By Dr. Nigel Day, Technical Director, Polyhedra Plc,
Dave Morse, President, Polyhedra, Inc.

Real-time Embedded Database Fault Tolerance on Two Single-board Computers

This document describes a generic arbiter written in C which is supplied as demonstration code, which calls a function to determine whether the board on which it runs should be master. A dummy implementation of the function is provided, which would have to be replaced for use on an embedded system. The architecture of the overall mechanism is as illustrated in figure 1.

Each arbiter task services the local JCP. The two arbiters do not communicate with each other, and yet, due to the services provided by the system software (which itself will be relying on hardware mechanisms), they are able to offer a reliable, distributed arbitration service to the JCPs. 'Reliable' here means that one will not get a situation whereby both JCPs think their database should be master, except perhaps for a brief moment when a switchover occurs.

INTRODUCTION

Real-time database fault tolerance is provided by Polyhedra with an Active and redundant Standby database configuration. The Standby database is kept up to date with the changes that occur in the Active database, and so it is ready to go live as soon as it is told to do so - either as a result of a problem with the Active or because a controlled switch-over was requested. The roles of the databases can be switched at any time.

If you are developing an embedded 2-machine fault-tolerant real-time database system you will need a reliable way of detecting which of the two computers is 'master' at any time, and which one is 'standby' - for chaos can ensue if both machines consider themselves master. The way this decision is made will vary widely; a common technique is to design boards with a hardware-supported watchdog mechanism, whereby a software function can query a hardware latch to see if the board is master - if so, the software on the board is expected to call a watchdog routine every so often to prevent the latch tripping over (which would signal the partner board to become master).

Various other mechanisms are possible, each with their own advantages and disadvantages. When one or more Polyhedra databases are running on a board it is not the role of Polyhedra either to supply a mech-

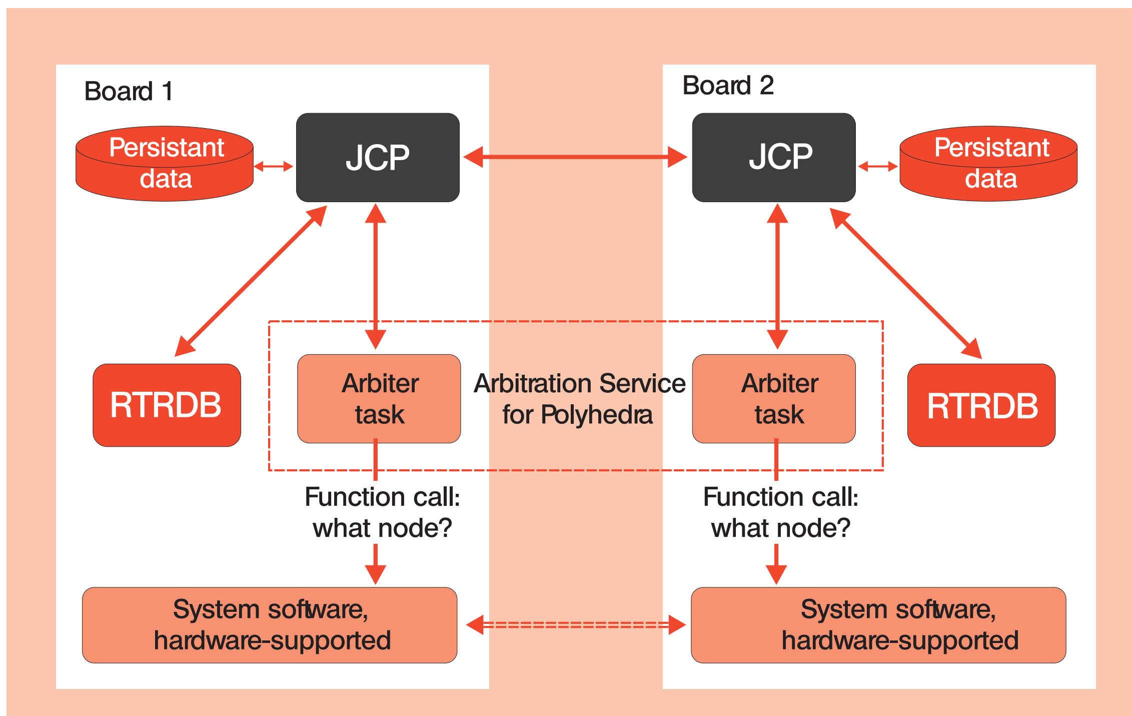

Figure 1. Architecture of the overall mechanism.

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

anism to determine whether the board is master or standby, nor should we dictate which mechanism should be used. Instead, the Polyhedra Journal Control Process component (JCP) on each board expects that there is an arbiter process/task to which it can connect via TCP/IP, and with which it can communicate to determine the board's master/standby status; the arbiter tasks on the two machines between them provide an 'arbitrator service' to the Polyhedra JCPs, even though they might not directly communicate. The protocol of messages between the JCP and the arbitrator service is described in an appendix to this document.

A SIMPLE ARBITER, USING A SYSTEM-SUPPLIED FUNCTION

Suppose there is a system-supplied function named, say, CheckIfActive() which whenever it is called returns 0 on the board that is the standby, 1 if it is the master board, and which suspends until it knows if the machine does not yet know if it should be master. In this case, the arbiter task is very simple. The routine takes one argument, a port number, and it claims this TCP/IP port to allow the JCP to connect to it. If the open fails, the routine can give up with a suitable return code; in other cases, it can drop into its main loop as described below.

THE MAIN LOOP

The main loop of the program calls a standard library function, select to wait for a connection attempt, an incoming message over an existing connection. When the select function returns, the program looks to see what woke it up. If it is a connection attempt, it accepts the connection and creates a little data structure to record the socket ID of the newly-opened connection. If it is an incoming message from a JCP, it finds out - by calling the CheckIfActive function - whether to JCP should be master or standby, and sends the JCP a message telling it the mode in which it should be operating. If the arbiter realises a connection has been closed by the other end, it simply closes its end of the connection and deletes the associated data structure that it had created earlier. Once the reason the program had been woken up has been determined and appropriate action taken, the program goes back to the start of the loop and calls select again.

THE CheckIfAlive IMPLEMENTATION

In order to test operations in the absence of a general-purpose arbitration mecnanism, the sample arbiter program stores a character flag, and the CheckIfAlive checks this flag against the last character of the 'name' field in the messages from the JCP: if there is a match, the JCP is assumed to be master. This implementation has the benefit that the arbiter program can be used in test environments where both databases are running on the same computer; typically, one might have one jcp announcing itself as jcp1 and the other as jcp2.

To test dynamic change of the master/standby status, the sample arbiter described in this document has extra code to allow for a client to connect in via, say, telnet; the first letter of each incoming line is used to select a new mode.

IMPROVING THE ARBITER FOR REAL USE

Before using this code in earnest in an embedded environment, some changes are needed, to remove code introduced to aid testing on Unix & Windows, and to integrate into the on-board support for determining board status. In particular, you should:

- remove the code in the main loop (shaded green in the listings below) that handles connections from tasks that are not JCPs;
- remove the code (again shaded green) for initialising the board_mode global variable; and, most importantly,
- remove the code (shaded red) that defines the CheckIfActive function, replacing all instances of CheckIfActive by the name of the system-supplied function that allows a task to find out the board status.

Various other improvements would be needed, such as making more use of non-blocking socket operations, and ensuring that all resources are freed whenever leaving the program. The code given here should be treated as a prototype.

CONFIGURING POLYHEDRA TO USE THE ARBITER

Assuming the arbiter has been modified as indicated above to work on emebedded systems, then the Polyhedra configuration file used on each of the two

```
JCP:
type                    = jcp
arbitrator_port         = 7200
rtrdb_heartbeat_interval = 2000
rtrdb_heartbeat_timeout  =  1500
startup_timeout          =  7000
data_service            = 7202
other_jcp_service        = $(OTHER_MC):7202
rtrdb_command            = rtrdb db
log_command              = jcplog log

J:
jcp_service             = 7202

LOG:j
type                    = jcplog

DB:j
suppress_dvi            = yes
suppress_log            = yes
type                    = rtrdb
load_file               = test.dat
data_service            = 7002
```

Table 1. The poly.cfg file.

machines can simplify greatly compared to the full version used in the fault-tolerant demos: for example, as they are running on separate machines, the JCPs can use the same port number on each machine, as can the RTRDB. In fact, the only difference between the configurations on the two machine is that each JCP knows to know the address of the machine running the other database. The file poly.cfg could simplify enormously (See Table 1).

One would EITHER substitute the correct IP address of the 'other' machine for $(OTHER_MC) on each machine, OR remove the line defining the other_jcp_service resource from the poly.cfg file, and arrange to start each jcp with a command such as...

```
jcp -r other_jcp_service=10.1.2.101:7202 jcp
```

THE PROGRAM CODE

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

/*                  -----------------------------------------------------------------                  */
/*                                   C h e c k I f A c t i v e                                         */
/*                                   ------------------------                                          */
/*        define a function that will return information about the board mode;                        */
/*        in this simple example code, we check whether the last character of the                      */
/*        string supplied by the jcp in its heartbeat matches the global constant                      */
/*        board_mode. In a REAL implementation, we would call a system function to                     */
/*        determine the mode, ignoring the string supplied by the jcp.                                 */
/*                                                                                                     */
/*              REPLACE THIS DEFINITION on a real implementation.                                      */
/*                -----------------------------------------------------------------                    */

unsigned char board_mode;

int CheckIfActive (unsigned char c)
{
  return (c == board_mode);
}

/*                  -----------------------------------------------------------------                  */
/*                                s t r e a m h a n d l e                                              */
/*                                ------------------------                                             */
/*        define a structure for recording connection info, and a function to send                     */
/*        a message to a nominated connection to tell it what mode it should be in.                    */
/*                  -----------------------------------------------------------------                  */

struct streamhandle
{
  int           id;
  char          letter;
  int           tnum;
  struct streamhandle* next;
};

void BuildAndSendMessage (struct streamhandle* sh)
{
  unsigned char buf[13];

  buf[0] = 'A';

  /* mode */

  if (CheckIfActive (sh->letter))
    {
```


Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

```
      buf[1] = 1;
      buf[2] = 0;
      buf[3] = 0;
      buf[4] = 0;
    }
  else
    {
      buf[1] = 2;
      buf[2] = 0;
      buf[3] = 0;
      buf[4] = 0;
    }

/*          transaction number - let's make byte ordering explicit!                  */

  buf[5] = (sh->tnum      ) & 255;
  buf[6] = (sh->tnum >>  8) & 255;
  buf[7] = (sh->tnum >> 16) & 255;
  buf[8] = (sh->tnum >> 24)      ;

/*          heartbeat                                                   */
/*          here, 1 second (1,000,000 microseconds, hex 0F4240)                      */

  buf[ 9] = 0x40;
  buf[10] = 0x42;
  buf[11] = 0x0F;
  buf[12] = 0x00;

  write(sh->id, buf, 13);
}

/*          ----------------------------------------------------------------         */
/*                               m a i n                                           */
/*                               -------                                           */
/*     on embedded platforms, do NOT define main in this program!        */
/*           ----------------------------------------------------------------        */

#ifndef VXWORKS
int main (int argc, char* argv [])
{
  return arbiter (argc, argv);
}
#endif

/*                    ----------------------------------------------------------------        */
/*                                      a r b i t e r                                      */
/*                                      ------------                                       */
/          define the main work routine of this program, which prepares to listen  */
/*          for connections, and then waits for connection attempts, disconnections  */
/*          and messages.                                                          */
/*                    ----------------------------------------------------------------        */

int arbiter (int argc, char* argv [])
{
  int                   s;                   /* connection socket         */
  short                 p;                   /* port number to use        */
  struct sockaddr_in    sa;
  struct servent        *sp;
  struct hostent        *hp;
  fd_set                rd_fds;
  struct streamhandle* streamchain = 0;
```

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

```
  if (argc != 3)
    {
      fprintf (stderr,
               "please give %s two arguments, a port and 0/1.\n",
               argv[0]
               );
      return 10;
    }

  board_mode = (argv[2])[0];

/*          find out info about the socket I am supposed to claim.          */

  if ((sp = getservbyname(argv[1],"tcp")) == NULL)
    {
      p = htons (atoi (argv[1]));
      if (p == 0)
        {
          fprintf(stderr, "service '%s' not registered on this machine\n",
                  argv[1]);
          return 10;
        }
    }
  else
    {
      p = sp->s_port;
    }

/*          port OK, so claim it and listen for connections.          */

  if ((hp = gethostbyname("127.0.0.1")) == NULL)
    {
      fprintf(stderr, "can't get local host info\n");
      return 10;
    }
  sa.sin_addr.s_addr = INADDR_ANY;
  sa.sin_port        = p;
  sa.sin_family      = hp->h_addrtype;

  if ((s = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0)
    {

/*                    failed to create socket. Give up.          */

      perror("Socket");
      return 10;
    }
  if (bind(s, &sa, sizeof sa) < 0)
    {

/*          failed to bind the port to the socket; give up, leaving O/S to          */
/*          tidy up the socket (lazy, but OK on UNIX, Windows).          */
      perror("Bind");
      return 10;
    }

  listen (s, 5);

/*                    ----------------                    */
/*                    M A I N   L O O P                    */
```


Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

```
/*                              ----------------                               */
/*          reacting both to connection attempts and                 */
/*          also to traffic over open connections.                         */

  while (1)
    {
      int                  n;
      char*                 c;
      int                  dbg = 0;
      struct sockaddr_in   isa;
      int                  i = sizeof isa;
      unsigned char        buf[1000];
      struct streamhandle* sh = streamchain;

/*                              -------                              */
/*                            W A I T                               */
/*                              -------                              */
/*        wait for something to happen; on return                    */
/*        from the select, rd_fds will indicate                             */
/*        which stream(s) something happened on.                     */

      FD_ZERO(&rd_fds);
      FD_SET(s, &rd_fds);

      while (sh != 0)
        {
          FD_SET(sh->id, &rd_fds);
          sh = sh->next;
        }
      n = select(FD_SETSIZE, &rd_fds, NULL, NULL, NULL);

      /* is there an incoming connection or message (or is a stream closing)?          */

      sh = streamchain;
      while (sh != 0)
        {
          if (FD_ISSET(sh->id, &rd_fds))
            {
              n = read(sh->id, buf, 1000);
              if (n <= 0)
                {

/*          we were told there was something on this stream - but it                        */
/*          is empty, so it must have been closed by the far end,                                   */
/*          or broken. Close our end of the connection.                                             */

                  close(sh->id);
                  if (streamchain==sh)
                    streamchain = sh->next;
                  else
                    {

/*          find the streamhandle that points at sh, and make its 'next' pointer skip over sh        */

                      struct streamhandle* sh2 = streamchain;
                      while (sh2->next != sh)
                        sh2 = sh2->next;
                      sh2->next = sh->next;
                    }
                  free (sh);
                  break;
```

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

```
                }
              else if (buf[0] == 'J')
                {
                  unsigned char* str = buf+9;
                  sh->letter = str[strlen(str)-1];
                  sh->tnum    = ((buf[8] << 24) |
                                 (buf[7] << 16) |
                                 (buf[6] <<  8) |
                                 (buf[5]       ));

/*        build up the response to the JCP, calling the                    */
/*        function CheckIfActive to determine if the              */
/*        JCP is to be told active or standby.                             */

                  BuildAndSendMessage (sh);
                }
              else
                {

/*        to assist debugging and to allow mode swapping,         */
/*        allow for a connection asking me to swap mode.          */
/*        Thus one can connect in via, say telnet (by a                    */
/*        command such as 'telnet localhost 7200') and                     */
/*        the first letter of a line is significant                        */

                  struct streamhandle* sh2 = streamchain;
                  board_mode =  buf[0];

/*        tell all JCPS about the changed mode                    */

                  while (sh2 != 0)
                    {
                      if (sh2->letter != 0)
                        BuildAndSendMessage (sh2);
                      sh2 = sh2->next;
                    }
                }
            }

/*        move onto the next streamhandle, to check for input     */

          sh = sh->next;
        }
/*        is someone trying to connect?                           */

      if (FD_ISSET(s, &rd_fds))
        {
          int v;
          if ((v = accept(s, &isa, &i)) < 0)
            {
              perror("Accept");
              return 10;
            }
          sh = (struct streamhandle *) malloc (sizeof (struct streamhandle));
          sh->id       = v;
          sh->letter  = 0;
          sh->next     = streamchain;
          streamchain = sh;
        }
    }
}
```


Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

AD SYSTRAN

APPENDIX: THE JCP-ARBITRATOR PROTOCOL

When is starts up, the JCP will attempt to connect to the posrt indicated by the arbitrator_port resouce; if this connection attempt fails, the JCP stops. The JCP then sends a single message to announce itself and to prompt the arbitrator to tell it (a) the operating mode (standby, master) in which it should be running, and (b) the heartbeat interval between messages from the JCP to the arbitrator.

The messages from JCP to arbitrator are all of the form:

'J'	1 byte
‹mode›	4 byte
‹trans›	4 byte
‹name›	C string, including null byte

The mode values indicate what mode the JCP thinks it ought to be running in:

0	Unknown
1	Master
2	Standby

The transaction number indicates the number of the most recently completed transaction. The name field is the service name of the JCP that has sent the message, NOT that of the DBMS process controlled by the JCP.

Note that the byte ordering is fixed, regardless of the platforms being used: thus the message represented in hex by 4a02000000320100003a3732303100 can be broken down as follows:

4a	the letter J
02000000	mode 2: standby
32010000	transaction 306 (0x132)
3a3732303100	":7201" as a C string

Likewise, the messages from the arbitrator to the JCP are also all of the same form:

'A'	1 byte
‹mode›	4 byte
‹trans›	4 byte
‹interval›	4 byte (micro-seconds)

The mode values are as before...

1	Master
2	Standby

... except the arbitrator is not expected to set the mode to 0 (unknown). The transaction number is the latest transaction number reported by the JCP to the arbitrator, so the JCP is able to recognise if the arbitrator is getting behind. The interval field defines (as a number of microseconds) the interval between heartbeat messages to be sent from the JCP to the RTRDB; a value of zero indicates that no heartbeat messages are to be sent by the JCP. As with messages from the JCP to the arbitrator, the least significant byte of the 4-byte fields appears before the other bytes of the field, so the message represented in hex by

4101000000320100004042 0f00 can be broken down as follows:

41	the letter A
01000000	mode 1: master
32010000	transaction 306, hex 00000132
40420f00	1 second (1,000,000 microseconds, hex 000f4240)

If the stream is closed or broken, the arbitrator is entitled to assume the JCP - together with the database it controls - has stopped. If no heartbeat messages are received for some time - say, twice the heartbeat interval - an arbitrator service could assume that the JCP is dead - however, where there are other means of determining liveness of the board, the arbitrator is at liberty to ignore the heartbeat messages or set the heartbeat interval to zero or to a very large value. The only requirements on the arbitrator are

(a) that it is to confine itself to the form of messages described above,

(b) it must respond to the initial message from the JCP,

(c) it must avoid a build-up of unread messages,

(d) it should send a message confirming the heartbeat interval, last known transaction number and required state at approximately the heartbeat interval (pehaps in response to the message from the JCP) so that the JCP knows it is still alive; and,

(e) it must avoid closing the stream except where essential (for the JCP can treat this as a signal to stop immediately, though the JCP is entitled to reopen the connection if it wishes).

The arbitrator can send its messages whenever it wants - it does not to have to wait for a heartbeat from the JCP, for example, in order to signal a change in the operating mode ■

Nigel Day, Ph.D. Since gaining his PhD from Cambridge University Nigel has been involved in a wide range of leading edge software design and consultancy projects. Experience includes compiler and operating system design and implementation, and computer security research. Nigel is currently Technical Director of Polyhedra Plc., serving on the Board of Directors, and as Vice President of Engineering for Polyhedra, Inc.

David C. Morse. Mr. Morse has been working in various marketing and management positions in the embedded database industry for the past 15 years. As the founder of database tools and technology company in 1991, his experience with a variety of database and RTOS environments has allowed him to produce articles and technical papers on various subjects in this industry. He is the founder of Polyhedra, Inc. the US subsidiary of UK-based Polyhedra, Plc.

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)