OSF DCE SIG **R. Salz (OSF)**
Request For Comments: 36.0 **January 1993**

MIGRATING DCE 1.1 SERVICEABILITY TO EVS

1. INTRODUCTION

One of the requirements of the DCE 1.1 serviceablity work is that it should be reasonable to migrate to DME's EVS. This note describes a transition plan so that this could be done for a future release. It is based on the following documents:

(a) ''DME Event Services Architecture'' [Ferr 92].

(b) ''OSF DME Event Services Functional Specification'' [DME 92a].

(c) ''OSF DME Event Services Design Specification'' [DME 92b].

(d) ''So... You Want to Use the DME Event Services??'' [Ferr 93].

(e) ''DCE 1.1 Serviceability Proposal'' [RFC 24.0].

The last two documents should provide enough background for this note.

2. GOALS AND WAYS TO ACHIEVE THEM

The goal of the migration is that nothing should have to be changed:

(a) Any existing routing specifications work, including those on the command-line (shell scripts) and in the configuration file.

(b) Any existing use of the remote interface (e.g., the new `dce_log` command) still work.

(c) No changes to any calls on the serviceability routines.

(d) No changes to any program that already uses EVS.

There are two ways to handle the migration: gateway or conversion.

Under the first method, both sets of code are untouched. Instead DCE would provide ''gateway'' code. The major example of such code would be a tool to convert a binary serviceability log into an entity that could be imported into EVS and managed using its LVM facilities. In addition, it would be necessary to add an `EVS` routing specifier (and code to handle it) to the serviceability run-time so that an administrator could explicitly decide to route some serviceability notifications to EVS.

The primary benefit of this method is that existing code is not modified.[1] The primary drawback is that this is a very shortsighted approach. If EVS is going to become the primary distributed notification system provided by OSF, then it makes little sense to have another, more limited, system in use within DCE. Further, gateways are rarely clean solutions, and it will no doubt be necessary to add more and more ''glue'' code to make the fit between serviceability and EVS more seamless.

It is clear, then that it is much better to do a full conversion if this is possible. A conversion would maintain the serviceability API (e.g., `dce_svc_printf()`), but replace the internals with EVS routines. This preserves the work the providers will have done to meet the DCE 1.1 requirements.

This is a clean solution that provides full EVS functionality to existing code. The primary drawback is that there is additional overhead for each serviceability message point, notably packing the message into the ANS.1 format used

1. As the author of that code, this has its appeal.

by EVS. To route the message, the Event Controllers will have to unpack parts of the message; in serviceability the routine is done via in-line C code. If the new overhead is not acceptable, we will have to design some sort of ‘‘fast path’’ through the EVS routing. On way to do this would probably be to hard-code knowledge of serviceability events into the EVS API.

3. COMPARISON OF MODELS

The serviceability model was deliberately kept simple. Notifications are partitioned into a finite set of categories. They have a specified set of fixed attributes (component name, time, etc.), and variable data. Notifications can be routed to local log files based on their category. There is very limited filtering: debug messages can be filtered based on an arbitrary small index called the level.

The EVS model is much richer. Events have system (fixed) data and variable data. Event Controllers (EC's) can make decisions based on any of the data fields. Messages can be routed to local or remote log files, or sent out as DCE RPC calls to interested agents.

For example, the serviceability routing specification `FATAL:FILE:/opt/dce/var/fatal.log` could be replaced by an EC that looks like this:

```
action_table {
    fatal { log_local("/opt/dce/var/fatal.log") }
}
filter {
    or { $1 = "FATAL" }
    actions { fatal }
}
```

In order to make this consistent, DCE will provide the fixed serviceability attributes as the first ‘‘n’’ variable parameters of an EVS event.[2] We can call these initial parameters the ‘‘semi-fixed’’ part of the notification.

4. IMPLEMENTATION

The two major serviceablity routines, `dce_svc_printf()` and `dce_svc__debug()`, will be rewritten to pack up their arguments using `evs_ntf_arg_set` and call `evs_ntf_notify()`. This is a simple process — the EVS documents contain sample code. The routines will still do the checks for the various serviceability actions such as `SVC_ACTION_ABORT`, `SVC_ACTION_EXIT_`*xxx*, and `SVC_ACTION_STDERR`; the `SVC_ACTION_CONSOLE` and `SVC_ACTION_NOLOG` will be ignored.

We will provide a new routine, `dce_svc_exit()`.[3] All programs making serviceability calls should call this routine before existing so that any EVS state can be cleanly torn down. The routine will also register itself as an exit handler via `atexit()`.

An EVS event is named by a <UUID, Index> pair, where the UUID is a standard DCE UUID and the Index is an index into an XPG/4 message catalog. All DCE serviceability events will use a single UUID. The Index will not be the typical small integer, but will instead be a 32 bit DCE status code. In order for this to be transparent, minor changes will have to be made to the `evs_cms_open()` and `evs_cms_entry_get()` routines, as detailed in the Appendix.

It is not clear that mapping all serviceability events into a single EVS event set is necessarily the best thing to do. It is probably a better idea for each serviceability component to have its own event set. There are no real technical reasons for this, however, and it can always be done later. Since the number of serviceability components is small, `dce_svc_printf()` can be modified to maintain the <component, UUID> mapping table.

---

2. The actual mapping will be determined later, after we know the fixed prolog for all serviceability messages, but it is likely that the severity level and component name will be the first two.

3. While writing this, it became clear that this would be a good thing to have for DCE 1.1 anyway.

It also seems natural to map each component into a separate application class. We expect that much routing will be based on the component, and moving the component name out of the semi-fixed part and into a separate application class would make the EC's more efficient. The drawback to doing this is that each application class presently requires its own client-ERB (EVS Event Request Broker) communication link, and this would probably consume too many resources in a typical DCE program. It also adds more special cases to the `evs_cms_*()` functions, as described below. If filtering is too inefficient, however, this should be reconsidered.

For the initial work, then, all DCE Serviceability messages will be in the EVS Application Class `dce-svc`. When components move to their own class, they will named `dce-svc-`*xxx* where *xxx* is the component name.

We would like to dispose of all serviceability routing. If ''the market'' prevents this, then it is easy to create an EC from a routing specifier, so the serviceability code will do that. In addition, we will add a new routing specifier: `EVS:{xxx}`. This specifies the UUID of the private EC to be used when the EVS is first contacted.

Initial configuration of routing for a program may be difficult. DCE will provide a default set of EC's the mimic the default serviceability routing. The run-time will use these defaults if no other routing is specified. The serviceability routines must make sure that any serviceability routings are mapped to EC, and that the EC's exist before EVS is first contacted. This may be difficult. While the API to do this exists, it might be too much overhead to add to all DCE servers. If this is the case, then we will have to off-load this work by making an RPC call to a DCE daemon.[4] We cannot know for sure until this part of the EVS code is available.

All filtering can be done by Event Controllers. The `dce_svc_*()` filter routines will still be available, although vendors will be encouraged to replace them with appropriate EC's.

The `dce_log` command will be rewritten. The most common use of this command will probably be to change the routing of (remote) servers. Rather than using the remote serviceability interface defined in the RFC, it will talk to the EC Management Server (ECMS) on the remote host and have it install new EC's there. If the ECMS is not available, it will fall back to using the serviceability interface. This guarantees that the new program can talk to both old and new hosts. Over time, the program should be phased out in favor of the EVS administrative programs.

We will provide a tool to convert binary serviceability logs into EVS logs. This should not be difficult. It will probably be necessary to ''steal'' some EVS source so that the internal format can be duplicated. The DCE 1.1 serviceability work will include an API to access serviceability logs, and we must make sure that this API can be layered on top of the EVS LVM functions, in particular `evs_lvm_record_read()`.

APPENDIX A. EVS CODE CHANGES

Here is sample code intended to show what changes will have to be made to the EVS CMS facility:

```
#include <dce/dce.h>
struct _evs_cms_handle_t dce_svc_evs_cms_handle;

void evs_cms_open(
    evs_ntf_id_p_t p_event_id,
    evs_cms_handle_t handle,
    evs_status_p_t p_status
)
{
    static int setup;
    static uuid dceuuid;
    unsigned32 st;

    if (!setup) {
        uuid_from_string(DCE_SVC_UUID, &dceuuid, &st);
```

4. The general DCE server daemon, `dced`, will probably exist by the time this migration is started.

```
            setup = 1;
        }
        if (uuid_equal(&dceuuid, &p_event_id->evs_eventset) {
            handle = &dce_svc_evs_cms_handle;
            *p_status = evs_s_ok;
            return;
        }
        /* .
           .
           . */
    }

    void evs_cms_entry_get(
        evs_cms_handle_t handle
        evs_ntf_id_p_t p_event_id,
        evs_uchar_p_t *p_msg_buf,
        evs_status_p_t p_status
    )
    {
        if (handle == &dce_svc_evs_cms_handle) {
            *p_msg_buf = dce_msg_get_msg(p_event_id->evs_id,
                                         p_status);
            return;
        }
        /* .
           .
           . */
    }
```

REFERENCES

[DME 92a] Open Software Foundation and Zeitgeist, Inc., ‘‘OSF DME Event Services Functional Specification’’, December 18, 1992.

[DME 92b] Open Software Foundation and Zeitgeist, Inc., ‘‘OSF DME Event Services Design Specification’’, December 18, 1992.

[Ferr 92] Lisa Ferrante and Michael Santifaller, ‘‘DME Event Services Architecture’’, November 4, 1992.

[Ferr 93] Lisa Ferrante, ‘‘So... You Want to Use the DME Event Services??’’, January 8, 1993.

[RFC 24.0] R. Salz, ‘‘DCE 1.1 Servicieabiltiy Proposal’’, November 1992.

AUTHOR’S ADDRESS

Rich Salz
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: rsalz@osf.org
Telephone: +1-617-621-7253