CONTROLLING POLYHEDRA® VIA ELEMENT
How Enea® Element can be configured to manage a fault-tolerant pair of Polyhedra database servers

Abstract

Polyhedra can be configured in a high-availability mode, where a master server is shadowed by a standby server that is ready to take over at a momentôs notice. There is an arbitration interface that allows external code to monitor the servers, and to tell each of them whether it should be master or hot standby. Where Polyhedra is being run in a system that is using Eneaôs Element application development framework, it is possible to use Element to act as the arbitrator for Polyhedra, allowing a consistent approach for controlling and monitoring the HA features of the system. This technical note shows the main features of a sample Element component that can perform this integration.

Contents

Introduction

An overview of Polyhedra®

Polyhedra is a family of relational database management systems designed for use in embedded applications. It has two main flavours: Polyhedra IMDB and Polyhedra Flash DBMS, with the former being available for 32-bit platforms (Polyhedra32) and 64-bit platforms (Polyhedra64). The difference between these products is that Polyhedra IMDB is in-memory for speed, and has journalling and fault-tolerant mechanisms to ensure data persistence and system availability, whereas Polyhedra Flash DBMS trades performance against RAM footprint by using a tuneable in-memory cache in front of a file-based database (assumed to be on a flash-based file system). In 2012 a new product was released: Polyhedra Lite, a free version (subject to license conditions) of Polyhedra32 IMDB, but omitting some functionality such as support of fault-tolerant configurations.

All Polyhedra products support an extended SQL-92 subset and the ODBC and JDBC client libraries conform to the international standards. They have object-oriented features, such as table inheritance (which simplifies and speeds up many queries and updates) and behaviour (to perform knock-on actions and additional integrity checks). Polyhedra was designed from the start for client-server use, with full interoperability between 32-bit and 64-bit versions of Polyhedra. When all the software is running on the same machine, the client-server architecture has three main benefits:

- the software naturally allows for multiple clients, which could be running the same application software (on behalf of different users, say) or performing different parts of the overall application;
- where supported by the operating system and hardware platform, the client application(s) and the server will run in separate address spaces, protected from each other - so a failure of an application will not damage the integrity of the database, nor stop other parts of the application running. Also, on an SMP multicore or multiprocessor machine, clients can be running on separate cores from each other and from the server, and thus allow the overall application to make more use of the available CPU cycles;
- the client-server architecture naturally extends to allowing client and server to be on separate machines, to allow the application to be distributed. When clients are remote, Polyhedra automatically handles issues such as differences in endianism.

Polyhedra servers can operate stand-alone, with each instance handling a separate database, and in addition both Polyhedra IMDB and Polyhedra Flash DBMS can operate in master-standby mode, where one server is acting as a hot standby of another server and has a read-only copy of the database. Polyhedra is fully transactional, and satisfies the Atomic, Consistent and Isolated properties needed for ACID compliance (see sidebar). Polyhedra Flash DBMS transactions are Durable, but in Polyhedra IMDB durability can be balanced against performance: critical changes to the data are preserved by streaming journal records to a log file, and client applications can choose whether the success of a transaction is to be reported immediately or when the log file has been flushed.

Polyhedra has a special feature called ócactive queriesô that allows clients to keep up to date without polling. Basically, for the lifetime of the query the server remembers enough about the query (and the previously-transmitted result set) to know when it has become out of date ï so when the transaction completes it can send a ódeltaô to the client that launched the query to bring it up to date. There is a ódelta mergingô mechanism to avoid problems if the client is slow.

"The ACID Test"

ACID is a commonly-used abbreviation for **A**tomic, **C**onsistent, **I**solated and **D**urable, relating to desirable properties of transactional data management systems:

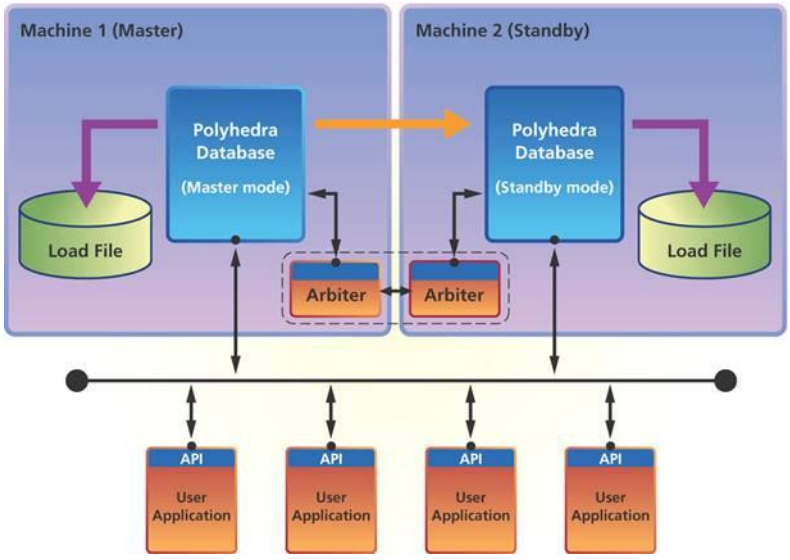
- **Atomic –** Transactions are all-or-nothing: the system never does just part of what is asked.
- **Consistent –** Transactions take the system from one consistent state to another: the system rejects any transactions that break the rules.
- **Isolated –** Transactions operate independently, as though fully serialized, with intermediate, mid-transaction states hidden.
- **Durable –** If a database indicates that a transaction has successfully completed, the application can rely on the changes being preserved - even through a variety of system failure scenarios

Implicit in the above is that the ONLY way of reading or writing to the data is via transactions. All of these transactional properties are vital for a reliable data storage system. Most database systems will offer ACID-compliance, though this is much less common in simpler data stores.

Polyhedra's arbitration mechanisms

When Polyhedra is to be used in an HA configuration, the servers are started up under the control of an external arbiter, whose role is to tell the server whether it should be master or standby. The original mechanism used a third database server containing special records: the fault-tolerant servers that were providing the main database service

would quietly connect to the control server and monitor their control records (using Polyhedraôs active query mechanism) to find out what state each should be in.



Using a database as the arbitrator usually requires a 3-machine set-up to avoid a single point of failure, and often embedded systems want to minimise the hardware cost. However, such systems will normally have a hardware-supported mechanism for determining if the ôotherô board is live (to avoid the risk of accidentally considering both boards as master), so Polyhedra was enhanced to take advantage of this. Each server can be instructed to connect to a local process that acts as the local agent of a decentralised arbitration service, poll it regularly to confirm it is alive, and be ready at any time to receive messages confirming whether it should be master or standby. Communication with the local agent can be via TCP, and on OSE and some Linux platforms Eneaôs LINX Inter-Process Communication (IPC) can also be used.

An overview of Element

The Element product is a middleware software layer situated between the operating system (OS) and application layer providing infrastructure services that address common application needs beyond those supplied by the OS. Application developers can use Elementôs powerful application development framework to simplify the design and implementation of complex, highly available applications in high performance, distributed, heterogeneous systems.

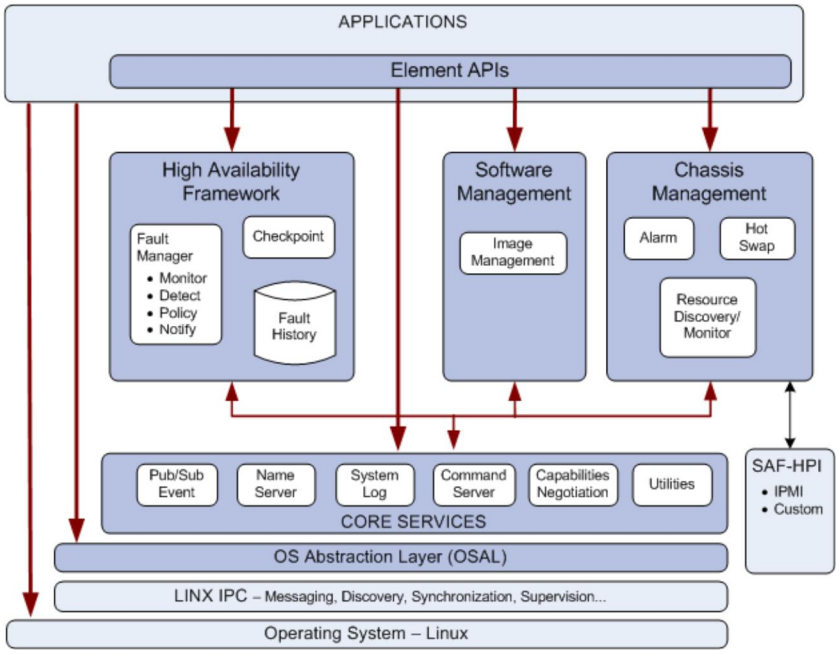
Element is integrated with Eneaôs LINX Inter-Process Communication (IPC) to provide a commercial, carrier-grade middleware package that is available for the Linux OS. LINX enables Elementôs distributed infrastructure that provides reliable messaging, processor domain discovery, synchronization and monitoring, and process location transparency.

Element works with LINX to include the following services:

- discovery, communication, and synchronization that is suitable for distributed, clustered environments;
- transparent, reliable inter-process communication in the same processing domain or distributed domains connected by a network;
- debugging and low-level application monitoring and control;
- partitioning, protection, restart, load and unload capabilities for applications;
- fault monitoring, detection, recovery, notification, and data check-pointing for highly available applications;
- software life-cycle management that includes stopping, starting, re-starting, monitoring, and distributing redundancy-role information to applications;
- hardware chassis and module asset management and monitoring; and,
- software image management.

The figure below shows the Element services and the relationship Element and LINX have with the underlying OS and the customer application layer. Element consists of a suite of core services which serve as a foundation for other

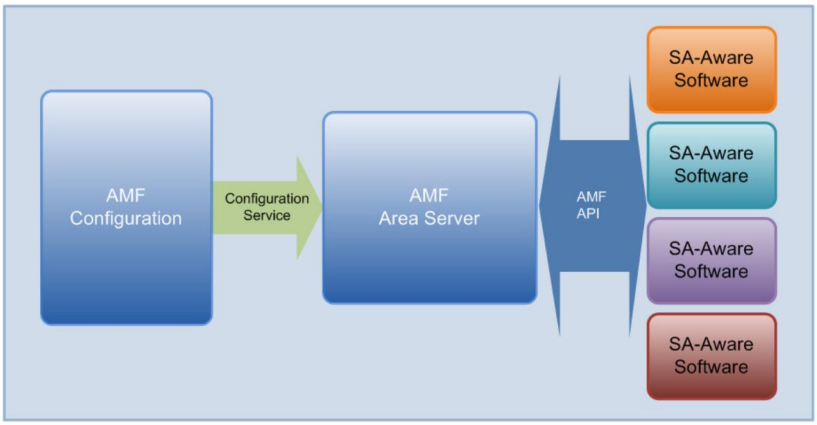
aggregate services. The entire complement of services is available to applications through a collection of service-specific Application Programming Interfaces (APIs).



Element's AMF module

The Availability Management Framework (AMF) provides a high availability (HA) framework that is integrated with the Element Service Suite. The AMF is a standards-based implementation, defined by a specification from the Service Availability Forum (SA Forum) called the SAI-AIS-AMF.

The AMF consists of the AMF Configuration, the AMF Area Server, and the AMF API. The AMF Configuration defines the availability model and is provided to the area server via a configuration service. The AMF Area Server, or simply the area server, is a distributed service that maintains the run-time availability model of the hardware/software system. The area server controls the life cycle of the resources defined in the availability model, assigns services defined in the availability model to those resources, and recovers from errors associated with those resources. Most resources are software entities that are under the direct control of the AMF, have a high level of integration with the framework and are termed SA-Aware. These SA-Aware software entities interact with the area server through the AMF API. The figure below shows the AMF Service, the AMF Configuration Model, and the AMF API.



An Element shim for Polyhedra

Before we start, it is important to emphasise that Polyhedra is not dependent on Element, nor vice versa ï and that fault-tolerant Polyhedra configurations are quite possible without using Element! Having said that, on systems that are using Element (and in particular the HA and AMF sub-systems) it makes sense to use Element to configure, control and monitor the HA Polyhedra service alongside the other services it controls.

As the Polyhedra server executable is not a native Element application, the best way of integrating Polyhedra into the Element framework is to produce a óshimôï an Element application that runs on each of the two boards on which you want to host the HA Polyhedra service, that acts as the arbitrator for the local instance of the Polyhedra server, and that uses the Element services for

- reporting the status of the server;
- logging significant events;
- determining whether the local Polyhedra server should be master or standby;
- determining where the partner Polyhedra sever is located; and
- publishing information about the availability of the Polyhedra data service.

The complete example shim also contains some example code that can be used (from the Element command shell or its web interface) to list the tables in the database or to shut it down, and some prototype code to handle a field upgrade. The complete shim is about 2000 lines long (including comments and lots of calls to the Element logger), so we wonôt go through it line by line in this technical note. Instead, we shall concentrate on the salient details, so that this note both acts as a guide to those who want to adopt and adapt the example shim for use in their own Element Application that uses Polyhedra in an HA fashion, and also illustrates some of the issues involved in controlling third-party HA-aware software from Element.

The core of the shim: telling Polyhedra the mode

This is relatively easy, as Polyhedra can be instructed to use LINX for its control messages ï but one needs to do a slight patch to Element to ensure it is not confused by messages coming from Polyhedra. The alteration is in the file `src/os_specific/linux/linx/ose_emul_linx/ose_emul_linux_linx.c`, where we have to alter the logic of the the `ElemMsgIsWrongEndian()` function:

```
/* two signal numbers are defined by Polyhedra and must have this value:
 *      0x504F4C59 client-server communications ('POLY')
 *      0x504F4C41 server-arbitrator comms      ('POLA')
 */
#define POLYHEDRA_ARB_SIG   0x504F4C41
#define POLYHEDRA_COMMS_SIG 0x504F4C59

OSBOOLEAN
ElemMsgIsWrongEndian(union SIGNAL **sig)
{
#if 0
   /* original version */
   UINT32_T size = ElemMsgSigSize(sig);
   if (((*sig)->sigNo != LINX_OS_LINK_SIG && (*sig)->sigNo != LINX_OS_NEW_LINK_SIG) &&
       ((unsigned char *)(*sig))[size + 1] ^ ElemEndianGetInfoByte())
      {
      return TRUE;
      }
#else
   /* altered version that looks at the signals BEFORE calling ElemMsgSigSize(),
    * and thus avoids problems when a Polyhedra message is inspected.
    */
   if ((*sig)->sigNo == LINX OS LINK SIG  || (*sig)->sigNo == LINX OS NEW LINK SIG ||
       (*sig)->sigNo == POLYHEDRA_ARB_SIG || (*sig)->sigNo == POLYHEDRA_COMMS_SIG   )
      {
      return FALSE;
      }
       UINT32_T size = ElemMsgSigSize(sig);
       if (((unsigned char *)(*sig))[size + 1] ^ ElemEndianGetInfoByte())
          {
          return TRUE;
          }
#endif
```

```
    return FALSE;
}
```

(For the shim we are describing here, we don't actually have to filter out the `POLYHEDRA_COMMS_SIG` – but it is essential if developing an application which wants to wait for either an Element signal or a Polyhedra client-server communication, and it could be needed in a more powerful shim that provided database inspection tools, so we include it here to cover future needs.) Once this patch is done, the Polyhedra shim can follow the standard Element pattern and have a control loop that just waits for signals, and rely on registered call-back functions to take the appropriate action:

```
    for (;;)
        {
        static const SIGSELECT selectAll[] = {0};
        union SIGNAL          *receiveSig;

        /* block waiting for messages */
        receiveSig = ElemMsgRecv((SIGSELECT *)selectAll);
        status = ElemSigDispatch(pEnv->pEsdRegistry, &receiveSig);
        H_ASSERT_M(status == ELEM_STATUS_OK);
        if (receiveSig)
            ElemSigDiscard(pEnv->pEsdRegistry, &receiveSig);
        }
```

(In the above and all subsequent extracts, Element API functions are shown in red, and `pEnv` will point at the information structure used by the shim to keep track of the operational information. `H_ASSERT_M` is a macro in the Element library whose job is to kill the process and generate a report if the condition is not satisfied; for simplicity, invocations of this macro have usually been removed from later extracts – but they are present in the full example shim, along with various calls to the Element log functions.)

Before entering this loop, you have to register a handler for the arbitration request messages coming from the local Polyhedra database server:

```
    status = ElemSigReg(pEnv->pEsdRegistry,
                        POLYHEDRA_ARB_SIG,
                        ELEM_SIG_SCOPE_SIGNAL,
                        polyShimHandleArbSig,
                        pEnv);
    H_ASSERT_M(status == ELEM_STATUS_OK);
```

This tells Element that when a signal of the right type is received it should invoke the `polyShimHandleArbSig()` callback handler, with one of its arguments pointing at `pEnv` (though it will have to be cast to the right type within the callback function). The Polyhedra server will expect a message in response indicating the mode in which it should operate, so a suitable definition of the signal structures, the `polyShimHandleArbSig()` function and its support function could be:

```
#define POLY_ARB_SIGNAL_TYPE_STATUS     'J'
#define POLY_ARB_SIGNAL_TYPE_CONTROL    'A'

#define POLY_ARB_SIGNAL_MODE_UNKNOWN    0
#define POLY_ARB_SIGNAL_MODE_MASTER     1
#define POLY_ARB_SIGNAL_MODE_STANDBY    2
/* Set the arbiter's reporting interval to once per minute for now, it's in microseconds
 * Also, have a faster interval when we need the rtrdb to acknowledge a mode change.
 */
#define POLY_ARB_STAT_INTERVAL          (1000*1000 * 60)
#define POLY_ARB_STAT_INITIAL_INTERVAL  ( 500*1000)

#define POLY_ARB_MAX_SERVICE_NAME       255

typedef struct POLY_ARB_DB_STATUS_SIG
    {
    SIGSELECT    sigNo;
    INT32_T      Type;
    INT32_T      Mode;
    UINT32_T     TransNo;
    char         name[POLY_ARB_MAX_SERVICE_NAME + 1];
    } POLY_ARB_DB_STATUS_SIG;
```

```
typedef struct POLY_ARB_DB_CONTROL_SIG
   {
   SIGSELECT   sigNo;
   INT32_T     Type;
   INT32_T     Mode;
   UINT32_T    Interval;
   char        name[POLY_ARB_MAX_SERVICE_NAME + 1];
   } POLY_ARB_DB_CONTROL_SIG;
```

(The structure of the status signal that comes from the Polyhedra server is the same as the control signal that needs to be sent back, which allows us to reuse the signal when appropriate!)

```
PRIVATE_T void
polyShimHandleArbSig(void *refPtr, union SIGNAL **pSig)
   {
   POLYHEDRA_SHIM_ENV_ST   *pEnv        = (POLYHEDRA_SHIM_ENV_ST *)refPtr;
   POLY_ARB_DB_STATUS_SIG  *pArbSigStat = (POLY_ARB_DB_STATUS_SIG *) *pSig;

   if (pArbSigStat->Type == POLY_ARB_SIGNAL_TYPE_STATUS)
      {
      /* remember who sent the message, and the mode reported by the server */
      pEnv->dbMode        = pArbSigStat->Mode;
      pEnv->rtrdbPid      = ElemMsgSender(pSig);
      pEnv->rtrdbPidKnown = TRUE;
      /* call polyShimTellModeToRtrdb() to tell the rtrdb what mode it should be in. */
      polyShimTellModeToRtrdb (pEnv, pSig);
      }
   }
```

(We introduce the second function `polyShimTellModeToRtrdb()` to allow the functionality to be reused in another part of the shim: more of that later.)

```
PRIVATE_T void
polyShimTellModeToRtrdb(POLYHEDRA_SHIM_ENV_ST  *pEnv,
                        union SIGNAL          **pSig)
   {
   POLY_ARB_DB_CONTROL_SIG *pArbSigCtrl = (POLY_ARB_DB_CONTROL_SIG *)*pSig;

   pArbSigCtrl->sigNo = POLYHEDRA_ARB_SIG;
   pArbSigCtrl->Type  = POLY_ARB_SIGNAL_TYPE_CONTROL;
   pArbSigCtrl->Interval = POLY_ARB_STAT_INTERVAL;
   /* have a fast heartbeat until we know the rtrdb is in the right mode */
   pArbSigCtrl->Interval = (pEnv->dbMode == pArbSigCtrl->Mode) ?
                            POLY_ARB_STAT_INTERVAL : POLY_ARB_STAT_INITIAL_INTERVAL;
   pArbSigCtrl->Mode  = (pEnv->amfHaState == SA_AMF_HA_ACTIVE) ?
                        POLY ARB SIGNAL MODE MASTER : POLY ARB SIGNAL MODE STANDBY;
   strncpy(pArbSigCtrl->name, sizeof(pEnv->pArbSigCtrl), pEnv->otherJournalService);
   ElemMsgSend(pSig, pEnv->rtrdbPid);
   }
```

The `polyShimHandleArbSig()` function checks the signal is of the expected type, remembers certain information in the shimôs main control structure for later use, and calls the `polyShimTellModeToRtrdb()` function to send the message back. The `polyShimTellModeToRtrdb()` function looks inside the control structure (at `pEnv->amfHaState` and `pEnv->otherJournalService`) to decide what information to put in the message that is to be sent back to the Polyhedra server. This leaves open the questions: where do we get the information that needs to be sent to the serveré and what starts off the server, anyway?

Finding out the mode

To determine the mode the Element wants the local Polyhedra server to act, it is necessary (a) to tell Element that Polyhedra is an HA service that supports master-standby configurations, and (b) for the shim to register with Element to be told the initial mode (and again whenever the required mode changes). The first can be done using the AMF configuration wizard (described in the Element AMF guide that is distributed in Element release kits); alternatively, you can adapt an AMF óscript fileô such as the following:

```
add app Polyhedra
set EXTRA_ARGS "-app Polyhedra"
add service Redundant 2N -cbs
add service Non_Redundant NONE

set EXTRA_ARGS "-app Polyhedra -service Redundant -red 2N -scope 1"
add component polyShim COMPONENT_FAILOVER
```

In essence, this tells the Element AMF that Polyhedra can be run in a master + hot standby configuration by starting off the polyShim application on two control blades, and if the master instance of polyShim fails then the other one is to be promoted.

To find out the mode assigned to it, the shim has to make its control object global[1]é

```
static POLYHEDRA_SHIM_ENV_ST *pGlobalEnv;
```

é and then (in addition to other stuff needed of all Element components) register itself and some call-back functions with the AMF:

```
ELEM_MAIN(polyShim)
   {
   POLYHEDRA_SHIM_ENV_ST *pEnv;
   ELEM_STATUS_T          status;
   SaVersionT             polyShimAmfVersion = {'B', 2, 1};
   SaAisErrorT            aisErr;
   SaAmfCallbacksT        polyShimAmfCallbacks = { NULL,
                                                   polyShimComponentTerminateCb,
                                                   polyShimCSISetCb,
                                                   polyShimCSIRemoveCb,
                                                   NULL,
                                                   NULL,
                                                   NULL
                                                 };
   pGlobalEnv = pEnv = (POLYHEDRA_SHIM_ENV_ST *) malloc(sizeof(POLYHEDRA_SHIM_ENV_ST));
   memset (pEnv, 0, sizeof(POLYHEDRA_SHIM_ENV_ST));
   ...
   aisErr = saAmfInitialize(&pEnv->amfHandle, &polyShimAmfCallbacks, &polyShimAmfVersion);
   aisErr = saAmfComponentNameGet(pEnv->amfHandle, &pEnv->amfCompName);
   aisErr = saAmfComponentRegister(pEnv->amfHandle, &pEnv->amfCompName, NULL);
   ...
```

Two of the call-backs are present simply because the AMF state model requires them for fault-tolerant 2N components, but in our example we shall provide minimal implementations (and do all the work in the remaining call-back function)

```
PRIVATE_T void
polyShimCSIRemoveCb(SaInvocationT    invocation,
                    const SaNameT   *compName,
                    const SaNameT   *csiName,
                    SaAmfCSIFlagsT   csiFlags)
   {
   POLYHEDRA_SHIM_ENV_ST  *pEnv = pGlobalEnv;

   /* Respond to AMF to accept assignment */
   saAmfResponse(pEnv->amfHandle, invocation, SA_AIS_OK);
   }

PRIVATE_T void
polyShimComponentTerminateCb(SaInvocationT invocation, const SaNameT *compName)
   {
   POLYHEDRA_SHIM_ENV_ST       *pEnv = pGlobalEnv;

   /* Respond to AMF to accept assignment */
   saAmfResponse(pEnv->amfHandle, invocation, SA_AIS_OK);
   }
```

Now let us look at the other call-back function referenced in the `SaAmfCallbacksT` structure, `polyShimCSISetCb()`. The AMF knows (by its position in the structure) that it should call this function to say

[1] *When registering a callback function with Element, you can supply a pointer whose value will be passed over when the callback function is invoked. However, as mentioned in the introductory section, the Element AMF functions are based on the standard API published by the SA Forum, and they don't provide this useful facility.*

whether the component should be active or standby, and also call it again whenever the required mode changes. The call-back function (or something it triggers) has to acknowledge to the AMF that it has received the message and that appropriate actions have been initiated. In our case, these other actions are both to propagate the status information to the Polyhedra server, and also to ensure that the standby server can find out where the master server is; this allows the standby to connect to the masterôs ójournal portô to get an up-to-date copy of the database, and to receive information about subsequent changes. Thus:

- If the AMF is telling us to be active (or, in Polyhedra parlance, ómasterô), then:
  - if the local server was previously in standby mode, promote it (by calling the `polyShimTellModeToRtrdb()` function described earlier: we said we would need it again!);
  - if we had previously registered to be told information about the journal service of the other instance of the Polyhedra server, cancel the registration; and,
  - publish information about the journal port for the local server so the standby can connect to us.
- If the AMF is telling us to go standby, then:
  - we should cancel the publication about the journal port of the local server (if we had done so);
  - we should tell the Polyhedra server to go to standby (if it is currently master); and,
  - we should register to be told about the journal port of the master, whenever if becomes available.

We shall ignore other state changes that are supported by the SA Forum HA model.

```
PRIVATE T void
polyShimCSISetCb(SaInvocationT        invocation,
                 const SaNameT       *compName,
                 SaAmfHAStateT        haState,
                 SaAmfCSIDescriptorT  csiDescriptor)
   {
   POLYHEDRA_SHIM_ENV_ST *pEnv = pGlobalEnv;
   ELEM_OBJ_HANDLE_T      myJournalServiceObj = 0;
   SaAmfHAStateT          prevHaState = pEnv->amfHaState; /* remember the previous state)*/

   /* Respond to AMF to accept assignment, and record the new state locally */
   saAmfResponse(pEnv->amfHandle, invocation, SA_AIS_OK);

   if (haState == SA_AMF_HA_ACTIVE)
      {
      char slotLabel[ELEM_BCS_SLOT_LABEL_LEN];
      pEnv->amfHaState = haState;

      /* if the server had previously told us it was in standby mode, promote it. */
      if ((pEnv->rtrdbPidKnown == TRUE) && (pEnv->dbMode == POLY_ARB_SIGNAL_MODE_STANDBY))
         {
         union SIGNAL *pElemSig;
         pElemSig = ElemMsgAllocBuf(sizeof (POLY_ARB_DB_CONTROL_SIG), POLYHEDRA_ARB_SIG);
         polyShimTellModeToRtrdb(pEnv, &pElemSig);
         }

      /* ensure we are not subscribed to notifications about 'other journal service'. */
      if (pEnv->otherJournalServiceOptNSHandle)
         {
         ElemNssUnsubscribe(&pEnv->otherJournalServiceOptNSHandle);
         pEnv->otherJournalServiceOptNSHandle = 0;
         }

      /* publish information so that the standby can find the 'other journal service'. */
      ElemBcsGetSlotLabel(slotLabel);
      snprintf(pEnv->myJournalService, sizeof(pEnv->myJournalService),
               "linx/%s/polyJ", slotLabel);
      myJournalServiceObj = ElemObjNewRootContainer((char*)"journal_service", 0);
      ElemObjNewString(myJournalServiceObj,
                       (char*)"journalServiceName", pEnv->myJournalService);
      ElemNssPublishOpt((char*)POLY_JOURNAL_SERVICE,
                        ELEM_NSS_SCOPE_CLUSTER,
                        myJournalServiceObj);

      /* Flag as ready. */
      pEnv->isReady              = TRUE;
      pEnv->journalnamePublished = TRUE;
      }
```

```
   else if (haState == SA_AMF_HA_STANDBY)
      {
      pEnv->amfHaState = haState;
      if (prevHaState == SA_AMF_HA_ACTIVE)
         {
         /* We were the active/master shim but (for some reason) we're back into
          * standby mode - so we should unpublish some stuff and tell the rtrdb
          */
         if (pEnv->journalnamePublished == TRUE)
            {
            ElemNssUnpublish((char*) POLY_JOURNAL_SERVICE);
            pEnv->journalnamePublished = FALSE;
            }
         if ((pEnv->rtrdbPidKnown == TRUE) && (pEnv->dbMode == POLY_ARB_SIGNAL_MODE_MASTER))
            {
            union SIGNAL *pElemSig;
            pElemSig = ElemMsgAllocBuf(sizeof (POLY_ARB_DB_CONTROL_SIG), POLYHEDRA_ARB_SIG);
            polyShimTellModeToRtrdb(pEnv, &pElemSig);
            }
         }

      /* try and find out the journal service of the other server */
      if (!pEnv->otherJournalServiceOptNSHandle)
         {
         pEnv->otherJournalServiceOptNSHandle = ElemNssSubscribeOpt(pEnv->pEsdRegistry,
                                                (char*) POLY_JOURNAL_SERVICE,
                                                polyShimHandleOptAdded,
                                                polyShimHandleOptRemoved,
                                                pEnv,
                                                ELEM_NSS_SUBSCRIBE_OPT_ATTROBJ,
                                                NULL);
         }
      }
   }
```

The `polyShimHandleOptAdded()` function registered above will be called by Element when information has been published about the journal port, and the function stores in `pEnv->otherJournalService` so that it can be passed to the standby server in the arbitration messages generated by `polyShimTellModeToRtrdb()`:

```
PRIVATE T_void
polyShimHandleOptAdded(void             *userParam,
                       PROCESS           pid,
                       char             *path,
                       ELEM_OBJ_HANDLE_T attrObj)
   {
   POLYHEDRA_SHIM_ENV_ST  *pEnv = (POLYHEDRA_SHIM_ENV_ST *)userParam;
   char                   *pOtherJournalService;
   ElemObjGetStringRef(attrObj, (char*) "journalServiceName", &pOtherJournalService);
   strncpy(pEnv->otherJournalService, sizeof(pEnv->otherJournalService),
           pOtherJournalService);
   pEnv->isReady = TRUE;
   ElemObjFree(attrObj);
   }
```

Starting the Polyhedra server

The simplest way to do this is for the shim to make use of the POSIX `fork()` command to fire off a separate process, and within that separate process to use `chdir()` to move to the right directory and `execl()` to load and run the Polyhedra server executable. Timing is important, though: we have to wait until we know enough about how we want to run the executable. For example, we need to know the directory tree in which Element is running, so we can fire off the right instance of the executable; we also have to know whether the Polyhedra server is to start off as master or standby, as in the latter case we have to wait until the master is ready for the standby to connect to it. The simplest way to do this is:

- first to register to be told about all the information we need to start up the Polyhedra server;
- next, to enter a loop waiting until we have all the information; and, finally,
- fork the shim, with one process executing the Polyhedra server and the other dropping through into the main loop described above.

You may have noticed the `polyShimCSISetCb()` and `polyShimHandleOptAdded()` functions shown above both contained the lineé

```
    pEnv->isReady = TRUE;
```

This is used as a signal to the initial control loop that we know either that the Polyhedra server is to start up as master (which would normally only occur when doing a cold boot of the system) or that it is to start as standby and we know where the master is running. A second flag, `pEnv->isInited`, is set when we know the working directory: the command to ask for this is:

```
    ElemRelRegisterReleasePath(pEnv->relHandle, polyShimReleasePathCb, pEnv);
```

é and the corresponding call-back function simply records the value and set the flag:

```
polyShimReleasePathCb(void *pUser, char *pRelPath)
    {
    POLYHEDRA_SHIM_ENV_ST *pEnv = (POLYHEDRA_SHIM_ENV_ST *)pUser;
    strncpy(pEnv->relPath, sizeof(pEnv->relPath), pRelPath);
    snprintf(pEnv->execDir, sizeof(pEnv->execDir), "%s/%s", pRelPath, TARGET_BIN);
    pEnv->isInited = TRUE;
    }
```

(This records not just the directory containing the Element release, but also the subdirectory that should contain the executable; TARGET_BIN is set by the make file and will depend on the hardware architecture on which the software will be running.) So, the initial control loop tests both the flags described above and exits when both are satisfied and we are ready to proceed to the `fork()` stage:

```
    while  (!(pEnv->isInited) || !(pEnv->isReady))
       {
       static const SIGSELECT selectAll[] = {0};
       union SIGNAL           *receiveSig;

       /* block waiting for messages */
       receiveSig = ElemMsgRecv((SIGSELECT *)selectAll);
       ElemSigDispatch(pEnv->pEsdRegistry, &receiveSig);
       if (receiveSig)
          ElemSigDiscard(pEnv->pEsdRegistry, &receiveSig);
       }
```

```
    kidPid = fork();
    if (kidPid == 0)
       {
       /* child fork */

       /* All the descriptors got duplicated by the fork - I have to close
        * this one or it will hang around and cause BAD things to happen.
        */
       inchild = true;
       (void) close(ElemIpcGetDescriptor());

       /* Polyhedra operates best if the database server is fired off in the directory
        * containing the poly.cfg file, as that way the locations for the saved database
        * file and any CL file can all be expressed in relative form.
        */
       if (chdir (pEnv->relPath) != 0)
           printf ("chdir (<relpath>) call failed.\n");
       if (chdir ("polyhedra") != 0)
           printf ("chdir ('polyhedra') call failed.\n");
       snprintf(path, sizeof(path), "%s/%s", pEnv->execDir, POLY_SERVER_IMAGE_NAME);
       snprintf(pEnv->arb_service, sizeof(pEnv->arb_service),
                "arbitrator_service=%s", POLY_SHIM_NAME_QUOTED);
       execl( path, POLY_SERVER_IMAGE_NAME, "-r", pEnv->arb_service, pEnv->modName, NULL );
       printf ("execl call failed.\n");
       exit (10);
       }
```

```
else
   {
   /* save the Pid of the child process for later use. */
   pEnv->rtrdbLinuxPid = kidPid;
   /* finally, we are ready to enter the main control loop of the shim! */
   for (;;)
      {
      static const SIGSELECT selectAll[] = {0};
      union SIGNAL          *receiveSig;

      /* block waiting for messages */
      receiveSig = ElemMsgRecv((SIGSELECT *)selectAll);
      status = ElemSigDispatch(pEnv->pEsdRegistry, &receiveSig);
      H_ASSERT_M(status == ELEM_STATUS_OK);
      if (receiveSig)
         ElemSigDiscard(pEnv->pEsdRegistry, &receiveSig);
      }
```

(The above code snippet incorporates the main control loop given earlier in this technical paper).

There are few other details that have to be filled in: for example, the shim should catch POSIX signals so it can detect when the child process has failed ï at which time it should reset the original signal handling and reissue the signal at itself; Element will detect the failure, promote the standby (if running) and restart the shim. However, what has been given above should be enough to give a feel for what needs to be done in a basic shim.

Other functionality

Having got the overall framework of the shim in place, there are a number of ways this can be enhanced. For example, the shim could publish when the overall service becomes available, so that client applications can register to be told that connections are possible. The shim can also provide Element commands that inspect or update the database, generate snapshots, or perform a controlled shut-down. And of course the shim can make use of Elementôs logging functions to record what it is doing; this can not only help in debugging the shim itself, but can be also of benefit in monitoring the behaviour of the overall system. Some of these features are illustrated in the complete example shim.

Summary

This technical note has shown the core elements of an Element shim that can be used to monitor and control Polyhedra servers that are to be run in a fault-tolerant configuration. It illustrates that standard Element facilities can be used to control third-party HA-aware software, and also that Polyhedra makes it easy for system administrators to integrate it into their HA platform. Integration of Element and Polyhedra is particularly easy as Polyhedra allows LINX to be used for HA control messages! A complete copy of the source code of a working shim is available to Enea customers in machine readable form, along with supporting information such as make files and suitable configuration files.