|---|
| service_type | The `service_type` parameter is used to pass the service type of the server that is to be encoded into the DACS access lock. This value is a unique numeric ID assigned to a service by the service's `serviceId` parameter in the global configuration file used by core infrastructure components within the Thomson Reuters Enterprise Platform for Real-Time. |
| ProductCodeList | The `ProductCodeList` parameter is used to pass the list of entitlement codes which shall be encoded into the DACS access lock. |
| LockPtr | The `LockPtr` parameter is a pointer to an unsigned char pointer that shall point to the generated access lock that represents the entitlement code list in DACS lock format. If the pointer is NULL on entry to the `DACS_GetLock()` function, space for the new access lock will be dynamically allocated using `malloc()`. It is the responsibility for the caller of `DACS_GetLock()` to `free()` the memory allocated when the newly generated access lock is no longer required.<br><br>**Note:**   If the `DACS_GetLock()` function returns an error, then no space for the access lock will have been allocated. |
| LockLen | The `LockLen` parameter is a pointer to an int. This parameter is updated to reflect the length of the newly generated access lock. If the `LockPtr` parameter does not point to a NULL pointer on entry, then the user must supply the `LockLen` with the maximum size of the user-supplied access lock buffer. If the user-supplied `LockLen` is less then the length required to fit the access lock, then a **DACS_FAILURE** error is returned. |
| Dacs_Error | The `Dacs_Error` parameter points to a data structure of type `DACS_ERROR_TYPE` in which returned errors will be placed. |

Table 9: `DACS_GetLock()` Functional Arguments

## 5.3 Data Structure: PRODUCT_CODE_TYPE

The **PRODUCT_CODE_TYPE** data structure is defined to be as follows:

```
typedef struct {
    char            operator;
    unsigned short  pc_listLen;
    unsigned long   pc_list[/* pc_listLen */];;
    } PRODUCT_CODE_TYPE;
```

where:

- *operator*: is the user-supplied operation that shall be imposed on the entitlement code list that is supplied. The operator field must have one of the following formats:

  - The ampersand character '**&**' (for an "AND" list): The implication is that the user of the item must be permissioned for all the entitlement codes contained within the access lock.
  - The pipe character '**|**' (for an "OR" list): The implication is that the user of the item only needs access to any one of the entitlement codes contained within the access lock.

- *pc_listLen*: is used to flag the number of entries that are contained within the **PRODUCT_CODE_TYPE** array.

- *pc_list*: is a numerically ascending sorted list of **unsigned long** values that shall be interpreted as the entitlement codes to be encoded into a DACS access lock.

## 5.4 Data Structure: DACS_ERROR_TYPE

The **DACS_ERROR_TYPE** data structure is defined to be as follows:

```
typedef struct {
    short dacs_error;
    short dacs_suberr;
    } DACS_ERROR_TYPE;
```

## 5.5 Return Value

This function returns **DACS_SUCCESS** if the function does not encounter a fatal error.

**DACS_FAILURE** is returned if a fatal unrecoverable error was encountered. An ASCII explanation of the error can be further determined by passing the **Dacs_Error** data structure to the **DACS_perror()** function.

# Chapter 6   DACS_CmpLock()

## 6.1      Synopsis

The `DACS_CmpLock()` function compares two access locks for equality. You can use this function to verify whether a new access lock is different from a previously-generated access lock, thus reducing the possible overhead incurred in redistribution of an unchanged access lock.

```
DACS_CmpLock (Lock1Ptr, Lock1Len, Lock2Ptr, Lock2Len, Dacs_Error)
    unsigned char      *Lock1Ptr;
    int                Lock1Len;
    unsigned char      *Lock2Ptr;
    int                Lock2Len;
    DACS_ERROR_TYPE    *Dacs_Error;
```

## 6.2      Function Arguments

| ARGUMENT | DESCRIPTION |
|---|---|
| Lock1Ptr | The `Lock1Ptr` parameter is a pointer to the first access lock that is to be compared. |
| Lock1Len | The `Lock1Len` parameter is an integer. This parameter is the length of the first access lock that is to be compared. |
| Lock2Ptr | The `Lock2Ptr` parameter is a pointer to the second access lock that is to be compared. |
| Lock2Len | The `Lock2Len` parameter is an integer. This parameter is the length of the second access lock that is to be compared. |

Table 10: `DACS_CmpLock()` Functional Arguments

## 6.3      Return Value

This function returns **DACS_SUCCESS** if the function did not encounter a fatal error and the access locks were logically identical.

**DACS_DIFF** is returned if a fatal error was not encountered but the two access locks were logically different.

**DACS_FAILURE** is returned if a fatal unrecoverable error was encountered. An ASCII explanation of the error can be further determined by passing the `Dacs_Error` data structure to the `DACS_perror()` function.

# Chapter 7    DACS_perror()

## 7.1    Synopsis

The `DACS_perror()` function creates a textual message describing the last error generated by a DACS Library call. The message is saved to the supplied buffer. The error number is taken from the location `Dacs_Error -> dacs_errno` variable pointed to by the user application, when a library function returns a **DACS_FAILURE**.

```
DACS_perror (err_buffer, buffer_len, text, Dacs_Error);
    unsigned char      *err_buffer;
    int                buffer_len;
    unsigned char      *text;
    DACS_ERROR_TYPE    *Dacs_Error;
```

## 7.2    Function Arguments

| ARGUMENT | DESCRIPTION |
|---|---|
| err_buffer | The `err_buffer` parameter contains the destination address for the generated error string. If the supplied buffer is smaller then the generated error string, DACS_FAILURE will be returned. |
| buffer_len | The `buffer_len` is a variable that indicates the maximum size of the `err_buffer`. |
| text | The `text` parameter is a pointer to a null-terminated character string that is placed into the buffer before the DACS error message. This string and the error message will be separated by a colon and a blank space. If an empty character string is specified, only the DACS Library error message will be placed into the `err_buffer`. |
| Dacs_Error | The `Dacs_Error` parameter points to a data structure of type `DACS_ERROR_TYPE` into which returned errors shall be placed. |

**Table 11: `DACS_perror()` Functional Arguments**

## 7.3    Return Values

This function returns **DACS_SUCCESS** if the function did not encounter a fatal error.

**DACS_FAILURE** is returned if a fatal unrecoverable error was encountered.

THOMSON REUTERS