OSF DCE SIG **M. Hubbard (IBM)**
Request For Comments: 12.0 **August 1992**

DCE SERVICEABILITY STRATEGY/DESIGN DISCUSSION PAPER

1. INTRODUCTION

Serviceability refers to those functional enhancements that improve the diagnostic features of software. These enhancements involve (1) message logging, (2) tracing and (3) error logging.

This paper stems from the Serviceability Requirements document issued within the DCE SIG by the Serviceability workgroup (see [RFC 11.0]). It is recommended that the reader be familiar with the requirements document before reviewing this paper.

This document is not an official IBM proposal to OSF. It represents the ideas of the author and is only an informal starting point intended to spur discussion on how to implement serviceability within DCE. The proposals/descriptions are not necessarily complete or final. Comments, questions and disagreements are welcome.

1.1. Proposed Minimum V1.1 Content

(a) General

- (i) Replace `printf()` calls with serviceability interfaces (‘‘hooks’’) proposed within this document to facilitate better integration with the base platform.
- (ii) Establish guidelines for the placement of these serviceability hooks.

(b) Messaging

- (i) Move all message format strings into message catalogs.
- (ii) Define a common format for all messages.

(c) Error Logging

- (i) Define error scenarios where special error logging (beyond just logging a message) is required.
- (ii) Define a common format for all local error log records.
- (iii) Enable remote notification of errors via the DME event logger.
- (iv) Define contents of remote notification events.

(d) Tracing

- (i) Enable the remote activation/deactivation of tracing.
- (ii) Define a set of trace types.
- (iii) Define trace points within each DCE component.
- (iv) Define a common format for all trace records.

(e) Documentation

- (i) Create a Problem Determination Guide that at a minimum explains all messages, trace options and remote error notification events.

1.2. Terminology

(a) *log entry*

A data structure containing a header of identifying information plus several bytes of defined data (e.g., error entry, trace entry) which is recorded to disk, screen or memory.

(b) *log point*

The location at which a (group of) code statement(s) that generate an entry is placed within a software program (e.g., trace point, error point).

(c) *hook*

An interface called to record an entry at a given point in a software program.

(d) *hook code*

The code that implements the hook.

(e) *ID*

A unique combination of bytes that identifies the component and error/trace logging point within the software program.

2. MESSAGE/ERROR LOGGING

2.1. Purpose

Message and error logging serve a common purpose which is to notify the end user or system administrator about software program activity.

Messages are logged to give immediate local notification of an error or to provide informational feedback. In the case of an error situation, they concisely describe the error that occurred. The administrator/user will then refer to a Problem Determination Guide for a list of causes for the error and the suggested action to be taken. For usability reasons, messages are ''localized'' (it is the responsibility of each DCE system vendor to translate messages).

The main purpose of error logging is to locally log control information when an error occurs. This information supplements any error message and is aimed at reducing/avoiding the need for any diagnostic investigation requiring tracing (recreating serious errors should be avoided and tracing can impact system performance).

A secondary element of message/error logging is remote notification. Remote notification is intended to allow:

(a) centralized problem management

(b) unattended operation of DCE server nodes

2.2. Activation/Deactivation

(a) Control over the message/error log and the filtering/routing of messages/errors is controlled by the local operating system (and any end user interface it offers). A generalized server that exports an RPCable serviceability control interface (which DME may define) for remotely activating local commands that control serviceability related features could be created.

(b) Logging within client/server stubs and runtimes require special consideration:

   (i) Multi-user systems will need to distinguish between executables running as production system services and executables running as end user applications (or system applications under development) in order to avoid centralized system logs from being overrun with user error activity (vs production services activity). The logging service would need to distinguish between these two cases (and route to the appropriate private/central log).

   (ii) User applications may have to explicitly initialize/terminate the various system logging facilities unless this can be made part of RPC (and DUA) initialization/termination.

   (iii) Hooks within stubs will likely be manually inserted unless the IDL compiler can be updated.

(c) A way to set the minimum severity level for message/error logging is required.

(d) Catalogs should automatically be opened within the message logging hook code.

(e) Catalogs may exist for each component or for each executable.

(f) The error logging hook must handle the case where the local error log is filled. One possible solution is to maintain two files where logging can alternate (i.e., when one file is filled, logs entries can be written to the secondary file while the first can be archived and cleared becoming the new secondary file).

(g) A serviceability profile file, a set of environment variables or new/extended runtime verbs (RPC and DUA) is required to identify the local log file(s) and to set the minimum recording severity.

(h) Local error logging can be considered optional (leaving just message logging and remote error notification). However, this decision depends on the implementation of the hook, not the code calling it.

2.3. General Message/Error Logging Guidelines

(a) Each DCE executable should always log the following general informational messages:

  (i) process has successfully started (i.e., ready for requests)

  (ii) tracing is active (lists all trace flags set)

  (iii) process has normally/abnormally ended

  (iv) ...(TBD)

(b) Other informational messages specific to a server/component that would be of interest to an administrator should also be logged:

  (i) deletion of directory entries

  (ii) login requests

  (iii) successful completion of replication activity

  (iv) ...(TBD)

(c) Two levels of informational severities should exist to allow suppression of detailed informational messages but still allowing key general informational messages to be seen.

(d) A message must be logged whenever an error occurs.

(e) Errors must be logged whenever an error situation results that is not part of normal processing (e.g., cds returning an ‘‘entry not found’’ error is considered normal whereas an ‘‘out of memory’’ error is considered abnormal). Places where unexpected errors are detected include:

  (i) system/library calls returning negative return codes (or `NULL` pointers)

  (ii) DCE client API calls returning negative status codes

  (iii) signal handlers

  (iv) thread recovery environments (established via `pthread_cleanup_pop()` and `pthread_cleanup_push()`)

  (v) process recovery environments (established via `atexit()`)

  (vi) code segment recovery environments established using `CATCH`/`CATCHALL` exception handling clauses

  (vii) points where non-zero RPC status and comm failures are received by the runtime

Note: It is assumed that the establishing of recovery environments will increase as part of the V1.1 code cleanup.

(f) Errors must be logged whenever manual (immediate or eventual) intervention is required.

(g) Errors must be logged whenever a server or server function becomes unavailable.

(h) Multiple error log entries may result from a single error condition.

(i) Error log entries should be filled with keywords and data information rather than conversational language descriptions (a stand-alone formatter can be used to generate easy-to-read reports).

(j) No message text should be sent remotely to the event logger. The logger should be able to reconstruct a text message based on the hex code points it receives.

(k) The actual recording of error data to memory/disk may be done inline or by issuing requests to a separate thread/daemon depending on the particular operating system. This decision depends on the implementation of the error logging hook code.

2.4. Logging Interfaces

The proposed message/error logging interfaces are the following:

```
int logmsg(
    nl_catd     cd,        /* catalog descriptor (optional)    */
    unsigned32 id,        /* message id                       */
    unsigned32 severity, /* message severity                 */
    uuid_t     *e_uuid,  /* error correlator                 */
    unsigned32 line,     /* line number where error occurred */
    char       *file,    /* source file where error occurred */
    ...                  /* message substitution variables   */
);

int logerr(
    unsigned32 id,        /* error id                      */
    unsigned32 severity, /* error severity                */
    unsigned32 e_info,   /* error codepoints              */
    unsigned32 e_status, /* errno/status/exception/signal */
    uuid_t     *e_uuid,  /* error correlator              */
    unsigned32 e_len,    /* length of error data          */
    char       *e_data   /* hex error data stream         */
);
```

More discussion and design effort may result in these 2 interfaces being combined:

```
int lognotice(
    nl_catd     cd,        /* catalog descriptor (optional)    */
    unsigned32 id,        /* message/error id                 */
    unsigned32 severity, /* message severity                 */
    unsigned32 e_info,   /* error codepoints                 */
    unsigned32 e_status, /* errno/status/exception/signal    */
    uuid_t     *e_uuid,  /* error correlator                 */
    unsigned32 line,     /* line number where error occurred */
    char       *file,    /* source file where error occurred */
    ...                  /* message substitution variables   */
                         /*   or error log hex data          */
);
```

New RPC interfaces should be defined to allow passing of error uuids between RPC and user code portions of executor threads. For example:

```
rpc_get_err_uuid(
    uuid_t     *e_uuid, /* creates a uuid if one       */
    unsigned32 *status  /*   doesn't already exist     */
);                      /*   else returns existing one */

rpc_test_err_uuid(
    uuid_t     *e_uuid, /* tests if a uuid has been */
    unsigned32 *status  /*   created and returns it */
);
```

2.5. Format

(a) Message/error identifiers (passed into the hooks) need to be architected to allow for vendor specific errors (as part of DCE extensions or operating system differences). These identifiers are used for remote notification as well as catalog access.

```
---------------------------------------------------------------
Bits   Usage
---------------------------------------------------------------
00-04: DCE executable identifier
05-08: DCE subcomponent identifier
09-16: Vendor identifier (OSF is the vendor of base DCE code)
17-32: Message/error number
---------------------------------------------------------------
```

(b) Informational messages need only include:

(i) message identifier

(ii) message text (retrieved from a catalog) and user substitution variables

(c) Error messages must include:

(i) message identifier

(ii) message text (retrieved from a catalog) and user substitution variables

(iii) error UUID (to correlate with other messages and local/remote error logs)

(iv) where the error occurred (`__FILE__`, `__LINE__`)

(d) Messages may be prefaced by various system information depending on the logging implementation:

(i) date and time (locale specific)

(ii) process id

(iii) severity indicator

(iv) vendor specific local message identifiers

(e) Messages may require multiple lines.

(f) Catalogs should store I18N enabled format strings (see the XPG3 guidelines regarding ordered substitution in format strings). Following is an example message format ('\' at the end of a line indicates line-continuation):

```
----------------------------------------------------------------
02/21/92  9:29:45 [1948382]  rpcd:     RPC0001I: \
RPC daemon successfully started.
02/21/92 10:29:31 [1948382]  cdsd:     RPC0034E: \
Error receiving IP packet, uuid=<8888...>.
02/21/92 10:29:45 [1948382]  cdsd:     RPC0034I: \
uuid=<8888...>,line=32,file=<dce/rpc/runtime/comnlsn.c>
02/21/92 10:45:30 [1948383]  cdsclerk: CDS0002E: \
No more heap memory available, uuid=<9999...>.
02/21/92 10:45:42 [1948382]  cdsclerk: CDS0002I: \
uuid=<9999...>,line=99,file=<dce/directory/cds/server/db_xyz.c>
----------------------------------------------------------------
```

Note: RPC0001I, RPC0034E, CDS0002E, ..., are vendor specific prefixes that represent the message id (I=informational, E=error). They are part of the message text in the catalog.

(g) Error log entries must include:

   (i) error identifier (defines the error that occurred)

   (ii) severity

   (iii) error UUID (to correlate local and remote error logs)

   (iv) an architected hex value indicating the error type

   (v) an architected hex value indicating the likely cause

   (vi) an architected hex value indicating the suggested action

   (vii) the errno/status/exception value that led to the error

(h) Suggested architected error code points (type, cause and action):

```
----------------------------------------------------------------
Value                Code Point  Description
----------------------------------------------------------------
  [Error Types:]
S_NO_ERR             x'00000000' No error occured.
S_SYSTEM_ERR         x'01000000' Operating system failure.
S_DEVICE_ERR         x'02000000' Device failure.
S_NETWORK_ERR        x'03000000' Failure during communications.
S_DATA_ERR           x'04000000' Failure during data processing.
...
  [Causes:]
S_UNKNOWN            x'00010000' Cause is unknown.
S_CONFIG_ERR         x'00020000' Improper network configuration.
...
  [Suggested Actions:]
S_NO_ACTION          x'00000000' No action required.
S_ATTEMPT_RETRY      x'00000100' Attempt to retry.
S_CORRECT_AND_RETRY  x'00000200' Correct problem and retry.
S_CONTACT_REP        x'00000300' Contact service representative
                                 and report keywords.
S_CONTACT_ADMIN      x'00000400' Contact system administrator.
...
----------------------------------------------------------------
```

(i) Error UUIDs should be passed back to the client (this may not be possible in V1.1 since it affects DCE protocols, but it is a good recommendation for new servers).

(j) Example error log format:

```
---------------------------------------------------------------
Date     Time     ID    Sev CodePts  Status   UUID             \
Len Data
02/21/92 9:29:45 00003 03  02000100 0000002A 123456782003FA10 \
005 002003FA38
02/21/92 9:29:45 00423 04  02000200 00000042 123456783003FA10 \
004 12345678
---------------------------------------------------------------
```

2.6. Notes

(a) All messages should be reviewed for consistent terminology.

(b) Systems with multiple destinations can route/filter the message based on severity or message number.

(c) It is assumed that the DME event logger has architected event types to support the creation of specialized servers to separately handle error events, security events (e.g., centralized auditing), ...

(d) OSF may choose to define the format of an error report to be used as a guide by vendors creating error log formatters (error logs only contain raw hex data).

(e) No design of the remote ‘‘catcher’’ or ‘‘agent’’ of remote events is provided within this document.

(f) No design of a local serviceability command catcher or its RPCable interface is provided within this document.

3. TRACING

3.1. Purpose

Tracing offers a way to achieve further information to isolate an error or irregular behavior. There are several uses for tracing:

(a) port verification (aimed at DCE system vendors)

(b) application debugging (aimed at application developers)

(c) diagnostic investigation (aimed at service personnel)

(d) performance monitoring (aimed at system administrators)

Each type of tracing has a different main audience (certainly each could be used in conjunction with another) and attempts to record different information (and sometimes, in a different way).

3.2. Activation/Deactivation

(a) Port verification / application debugging

(i) These types of tracing are activated using a compile option in combination with a series of event flags controlled via a serviceability profile file (or optionally via environment variables). For example:

```
#ifdef _DEBUG
    if (tracing_on & debug_flag1) /* optional to allow
                                     for scoped debug
                                     tracing */
        fprintf(dbgfh,
            "cds: about to bind to the server\n");
#endif

#ifdef _DEBUG
    if (tracing_on & api_check)   /* print out all input
                                     parameters */
        fprintf(dbgfh,"cds: %d %x %x %c %s\n");
#endif
```

(ii) The debug/event flags should be settable during runtime by using environment variables or a separate event file loaded during initialization. The code itself can also set these flags.

(iii) The CDS code has created a flexible trace utility that implements the above concepts using a set of macro hooks which could be used by all DCE components (processes and runtimes):

[a] `DEBUG_EVENT(event_flag, event_message_string)` — if tracing is on, `fprintf()`'s a message to the debug file

[b] `DEBUG_TRACE(message_string)` — if tracing is on, checks an event flag and conditionally `fprintf()`'s to debug file

[c] `LOG_EVENT(event_flag, event_message_string)` — same as `DEBUG_TRACE` but can't be compiled out

[d] `LOG_TRACE(message_string)` — same as `DEBUG_EVENT` but can't be compiled out

`LOG_EVENT` and `LOG_TRACE` should still permit other system vendors to compile them out (perhaps using a `#define _DEBUG2` flag).

(iv) `Assert()` calls should be used for all input parameters within DCE client stubs (possible IDL compiler enhancement).

(b) Diagnostic investigation / performance monitoring

(i) A new set of rpc management interfaces should be created for controlling tracing (since tracing can affect multiple users, it should be treated as an authorized interface):

```
/* turns trace on/off */
void rpc_mgmt_set_trc_flags(
    rpc_binding_handle_t binding,
    unsigned32           flags,
    unsigned32           options,
    unsigned32           *status
);

/* queries trace flags */
void rpc_mgmt_inq_trc_flags(
    rpc_binding_handle_t binding,
    unsigned32           *flags,
    unsigned32           *status
);
```

Notes:

[a] The meaning of each bit in the `flags` variable is defined by each server that supports this interface.

[b] Both these interfaces must be added to the `rpc_mgmt_set_authorization_fn` routine.

[c] Non-RPCable processes (e.g., cdsclerk) can only be controlled via local trace commands (using local sockets??).

[d] What about the GDS processes?

(ii) Each of the DCE server control programs or admin packages should support a new `trace` command, to set trace flags and trace options:

```
trace <component> on  -a -d -p -all -e <flag1,flag2,...>
trace <component> off          -all -e <flag1,flag2,...>
```

where:

`-a` — archive wraparound buffer when full
`-d` — defer recording until a subsequent `trace on` command is issued
`-p` — trace flags should be persistent across server restarts
`-all` — set all trace flags on/off
`-e` — list of event flags

(iii) A set of trace flags common to all DCE components may be created in addition to the component specific trace points.

(iv) A serviceability profile file (or a set of environment variables) is required to store the on/off state of trace points (so they are persistent across restarts) as well as identifying the trace file (for trace data archiving).

(v) The actual recording of trace data to memory/disk may be done inline or by issuing requests to a separate thread or daemon depending on the particular operating system. This decision depends on the implementation of the trace hook code.

3.3. General Trace Point Guidelines

(a) Port verification:

(i) anywhere/everywhere

(b) Application debugging:

(i) Values of input parameters upon entry to key subroutines should be recorded.

(ii) Values of return parameters and return code upon exit should be recorded.

(iii) Contents of de-referenced storage (and not just the value of pointers) should be recorded.

(iv) Key low level algorithm processing should be recorded.

(c) Diagnostic investigation:

(i) process to process communications within a component

(ii) cross component interactions (e.g., login requests)

(iii) database interactions

(iv) network interactions

(v) replication activity

(vi) major events

(vii) flow of control (e.g., subroutine entry/exit)

(viii) thread creation/termination

(ix) synchronization events (mutexes, condition variables, semaphores)

(x) internal API boundary crossings (e.g., cdsapi)

(xi) executor thread activity

(xii) RPC runtime system threads activity (e.g., listener thread)

(d) Performance monitoring:

(i) event counting (e.g., number of cache hits/misses)

(ii) frequency measurements (e.g., number of requests per hour)

(iii) elapsed time measurements (e.g., lifespan of executor threads)

3.4. Logging Interfaces

(a) For port verification, `fprintf()` may be used (or better yet, exploitation of the CDS debug facility).

(b) For application debugging, `fprintf()` may be used (or better yet, exploitation of the CDS debug facility).

(c) For performance monitoring involving counts, a data structure of performance data can be created where no interface is required (updates to the variables occur directly). Logging of activity can then occur on every increment/decrement or at various threshhold levels.

(d) For performance and diagnostic tracing, the following trace logging interfaces should be used:

```
/* create a trace buffer - multiple buffers are supported */
int crttrcbuf (
    int num_entries,
    int entry_length,
    int *trcbh
);

/* destroy a trace buffer */
int dsttrcbuf (
    int trcbh
);

/* log a diagnostic/performance trace record */
int logtrc (
    int         trcbh,
    unsigned16  compid,
    unsigned16  trctype,
    unsigned32  trcword,
    unsigned32  trclen,
    char       *trcbuf
);
```

Notes:

(i) A buffer may be global to a process or local to a thread.

(ii) For performance reasons, all entries are the same length (allows for efficient locking of a single entry by a single thread rather than locking the entire trace buffer).

3.5. Format

(a) Port verification / application debugging

(i) I18N considerations are not mandatory for this type of tracing.

(ii) Each entry should include:

[a] component prefix string

[b] where the trace entry was recorded (__FILE__, __LINE__)

[c] thread id

[d] event name

Following is an example debug tracing format:

```
-----------------------------------------------------------------
pfx: <event_name>   <__FILE__>             <__LINE__ > \
<thread:xx> <debug_text>

CDS: <db_trc>       </dce/src/cds/cdsd.c>  <line:123>  \
<thread:22> <updating database>
RPC: <listener_trc> </dce/src/cds/rpcrt.c> <line:999>  \
<thread:01> <connection accepted from 111.222.333.444>
-----------------------------------------------------------------
```

(b) Diagnostic investigation / performance monitoring

The structure of each entry in the diagnostic trace buffer would be:

```
-----------------------------------------------------------------
Date      Time    Comp/Type Word     Len Data
02/21/92  9:29:45 02100006  12345678 004 01020304
02/21/92  9:32:41 02100099  87654321 008 0102030405060708
-----------------------------------------------------------------
```

3.6. Notes

(a) Other than port verification tracing, message-like text phrases should not be used.

(b) This interface can be a macro vs a subroutine hook.

(c) OSF may choose to define the format of a trace report to be used as a guide by vendors creating trace log formatters (trace logs only contain raw hex data).

4. PROBLEM DETERMINATION GUIDE

The following figure represents an example message description within the PDG:

```
---------------------------------------------------------------------
RPC00001 FAILURE SENDING DATAGRAM TO NETWORK ADDRESS xxx.xxx.xxx.xxx
Explanation: The RPC runtime was unable to send a datagram to the
  specified network address xxx.xxx.xxx.xxx
Severity: WARNING
Problem Determination: Likely cause is that the node indicated by the
  network address is unavailable.
User Response: Contact the network administrator and report the
  failure.
Operator Response: Check the error log for the cause of the node
  becoming unavailable and make appropriate adjustments.
System Action: The system is not halted.  The application program
  that discovered the error does not need to be restarted.
---------------------------------------------------------------------
```

Note: The following severities exist:

(a) `ALERT` — Immediate intervention required. System or DCE service is unusable until action is taken. Usually reserved for permanent errors (non-retriable conditions).

(b) `CRITICAL` — Severe problem exists that makes some DCE services unavailable. Intervention is required. Usually reserved for permanent errors (non-retriable conditions).

(c) `ERROR` — Failure occurred fulfilling a user request. Service is still available but without intervention, any retries of a similar request may fail.

(d) `WARNING` — Eventual intervention required. System is still usable.

(e) `NOTICE` — No error condition exists but some action needs to be taken.

(f) `INFORMATIONAL` — Informational message that requires no action to be taken.

In addition, the following items should be described:

(a) Diagnostic/Performance trace flags common to all DCE processes.

(b) Diagnostic/Performance trace flags specific to each component.

(c) Error type, cause and suggested action code points.

(d) Control program trace command syntax.

(e) Trace log entry contents.

(f) Error log entry contents.

(g) RPC runtime status and comm failure codes.

(h) List of errors/messages that result in remote notification.

(i) Location of logs (operating system specific).

(j) Description of formatting tools (operating system specific).

APPENDIX A. LOGGING INTERFACE MANPAGES

To be done...

APPENDIX B. AIX IMPLEMENTATION EXAMPLE

This section describes a possible AIX V3 implementation using existing facilities. This example is only meant as a means to more clearly demonstrate what the placement of ‘‘hooks’’ within DCE code would achieve.

B.1. Existing AIX Services

Within AIX, the existing set of serviceability related functions include:

(a) `openlog(), syslog(), closelog()`

(b) `catopen(), catgets(), catclose()`

(c) `printf()`

(d) `perror()`

(e) `errlog()`

(f) `assert()`

(g) `trcstart(), trchk(), trcgen(), trcon(), trcoff()`, ...

The following serviceability daemons exist:

(a) `syslogd`

(b) `trace`

(c) errdemon

B.2. Serviceability Header File

```
/* DCE serviceability header file: <dce/service.h> */

#include <sys/syslog.h>
#include <dce/uuid.h>

/* prototypes */
int  loginit(char *);
int  logterm();
int  logmsg(nl_catd, unsigned32, unsigned32, uuid_t *, unsigned32,
            char *, ...);
int  logerr(unsigned32, unsigned32, unsigned32, unsigned32, uuid_t *,
            unsigned32, char *);
int  logtrc(int, unsigned16, unsigned16, unsigned32,
            unsigned32, char *);
int  crttrcbuf(unsigned32, unsigned32, int *);
int  dsttrcbuf(int);
void settrcflags(unsigned32, unsigned32);

/* limits and defaults */
#define  S_MAX_MSG_LTH              132         /* maximum message
                                                   length */
#define  S_MAX_ERR_LTH              132         /* maximum error
                                                   string length */
#define  S_CATALOG_ERRMSG           "Error accessing catalog!\n"
#define  S_DEFAULT_MSG              "Message not found!\n"

/* example message/error identifiers */
#define RPC00001                    0x11000001  /* RPC R/T Error */
#define RPC00002                    0x12000002  /* RPCD Error    */
#define RPC00003                    0x12000003  /* RPCD Error    */
#define CDS00001                    0x21000001  /* CDSD Error    */

#define S_MSG_CATALOG_NUMBER(x)     (x)>>28     /* first nibble
                                                   determines the
                                                   catalog containing
                                                   the message */

#define S_MSG_COMP_MASK             0xF0000000
#define S_MSG_SUBCOMP_MASK          0x0F000000
#define S_MSG_VENDOR_MASK           0x00FF0000
#define S_MSG_NUMBER_MASK           0x0000FFFF

/* message severities */
#define  S_INFO1                    LOG_INFO
#define  S_INFO2                    LOG_INFO
#define  S_NOTICE                   LOG_NOTICE
#define  S_WARNING                  LOG_WARNING
#define  S_ERROR                    LOG_ERR
#define  S_CRITICAL                 LOG_CRIT
#define  S_ALERT                    LOG_ALERT

#define  S_MSG_NOTIFY_MASK          0x80000000
```

```
/* message string prefixes */
#define  S_RPCD_STR                  "rpcd:"
#define  S_RPCRT_STR                 "rpcrt:"
#define  S_SECD_STR                  "secd:"
#define  S_SEC_CLIENTD_STR           "sec_clientd:"
#define  S_CDSD_STR                  "cdsd:"
#define  S_CDS_CLERK_STR             "cds_clerk:"
#define  S_CDSADV_STR                "cds_adv:"

/* error types */
#define  S_NO_ERR                    0x00000000
#define  S_SYSTEM_ERR                0x01000000
#define  S_DEVICE_ERR                0x02000000
#define  S_NETWORK_ERR               0x03000000
#define  S_DATA_ERR                  0x04000000

/* error causes */
#define  S_UNKNOWN                   0x00000000
#define  S_CONFIG_ERR                0x00010000

/* suggested actions */
#define  S_NO_ACTION                 0x00000000
#define  S_ATTEMPT_RETRY             0x00000100
#define  S_CORRECT_AND_RETRY         0x00000200
#define  S_CONTACT_REP               0x00000300
#define  S_CONTACT_SYS_ADMIN         0x00000400

/* trace component ids */
#define  S_RPCD_ID                   0x0110
#define  S_RPCRT_ID                  0x0120
#define  S_CDSD_ID                   0x0210
#define  S_CDSCLERK_ID               0x0220
#define  S_CSDADV_ID                 0x0230
#define  S_SECCLIENTD_ID             0x0310
#define  S_SECSERVER_ID              0x0320

#define  S_TRC_COMP_MASK             0x0F00
#define  S_TRC_SUBCOMP_MASK          0x00F0

/* general trace types */
#define  S_SUBROUTINE_FLOW           0x0001
#define  S_DATABASE_TRACE            0x0002
#define  S_NETWORK_TRACE             0x0003
#define  S_THREADS_TRACE             0x0004
#define  S_SYNCH_TRACE               0x0005
#define  S_REPLICATION_TRACE         0x0006
#define  S_SECURITY_TRACE            0x0007
#define  S_MARSHALLING_TRACE         0x0008

/* specific component trace types */
#define  S_SECTRC_LOGINS             0x0200
#define  S_CDSTRC_CACHE              0x0100

/* example component trace words */
#define S_CDS_REPLICATION_STARTED    0x0001
```

```
#define S_CDS_REPLICATION_FINISHED  0x0002
#define S_SEC_LOGIN_REQUEST         0x0003
#define S_SEC_TGT_ISSUED            0x0004
```

B.3. Source Code

The following section provides a simple implementation of message/error logging and tracing. Since the AIX operating system already has a serviceability infrastructure, there is no need to implement a trace or error logging daemon. It is also assumed that the underlying services used by these hook implementations are thread-safe.

This example is *NOT* a recommendation of how these hooks should be implemented on AIX, but rather as an illustration of what the hooks should do.

For simplicity, thread-safing and use of a serviceability profile have not been incorporated.

B.3.1. Mainline

```
#include <nl_types.h>
#include <stdio.h>
#include <stdlib.h>
#include <dce/uuid.h>
#include <dce/service.h>

main()
{
  nl_catd    cd;
  uuid_t     e_uuid;
  char       *string1="string1";
  char       *string2="string2";
  unsigned32 errnum =42;
  unsigned32 status;

  printf("msg: start\n");
  uuid_create(&e_uuid, &status);
  loginit(S_RPCD_STR);

  printf("\nmsg: logging a message\n\n");
  logmsg(0, RPC00003, (S_INFO1 | S_MSG_NOTIFY_MASK), &e_uuid,
    __LINE__, __FILE__, string1, string2);

  printf("\nmsg: logging an error\n\n");
  logerr(RPC00003, S_ERROR, S_DEVICE_ERR | S_UNKNOWN |
    S_ATTEMPT_RETRY, errnum, &e_uuid, 4, 0x01020304);

  printf("\nmsg: logging a trace entry\n\n");
  logtrc(0, S_CDSD_ID,
    S_REPLICATION_TRACE, S_CDS_REPLICATION_STARTED, 4,
    0x01020304);

  logterm();
  printf("\nmsg: end\n");
}
```

B.3.2. Hook code

```
#include <nl_types.h>
#include <stdio.h>
#include <stdarg.h>
#include <sys/syslog.h>
```

```
#include <sys/errids.h>
#include <sys/trchkid.h>
#include <dce/service.h>
#include <dce/uuid.h>

#define  MAX_DTM_LTH  80
#define  ON            1
#define  OFF           0
#define  YES           1
#define  NO            0

static void _getdtm(char *);

static struct {
  int  cd;
  char *name;
} dce_catalogs[] =
{
  0, "mymsgs.cat",
  0, "mymsgs.cat",
  0, "mymsgs.cat"
};

static unsigned32 trcglobalflags;
static unsigned32 trcstate=OFF;

/* things normally in the serviceability profile file */
static int production=YES;
static int min_severity;
static int msgfh;
static int errfh;
static int trcfh;

/*=============================================================*/
/* routine:     loginit()                                      */
/* description: Initializes all resources and services related to  */
/*              logging.                                       */
/*=============================================================*/
int loginit(
  char *name     /* string name of executable */
)
{
  production     = YES;
  min_severity   = S_INFO1;
  trcstate       = ON;
  trcglobalflags = 0xFFFFFFFF;
  msgfh          = stdout;
  errfh          = stdout;
  trcfh          = stdout;

  return(openlog(name, LOG_PID, LOG_USER));

  /* Optionally, this routine would:                           */
  /* -open/read the serviceability profile file (/dev/service ??)  */
  /* -open the specified log files (message, error and tracing)    */
```

```
  /* -set the specified minumum severity level for message/error   */
  /*   logging                                                      */
  /* -initialize the global trace flags                             */
  /* -initialize the remote logging service                         */
  /* -set the production logging flag                               */
}

/*==================================================================*/
/* routine:     logterm()                                           */
/* description: Cleans up all resources and services related to    */
/*              logging.                                            */
/*==================================================================*/
int logterm()
{
  return(closelog());
}

/*==================================================================*/
/* routine:     logmsg()                                            */
/* description: Logs a message to the local message log.           */
/*==================================================================*/
int logmsg(
  nl_catd     cd,       /* catalog descriptor               */
  unsigned32  id,       /* message identifier               */
  unsigned32  severity, /* message severity                 */
  uuid_t      *e_uuid,  /* error correlator                 */
  unsigned32  line,     /* line number where error occurred */
  char        *file,    /* source file where error occurred */
  ...                   /* caller's substitution variables  */
)
{
  va_list arg_ptr;               /* caller's substitution
                                    variables              */
  char    *fmtstr;               /* format string          */
  char    msg[S_MAX_MSG_LTH];    /* formatted message      */
  char    errstr[S_MAX_ERR_LTH]; /* error entry string     */
  char    dtmstr[MAX_DTM_LTH];   /* date and time string   */
  int     catnum;                /* catalog array index    */

  /* check if message is to be suppressed */
  if (severity < min_severity)
    return(0);

  /* automagically open the catalog based on the identifier */
  if (cd == 0)
  {
    catnum = S_MSG_CATALOG_NUMBER(id & S_MSG_COMP_MASK);
    if (dce_catalogs[catnum].cd == 0)
    {
      cd = catopen(dce_catalogs[catnum].name, 0);
      if (cd == -1)
      {
        fprintf(msgfh, S_CATALOG_ERRMSG);
        return(-1);
      }
```

```
      dce_catalogs[catnum].cd = cd;
    }
  }

  /* log the message */
  fmtstr = catgets(cd, 1, (id & S_MSG_NUMBER_MASK), S_DEFAULT_MSG);
  va_start(arg_ptr, file);
  vsprintf(msg, fmtstr, arg_ptr);
  va_end(arg_ptr);
  if (production)
  {
    syslog(severity, "%s", msg);
    syslog(LOG_INFO, "uuid=<%.16X>,line=%d,file=<%s>\n", e_uuid,
      line, file);
  }
else
  {
     _getdtm(dtmstr);
     fprintf(msgfh, "%s: %s", dtmstr, msg);
     fprintf(msgfh, "%s: uuid=<%.16X>,line=%d,file=<%s>\n",
          dtmstr,e_uuid,line,file);
  }

  /* send a remote error notification to the event handler */
  if (severity & S_MSG_NOTIFY_MASK);
    /* NLNotify(); */       /* DME notification mechanism */
}

/*================================================================*/
/* routine:     logerr()                                          */
/* description: Logs error data to the local error log as well as  */
/*              sending a remote notification event to the DME     */
/*              event logger.                                     */
/*================================================================*/
int logerr(
  unsigned32  id,       /* message identifier            */
  unsigned32  severity, /* message severity              */
  unsigned32  e_info,   /* error codepoints              */
  unsigned32  e_status, /* errno/status/exception/signal */
  uuid_t      *e_uuid,  /* error correlator              */
  unsigned32  e_len,    /* length of user data           */
  char        *e_data   /* hex error data stream         */
)
{
  char        errstr[S_MAX_ERR_LTH];      /* error log entry      */
  char        dtmstr[MAX_DTM_LTH];        /* date and time string */

  /* log error information to the local error log */
  _getdtm(dtmstr);
  sprintf(errstr,"%s: %.5d %.2X %.8X %.8X %.16X %.3d %.*X", dtmstr,
    id, severity, e_info, e_status, e_uuid, e_len, e_len, e_data);
    /* memcpy would be better for production logging */
  if (production)
    errlog(errstr, (unsigned int)strlen(errstr));
  else
```

```
    fprintf(errfh, "%s\n", errstr);

  /* send a remote error notification to the event handler */
  /* NLNotify(...); */       /* DME notification mechanism */
}

/*==================================================================*/
/* routine:     logtrc()                                            */
/* description: Logs trace information based on trace flags set.   */
/*==================================================================*/
logtrc(
  int        trcbh,   /* trace buffer handle          */
  unsigned16 compid,  /* component/subcomponent id    */
  unsigned16 trctype, /* component trace type         */
  unsigned32 trcword, /* component defined trace word */
  unsigned32 trclen,  /* trace buffer length          */
  char       *trcbuf
)
{
  char  dtmstr[MAX_DTM_LTH];   /* date and time string */

  if (trcstate & ((compid | trctype) && trcglobalflags))
  {
    if (production)
    {
      if (trclen > 0)
        trcgent(0, ((compid<<16) | trctype), trcword, trclen,
          trcbuf);
      else
        trchkl(((compid<<16) | trctype), trcword);
    }
    else
    {
      _getdtm(dtmstr);
      fprintf(trcfh,"%s: %.8X %.8X %.3d %.*X\n",dtmstr,
        ((compid<<16) | trctype), trcword, trclen, trclen, trcbuf);
    }
  }
}

/*==================================================================*/
/* routine:     crttrcbuf()                                         */
/* description: Creates a buffer to be used for tracing.            */
/*==================================================================*/
int crttrcbuf(
  unsigned32 num_entries,
  unsigned32 max_entry_lth,
  int        *trcbh         /* trace buffer handle is returned */
)
{
  trcbh = 0;
  return(0);     /* support for multiple trace buffers is optional */
}

/*==================================================================*/
```

```
/* routine:     dsttrcbuf()                                        */
/* description: Destroys a trace buffer.                           */
/*=================================================================*/
dsttrcbuf(
  int trcbh
)
{
  return(0);
}

/*=================================================================*/
/* routine:     settrcflags()                                      */
/* description: Sets trace state (on/off) and trace flags.         */
/*=================================================================*/
void settrcflags(
  unsigned32 state,
  unsigned32 flags
)
{
  trcstate       = state;
  trcglobalflags = flags;
}

/*=================================================================*/
/* routine:     _getdtm()                                          */
/* description: Returns a formatted string.                        */
/*=================================================================*/
static void _getdtm(char *string)
{
  time_t     temp;
  struct tm  *timeptr;

  *string = '\0';
  temp    = time(NULL);
  timeptr = localtime(&temp);
  strftime(string, MAX_DTM_LTH-1, "%x %X", timeptr);
}
```

REFERENCES

[RFC 11.0] M. Hubbard, ‘‘DCE SIG Serviceability Requirements’’, August 1992.

AUTHOR’S ADDRESS

Mark Hubbard
Distributed System Services
IBM Canada Laboratory
Stn. 2G, 1150 Eglinton Avenue East
Toronto, Ontario
CANADA

Internet email: hubbard@torolab5.vnet.ibm.com
Telephone: +1-416-448-3919