Distributed Statistic Records Shared Library
–
API Guide for the writer

Last updated on
25 November 2002

Content

Concept of operation

The Distributed Statistic Records Shared Library (in what follows called ‘DSRlib’) is designed to be used by non R/3 applications (in what follows called ‘Components’) to write statistical and trace data. The library provides means to write data in a centralized, synchronized and standardized manner.

This data may be retrieved by using API calls and e.g. posted into a R/3 monitoring system. The DSRlib provides functionality to retrieve, filter and aggregate statistical and trace data.

Statistic data generated by the DSRlib is transported to a R/3 monitoring system with an SAP agent ‘sapccmsr’.

Data collected by the DSRlib is logically and physically grouped by the Name of the generating Component. A Component is uniquely identified by its Name. A Component may physically consist of several processes which may consist of several execution units as well. All of which may generate statistical and trace data. Synchronization among and within Components is handled by the DSRlib.

Statistic data generated by different components may be logically connected by means of a so called passport. Via the passport concept it is possible to collect from several components that kind of data which is logically associated with one piece of work spread over several component.

The DSRlib buffers statistical data in a flip flop buffer which is located in a shared memory segment. When the buffer is flipped the inactive part is flushed asynchronously into a file.

Statistic files are created on an hourly base. All statistic data of all execution units of a component is condensed into one file. Statistic files are kept by default for 48 hours. This value can be modified by an environment variable (see section ‘Prerequisites’).
NOTE: Statistic files may also be deleted from DSRlib on request from the monitoring system. So the number of statistic files may be 48 by default or less.

Statistic files are written to a subdirectory of the DSRlib’s home directory (see section ‘Prerequisites’). The name of the subdirectory matches the name of the component.

Statistic files obey the following naming scheme:

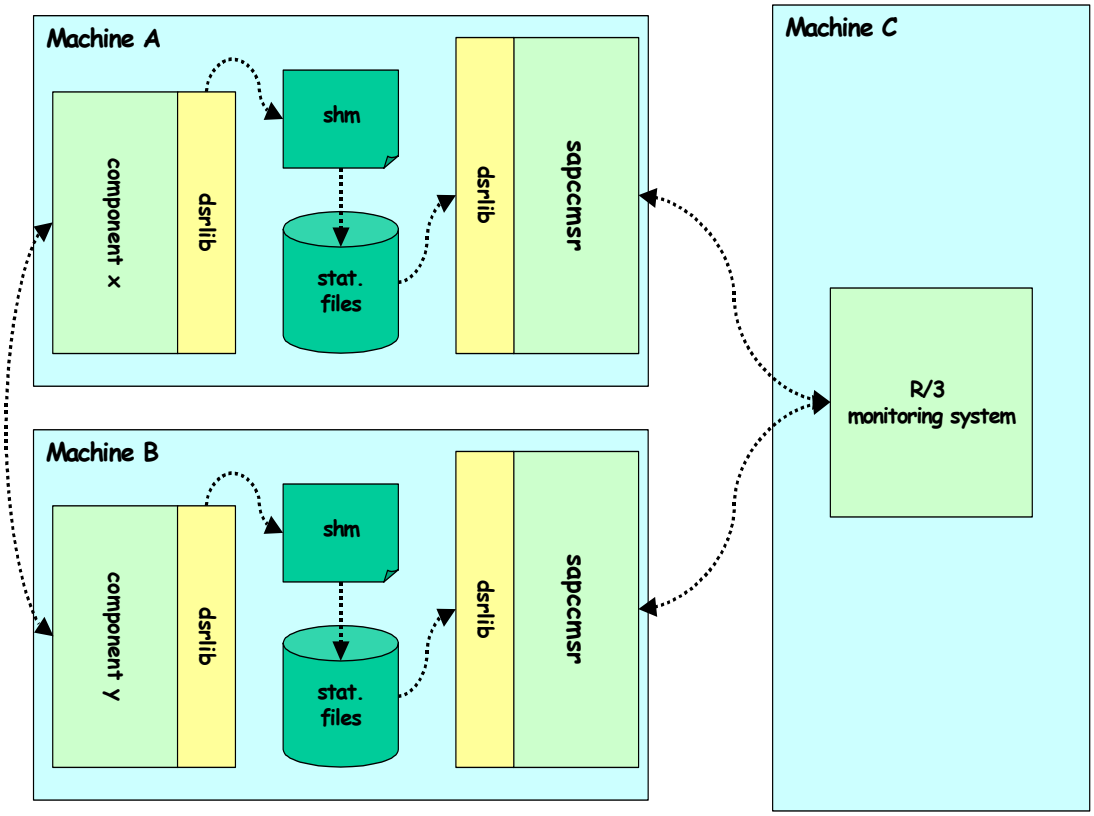
‘<name of component>_<dd><mm><yyyy><hh>’.

Each statistic file has an associated index file, which is used (but not required) to speed up read access. Index files obey the following naming convention:

<name of statistic file>.six

Both index and statistical file are not human readable.

In the following picture a short glimpse at the landscape is given:



Prerequisites

Version 1 of the DSRlib supports only Windows NT and Windows 2000.

The DSRlib requires the full qualified path of its home directory to be stored in the registry under key

HKEY_LOCAL_MACHINE\SOFTWARE\SAP\CCMS\DSR\DirDsr

as a string value. The specified directory has to exist and has to be accessible. The DSRlib creates subfolders for each component it collects statistical data for.

Usually statistic files are kept for 48 hours. One can change this value by setting the environment variable

dsr_max_stat_file_count

to the number of hours statistic files shall be kept.
NOTE: This parameter determines the maximum count of statistic files per component. Since it is possible to delete statistic files via DSRlib requests, the actual number may be less. Use transaction ST03G to remotely trigger deletion of statistic files.

Data types

Overview

The DSRlib uses heavily custom type definitions. Names of data types obey the following naming scheme:

1. The name of a data type is prefixed with the shortcut of the module name it is defined in. In case of the main header this is 'dsr'.
2. The name of a data type is suffixed with 'T'.

RFC table data types in 'dsrxxrfc.h' build an exception to this rule.

DSR basic data types

The DSRlib defines the following base data types:

- dsrServiceT;
- dsrActionTypeT;
- dsrTraceT;
- dsrTransIdT;
- dsrActionT;
- dsrUserT;
- dsrSystemT;
- dsrAddInfoT;
- dsrComponentNameT;
- dsrComponentTypeT;

- dsrDestinationT;
- dsrModuleNameT;
- dsrModuleTypeT;
- dsrExternalAccessT;
- dsrTraceMessageT;

These are derived from SAP_INT and SAP_CHAR.

Furthermore the following SAP types are used:

- SAP_INT4;
- SAP_SHORT;
- SAP_RAW;
- SAP_DATE;
- SAP_TIME;
- RFC_DATE;
- RFC_TIME;

DSR structured data types

Overview

Structured data types are used for the statistic records and for an additional component administration record used at DSRlib initialization time.

In order to write a set of statistic records the component has to

1. fill the desired records with data,
2. chain those records together and
3. handle them over to the write API call ‘dsrWrite()’.

In what follows a chained set of records used for input to the DSR write call dsrWrite() is called ‘atomic set’.

Each atomic set is associated with an action of a component. A component may call into other components in order to perform other actions. Each action and therefore each atomic set belongs to a Logical Unit of Work (LUW). A LUW may involve several actions and therefore several atomic sets spread over several components. A LUW is identified by an unique tag called ‘transaction Id’. A transaction Id is unique in time and over machine boundaries.

In what follows the component which is writing a set of statistic records associated with an action is called the generating component. A component called by the generating component is called subcomponent.

All data records are wrapped into type

dsrRecT.

It is declared as

```
typedef struct _dsrRecT
{
  dsrRecTypeT       recordType;
  dsrRecUnionT     *record;
  struct _dsrRecT  *next;
}
dsrRecT;
```

The Attribute 'recordType' identifies the type of the statistic record. dsrRecTypeT is defined as an enumeration type:

```
typedef enum _dsrRecTypeT
{
  dsrRecType_Main    = 1,
  dsrRecType_Cert,
  dsrRecType_Call
}
dsrRecTypeT;
```

The DSRlib knows in version 1 three types of records:

- The main record. Each atomic set has to have exactly one main record. This main record has to precede all other records in the chained list. An atomic set may contain nothing but the main record. That is why a main record is called main record. All other statistic records can not form for itself an atomic set. That's why they are called subrecords. The main record includes all important statistic information concerning a specific action.
- The certification or client info subrecord (in what follows called cert record). This subrecord cannot form an atomic set on its own. It contains data of the origin of a Logical Unit of Work (LUW). Each atomic set may have at most one cert subrecord.
- The call type 1 subrecord (in what follows referred to as the call subrecord). The call subrecord cannot form an atomic set on its own. An atomic set can contain several call subrecords. If a component calls into a subcomponent (e.g. into a database), it attaches a call record to track performance of the called subcomponent.
  NOTE: Several calls into a component may be summed up in one call subrecord.

The attribute ‘next’ of an instance of type dsrRecT points to the next subrecord in an atomic set. **NOTE:** The last record in this chained list has to have a NULL next pointer.

The attribute ‘record’ of an instance of type dsrRecT points to an instance of an union data type containing the actual statistic record. dsrRecUnionT is defined as follows.

```
typedef union _dsrRecUnionT
{
  dsrRecMainT     mainRecord;
  dsrRecCertT     certRecord;
  dsrRecCallT     callRecord;
}
dsrRecUnionT;
```

Depending on the specified record type ‘recordType’ in dsrRecT the attribute ‘mainRecord’ or ‘certRecord’ or ‘callRecord’ of an instance of dsrRecUnionT has to be filled.

NOTE: Character array attributes in statistic records are expected to be not zero terminated. This is because statistic records are directly mapped onto RFC tables. A user or the DSRlib can of course zero terminate them if not the whole length of the array is needed.

NOTE: Character arrays in the component static data record are zero terminated.

Main record

The main record is defined as follows:

```
typedef struct _dsrRecMainT
{
  SAP_RAW             recVersion;
  SAP_INT4            startTimeSec;
  SAP_INT4            startTimeMilliSec;
  SAP_INT4            endTimeSec;
  SAP_INT4            endTimeMilliSec;
  SAP_DATE            startDate;
  SAP_TIME            startTime;
  SAP_DATE            endDate;
  SAP_TIME            endTime;
  dsrTransIdT         transId;
  dsrServiceT         service;
  dsrActionT          action;
  dsrActionTypeT      actionType;
  dsrUserT            userId;
  SAP_INT4            processId;
  SAP_INT4            threadId;
  SAP_INT4            cpuTime;
  SAP_INT4            queueTime;
  SAP_INT4            loadTime;
  SAP_INT4            genTime;
  SAP_INT4            netTime;
  SAP_INT4            waitTime;
  SAP_INT4            maxMem;
```

```
  SAP_INT4          respTime;
  SAP_RAW           luwInfo;
  dsrAddInfoT       addInfo;
}
dsrRecMainT;
```

It contains performance and administrative information about an action of a component:

- **recVersion**: Contains the version of the main record. The value of this attribute is provided from the DSRlib and currently set to 1.
- **startTimeSec**: Has to be provided by the generating component. It contains the start time in seconds since 1.1.1970 UTC of the start of the recorded action. An UTC timestamp can be retrieved by using the DSR API call 'dsrGetTime()'.
- **startTimeMilliSec**: Has to be provided by the component. It contains the start time in milliseconds since startTimeSec. Use here DSR API call 'dsrGetTime()', as well.
- **endTimeSec**: Provided by DSRlib. It contains the end time in seconds since 1.1.1970 UTC of the action. Before storing the atomic set, the DSRlib fills in the current UTC time.
- **endTimeMilliSec**: Provided by DSRlib. It contains the end time in milliseconds of the performed action since endTimeSec. Before storing the atomic set, the DSRlib fills in the current milliseconds value.
- **startDate**: Provided by the component. 8 character string not zero terminated. Format 'yyyymmdd'. Start date of action.
- **startTime**: Provided by the component. 6 character string not zero terminated. Format 'hhmmss'. Start time of action.
- **endDate**: Provided by DSRlib. 8 character string not zero terminated. Format 'yyyymmdd'. End date of action. Written at store time.
- **endTime**: Provided by DSRlib. 6 character string not zero terminated. Format 'hhmmss'. End time of action. Written at store time.
- **transId**: Character array type of length 32. Not zero terminated. Transaction Id. Provided by the generating component. A transaction Id is a unique identifier used to identify a Logical Unit of Work (LUW). A LUW may contain several component actions (normally an atomic set is written for each component action). A LUW may be spread over several components. When trying to identify all actions of a LUW the atomic sets of all participating components are searched for the associated transaction Id. A transaction Id is sometimes also called GUID. If a component starts a new LUW it uses DSR API call dsrGetGuid() to generate a fresh transaction Id. If a component is performing some action for an already existing LUW it has to use the already existing transaction Id in attribute transId. Transaction Ids are transported in a passport via RFC and DIAG. See DSR API calls dsrGetNetPassport() dsrSetNetPassport().
- **service**: Integer type. Provided by the generating component. Contains the service type of an action performed by the component. R/3 for example knows such service types like Batch, Dialog, Update. When a component registers itself at the DSRlib it specifies its known service types together with a short description (see section 'Component static data record').

- **action**: Character array type of length 128. Not zero terminated. Provided by the generating component. This attribute contains the name of an action performed by the component. A R/3 analogy would be e.g. the name of a batch job or a transaction code.
- **actionType**: Integer type. Provided by the generating component. This attribute contains the type of an action. A R/3 analogy would be 'batch job', 'transaction', 'report'.
- **userId**; Character array type of length 32. Not zero terminated. Provided by the generating component.
- **processId**: Integer type. Provided by the generating component. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **threadId**: Integer type. Provided by the generating component. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **cpuTime**: Integer type. Provided by the generating component. Pure CPU time spent for the associated action. Measured in milliseconds. Timestamps can be retrieved by using DSR API call dsrGetTime(). Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **queueTime**: Integer type. Provided by the generating component. This attribute contains the time measured in milliseconds the component has to wait to start performing the monitored action, e.g. in a message or job queue. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **loadTime**: Integer type. Provided by the generating component. This attribute contains the time measured in milliseconds the component has to spend for loading information it needs for processing. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **genTime**: Integer type. Provided by the generating component. This attribute contains the time measured in milliseconds the component has to spend for some generating processes like generating JAVA applets. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **netTime**: Integer type. Provided by the generating component. This attribute contains the time measured in milliseconds that was spent calling the generating component. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **waitTime**: Integer type. Provided by the generating component. Contains the total wait time in milliseconds spent waiting by the generating component for called components. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **maxMem**: Integer type. Provided by the generating component. Contains the maximum amount of memory in kb allocated by the generating component used to perform the associated action. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **respTime**: Integer type. Provided by the generating component. Contains the total time in milliseconds the generating component spent to perform the associated action. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **luwInfo**: Raw type. Provided by the generating component. Contains information of the status of the associated action in respect to the associated LUW. Use the following macros when populating this attribute:

dsr_LUW_TYPE_START
dsr_LUW_TYPE_END
dsr_LUW_TYPE_1ST_STEP_IN_HOUR
dsr_LUW_TYPE_1ST_LUW_OF_SESS
dsr_LUW_TYPE_1ST_STEP_IN_REM_SESS

- **addInfo**: Character array type of length 256. Not zero terminated. Provided by the generating component. This attribute may contain any data the generating component regards as meaningful or important. Use the following format: <identifier 1>=<value 1>, <identifier 2>=<value 2>, ...

Subrecords

Subrecords always have to be attached or chained to a main record. A main record may have several or zero calltype1 subrecords and one or zero certificate subrecord.

A certificate subrecord indicates the origin of a LUW. Data contained in the cert record and a transaction Id are building up the passport. A passport is send via RFC or DIAG from the generating component to subcomponents. Subcomponents use the transaction Id of the passport to logically connect to an existing LUW.

If a passport is included into the call to a subcomponent, the subcomponent has to

1. extract the transaction Id out of the passport (DSR API call dsrGetNetPassport()) and use it in all main records of all atomic sets written associated with the call and
2. write a cert subrecord containing the passport information in one of the atomic sets associated with the call. The information contained in this cert subrecord is fully provided by the passport. If a subcomponent writes a cert subrecord this information must not be altered. If subcomponent A calls an other component B within a LUW, component A has to alter the attribute ‘preSysId’ of the passport sent to component B. See definition of attribute ‘perSysId’ of the cert subrecord.

A subcomponent may attach a cert subrecord to all atomic sets written associated with the call. But it is required and recommended only for one atomic set. It is recommended to attach the cert subrecord to the first atomic set written.

If a passport is not included into the call of a subcomponent, the subcomponent starts its own LUW and has to request a fresh Transaction Id form the DSRlib (dsrGetGuid() call).

If a component is the creator of a LUW and calls into subcomponents in order to perform parts of the LUW, it has to create a passport and handle it over to the subcomponent (see DSR API call dsrSetNetPassport()). In such a case the LUW starting component may write a cert record with an **EMPTY** preSysId attribute but it is not recommended to do so.

If a component is the creator of a LUW and does not call into subcomponents it is not necessary for the LUW starting component to write a cert subrecord.

Calltype1 subrecord

The call type 1 subrecord is defined as follows:

```
typedef struct _dsrRecCallT
{
  SAP_RAW             recVersion;
  dsrComponentTypeT   componentType;
  dsrComponentNameT   componentName;
  dsrDestinationT     destination;
  SAP_INT4            callTimestamp;
  SAP_INT4            receiveTimestamp;
  SAP_INT4            sentBytes;
  SAP_INT4            receivedBytes;
  SAP_SHORT           numberOfCalls;
  dsrAddInfoT         addInfo;
}
dsrRecCallT;
```

It contains information of a called subcomponent:

- **recVersion**: Provided by DSRlib. Contains the version of the call subrecord. Currently set to 1.
- **componentType**: Provided by the generating component. Character array of length 10. Not zero terminated. Type of the called subcomponent. Not zero terminated. When a component registers itself at the DSRlib (dsrInit() call), it provides information on its type. See section ‘Component static data record’. Use the same component type here.
- **componentName**: Provided by the generating component. Character array of length 32. Not zero terminated. Name of the called subcomponent. Not zero terminated. A component is identified by its name which has to be specified when registering at the DSRlib (dsrInit() call).
- **destination**: Provided by the generating component. Character array of length 128. Not zero terminated. Address used for calling the subcomponent e.g. IP-Address or RFC destination.
- **callTimestamp**: Integer type. Provided by the generating component. This is the time in milliseconds measured from the time the subcomponent is called until it returns. If a subcomponent is called several times the generating component has to sum up these intervals in the attribute callTimestamp instead of writing several call subrecords. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **receiveTimestamp**: Integer type. Provided by the generating component. This is the time in milliseconds measured from the time the subcomponent starts its work until it finishes its work. If a subcomponent is called several times the generating component has to sum up these intervals in the attribute receiveTimestamp

instead of writing several call subrecords. By substracting receiveTimestamp from callTimestamp one gets the time spent for communication between generating component and subcomponent. Use Macro STCM15() defined in saptypeb.h to write to this attribute.

- **sentBytes**: Integer type. Provided by the generating component. Total amount of data in kb sent by generating component to subcomponent. If a subcomponent is called several times the sent data volumes have to get summed up in this attribute instead of writing several call subrecords. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **receivedBytes**: Integer type. Provided by the generating component. Total amount of data in kb received by generating component from a called subcomponent. If a subcomponent is called several times the received data volumes can be summed up in this attribute instead of writing several call subrecords. Use Macro STCM15() defined in saptypeb.h to write to this attribute.
- **numberOfCalls**: Integer type. Provided by the generating component. The number of calls made from the generating component to the subcomponent this call subrecord refers to.
- **addInfo**: Character array of length 256. Not zero terminated. Provided by the generating component. This attribute may contain any data the generating component regards as meaningful or important. Use the following format: <identifier 1>=<value 1>, <identifier 2>=<value 2>, ...

Certificate subrecord

All information the component has to fill into the cert subrecord comes from the passport. The cert subrecord is defined as follows:

```
typedef struct _dsrRecCertT
{
  dsrTraceT          traceFlag;
  dsrSystemT         sysID;
  dsrServiceT        service;
  dsrActionT         action;
  dsrActionTypeT     actType;
  dsrUserT           userID;
  dsrSystemT         preSysID;
}
dsrRecCertT;
```

It contains information on the origin of a LUW:

- **traceFlag**: Provided by the generating component. The trace flag is also transported in the passport. Because of this fact it is possible to trace a LUW spread over several components. Usually the initiator of a LUW turns on tracing. All subcomponents involved in the associated LUW will receive the trace flag transported by the passport and in turn trace as well. The DSRlib defines the following values for the trace flag:

```
#define dsr_TRACE_TYPE_SQL                    0x0001
#define dsr_TRACE_TYPE_BUFFER                 0x0002
#define dsr_TRACE_TYPE_ENQEUE                 0x0004
#define dsr_TRACE_TYPE_RFC                    0x0008
#define dsr_TRACE_TYPE_PERMISSION             0x0010
#define dsr_TRACE_TYPE_FREE                   0x0020
#define dsr_TRACE_TYPE_C_FUNCTION             0x0040
#define dsr_TRACE_TYPE_ABAP_CONDENS_0         0x0100
#define dsr_TRACE_TYPE_ABAP_CONDENS_1         0x0200
```

- **sysID**: Provided by the generating component. Character string of length 32. Not zero terminated. This attribute contains the system Id (if available) of the component starting the associated LUW. If a subcomponent uses a passport to fill the attributes of a cert record, it simply takes the sysID value provided in the passport. If a R/3 like system Id is not available (which is likely to be the case in every not R/3 component), use the name of the component here.
  NOTE: If a system Id is not available to a component the component name may be used here as well.
- **service**: Integer type. Provided by the generating component. Contains the service type of an action starting a LUW (see attribute 'service' in main record). If a subcomponent uses a passport to fill the attributes of a cert record, it simply takes the service value provided in the passport. If a component is the initiator of a LUW, and therefore has to generate a passport, the value of the service attribute of the main record is used here.
- **action**: Provided by the generating component. Character array of length 128. Not zero terminated. Contains the name of the action starting a LUW (see attribute 'action' in main record). If a subcomponent uses a passport to fill the attributes of a cert record, it simply takes the action value provided in the passport. If a component is the initiator of a LUW, and therefore has to generate a passport, the value of the action attribute of the main record is used here.
- **actType**: Integer type. Provided by the generating component. Contains the action type of the first action of a LUW (see attribute 'actionType' in main record). If a subcomponent uses a passport to fill the attributes of a cert record, it simply takes the actType value provided in the passport. If a component is the initiator of a LUW, and therefore has to generate a passport, the value of the actType attribute of the main record is used here.
- **userID**: Provided by the generating component. Character array of length 32. Not zero terminated. Contains the name of the user associated with an action starting a LUW (see attribute 'userId' in main record). If a subcomponent uses a passport to fill the attributes of a cert record, it simply takes the userID value provided in the passport. If a component is the initiator of a LUW, and therefore has to generate a passport, the value of the userID attribute of the main record is used here.
- **preSysID**: Provided by the generating component. Character string of length 32. Not zero terminated. Contains the system Id of the component which has called the generating component. If a subcomponent uses a passport to fill the attributes of a cert record, it simply takes the preSysID value provided in the passport.

NOTE: If a subcomponent uses a passport to fill the attributes of a passport for another call to a second subcomponent it has to replace the value of this attribute with its system Id.
NOTE: If a system Id is not available to a component the component name may be used here as well.

Component static data record

When a component registers itself at the DSRlib it has to provide some administrative information about itself. This data is contained in the so called 'component static data record' (CSDR).

For each component exactly one CSDR is maintained.

The DSRlib remembers this data as long as the component is registered. If several instances of a component register itself, the CSDR of the last instance in respect to registration time is valid for all instances.

The CSDR is defined as follows:

```
typedef struct _dsrApplicationStaticDataT
{
  SAP_CHAR          componentName[dsr_COMPONENT_NAME_LEN+1];
  dsrIsSrvOrActionT serviceOrActionTypeType[dsr_SRV_OR_ACT_TYPE_COUNT];
  dsrServiceT       serviceOrActionType[dsr_SRV_OR_ACT_TYPE_COUNT];
  SAP_CHAR   serviceOrActionTypeShortText[dsr_SRV_OR_ACT_TYPE_COUNT]
                                    [dsr_ACT_SRV_TYPE_SHORT_TEXT_LEN+1];
  SAP_CHAR serviceOrActionTypeLongText[dsr_SRV_OR_ACT_TYPE_COUNT]
                                     [dsr_ACT_SRV_TYPE_LONG_TEXT_LEN+1];
  SAP_CHAR className[dsr_CLASS_NAME_LEN+1];
  SAP_CHAR componentType[dsr_COMPONENT_TYPE_LEN+1];
  SAP_CHAR componentVersion[dsr_COMPONENT_VERSION_LEN+1];
  SAP_CHAR linkedSystems[dsr_LINKED_SYSTEMS_COUNT]
                        [dsr_LINKED_SYSTEMS_STRING_LEN+1];
  SAP_CHAR tracePaths[dsr_TRACE_PATHS_COUNT]
                      [dsr_TRACE_PATHS_STRING_LEN+1];
  SAP_CHAR traceFileNames[dsr_TRACE_FILENAMES_COUNT]
                         [dsr_TRACE_FILENAMES_STRING_LEN+1];
  SAP_CHAR traceFilePatterns[dsr_TRACE_FILE_PATTERNS_COUNT]
                           [dsr_TRACE_FILE_PATTERNS_STRING_LEN+1];
  struct _dsrApplicationStaticDataT *next;
}
dsrApplicationStaticDataT;
```

All attributes have to be provided from the component when performing the dsrInit() call. Information contained in the CSDR is stored in the System Component Repository in the monitoring R/3 system.

The semantic of the attributes is described in what follows:

- **componentName**: Character array of length 32. Zero terminated. Contains the name of the component which registers itself.
- **serviceOrActionTypeType**: Enumeration type array of length 10. This array is tightly associated with the attributes 'serviceOrActionType', 'serviceOrActionTypeShortText' and 'serviceOrActionTypeLongText', which are also arrays of length 10. Each array element of one of those attributes is associated with an array element of the same number of each of the other named attributes. Each component has to specify which services and action types it knows and uses. These values depend usually on the type of component (see attribute 'componentType'). A component may register up to nine services or action types in one CSDR. In attribute serviceOrActionTypeType is specified for each array element whether it is regarded as a service or actionType. In attribute serviceOrActionType is the service or action type itself specified (for each array element). In attributes 'serviceOrActionTypeLongText' and 'serviceOrActionTypeShortText' is specified a description of the associated service or action type. A value of 'dsrIsSrvOrAction_Service' (see type definition below) in array element n of attribute 'serviceOrActionTypeType' indicates that array element n of attribute 'serviceOrActionType' contains a service value. It also indicates that array element n of attribute serviceOrActionTypeLongText' and serviceOrActionTypeShortText' contain descriptions of a service. A value of 'dsrIsSrvOrAction_Action' in array element n of attribute 'serviceOrActionTypeType' on the other hand indicates that the associated array elements contain information about an action type. When filling the 'serviceOrActionTypeType' array use 'dsrIsSrvOrAction_Unknown' as the value of the last element (see type definition below) to indicate the end of the list of the service or action type specification. This is why you can only specify nine services or action types in one CSDR although the array has ten elements. The type of the attribute 'serviceOrActionTypeType' is defined as follows:

```
typedef enum _dsrIsSrvOrActionT
{
  dsrIsSrvOrAction_Unknown = 1,
  dsrIsSrvOrAction_Service,
  dsrIsSrvOrAction_Action
}
dsrIsSrvOrActionT;
```

- **serviceOrActionType**: Integer array of length 10. In an array element of this attribute you specify the value of an service or action type. See explanation of attribute 'serviceOrActionTypeType' above.
- **serviceOrActionTypeShortText**: Character array of length 20. Zero terminated. Contains short information on the associated service or action type. See explanation of attribute 'serviceOrActionTypeType' above.
- **serviceOrActionTypeLongText**: Character array of length 128. Zero terminated. Contains information on the associated service or action type. See explanation of attribute 'serviceOrActionTypeType' above.
- **className**: Character array of length 32. Zero terminated. Further identification of a component. Used for classification in the System Component Repository. A

component should fill in its component type (see attribute ‘componentType’) here.

- **componentType**: Character array of length 10. Zero terminated. Identifies the type of a component e.g. ITS (Internet Transaction Server).
- **componentVersion**: Character array of length 32. Zero terminated. Identifies the version of a component.
- **linkedSystems**: String array of length 10. Each string with length 512, zero terminated. If the registering component is tightly associated with one or several R/3 systems or components, the names of these systems/components have to be passed in this array. If the total number n of linked systems is smaller than 10, the n+1 element has to be a zero string, i.e. the first character has to be hex zero.
- **tracePaths**: String array of length 10. Each string with length 256, zero terminated. To be filled if a component specifies a custom trace path at the dsrInit() call. This trace path has to be specified a second time in the first element of this array. The second string element has to be a zero string. If no trace path is specified in the dsrInit() call the first string has to be a zero string. See section ‘API functions – Trace functions and macros’.
- **traceFileNames**: Not used at the moment. String array of length 10. Pass zero string in first array element.
- **traceFilePatterns**: Not used at the moment. String array of length 10. Pass zero string in first array element.
- **next**: Not used at the moment. Pass NULL.

Example LUW scenario

Consider the following example:

1. Component A starts a LUW. In that LUW component A performs two actions a1, a2 and makes one call AB into a subcomponent B.
2. Component B performs action b1. Within this action a call BC into component C is performed.
3. Component C performs two actions c1 and c2 and returns.

The instrumentation of DSRlib leads to the generation of the following atomic sets:

Component A writes:

- For action a1: One atomic set containing a main record.
- For action a2: One atomic set containing a main record.
- For call AB: One atomic set containing a main record and a call subrecord.

Component B writes:

- For action b1: One atomic set containing a main record, a cert subrecord and a call subrecord.

Component C writes:

- For action c1: One atomic set containing a main record and a cert subrecord.
- For action c2: One atomic set containing a main record.

NOTE: Component A regards the call AB into component B as an action for itself, therefore an extra atomic set is written for call AB.
NOTE: Component B regards the call BC into component C as a part of action b1. Therefore only one atomic set is written.
NOTE: Component C attaches only to the first written atomic set a cert subrecord.

There are some additional tasks performed by:

Component A:

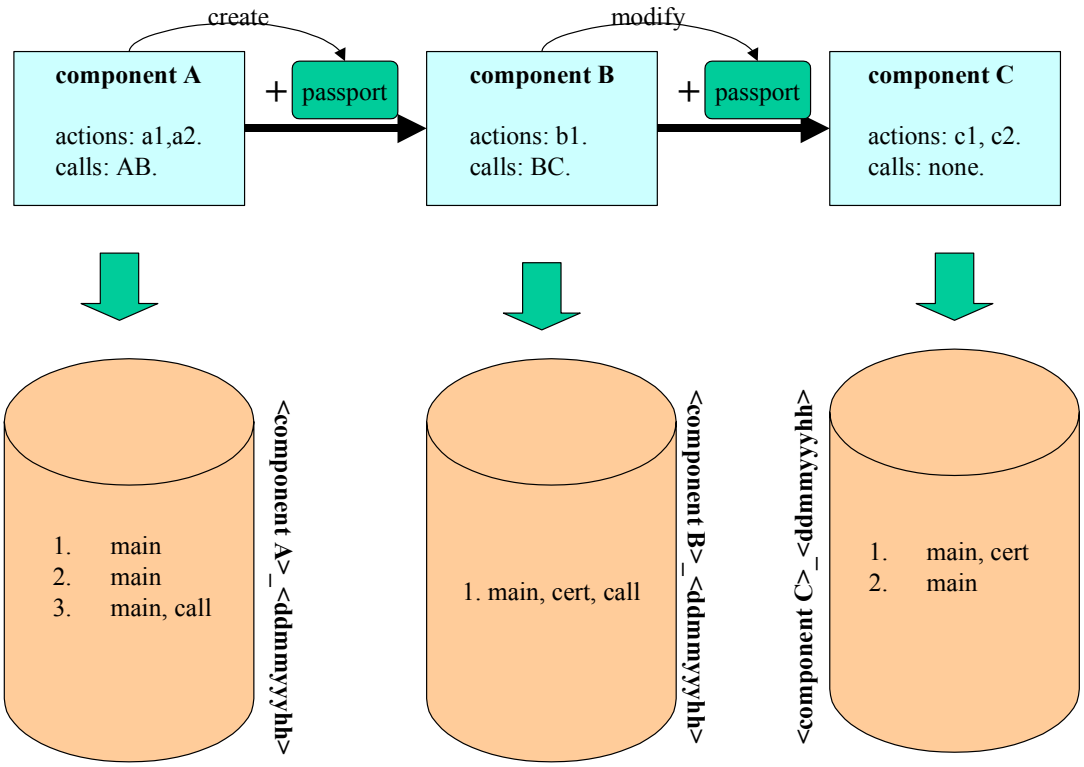
- As component A is starting a LUW it has to request a fresh transaction Id (dsrGetGuid() call).
- It also has to create a new passport and transport it to the called subcomponent B. Component A fills the attribute 'preSysId' of the passport since component A is the first component working on the LUW.

Component B:

- Component B receives the passport and writes it down in a cert subrecord.
- It uses the transaction Id provided by the passport in its main record.
- It updates the attribute 'preSysId' of the passport to its system Id.
- It sends the updated passport with its call to component C.

Component C:

- Component C receives the passport and writes it down in its cert subrecord of its first atomic set.
- It uses the provided transaction Id in all its written main records.


create
modify
component A
actions: a1,a2.
calls: AB.
+ passport
component B
actions: b1.
calls: BC.
+ passport
component C
actions: c1, c2.
calls: none.
1. main
2. main
3. main, call
<component A>_<ddmmyyyyhh>
1. main, cert, call
<component B>_<ddmmyyyyhh>
1. main, cert
2. main
<component C>_<ddmmyyyyhh>

API functions

Initialization

Before a component is able to use the DSRlib it has to perform an initialization call. During this call administrative data of the component is stored and an shared memory buffer reserved.

dsrInit() call

The function has the following signature:

```
dsrInitRcT dsrInit(SAP_CHAR                  *applicationName,
                   SAP_BOOL                   statOn,
                   SAP_BOOL                   userBlanking,
                   dsrApplicationStaticDataT *statData,
                   SAP_CHAR                  *customTracePath);
```

Its parameters are:

- **applicationName**: Character array of maximum length 32, zero terminated. If a longer string is specified the call will fail. Name of the calling component.
- **statOn**: Boolean type. If set to true, a dsrWrite() call will produce a statistic record otherwise dsrWrite() simply returns. Can be switched on and off dynamically via dsStatOn() and dsrStatOff() after successful initialization.
- **userBlanking**: Boolean type. If set to true, the userId attribute of the main record is blanked. Can be switched on and off dynamically via dsrLogUser() after successfull initialization.
- **statData**: Structured data type. Contains administrative information on the registering component. See section ‘Component Static Data Record’.
- **customTracePath**: Character array of length 256, zero terminated. If a component wants to specify a custom path for its trace files and not use the DSR default one, this can be done here. Set to NULL, if no custom trace Path is required. The entry should match the one of the first array element of attribute ‘tracePaths’ in the CSDR record.

dsrInit() returns type:

```
typedef enum _dsrInitRcT
{
  dsrInitRc_Success   = 0,
  dsrInitRc_NoMemory,
  dsrInitRc_NoMoreApps,
  dsrInitRc_NoSemaphore,
```

```
  dsrInitRc_DuplicateInitialization,
  dsrInitRc_NoDSRPath,
  dsrInitRc_DirectoryError,
  dsrInitRc_Error
}
dsrInitRcT;
```

dsrExit() call

The dsrExit() call is defined as follows:

```
void dsrExit(void);
```

When dsrExit() is called the following actions are performed:

- The shared memory buffers associated with this component are flushed and cleared for reuse if no further instance of the component is active.
- The associated CSDR is cleared if no further instance of the component is active.
- Further component administrative data is cleared.

NOTE: It is very important for a component not to forget to deregister when going down. Since most of the component data is kept in a shared memory segment, component data is not automatically cleared when the associated process dies. The DSRlib provides means to identify such cases of a forgotten or not possible deregistration. But an identification and a cleanup is not always possible. Garbage in the DSRlib's shared memory segment may lead to denial of initialization requests without an obvious reason. It is recommended to include a dsrExit() call into signal handler routines leading to controlled program termination.

Statistic functions

With this group of functions statistic data can be written and the way of writing can be influenced.

dsrWrite() call

This call is used to write an atomic set. The signature of the dsrWrite() function is defined as follows:

```
void dsrWrite(dsrRecT *statRec);
```

For a definition of the type ‘dsrRecT’ see section ‘DSR structured data types’.

When using this function all static data contained in an atomic set has to be collected. The records part of the atomic set have to be correctly chained and the chain NULL terminated (see section ‘DSR structured data types - Overview’).

NOTE: dsrWrite() simply returns if the statistic is turned off. See functions calls ‘dsrStatOff()’, ‘dsrStatOn()’ and ‘dsrInit()’.

dsrLogUser() call

The signature of function dsrLogUser() is defined as follows:

```
void dsrLogUser(SAP_BOOL logUser);
```

Switches dynamically on and off the logging of a user Id in a main record. Can only be used after successful registration via dsrInit().

NOTE: Not supported at the moment. A component has to specify at init time whether to log the user Id or not and cannot change this setting afterward.

dsrStatOn() call

The signature of function dsrStatOn() is defined as follows:

```
void dsrStatOn(void);
```

Enables the statistic for a component. Can only be used after a successful init call. If statistic is enabled a dsrWrite() call produces a written statistic record. If statistic is not enabled dsrWrite() simply returns. See dsrStatOff() call.

NOTE: You have to issue this call if you want to write statistic data and

- you have initially turned off statistics in the dsrInit() call.
- you have issued dsrStatOff() before.

dsrStatOff() call

The signature of function dsrStatOff() is defined as follows:

```
void dsrStatOff(void);
```

Disables the statistic for a component. Can only be used after a successful init call. If statistic is enabled a dsrWrite() call produces a written statistic record. If statistic is not enabled dsrWrite() simply returns. See dsrStatOn() call.

dsrInitMainRec() call

The signature of function dsrInitMainRec() is defined as follows:

```
void dsrInitMainRec(dsrRecUnionT *rec);
```

This function can be used to clear a main record variable for reuse in your coding.

NOTE: The argument type is not the actual main record 'dsrRecMainT' but the wrapping type 'dsrRecUnionT'.

dsrInitCallRec() call

The signature of function dsrInitCallRec() is defined as follows:

```
void dsrInitCallRec(dsrRecUnionT *rec);
```

This function can be used to clear a call subrecord variable for reuse in your coding.

NOTE: The argument type is not the actual call subrecord 'dsrRecCallT' but the wrapping type 'dsrRecUnionT'.

InitCertRec() call

The signature of function dsrInitCertRec() is defined as follows:

```
void dsrInitCertRec(dsrRecUnionT *rec);
```

This function can be used to clear a cert subrecord variable for reuse in your coding.

NOTE: The argument type is not the actual cert subrecord ‘dsrRecCertT’ but the wrapping type ‘dsrRecUnionT’.

Trace functions and macros

In addition to its main function of writing statistic data, the DSRlib offers also the possibility of writing trace files.

The trace functionality can only be used after successful initialization of the DSRlib. This is likely to change in following releases.

Trace files are normally located in a components subdirectory. They are created on a hourly base and follow the same naming scheme as the statistic files do except of a ‘.trc’ suffix:

<component name>_<dd><mm><yyyy><hh>.trc

Trace files are human readable. Here is an example of one trace entry:

20011214 08:00:28:980 Module-Name(8): test.exe Module-Type(9): test-type PID: 0000001740 TID: 0000001740 Message(23): Wer das liest ist doof. Level: 0000000001 Type: 0000000032 Line#: 000000 Duration: 004711 Ext-Access(9): extAccess GUID: 4F4A9C3A09C16643A6A6547282B7B79C

The DSRlib also offers a read interface for trace files with several filter functionality. Please refer to the ‘API Guide for the reader’.

NOTE: A component is expected to set its trace level and trace its actions accordingly to the trace flag it receives via the passport. See section ‘dsrSetTraceLevel()’.

dsrSetTraceLevel() call

The signature of function dsrSetTraceLevel() is defined as follows:

```
void dsrSetTraceLevel(dsrTraceT traceLevel);
```

This function sets the trace level, which is valid for the whole process context. The following levels are provided by the DSRlib:

```
#define dsr_TRACE_OFF                    0x0000
#define dsr_TRACE_LOW                    0x0001
#define dsr_TRACE_MIDDLE                 0x000F
#define dsr_TRACE_HIGH                   0x00FF
```

The following trace flag values may be transported via the passport:

```
#define dsr_TRACE_TYPE_SQL                  0x0001
#define dsr_TRACE_TYPE_BUFFER               0x0002
#define dsr_TRACE_TYPE_ENQEUE               0x0004
#define dsr_TRACE_TYPE_RFC                  0x0008
#define dsr_TRACE_TYPE_PERMISSION           0x0010
#define dsr_TRACE_TYPE_FREE                 0x0020
#define dsr_TRACE_TYPE_C_FUNCTION           0x0040
#define dsr_TRACE_TYPE_ABAP_CONDENS_0       0x0100
#define dsr_TRACE_TYPE_ABAP_CONDENS_1       0x0200
```

It is suggested that a component sets the trace flag for its trace via dsrSetTraceLevel() to

- dsr_TRACE_OFF if the transported trace flag is zero,
- dsr_TRACE_LOW if the transported trace flag equals dsr_TRACE_TYPE_SQL,
- dsr_TRACE_MIDDLE if the transported trace type equals dsr_TRACE_TYPE_BUFFER or dsr_TRACE_TYPE_ENQEUE or dsr_TRACE_TYPE_RFC or any combination of them and if no other trace type is set.
- dsr_TRACE_HIGH if no one of the upper cases applies.

dsrTrcLow(), dsrTrcMid() and dsrTrcHi() macros

A component is supposed to use macros dsrTrcLow(), dsrTrcMid() and dsrTrcHi() in order to write traces. The underlying API functions are not meant to be directly called for performance reasons.

The mentioned macros first check a global trace flag (set via dsrSetTraceLevel()) before they call into the trace writing functions. If the trace level doesn't match no function call is performed.

The trace is invoked if the desired trace level indicated by the associated macro name is less or equal to the one indicated by the global trace level (set via dsrSetTraceLevel()).

NOTE: As mentioned before the trace level set by a call to dsrSetTraceLevel() is a global one in every execution context of the calling process.

It is suggested that components pprovide the following trace information:

- **dsrTrcLow():** Should contain information on the control flow on component level.
- **dsrTrcMid():** Should contain information on the control flow on component function level.

- **dsrTrcHi():** Should contain as much information as possible like invocation parameters on function level.

Each of the trace macros is intended to be invoked in the following fashion using dsrTrcLow() as an example:

```
dsrTrcLow((dsrTraceT           traceType,
           SAP_INT             pid,
           SAP_INT             tid,
           dsrModuleNameT      moduleName,
           SAP_INT             lineNumber,
           dsrModuleTypeT      moduleType,
           SAP_INT             version,
           dsrExternalAccessT  extAccess,
           dsrTransIdT         transID,
           SAP_INT             duration,
           SAP_CHAR           *messageFormat,
           ...));
```

The parameter list is written down in a function declaration like style.
NOTE: The parameter list enclosed in double brackets because of the ellipse at the end.

The following parameters are expected:

- **traceType**: Integer type. The trace writing component should provide in this parameter the trace type received from the passport.
- **pid**: Integer type. Process Id of the calling process.
- **tid**: Integer type. Thread Id of the calling thread.
- **moduleName**: Character array of length 60. Zero terminated. Name of the calling module.
- **lineNumber**: Integer type. Number of the line containing the calling statement.
- **moduleType**: Character array of length 32, zero terminated. Type of the calling module (e.g. ABAP report), component specific.
- **version**: Integer type. Version of the module.
- **extAccess**: Character array of length 60, zero terminated. If the traced action is related to the access of an external object (e.g. a database table), the name of the object may be traced here.
- **transID**: Transaction Id provided by the passport or by DSR API function dsrGetGuid().
- **duration:** Integer type. Duration in microseconds of the traced action.
- **messageFormat**: Character array, zero terminated. Format of the message to be traced. See format specifications of ANSI C function call printf(). The formatted message string may contain up to 255 characters. The tracing routine will write more than 255 characters, but the trace read routine won't read more than 255 characters to be able to push the message into a fixed format RFC table.
- **...** : Parameters of the message. See parameter 'messageFormat'.

Service functions

This sections introduces API functions which are needed by the generating component to produce or convert data to fill into statistic records.

dsrGetGuid() call

This function creates a new transaction Id or GUID. A transaction Id is unique in time and space (i.e. in respect to the generating machine).

A transaction Id is generated if a component starts a new Logical Unit of Work (LUW). It is transported to subcomponents in the passport.

The signature of function dsrGetGuid() is :

```
dsrOtherRcT dsrGetGuid(dsrTransIdT guid);
```

In parameter guid the transaction Id is returned by dsrGetGuid() in case of success.

Type dsrTransIdT is defined as a character array of length 32, not zero terminated.

dsrGetGuid() returns 0 if successful.

dsrGetTime() call

This function returns an UTC timestamp, which directly can be used in a main record (attributes startTimeSec, startTimeMilliSec, endTimeSec, endTimeMilliSec).

Its signature is:

```
int dsrGetTime(SAP_INT4    *secUTC,
               SAP_INT4    *milliSec);
```

dsrGetTime() returns in parameter secUTC the number of seconds since 1.1.1970 UTC and in parameter milliSec the number of milliseconds since secUTC. In case of success dsrGetTime() returns zero.

dsrGetLibVersion() call

This call returns the version number of the used DSRlib as well as the maximum number of concurrent registered or active processes or components.

Its signature is:

```
void dsrGetLibVersion(int *version,
                      int *maxRegisteredApp);
```

dsrGetNetPassport() call

This function is used by a subcomponent to convert a passport transported via RFC or DIAG from network byte order to a normal representation.

Its signature is:

```
int dsrGetNetPassport(SAP_RAW         *id,
                      SAP_UTF8       **sysid,
                      dsrServiceT     *service,
                      SAP_UTF8       **userid,
                      SAP_UTF8       **action,
                      dsrActionTypeT  *acttype,
                      SAP_UTF8       **presysid,
                      SAP_UTF8       **trans_id,
                      dsrTraceT       *traceFlag);
```

Input:

- **id**: Buffer to passport in network byte order as the subcomponent receives it from RFC or DIAG.

Output:

- **sysid**: UTF-8 character buffer. Should be able to hold at most 32 characters. In this parameter the system Id (if available) of the component starting the associated LUW is returned. May be NULL if no sysid is provided by the LUW starting component. A subcomponent takes this sysid and uses it in its cert subrecord.
- **service**: In this parameter the service type of an action starting a LUW (see attribute ‘service’ in main record) is returned. For subcomponent usage in its cert subrecord.
- **userid**: UTF-8 character buffer. Should be able to hold at most 32 characters. In this parameter the user Id of the user starting the LUW is returned. May be NULL if no userid is provided by the LUW starting component. For subcomponent usage in its cert subrecord.
- **action**: UTF-8 character buffer. Should be able to hold at most 128 characters. In this parameter the name of an action starting a LUW is returned. May be NULL if

no action is provided by the LUW starting component. For subcomponent usage in its cert subrecord.

- **acttype**: In this parameter the type of an action starting the associated LUW is returned. For subcomponent usage in its cert subrecord.
- **presysid**: UTF-8 character buffer. Should be able to hold at most 32 characters. This parameter returns the sysid of the previous component working on the associated LUW. May be NULL if this field is not provided by the previous component. A component should fill its sysid or component name into this field when generating a passport via dsrSetNetPassport().A subcomponent should fill its sysid or component name into this field when forwarding a passport via dsrSetNetPassport().
- **trans_id**: UTF-8 character buffer. Should be able to hold at most 32 characters. In this parameter the transaction Id of the associated LUW is returned. May be NULL if not provided by the calling component. For subcomponent usage in its main record.
- **traceFlag**: In this parameter the trace type associated with this LUW is returned. A component should set its trace level accordingly (see section 'dsrSetTraceLevel() call'). For subcomponent usage in its cert subrecord.

NOTE: All output string buffers have to exist. No buffer will be allocated by dsrGetNetPassport(). So e.g. parameter sysid is the address of a pointer pointing to a UTF-8 string buffer expected to hold the output of a dsrGetNetPassport() call.

dsrGetNetPassport() returns zero in case of success.

NOTE: Data contained in the passport is not dependent on locale or machine settings. Therefore all character array types are returned in UTF-8. The conversion into the local code page has to be done by the component itself.

dsrSetNetPassport() call

A call to dsrSetNetPassport() is done by a component in order to pack passport data into a format, which can be used by RFC or DIAG. dsrSetNetPassport() has the following signature:

```
int dsrSetNetPassport(SAP_RAW        **id,
                      size_t          *len,
                      dsrTraceT        traceFlag,
                      SAP_UTF8        *sysID,
                      dsrServiceT      service,
                      SAP_UTF8        *action,
                      dsrActionTypeT   actType,
                      SAP_UTF8        *userID,
                      SAP_UTF8        *preSysID,
```

```
                        SAP_UTF8        *transID);
```

dsrSetNetPassport() returns zero in case of success.

NOTE: All character string input parameters have to be specified in UTF-8.

NOTE: All character string parameters have to be zero terminated. The maximum length of the arrays refers to the maximum number of characters without the trailing zero.

dsrSetNetPassport() has the following parameters:

Output:

- **id**: Address of pointer pointing to buffer supposed to hold the generated network byte order passport. Buffer has to be allocated by caller. For determination of the required size of the buffer use API call dsrGetNetPassportLen().
- **len**: In this parameter the actual length of the generated passport is returned.

Input:

- **traceFlag**: This parameter should contain the trace type. Use the trace type received from a passport or use a 'or combination' of the following values if you are generating a new passport:

```
#define dsr_TRACE_TYPE_SQL                  0x0001
#define dsr_TRACE_TYPE_BUFFER               0x0002
#define dsr_TRACE_TYPE_ENQEUE               0x0004
#define dsr_TRACE_TYPE_RFC                  0x0008
#define dsr_TRACE_TYPE_PERMISSION           0x0010
#define dsr_TRACE_TYPE_FREE                 0x0020
#define dsr_TRACE_TYPE_C_FUNCTION           0x0040
#define dsr_TRACE_TYPE_ABAP_CONDENS_0       0x0100
#define dsr_TRACE_TYPE_ABAP_CONDENS_1       0x0200
```

- **sysID**: UTF-8 character array of maximum size of 32. This parameter should contain the system identifier of the system originating the associated LUW. A component which is only forwarding a passport should not modify this parameter. A component which starts a LUW should pass its system Id or component name in this parameter.
- **service**: This parameter should contain the service type of an action which builds the start of a LUW. A component which is only forwarding a passport should not modify this parameter. A component which starts a LUW should pass the associated service type in this parameter. See attribute 'service' in description of the main record.
- **action**: UTF-8 character array of maximum length of 40. This parameter should contain a description of the initial action starting a LUW. A component which is only forwarding a passport should not modify this parameter. A component which starts a LUW should pass the associated action description in this parameter. See attribute 'action' in description of the main record.
  NOTE: In the passport the length of the character array attribute action is 40. The length of the action attribute in the main record as well as in the cert subrecord is 128. Be sure to handle over only the first 40 characters of the action attribute otherwise the conversion will fail.

- **actType**: This parameter should contain the action type of an action which builds the start of a LUW. A component which is only forwarding a passport should not modify this parameter. A component which starts a LUW should pass the associated action type in this parameter. See attribute ‘actionType’ in description of the main record.
- **userID**: UTF-8 character array of maximum length of 32. Name of the user associated with a LUW. A component which is only forwarding a passport should not modify this parameter. A component which starts a LUW should pass the associated action type in this parameter.
- **preSysId**: UTF-8 character array of maximum length of 32. Should contain the system identifier of the system previously working on an action of this LUW. Before forwarding the passport a component fills into this field its system identifier or component name.
- **transID**: UTF-8 character array of length 32. Should contain the transaction Id or GUID associated with the LUW. See dsrGetGuid() call. A component starting a LUW should generate a transaction Id with dsrGetGuid(), a component forwarding a passport should not modify this parameter.

dsrGetNetPassportLen() call

The function dsrGetNetPassportLen() returns the length of a network byte order passport.

This length is required to be known when using e.g. dsrSetNetPassport() in order to determine the required size of the output buffer to hold the generated net passport.

The signature of dsrGetNetPassportLen() is

```
size_t dsrGetNetPassportLen(void);
```

Example instrumentation

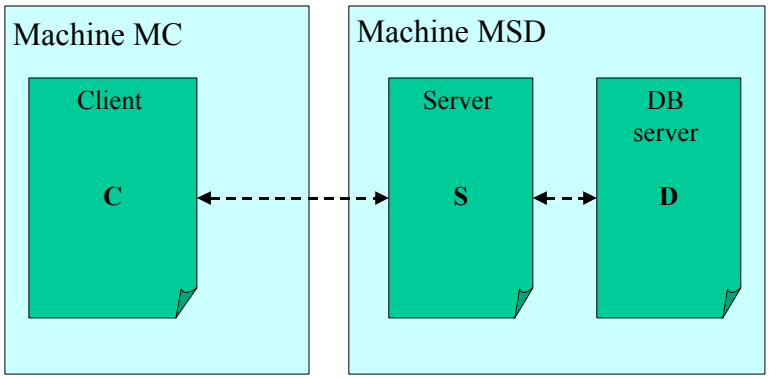
In what follows a simple instrumentation of the DSR library shall be given.

Scenario

Consider a classical client-server scenario: Client C wants to retrieve data e.g. on a certain user and asks server S for it. Server S performs some authentication work, fetches the required information from a database server D and sends it back to client C.

Physical view

Let client C reside on machine MC and server S as well as database server D reside on another one called machine MSD:



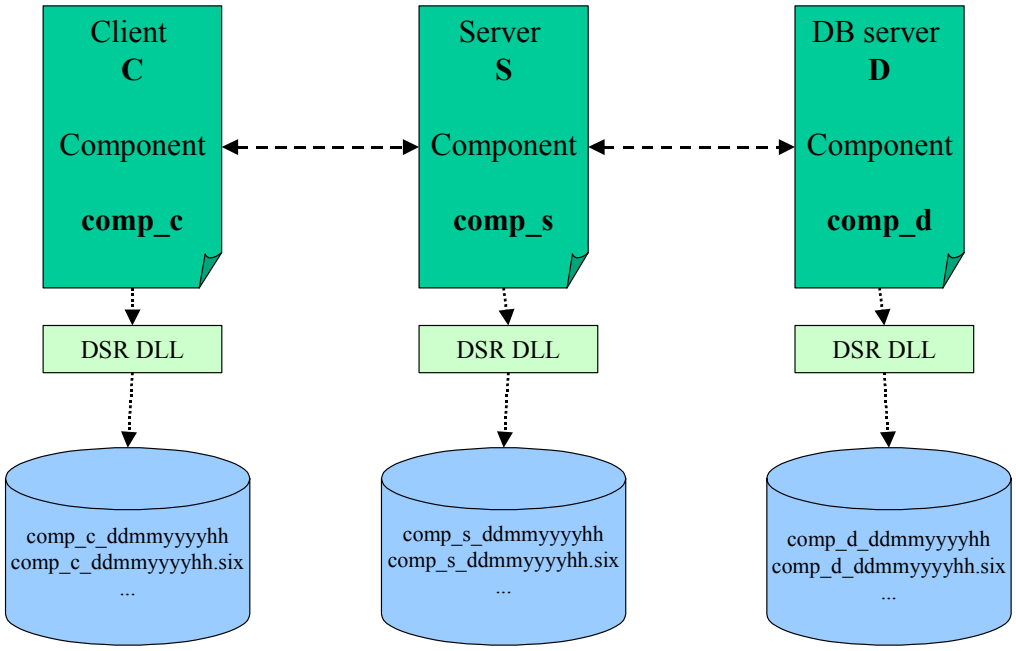
Logical view

From a logical point of view client C, server S and database server D are treated as three different components:

- Client C → Component C. Component name used with DSR lib: ‘comp_c’. Component type used with DSR lib: ‘clnt’.

- Server S → Component S. Component name used with DSR lib: ‘comp_s’. Component type used with DSR lib: ‘srv’.
- Database server D → Component D. Component name used with DSR lib: ‘comp_d’. Component type used with DSR lib: ‘dbsrv’.

For each component a separate set of statistic files is generated:



NOTE: In order to be able to retrieve statistical data, on both machines MC and MSD a CCMS agent (sapccmsr.exe) has to be installed and running:

Statistical data generated

Let’s have a look at the statistical data the components are about to write:

- **Component C**: The client C performs some actions and records its CPU time, the consumed memory and since it performs a call into a subcomponent (server S) it also records the time spent waiting for the returned answer as well as the total time spent for the action. It generates a main record and as it calls into a subcomponent also a call subrecord.

  Note: Client C doesn’t create a cert subrecord.

  Of course client C occupies several other record input fields like start time, end

time, process id, etc. Here is an overview:

Main record:

Field name	set by DSRlib	set by client C	not used by client C
recVersion	•		
startTimeSec		•	
startTimeMilliSec		•	
endTimeSec	•		
endTimeMilliSec	•		
startDate		•	
startTime		•	
endDate	•		
endTime	•		
transId		•	
service		•	
action		•	
actionType		•	
userId		•	
processId		•	
threadId		•	
cpuTime		•	
queueTime			•
loadTime			•
genTime			•
netTime			•
waitTime		•	
maxMem		•	
respTime		•	
luwInfo		•	
addInfo			•

Call subrecord:

Field name	set by DSRlib	set by client C	not set by client C
recVersion	•		
componentType		•	
componentName		•	
destinantion		•	
callTimestamp		•	

receiveTimestamp			•
sentBytes		•	
receivedBytes		•	
numberOfCalls		•	

- **Component S**: Server S writes a main record and since it performs a call to a database server D it also writes a call subrecord. Server S got called from client C and therefore acts as a subcomponent to client C, it also writes a cert subrecord. Lets say that server S fills the same main record fields as client C does, and additionally field 'loadTime', since a user table may be loaded in order to authenticate the client request:

  Main record:

Field name	**set by DSRlib**	**set by server S**	**not used by server S**
recVersion	•		
startTimeSec		•	
startTimeMilliSec		•	
endTimeSec	•		
endTimeMilliSec	•		
startDate		•	
startTime		•	
endDate	•		
endTime	•		
transId		•	
service		•	
action		•	
actionType		•	
userId		•	
processId		•	
threadId		•	
cpuTime		•	
queueTime			•
loadTime		•	
genTime			•
netTime			•
waitTime		•	
maxMem		•	
respTime		•	
luwInfo		•	

addInfo			•

Call subrecord:

Field name	set by DSRlib	set by server S	not set by server S
recVersion	•		
componentType		•	
componentName		•	
destinantion		•	
callTimestamp		•	
receiveTimestamp			•
sentBytes		•	
receivedBytes		•	
numberOfCalls		•	
addInfo			•

Cert subrecord:

All fields of the cert subrecord are set by server S, as it receives it from the passport.

NOTE: Before server S handles the passport to the called DB server D, it changes the field 'preSysId' to its system Id.

- **Component D**: The DB server D doesn't call any subcomponents, therefore it doesn't write a call subrecord. It writes a main record and a cert record. It occupies the same fields of the main record as the server S does, except an additional field 'queueTime':

Main record:

Field name	set by DSRlib	set by DB server D	not used by DB server D
recVersion	•		
startTimeSec		•	
startTimeMilliSec		•	
endTimeSec	•		
endTimeMilliSec	•		
startDate		•	

startTime		•	
endDate	•		
endTime	•		
transId		•	
service		•	
action		•	
actionType		•	
userId		•	
processId		•	
threadId		•	
cpuTime		•	
queueTime		•	
loadTime		•	
genTime			•
netTime			•
waitTime		•	
maxMem		•	
respTime		•	
luwInfo		•	
addInfo			•

Cert record:

All fields of the cert subrecord are set by DB server D, as it receives it from the passport.

Component static data

When the three components register itself at the DSRlib they have to specify the kind of service and action types they know and are capable to report (for an explanation of the terms ‘service’ or ‘service type’ and ‘action type’ see sections ‘Data types – main record‘ and ‘Data types – certification subrecord’).

Here is what client C, server S and DB server D register as service and action types:

Client C:

Service Types:

- Query of administrative data (referred to in this example). Numeric value: 1.

- Setting administrative data. Numeric value: 2.

Action Types:

- Query of **user** administrative data (referred to in this example). Numeric value: 1. May be performed within service #1.
- Query of **server** administrative data. Numeric value: 2. May be performed within service #1.
- Setting of **user** administrative data. Numeric value 3. May be performed within service #2.

Server S:

Service Types:

- Data push from server. Numeric value: 1. Let's say our server is also able to actively push data to a client. Service type 1 describes this action.
- Data request from client. Numeric value: 2. This service type describes the 'classical' work of a server, data is sent on demand.

Action Types:

- Data push type A. Numeric value: 1. May be performed within service #1.
- Data push type B. Numeric value: 2. May be performed within service # 1.
- Data request type A. Numeric value 3. May be performed within service #2.
- Data request type B. Query for user data (this request is treated in our example here). Numeric value: 4. May be performed within service #2.

DB server D:

Service Types:

- DB data lookup. Numeric value: 1.
- DB administration query. Numeric value: 2.

Action Types:

- DB table lookup. Numeric value: 1. May be performed within service #1.
- DB table meta data lockup. Numeric value: 2. May be performed within service #1.
- DB table reorg. Numeric value: 3. May be performed within service #2.
- DB table create. Numeric value: 4. May be performed within service #2.
- DB table delete. Numeric value: 5. May be performed within service #2.

If a component knows that it is going to delegate work to some subcomponents, it may register those subcomponents at DSRlib initialization time:

- Client C registers server S as a subcomponent:
- Server S registers DB server D and client C as a subcomponent (remember that server S is also able to push data to a client, therefore it registers client C as a subcomponent).
- DB server D does not register a subcomponent.

If a component specifies a custom directory for its DSR trace files (see parameter 'customTracePath' in call dsrInit()) this information may also be registered at DSR initialization time. In our example only client C decides to use this feature and specifies the following custom trace path:

c:\ramba\zamba\

Since no one of the components uses a proprietary tracing mechanism, no data concerning trace file names and trace file patterns is reported at DSR initialization time from the components.

Passport handling

In our example client C is the initiator of a LUW. Therefore client C creates a passport and handles it to server S. Server S modifies this passport (it updates the 'preSysId' attribute) and handles it over to its subcomponent DB server D.

Server S and DB server D are both writing a cert subrecord containing the passport information. Client C doesn't write a cert subrecord.

Coding excerpts

In what follows some coding excerpts are presented for typical actions a component has to take when instrumenting the DSRlib.

Filling component static data record

Filling the static data record is often done incorrectly e.g. incorrect termination of string or action/service type arrays. Here is an example how our components fill this record:

Client C:

```
...
#define my_COMPONENT_NAME     "client_c"
#define my_COMPONENT_TYPE     "clnt"
#define my_COMPONENT_VERSION  "1.0"
```

```
...

dsrApplicationStaticDataT      staticData;
int                            i;


...


/* init record; do this in any case because:
   -> string arrays will be automatically correctly terminated.
   -> static data record will be automatically correctly terminated
      (attribute: next).
*/
memset((void*) &staticData,0,sizeof(dsrApplicationStaticDataT));

/* write component static data.
*/

/* write component name, etc.
*/
strcpy(staticData.componentName,my_COMPONENT_NAME);
strcpy(staticData.componentType,my_COMPONENT_TYPE);
strcpy(staticData.className,my_COMPONENT_TYPE);
strcpy(staticData.componentVersion,my_COMPONENT_VERSION);

/* write service/action type meta info; 2 service types, 3 action types.
*/
for (i=0;i<2;i++)
{
  staticData.serviceOrActionTypeType[i] = dsrIsSrvOrAction_Service;
}
for (i=2;i<5;i++)
{
  staticData.serviceOrActionTypeType[i] = dsrIsSrvOrAction_Action;
}

/* terminate service/action type array.
*/
staticData.serviceOrActionTypeType[i] = dsrIsSrvOrAction_Unknown;


/* write values of action and service types
*/
staticData.serviceOrActionType[0] = 1;
staticData.serviceOrActionType[1] = 2;
staticData.serviceOrActionType[2] = 1;
staticData.serviceOrActionType[3] = 2;
staticData.serviceOrActionType[4] = 3;

/* write additional short info for known service and action types.
*/
strcpy(staticData.serviceOrActionTypeShortText[0],"Query adm data");
strcpy(staticData.serviceOrActionTypeShortText[1],"Set adm data");
strcpy(staticData.serviceOrActionTypeShortText[2],"Query usr adm data");
strcpy(staticData.serviceOrActionTypeShortText[3],"Query srv adm data");
strcpy(staticData.serviceOrActionTypeShortText[4],"Set usr adm data");


/* write additional long info for known service and action types.
*/
strcpy(staticData.serviceOrActionTypeLongText[0],
       "Query of administrative data from server.");
strcpy(staticData.serviceOrActionTypeLongText[1],
       "Submitting of administrative data to server.");
strcpy(staticData.serviceOrActionTypeLongText[2],
       "Query of user administartive data from server.");
strcpy(staticData.serviceOrActionTypeLongText[3],
       "Query of server administrative data from server.");
strcpy(staticData.serviceOrActionTypeLongText[4],
       "Submitting of user adinistrative data to server.");
```

```
/* write linked systems.
*/
strcpy(staticData.linkedSystems[0],"server_s");


/* write trace path.
*/
strcpy(staticData.tracePaths[0],"c:\\ramba\\zamba\\");

/* no trace file name information (DSR lib is handling trace file naming
   scheme).
*/

/* no trace file pattern information (DSR trace file format used).
*/

...
```

Main record and subrecord handling

Chaining an atomic set together

Client C:

```
...

dsrRecT                         r[2];
dsrRecUnionT                    ru[2];

...


/* chain records: main - call - 0
*/
r[0].recordType = dsrRecType_Main;
r[0].record     = &ru[0];
r[0].next       = &r[1];
r[1].recordType = dsrRecType_Call;
r[1].record     = &ru[1];
r[1].next       = NULL;

...
```

Server S:

```
...

dsrRecT                         r[3];
dsrRecUnionT                    ru[3];

...
```

```
/* chain records: main - cert - call - 0
*/
r[0].recordType = dsrRecType_Main;
r[0].record     = &ru[0];
r[0].next       = &r[1];
r[1].recordType = dsrRecType_Cert;
r[1].record     = &ru[1];
r[1].next       = &r[2];
r[2].recordType = dsrRecType_Call;
r[2].record     = &ru[2];
r[2].next       = NULL;

...
```

DB server D:

```
...

dsrRecT                         r[2];
dsrRecUnionT                    ru[2];

...


/* chain records: main - cert - 0
*/
r[0].recordType = dsrRecType_Main;
r[0].record     = &ru[0];
r[0].next       = &r[1];
r[1].recordType = dsrRecType_Cert;
r[1].record     = &ru[1];
r[1].next       = NULL;

...
```

Filling statistic records

Server S:

```
...

#define my_min(a,b) (((a)<(b))?(a):(b))
#define MY_STR_CP(a,b) strncpy(a,b,my_min(strlen(b),sizeof(a)))

...

dsrRecT                         r[3];
dsrRecUnionT                    ru[3];
struct _timeb                   t;
struct tm                      *tUTC;


...

dsrInitMainRec(&ru[0]);
dsrInitCertRec(&ru[1]);
dsrInitCallRec(&ru[2]);
```

```
...

/* fill main record.
*/

/* fill in start time
*/
_ftime(&t);
tUTC = gmtime(&(t.time));
sprintf(b,
        "%4.4i%2.2i%2.2i",
        tUTC->tm_year+1900,
        tUTC->tm_mon+1,
        tUTC->tm_mday);
SETDATE(r[0].record->mainRecord.startDate,b);
sprintf(b,
        "%2.2i%2.2i%2.2i",
        tUTC->tm_hour,
        tUTC->tm_min,
        tUTC->tm_sec);
SETTIME(r[0].record->mainRecord.startTime,b);
(void) dsrGetTime(&(r[0].record->mainRecord.startTimeSec),
                  &(r[0].record->mainRecord.startTimeMilliSec));

/* fill in user id
*/
MY_STR_CP(r[0].record->mainRecord.userId,getUserId());

/* fill in action
*/
MY_STR_CP(r[0].record->mainRecord.action,"Type B request");

/* fill in trans Id
*/
memcpy((void*) r[0].record->mainRecord.transId,
       (const void*) getTransIdFromPassport(),
       sizeof(dsrTransIdT));

/* fill in action type. see section:
   'example instrumentation - component static data'
*/
r[0].record->mainRecord.actionType = 4;

/* fill in service type. see section:
   'example instrumentation - component static data'
*/
r[0].record->mainRecord.service = 2;

/* fill in measurement data
*/
STCM15(r[0].record->mainRecord.cpuTime,getCPUTime());
STCM15(r[0].record->mainRecord.loadTime,getLoadTime());
STCM15(r[0].record->mainRecord.maxMem,getMaxMem());
STCM15(r[0].record->mainRecord.respTime,getRespTime());
STCM15(r[0].record->mainRecord.waitTime,getWaitTime());

/* fill in pid and tid
*/
STCM15(r[0].record->mainRecord.processId,getpid());
STCM15(r[0].record->mainRecord.threadId,gettid());

/* fill in luw info; no special luw type;
*/
r[0].record->mainRecord.luwInfo = 0;



/* fill cert subrecord.
*/

/* fill in user id.
```

```
*/
MY_STR_CP(r[1].record->certRecord.userID,getPassportUserId());

/* fill in action.
*/
MY_STR_CP(r[1].record->certRecord.action,getPassportAction());
/* fill in action type.
*/
r[1].record->certRecord.actType = getPassportActionType();

/* fill in trace flag.
*/
r[1].record->certRecord.traceFlag = getPassportTraceFlag();

/* fill in service type
*/
r[1].record->certRecord.service = getPassportService();

/* fill in preSysId.
*/
MY_STR_CP(r[1].record->certRecord.preSysID,getPassportPreSysId());

/* fill in sys id.
*/
MY_STR_CP r[1].record->certRecord.sysID,getPassportSysId());



/* fill call subrecord.
*/

/* fill in call time.
*/
STCM15(r[2].record->callRecord.callTimestamp,getCallTime());

/* fill in number of calls.
*/
r[2].record->callRecord.numberOfCalls = getNumberOfCalls();

/* fill in received bytes.
*/
STCM15(r[2].record->callRecord.receivedBytes,getNumberReceivedBytes());

/* fill in number of sent bytes.
*/
STCM15(r[2].record->callRecord.sentBytes,getNumberSentBytes());

/* fill in destinantion of subcomponent.
*/
MY_STR_CP(r[2].record->callRecord.destination,getIpOfDbServer());

/* fill in component name of subcomponent.
*/
MY_STR_CP(r[2].record->callRecord.componentName,"comp_d");

/* fill in component type of subcomponent.
*/
MY_STR_CP(r[2].record->callRecord.componentType,"dbsrv");
```

Sending an atomic set

```
...
```

```
dsrRecT                         r[3];
dsrRecUnionT                    ru[3];

...

dsrWrite(&r[0]);

...
```

Reusing a record

Server S:

```
...

dsrRecT                         r[3];
dsrRecUnionT                    ru[3];

...

/* write next atomic set of main - cert - call - 0. keep chaining.
*/

/* erase content of records.
*/
dsrInitMainRec(&ru[0]);
dsrInitCertRec(&ru[1]);
dsrInitCallRec(&ru[2]);

...
```

Contact

Please contact ...

- **Daniela Klausner** for questions concerning the semantics of statistical records.
- **Andreas Vogel** for questions concerning transaction ST03G (StatTrace), which is used for graphical presentation and administration of statistical data.
- **Markus Eichelsdörfer** for questions concerning the instrumentation of DSR lib.