



Raima Database Manager 12.0

Mirroring and High Availability User Guide

Trademarks

Raima[®], Raima Database Manager[®], RDM[®], RDM Embedded[®] and RDM Server[®] are trademarks of Raima Inc. and may be registered in the United States of America and/or other countries. All other names referenced herein may be trademarks of their respective owners.

This guide may contain links to third-party Web sites that are not under the control of Raima Inc. and Raima Inc. is not responsible for the content on any linked site. If you access a third-party Web site mentioned in this guide, you do so at your own risk. Inclusion of any links does not imply Raima Inc. endorsement or acceptance of the content of those third-party sites.

Contents

Contents	i
Introduction	1
Mirroring Architecture	2
Mirroring Process	2
Synchronous Mirroring	6
Mirroring Utilities	7
Mirroring Example	7
Dbmirror Usage	10
Installation as Service or Daemon Process	11
Dbget Usage	12
Mirroring Setup	14
Advanced Topics	17
Differences Between Master and Slave	17
In-Memory to On-Disk	17
Slave Database Setup	17
Synchronization Issues	18
Mirroring and HA API Functions	20
d_dbmir_init	21
d_dbmir_connect	23
d_dbmir_disconnect	25
d_dbmir_start	27
d_dbmir_stop	29
d_dbmir_term	30
HA Synchronous Notifications	31
Synchronous Callouts	31
Error Conditions	31
Non-error (Status) Conditions	32
Switchover	32
HA API Reference	33
ha_login	34
ha_logout	35
ha_status	36
ha_quiesce	37
ha_wakeup	38
ha_isTransActive	40
RDM Mirroring Example	42
Application Components	43
mirMasterExample	43

mirSlaveExample	45
Application Architecture	48
Model	48
CMirModel.h	48
CMirModel.cpp	48
CMirMasterModel.h	48
CMirMasterModel.cpp	48
CMirSlaveModel.h	48
CMirSlaveModel.cpp	49
View	49
CMirView.h	49
CMirViewConsole.h	49
CMirViewConsole.cpp	49
CMirMasterViewConsole.h	49
CMirMasterViewConsole.cpp	49
CMirSlaveViewConsole.h	49
CMirSlaveViewConsole.cpp	49
Controller	50
CMirController.h	50
CMirController.cpp	50
CMirMasterController.h	50
CMirMasterController.cpp	50
CMirSlaveController.h	50
CMirSlaveController.cpp	50
Application Database	51
Mirroring Utilities	52
dbmirror	53
dbget	55
dbrep	57
Appendix	59
Document Root (DOCROOT)	60
Database Name Specification (db_namespec)	61
Supported Transport Types (tfs_spec)	61
Default TFS Addresses	61
Database Name Specifications	62
Union Database Name Specifications	62
Antiquarian Bookshop Database	63
Replication/Mirroring Parameters (DBREP_CONNECT_PARAMS)	67
Replication/Mirroring Initialization Parameters (DBREP_INIT_PARAMS)	70
Shared Memory Transport:	72
Opening Slave Databases from Programs	73

Index	74
-------------	----

Introduction

RDM supports two unique but related methods for maintaining nearly real-time copies of databases in additional locations, referred to as *mirroring* and *replication*.

Both mirroring and replication use the same terminology for the roles of databases: the original, updateable database is called the *master*. From one master database, one or more *slave* copies can be created and dynamically maintained. The terminology comes from the idea that the master database controls the generation of data, and the slaves respond only when changes have been made on the master.

To mirror a database is to create a byte-for-byte copy of a database at a different location. Mirroring is different than copying or backing up a database in that a mirror database is updated at the same time as the original database (synchronous) or as quickly as possible after the original (asynchronous) database is updated. Page images from the master are applied to the slave (s) to implement mirroring.

There are three main purposes for mirroring:

1. To maintain another copy of a database for safe-keeping. The backup copy may be an on disk copy of an in-memory master database.
2. To offload reading of a database to another computer.
3. To be prepared to switch processing to another computer if the primary computer fails. This is often referred to as a Highly Available (HA) database.

Mirroring Architecture

The RDM Db Engine always generates change log files as part of the process of committing a transaction. These log files can also be used, after being committed to the original database, as the basis for updating mirrors of the original database. Whenever a log is written to the original, the log is copied to locations where mirrors exist, and then are applied to the mirror databases. This re-use of log files means that the databases are byte-for-byte identical, and therefore must be used on computer architectures that have the same integer byte ordering. The use of page images in the log files means that transaction integrity is maintained and the recovery mechanism is very reliable.

The mirroring process is primarily *asynchronous*, meaning that transactions applied to the master database will be propagated to the slaves as soon as possible. A *synchronous* mirroring process forces an acknowledgment from the slave that the transaction has been successfully applied prior to allowing the master to be updated again. Synchronous mirroring may be required in situations where the master and slave *must* be kept identical at all times, but it comes with a performance price. A restriction on synchronous mirroring is that a master may only have one synchronous slave, even if multiple slaves exist - all additional slaves are asynchronous.

Asynchronous mirroring is designed to optimize the process of maintaining extra copies of a database, allowing the master to be updated at a maximum rate without being impeded by the performance of the slaves. There can be "bursts" of changes to the master, and time afterward for the slave(s) to catch up. It also provides for intermittent connections, meaning that a slave database may be "offline" for a period of time and can get caught up (sometimes called a "synchronize" operation) when it reconnects. Log files are kept with the master database for the purpose of allowing intermittent connections to catch up. The time a log file is kept with the master can be configured such that it is deleted after a certain age. Logs can also be deleted, oldest first, if the total space taken by them exceeds a configurable size.

A slave may be on the same computer as the master (simply a different location), or on a different computer. The network connection can be a fast LAN within an office, or Internet from anywhere in the world (requires connectivity).

A utility named `dbmirror` is used to move log files from the master to the slave database. A slave database configuration will also run `dbmirror` if it is a mirrored slave, or `dbrep` if it is a replicated RDM slave, or `dbrepsql` if it is a replicated SQL slave.

Mirroring Process

Figure 12-1 below shows the generalized view of the mirroring process.

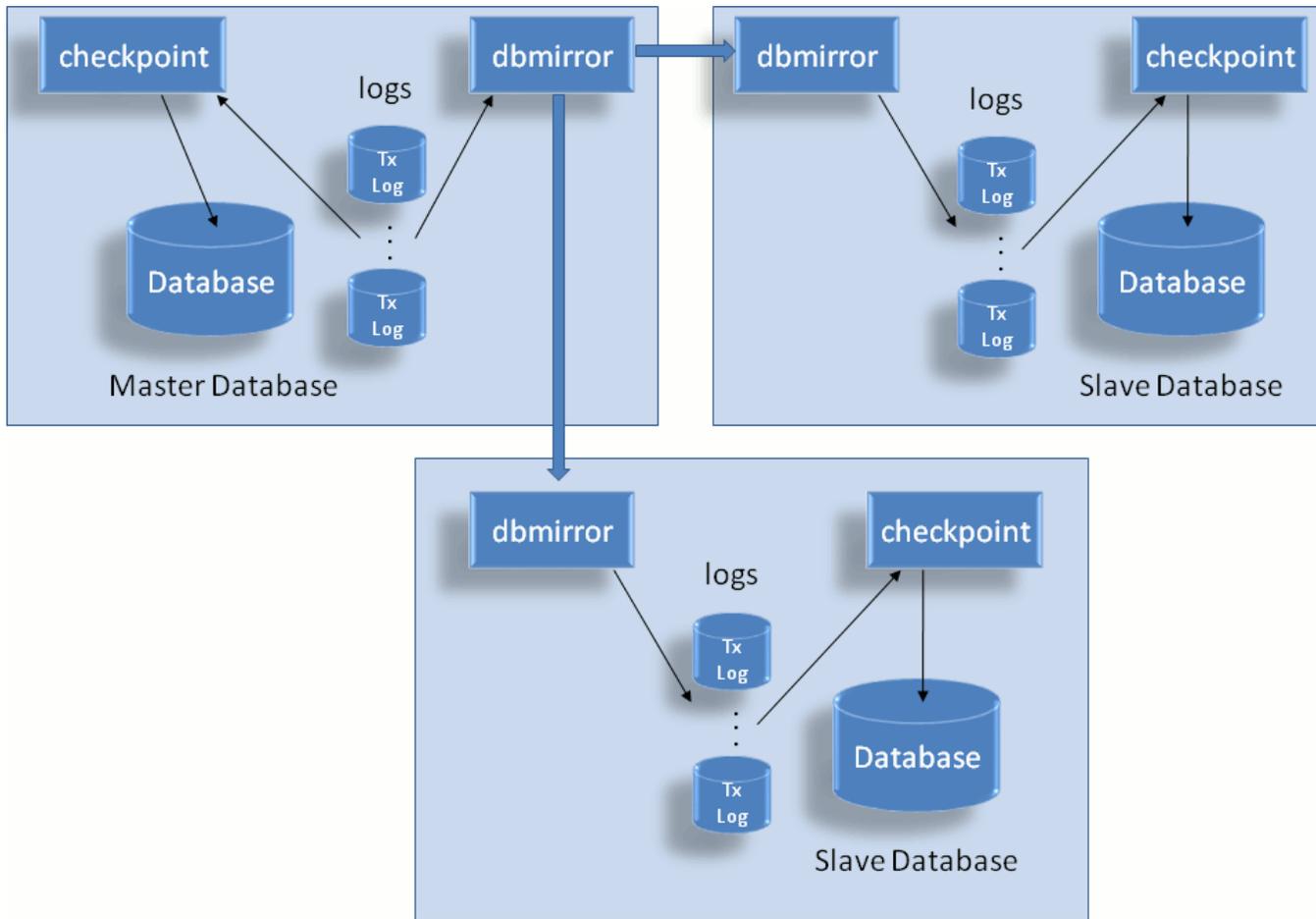


Fig. 12-1 Mirroring Architecture

The mirroring process is as follows:

1. Every transaction is committed to a master database by creating a log file (containing modified page images) in the database's directory. The log files are named after the transaction number they represent. Transactions are numbered serially with no gaps (see discussion about naming below).
2. The checkpoint process scans the directory for log files. When one or more new log files are found, they are "checked" into the database. The entire process is safe and repeatable so that there will be no loss of data.
3. To facilitate mirroring, the checkpoint process will not delete used log files, but will rename them so that they can be found by the `dbmirror` process.
4. The slave `dbmirror` process requests the "next" log from the master `dbmirror` process. When it receives it, it sends it to the local TFS, if a TFS is running, for normal transaction processing. If a TFS is not running together with the slave `dbmirror`, there may or may not be a checkpointing process running. If there is, the presence of the log causes the checkpointer to copy these changes into the slave database. This is repeated forever, or until the `dbmirror` slave process is terminated.
5. The master `dbmirror` process searches the database directory for log files and responds to slave `dbmirror`(s) when certain log files are requested.
6. The master `dbmirror` can respond to any number of slave `dbmirror`(s).

Certain special conditions exist:

1. When a *first-time request* for a database comes from a slave `dbmirror`, the entire database must be copied to the slave. Then the normal process of applying incremental changes through log files applies.

2. The master checkpointer will have rules by which it deletes or cleans up log files:
 - a. If mirroring is disabled, the log files are deleted immediately after they are checkpointed.
 - b. If mirroring is enabled, the checkpointer will rename them such that `dbmirror` will still find them, but they will not be checkpointed again.
 - c. The checkpointer will be given time and/or disk space rules for log file cleanup. If the log files are beyond a certain age, they will be deleted, or if the total space on disk taken by the log files exceeds a given number, log files will be deleted.
3. If a slave `dbmirror` requests a log that no longer exists (because of rule c above), it will be necessary to treat the request as a first-time request and send the entire database again. This situation can arise when a slave has an intermittent connection, or only connects to "synchronize" the database. There must be a balance between how long a slave may be disconnected and how long the log files will be kept by the master.

Transaction log files are transient. They are kept as long as they are useful. When mirroring is disabled, their usefulness ends as soon as their contents are written to the database. The name of a log file represents its current state. The list below shows the *life cycle* of a transaction log file:

1. **Creation** - while being created, the log file is opened in the database's directory and named `dbuserid.prelog`, where `dbuserid` has been automatically assigned by the TFS, or has been requested through the `d_dbuserid` call. Logs being simultaneously created by different RDM Db Engines will always have different names.
2. **Commitment** - Once a prelog file has been completely written, it is *synced*, meaning that all of its contents are written and committed to disk. Should the computer fail after this moment, the contents on the disk will be correct and complete. After the sync operation, the file is renamed. The renaming makes it visible to the checkpointing process. If a computer failure occurs after the sync but before the rename, then the transaction will not be considered to be committed, and will never be found in the database after the TFS restarts. In this situation, the RDM Db Engine submitting the transaction will not be notified that the transaction was successfully committed. The name of the log file will be the transaction ID (an 8 hexadecimal digit number) with a `.log` type, for example, `0000015d.log`. The log files are named strictly sequentially, and are always applied to the database in numeric order.
3. **Checkpoint** - The checkpoint process will find and apply the contents of a log file to the database on a regular basis. Once a log file (and any others that may be batched together) has been written to the database and the database files have been synced, the log files are no longer required by this TFS. If mirroring is disabled, the log file(s) are deleted now. If not, the log files are renamed again by adding a `.arch` suffix, for example, `0000015d.log.arch`. The `dbmirror` process looks for transactions in numeric order.
4. **Cleanup** - If archived log files exist, there should be time or space restrictions that determine when they can be deleted.

See Figure 12-2 below:

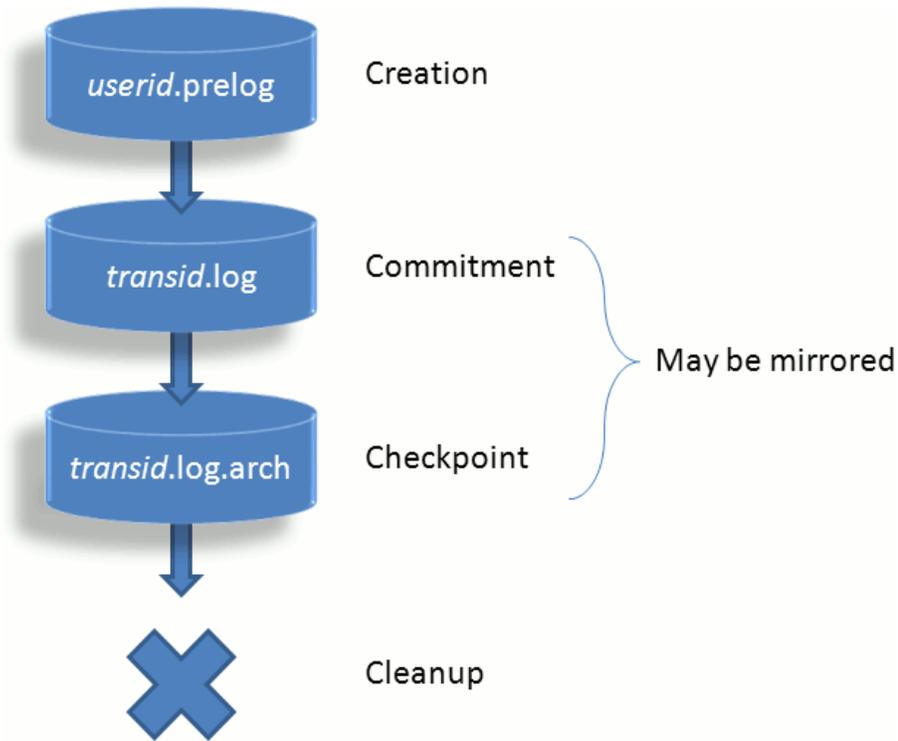


Fig. 12-2 Transaction Log Life Cycle

Synchronous Mirroring

By default, mirroring is asynchronous. Asynchronous operation is the fastest and most flexible mode for mirroring. Synchronous mirroring forces a program's `d_trend` call to wait until the slave `dbmirror` replies to the master `dbmirror` that the transaction has been committed in a durable manner on the synchronous slave database.

When a synchronous slave database is being used, the implied intent is that the slave database may be used as a primary database should anything happen to the master database, and it is essential that any transaction that has committed in the master can also be found in the slave. Asynchronous mirroring cannot guarantee this.

Because of the usage scenario for synchronous mirroring, once a synchronous mirror database has been established, the master database may not be updated unless the mirror is also being updated. In other words, the slave `dbmirror` utility becomes a required piece in updating the master database. If it is not running, then the master database is read-only. This enforces the notion that once a RDM Db Engine receives a successful "commit" message, it is committed on both master and slave.

Once a database is being synchronously mirrored, it will remain synchronously mirrored until this mode is deliberately cleared, even if the slave `dbmirror` utility stops and restarts.

Only one slave synchronous mirror may be established. Any other slaves must be asynchronous.

Mirroring Utilities

Two utilities exist to implement mirroring. The `dbmirror` utility introduced in the section above and another, `dbget`, which is used to initiate the mirroring process.

Follow the steps below to mirror a database:

1. Run `dbmirror` on the master TFS computer.
2. Run `dbmirror` on the slave computer.
3. Run `dbget` on the slave computer. Identify the master database and the slave `dbmirror`.

Step 3 is performed for each separate database that is to be mirrored.

The `dbmirror` utility spawns threads for each database connection. The master `dbmirror`'s threads respond to requests from the slave utilities, which will ask for the *next* transaction log file. The slave `dbmirror` keeps track of the last log file it received. The slave `dbmirror` may also be re-started after a termination or disconnection. It will determine the transaction ID of the last received log file and begin again (upon request by `dbget`) by requesting the next log file.

The `dbmirror` utility may serve as both master and mirroring slave at the same time. It may be a slave to create a mirror of a database for which it is also the master, for yet another slave requesting the same database. Thus a chain or tree of mirroring can be set up if necessary. It also can serve as master for multiple slaves of the same database, and for multiple databases.

Mirroring Example

The following getting started example sets up mirroring from a master database to a single slave database using on the same computer using the TCP transport on MS-Windows. This example uses the [Antiquarian Bookshop Database](#) SQL database example located in the `GettingStarted\examples\sql_db` directory.

You first need to create and load the `bookshop` database. Open a command window and create a directory that will serve as the *docroot* for the master bookshop database. This example will assume that it is called `\rdm\master`.

Now change to the `GettingStarted\examples\sql_db` directory and run `rdmsql` to create and load the `bookshop` database as shown below.

```
rdmsql -d c:\rdm\master -b bookshop.sql
rdmsql -d c:\rdm\master -b impdb_bookshop.sql
```

The results from a successful execution of these two commands is shown below.

```
c:\rdm\GettingStarted\examples\sql_db>rdmsql -tfs TFSR -b bookshop.sql

Raima Database Manager Interactive SQL Utility
Raima Database Manager 12.0.0 Build 1546 [8-6-2013] http://www.raima.com/
Copyright (c) 2013 Raima Inc., All rights reserved.

Document Root: c:\rdm\master\
RDMSQL completed

c:\rdm\GettingStarted\examples\sql_db>rdmsql -tfs TFSR -b impdb_bookshop.sql

Raima Database Manager Interactive SQL Utility
```

```
Raima Database Manager 12.0.0 Build 1546 [8-6-2013] http://www.raima.com/  
Copyright (c) 2013 Raima Inc., All rights reserved.
```

```
Document Root: c:\rdm\master\  

```

```
**** Importing data into the BOOKSHOP database
```

```
Importing data into acctmgr table...  
Importing data into patron table...  
Importing data into author table...  
Importing data into book table...  
Importing data into related_name table...  
Importing data into genres table...  
Importing data into subjects table...  
Importing data into genres_books table...  
Importing data into subjects_books table...  
Importing data into note table...  
Importing data into note_line table...  
Importing data into sale table...
```

```
**** BOOKSHOP database import complete
```

```
RDMSQL completed
```

Issue the following command to launch the `dbmirror` utility for the master database. Note that `dbmirror` here includes the TFS.

```
start dbmirror -trans tcp -p 21553 -d c:\rdm\master
```

A new command window should open with a display similar to the following:

```
DB Mirror Utility  
Raima Database Manager 12.0.0 Build 1546 [8-6-2013] http://www.raima.com/  
Copyright (c) 2013 Raima Inc., All rights reserved.
```

```
Document Root: c:\rdm\master\  

```

```
Setting up for listening...
```

```
TCP/IP: Port = 21553
```

```
Ready!
```

```
Document Root: c:\rdm\master\  

```

```
Setting up for listening...
```

```
TCP/IP: Port = 21553 (+1)
```

```
Ready!
```

Next, create the `docroot` directory which will contain the mirrored database. In this example, the directory is `c:\rdm\mirror`. The slave `dbmirror`, which also includes the TFS, is launched by the following command.

```
start dbmirror -trans tcp -p 21555 -d c:\rdm\mirror
```

A window similar to the one for the master `dbmirror` will be opened. The next command runs the `dbget` utility which will initiate mirroring of the master `bookshop` database (identified by its TFS specification) to the mirrored copy. Note that this `dbget` communicates with the slave's `dbmirror` (identified by port number 21555).

```
dbget -b -trans tcp -p 21555 tfs-tcp://:21553/bookshop
```

The output from a successful execution of this command is shown below.

```
DBGET Utility
Raima Database Manager 12.0.0 Build 1546 [8-6-2013] http://www.raima.com/
Copyright (c) 2013 Raima Inc., All rights reserved.

Began slave id 1
```

In addition, the slave's dbmirror window should have also displayed the following lines.

```
Slave server: Connected to local TFS
Slave server: Connected to master server
```

You can check out the mirroring results and behavior using rdmsql. Open two separate command windows and run rdmsql as follows in each.

```
rdmsql -tfs TFSR
```

In the first issue the commands shown in **red** in the following script. This opens and queries the master bookshop database.

```
001 rdmsql: .c 1
*** using statement handle 1 of connection 1
001 rdmsql: open "tfs-tcp://:21553/bookshop";
002 rdmsql: select * from acctmgr;

mgrid   name                hire_date                commission
ALFRED  Kralik, Alfred      1997-07-02                0.025
AMY     Zonn, Amy           1994-07-06                0.025
BARNEY  Noble, Barney       1972-05-08                0.035
FRANK   Doel, Frank         1987-02-13                0.030
JOE     Fox, Joe            1998-12-18                0.025
KATE    Kelly, Kathleen     1998-12-18                0.025
KLARA   Novac, Klara        1990-01-02                0.025
```

In the second rdmsql command window enter the commands in **red** shown below which opens and queries the mirrored/slave bookshop database. Note that slave databases can only be opened in read only mode.

```
001 rdmsql: .c 1
*** using statement handle 1 of connection 1
001 rdmsql: open "tfs-tcp://:21555/bookshop" read only;
002 rdmsql: select * from acctmgr;

mgrid   name                hire_date                commission
ALFRED  Kralik, Alfred      1997-07-02                0.025
AMY     Zonn, Amy           1994-07-06                0.025
BARNEY  Noble, Barney       1972-05-08                0.035
FRANK   Doel, Frank         1987-02-13                0.030
JOE     Fox, Joe            1998-12-18                0.025
```

Mirroring Utilities

```
KATE    Kelly, Kathleen      1998-12-18      0.025
KLARA   Novac, Klara         1990-01-02      0.025
```

Back in the first command window, issue the following SQL statements which add a new account manager to the `acctmgr` table.

```
003 rdmsql: insert into acctmgr values "JOHN", "Doe, John", date "2013-08-13", 0.05;
*** 1 rows affected
004 rdmsql: commit;
005 rdmsql:
```

Then, from the second command window enter the query shown below which should now include a row for John Doe.

```
003 rdmsql: select * from acctmgr;

mgrid  name                hire_date          commission
ALFRED  Kralik, Alfred      1997-07-02         0.025
AMY     Zonn, Amy           1994-07-06         0.025
BARNEY  Noble, Barney       1972-05-08         0.035
FRANK   Doel, Frank         1987-02-13         0.030
JOE     Fox, Joe            1998-12-18         0.025
JOHN    Doe, John           2013-08-13         0.050
KATE    Kelly, Kathleen     1998-12-18         0.025
KLARA   Novac, Klara       1990-01-02         0.025
```

As you can see, the row for John Doe has been successfully copied from the master `bookshop` database to the slave.

Issue a `.q` command to terminate each of the `rdmsql` programs. Then issue the following `dbget` command to end mirroring. After that, the two `dbmirror` windows can be closed to terminate them.

```
dbget -e -trans tcp -p 21555 tfs-tcp://:21553/bookshop
```

Dbmirror Usage

The `dbmirror` utility is run one time, even when there are two or more master databases that will be mirrored. It will start one thread for each mirrored master database.

The `dbmirror` utility will accompany a TFS that is controlling access to master database(s). If `dbmirror` is to operate as a slave only, the TFS is optional, but when the TFS is not running on a slave computer, only log files will be created in the database subdirectory under the document root. These log files can serve as a backup of a database should the database need to be reconstructed.

The `dbmirror` utility is itself a server that listens on its own separate port. However, when starting or referring to this utility, the *anchor port* (the port the TFS is using) is used. For those who need to make sure the proper ports are open in a firewall, this utility's port is the anchor port plus 1.

```
dbmirror [-d PATH] [-tfs type] [-trans trans_list] [-p port/name]
         [-v] [-start|-stop|-query|-install exepath|-uninstall]
         [-serviceuser username] [-servicepw password] [-nodisk]
```

```

-d           = PATH location of server document root (absolute, or relative
              to current directory)
-tfs type    = Specify the TFS type to use (TFST, TFSR, or TFSS) [default = TFST]
-trans       = Specifies a comma separated list of transport to listen on.
              Options are "tcp" and "shm" (default is "shm,tcp")
-p           = Specifies the server anchor port (TCP/IP) or anchor name
              (non-TCP-IP) for connections. The default anchor port is
              21553 and the default anchor name is RDM_21553. If a number
              is specified it will be used for the TCP/IP port and the
              server name will be RDM_port. If a string (non-number) is
              specified it will be used as the anchor server name and the
              TCP/IP transport will not be initialized.
-v           = Verbose output
-start       = Start the dbmirror (run as a daemon on Unix)
-stop        = Stop the dbmirror
-query       = Query the execution status of the dbmirror
-install     = Install the dbmirror as a service at the specified path
-uninstall   = Uninstall the dbmirror as a service
-serviceuser = User account to install service as (only valid with -install
              option)
-servicepw   = User account password to install service as (only valid with
              -install option and required with -user option)
-nodisk      = Don't use any disk I/O
    
```

The `-d` option should identify the same document root directory as the one being used by the TFS. If this option is not specified, the current directory is used.

The `-v` option will print a log of mirroring activity. Use this to verify that your configuration is working, but leave it off for normal operation because it will interfere with performance.

The `-nodisk` option is used when databases are defined to be inmemory and you want the Replication utility to keep its files only in memory also.

Installation as Service or Daemon Process

The following options are available for the `dbmirror`, `dbrep` or `dbrepsql` utilities for the purpose of starting them in the background:

```

dbmirror [-start|-stop|-query] [-stdout filename]

-start = Start utility as background process
-stop  = Shut down the utility
-query = Determine if the utility is running in the background or not
    
```

The `-stdout filename` option is used when `stdout` is not appropriate, such as when the utility is started in the background. All error and warning output will be written to `filename`.

The following options function on Windows systems in order to treat the utility as an automatic Windows service:

```

dbmirror [-install exepath|-uninstall]

-install exepath = Install utility as a service. The exepath is the directory
                  containing the utility, or directory\utility.EXE
-uninstall       = Uninstall this utility
    
```

When installed as a service, use the Windows Services Manager to start and stop, rather than the command-line options above. By default, the service will be automatic, and will be started when it is installed.

Dbget Usage

The `dbget` utility is used to make a request to a slave process (`dbmirror`, `dbrep` or `dbrepsql`) to initiate mirroring or replication for a particular database. The slave process must be running before `dbget`'s notification can be processed. For every database being mirrored or replicated, `dbget` must be invoked once. `dbget` is also used to cleanly stop mirroring or replication. Mirroring or replication may be started up again after it has been stopped.

```
dbget [-u DBUSERID] [-s SLAVE_HOSTNAME] [-trans trans_list] [-n name]
      [-p N] [-alias alias] [-id numeric_id] [-nocopy] [-sync]
      [-unsync] [-b] [-e] [-override_inmem] [-key key]
      [-dsn dsn;user;pswd] [-oracle|mssql|-mysql] database URL

-u          = DBUSERID to use during transactions
-s          = HOSTNAME of slave DBMIRROR (default is localhost)
-trans     = Specifies a comma separated list of transport to listen
            on. Options are "tcp" and "shm" (default is "shm,tcp")
-n         = Specifies the anchor name of the slave TFS for non-TCP/IP
            connections (default is "21553")
-p         = Specifies the anchor port of the slave TFS for TCP/IP
            connections (default is 21553)
-alias     = Specifies the database alias to use on the slave
-id        = [Required] Specifies the unique id for this database
            connection
-nocopy    = Don't ever copy the database (data can be lost)
-sync     = Mirrored database is synchronous
-unsync   = End persistent synchronous mirroring
-b         = Begin mirroring (default action)
-e         = End mirroring
-override_inmem = Force INMEMORY database files from master to be disk based
            on the slave
-key key   = Specify the encryption key for the database [algorithm:]passcode.

            The valid algorithms are xor, aes128, aes192, and aes256. If an
            algorithm is not specified the default is aes128
-dsn       = Specify a dsn for dbrepsql
-oracle    = Slave dbrepsql connects to Oracle server
-mssql     = Slave dbrepsql connects to Microsoft SQLServer
-mysql     = Slave dbrepsql connects to MySQL server
database URL = Name and location of master database TFS. Default domain
            is localhost. Default port is 21553.
```

Together, the `-s` and `-p` options identify the location of the slave utility that will "get" a database. When not specified, `localhost:21553` is used, which are the defaults used by the `dbmirror` utility.

It is not necessary to use `-u` to provide a `DBUSERID`. This option may be specified in order to distinguish this utility from other RDM Db Engines when monitoring system activity.

The `-sync` and `-unsync` options place a slave mirror database into or out of synchronous mirroring mode, respectively. If another synchronous mirror for this database already exists, the `-sync` request will be rejected.

A single synchronous slave can be started as follows:

```
dbget -sync -b sales@tfs.raima.com
```

To deliberately stop synchronous mirroring, dbget must be used:

```
dbget -unsync -b sales@tfs.raima.com
```

The `-id` option specifies a unique id for this database connection. In use cases where a single slave is mirrors from multiple masters this enforces the uniqueness of the master mirror or replication logs.

The `-nocopy` option disables the behavior in which the slave mirroring process copies the entire database from the master on a *first-time request* or upon requesting a mirror log id which no longer exists on the master. With this option the master transmits the transaction id of the oldest mirror or replication log. This option is provided for use cases where it is acceptable to not maintain complete duplication of the database content. For example, a master database on a multiple low power sensor based devices that stores status information in a circular table, replicating to an aggregation of devices statuses on a single slave database.

Mirroring Setup

Default Options with Masters and Slaves on Different Computers

When only one TFS is run on each computer, the default ports should be used, and may be left off the command-line. The default port for `tfserver` is 21553. The `dbget` utility refers to anchor port 21553 to connect to `dbmirror`. The following example assumes that two Windows and two Linux machines are being used.

Master 1 (Windows, `master1.tfs.raima.com`), controlling sales and mkt databases:

```
start tfserver -d \RDM\databases.win32
start dbmirror -tfs tfsr
```

Master 2 (Linux, `master2.tfs.raima.com`), controlling tims database:

```
tfserver -d /RDM/databases.lnx &
dbmirror & -tfs tfsr
```

Slave 1 (Windows, `slave1.tfs.raima.com`):

```
start tfserver -d \databases.win32
start dbmirror -tfs tfsr
dbget -b mkt@master1.tfs.raima.com
dbget -b sales@master2.tfs.raima.com
```

Slave 2 (Linux, `slave2.tfs.raima.com`):

```
tfserver -d /home/databases &
dbmirror -tfs tfsr &
dbget -b mkt@master1.tfs.raima.com
dbget -b tims@master2.tfs.raima.com
```

Using dbget From a Different Computer

The `dbget` utility will commonly be used from within the context of the slave database, but it doesn't need to be. It simply needs to address the correct `dbmirror` slave process. Assuming that the master and slave processes have been set up as above, then the following `dbget` invocations will achieve the same results and will work from any of the computers:

```
dbget -b -s slave1.tfs.raima.com mkt@master1.tfs.raima.com
dbget -b -s slave1.tfs.raima.com sales@master1.tfs.raima.com
dbget -b -s slave2.tfs.raima.com mkt@master1.tfs.raima.com
dbget -b -s slave2.tfs.raima.com tims@master2.tfs.raima.com
```

Non-Default Options with Masters and Slaves on the Same Computer

This example shows the opposite extreme from the first example, where all masters and slaves are now to operate on the same computer. We will not use default ports for anything, although they will all use `localhost` as the domain. All `tfserver` and `dbmirror` processes must use different ports, and the `dbget` utility must refer to the correct ports. The following table shows how this will be configured:

Table 12-1 Same-computer Mirroring Configuration

Role	Utility	Document Root	Port
Master 1	tfserver	c:\RDM\databases.win32	1730
Master 1	dbmirror	c:\RDM\databases.win32	1730
Master 2	tfserver	c:\databases	1830
Master 2	dbmirror	c:\databases	1830
Slave 1	tfserver	d:\databases.win32	1840
Slave 1	dbmirror	d:\databases.win32	1840
Slave 2	tfserver	d:\home\databases	1940
Slave 2	dbmirror	d:\home\databases	1940

Master 1, controlling `sales` and `mkt` databases:

```
c:
cd \RDM\databases.win32
start tfserver -p 1730
start dbmirror -p 1730
```

Master 2, controlling `tims` database:

```
c:
cd \databases
start tfserver -p 1830
start dbmirror -p 1830
```

Slave 1:

```
d:
cd \databases.win32
start tfserver -p 1840
start dbmirror -p 1840
dbget -b -p 1840 mkt@localhost:1730
dbget -b -p 1840 sales@localhost:1730
```

Slave 2:

```
start tfserver -p 1940 -d d:\home\databases
start dbmirror -p 1940 -d d:\home\databases
dbget -b -p 1940 mkt@localhost:1730
dbget -b -p 1940 tims@localhost:1830
```

Using dbget From a Different Computer, Again

This example corresponds to the example above, where all of the utilities are being run on the same computer (say, `tfs.raima.com`). The `dbget` utility can be run from anywhere, and in this case, it must specify the correct ports (for the `dbmirror` process and master database) as well as the domain:

```
dbget -b -s tfs.raima.com -p 1840 mkt@tfs.raima.com:1730
dbget -b -s tfs.raima.com -p 1840 sales@tfs.raima.com:1730
dbget -b -s tfs.raima.com -p 1940 mkt@tfs.raima.com:1730
dbget -b -s tfs.raima.com -p 1940 tims@tfs.raima.com:1830
```

Advanced Topics

Differences Between Master and Slave

Rules for creating safe differences between the master and slave databases are discussed in the following sections. All techniques involve creating a database dictionary (DBD) file that is different, but compatible between the master and slave.

Frequently, when a slave database is created, its location will be empty, and the DBD from the master will be used. But once a DBD file has been created and placed into the slave database location, future copies of the database will not overwrite the DBD. All of the techniques described below depend on the ability to create a slave DBD that will not be overwritten by the master DBD.

In-Memory to On-Disk

A common reason to mirror or replicate is to create a permanent, safe copy of a database while maintaining an in-memory master database for performance reasons.

To create a slave database that is on-disk, regardless of the storage media of the master database (either in-memory or on-disk), use the `-override_inmem` option to `dbget`. (where replication is to an RDM slave). When this option is specified, the initial copy of the master database to the slave location will also alter the DBD such that the slave database is marked as on-disk. Subsequent copies of the database (because logs get out of range) will not cause this DBD to be overwritten, so the condition is permanent unless the slave database is completely destroyed.

Note that if additional changes must be made, the `-override_inmem` option will not effect this change - it must be made part of the original slave DDL as shown below.

Slave Database Setup

This section summarizes the steps needed to initiate and maintain mirroring or replication in its various permutations.

Setup 1 - Normal Mirroring, RDM Slave

Assuming that the master TFS is running on `tfs.raima.com:21553` and the master Mirroring Utility is referencing the TFS at `tfs.raima.com:21553`, normal mirroring setup of the `mkt` database in the slave environment is as follows:

1. Start TFS.

```
tfserver -d /users/RDM/databases
```

2. Start Mirroring utility.

```
dbmirror -d /users/RDM/databases
```

3. Initiate Mirroring.

```
dbget -b mkt@tfs.raima.com:1730
```

Setup 2 - Mirroring, Override In-Memory, RDM Slave

Assuming that the master TFS is running on `tfs.raima.com`, the master database is in-memory, and the master Mirroring Utility is referencing the TFS at `tfs.raima.com`, mirroring setup of the `mkt` database in the slave environment is as follows:

1. Start Mirroring utility.

```
dbmirror -d /users/RDM/databases
```

2. Initiate Mirroring. Add the command-line option to make slave store database on disk.

```
dbget -override_inmem -b tfs-tcp://tfs.raima.com/mkt
```

Synchronization Issues

The mirroring feature will fetch entire databases when the master and slave are unable to synchronize. During the initial creation of a slave database, the database is copied from the master to establish a starting point, followed by the application of log files to keep the slave current. This section discusses what is going on when the attempts to remain current fail for some reason.

Mirroring keeps track of the last log file that they received and applied to the slave database. The operation of the slave utilities is to continuously request the "next" log from the master. If the "next" log is not available, the utility must refresh the database. There are differences in the way this refresh occurs, which will be described below.

The master database directory stores all mirroring logs which are available for request by slaves. Because it is not practical to store all log files for all time, there are configurable limits on the age of log files, or the total storage space taken by log files. When they are removed, they are always removed beginning with the oldest.

There are just a few reasons why slave databases get "out of sync:"

1. The slave has been disconnected for a period of time such that upon reconnection, the master has already cleaned up the next log in the sequence.
2. The slave is unable to keep up with the updates made by the master, so that even though the slave is connected, it is so far behind the master that the master has started deleting logs still needed by the slave.
3. For some reason, the slave's database directory has been "cleaned up" manually, destroying the records of which log is the next log. (Manual maintenance of the database directories is not recommended).
4. Manual cleanup of the logs in the master database can also create an out-of-sync condition.

Synchronization and Mirroring

The action taken by the slave `dbmirror` utility whenever it cannot obtain the "next" log file is always the same as the initial copy of the database. It will learn that the log file it is requesting is not available, so it will begin requesting pieces of the database instead. The database that is transferred to the slave is stamped with the ID of the last transaction that was applied to it, so following the successful copy of the database; the slave requests the log for transaction ID+1.

Balance

Some distributed database applications may be designed to be intermittently synchronized. For example, a corporate contact list may have very few changes to it (perhaps 10 per week). If an employee maintains a slave of this database on their laptop,

then it may be necessary to connect to the master every week or two. It will be easy to configure the master database to clean up log files that are two months old.

Another example, like the [Market](#) example, involves a database that has repeated changes to the same records. Very quickly, the size of the database can be overshadowed by the size of the change logs. At a certain point, it makes more sense to copy the entire database to slaves rather than copy all of the change logs. There is no formula for this, but the controls are the `LogFileAge` and `LogFileSpace` parameters.

Another balance issue is the relative speed of the master updates vs. the slave consumption of the updates. This system is made to handle bursts of updates on the master with pauses that allow the slave(s) to catch up. If the master perpetually outpaces the slaves, it may also create a cycle of re-copying the entire database, which just makes matters worse. If system testing reveals that the master log files are being produced faster than the slave(s) can consume them, it will be necessary to:

1. Reduce the number of slaves,
2. Speed up the slave computers, or
3. Speed up the connection between master and slaves.

If one of the slaves is a synchronous mirror, this can also slow down the consumption of log files significantly.

Mirroring and HA API Functions

Mirroring API Functions

<code>d_dbmir_init</code>	Initialize mirroring server
<code>d_dbmir_start</code>	Begin execution of an initialized mirroring server thread
<code>d_dbmir_stop</code>	Terminate mirroring thread
<code>d_dbmir_term</code>	Clean up resources allocated by mirroring initialization
<code>d_dbmir_connect</code>	Begin mirroring a database
<code>d_dbmir_disconnect</code>	Terminate a mirroring connection between master and slave

HA API Functions

<code>ha_login</code>	Log into an HA utility.
<code>ha_logout</code>	Log out of an HA utility.
<code>ha_quiesce</code>	Quiesce the Master TFS.
<code>ha_status</code>	Obtain the current HA status from an HA utility.
<code>ha_wakeup</code>	Wake up the Master TFS.
<code>ha_isTransActive</code>	Checks whether or not the Master TFS is processing active transactions.

d_dbmir_init

d_dbmir_init

Initialize mirroring server

Prototype

```
int32_t d_dbmir_init(  
    const DBREP_INIT_PARAMS *rparams,  
    REP_HANDLE                *hREP);
```

Unicode Function Prototypes

```
int32_t d_dbmir_initW(  
    const DBREP_INIT_PARAMSW *rparamsW,  
    REP_HANDLE                *hREP);
```

Arguments

rparams	(input)	Arguments required for dbmirror operation.
hREP	(output)	Handle used for running, stopping or terminating this server.

Description

This function initializes the functionality of the mirroring utility so that it can operate as a server, and provides a handle to control the utility.

When d_dbmir_init has been called, then d_dbmir_term should be called prior to termination of the application program.

Once this function has been called, the mirroring utility can be controlled through the functions d_dbmir_start, d_dbmir_stop, and d_dbmir_term.

Header File

```
#include "mirutils.h"
```

Library

mirroing Mirroring Library

Return Codes

Value	Name	Description
0	S_OKAY	normal return, okay
-904	S_NOMEMORY	out of memory
-924	S_INVNULL	invalid NULL parameter

d_dbmir_init

See Also

[DataMove Initialization Arguments \(DBREP_INIT_PARAMS\)](#)

[Document Root \(DOCROOT\)](#)

d_dbmir_start	Begin execution of an initialized mirroring server thread
d_dbmir_stop	Terminate mirroring thread
d_dbmir_term	Clean up resources allocated by mirroring initialization
d_dbmir_connect	Begin mirroring a database
d_dbmir_disconnect	Terminate a mirroring connection between master and slave

Example

(missing or bad snippet)

d_dbmir_connect

d_dbmir_connect

Begin mirroring a database

Prototype

```
int32_t d_dbmir_connect(  
    const DBREP_CONNECT_PARAMS *params,  
    uint16_t *slaveId)
```

Unicode Function Prototypes

```
int32_t d_dbmir_connectW(  
    const DBREP_CONNECT_PARAMSW *params,  
    uint16_t *slaveId)
```

Arguments

params	(input)	A pointer to a DBREP_CONNECT_PARAMS structure.
slaveId	(output)	Sequence number of the Replication utility's activity.

Description

This function performs the action of `dbget -b`, requesting a slave Mirror utility (`dbmirror`) to begin a mirroring connection with a master Mirroring utility. To end the connection, the corresponding function `d_dbmir_disconnect` is called.

Reference Manual

[Database Mirroring Utility](#)

[Database Get Utility](#)

Header File

```
#include "mirutils.h"
```

Library

datamove Data Move Library

Return Codes

Value	Name	Description
0	S_OKAY	normal return, okay
3	S_DUPLICATE	duplicate key
-4	S_INVDB	cannot open dictionary
-904	S_NOMEMORY	out of memory

d_dbmir_connect

-924	S_INVNULL	invalid NULL parameter
------	-----------	------------------------

See Also

[DataMove Connection Arguments \(DBREP_CONNECT_PARAMS\)](#)

<code>d_dbmir_start</code>	Begin execution of an initialized mirroring server thread
<code>d_dbmir_stop</code>	Terminate mirroring thread
<code>d_dbmir_term</code>	Clean up resources allocated by mirroring initialization
<code>d_dbmir_connect</code>	Begin mirroring a database
<code>d_dbmir_disconnect</code>	Terminate a mirroring connection between master and slave

Example

(missing or bad snippet)

d_dbmir_disconnect

d_dbmir_disconnect

Terminate a mirroring connection between master and slave

Prototype

```
int32_t d_dbmir_disconnect(  
    const DBREP_DISCONNECT_PARAMS *params);
```

Unicode Function Prototypes

```
int32_t d_dbmir_disconnectW(  
    const DBREP_DISCONNECT_PARAMSW *params)
```

Arguments

params (input) A pointer to a [DBREP_DISCONNECT_PARAMS](#) structure.

Description

This function performs the action of `dbget -e`, requesting a slave mirroring utility (`dbmirror`) to terminate a mirroring connection with a master mirroring utility. To initiate the connection, the corresponding function `d_dbmir_connect` is called.

Reference Manual

[Database Mirroring Utility](#)

[Database Get Utility](#)

Header File

```
#include "mirutils.h"
```

Library

datamove Data Move Library

Return Codes

Value	Name	Description
0	S_OKAY	normal return, okay
2	S_NOTFOUND	record not found
-214	S_TX_CONNECT	failed to connect to TFS

See Also

[DataMove Connection Arguments \(DBREP_CONNECT_PARAMS\)](#)

d_dbmir_disconnect

d_dbmir_start	Begin execution of an initialized mirroring server thread
d_dbmir_stop	Terminate mirroring thread
d_dbmir_term	Clean up resources allocated by mirroring initialization
d_dbmir_connect	Begin mirroring a database
d_dbmir_disconnect	Terminate a mirroring connection between master and slave

Example

(missing or bad snippet)

d_dbmir_start

d_dbmir_start

Begin execution of an initialized mirroring server thread

Prototype

```
int32_t d_dbmir_start(  
    REP_HANDLE hREP,  
    DB_BOOLEAN threaded,  
    uint16_t rep_done)
```

Arguments

hREP	(input)	Handle returned from a successful mirroring initialization call.
threaded	(input)	If TRUE, start a new thread for processing utility functionality.
rep_done	(input)	Set to TRUE when replication thread has terminated and cleaned up.

Description

This function begins mirroring server utility operation within the program space of the caller. The handle, hREP, must have been obtained through a mirroring initialization call, which must have been successful.

The control functions, d_dbmir_start, d_dbmir_stop, and d_dbmir_term all operate on a hREP handle that has been initialized as a dbmirror utility through the corresponding initialization function, d_dbmir_init.

If threaded is TRUE, this function will return immediately. This allows the calling program to proceed with other calls to the RDM core functions, or not. To terminate the server cleanly at a later time, the d_dbmir_stop function may be used. When threaded is FALSE, the calling program will not return from this function unless there is an error. To terminate the utility when threaded is FALSE, the program must be externally terminated.

Note that this does not begin mirroring. This only begins the server that will handle mirroring requests. See d_dbmir_connect to begin the mirroring process.

If database mirroring is to be used, the database access mode must be the *shared* mode.

Reference Manual

[Database Mirroring Utility](#)

Header File

```
#include "mirutils.h"
```

Library

datamove Data Move Library

d_dbmir_start

Return Codes

Value	Name	Description
0	S_OKAY	normal return, okay
-43	S_INVREPHANDLE	Invalid replication handle provided
-200	S_TX_ERROR	generic tx_error
-216	S_TX_LISTEN	TCP/IP listen failure in TFS
-904	S_NOMEMORY	out of memory
-924	S_INVNULL	invalid NULL parameter

See Also

d_dbmir_start	Begin execution of an initialized mirroring server thread
d_dbmir_stop	Terminate mirroring thread
d_dbmir_term	Clean up resources allocated by mirroring initialization
d_dbmir_connect	Begin mirroring a database
d_dbmir_disconnect	Terminate a mirroring connection between master and slave

Example

(missing or bad snippet)

d_dbmir_stop

d_dbmir_stop

Terminate mirroring thread

Prototype

```
int32_t d_dbmir_stop(  
    REP_HANDLE hREP)
```

Arguments

hREP (input) Handle returned from a successful replication or mirroring initialization call.

Description

This function terminates the mirroring server utility running within the program space of the caller. The handle, hREP, must have been obtained through a d_dbmir_init call, which must have been successful.

This function may be called if d_dbmir_start has been called with threaded == TRUE. It performs a clean shutdown of the threads that are servicing external database programs.

Reference Manual

[Database Mirroring Utility](#)

Header File

```
#include "mirutils.h"
```

Library

datamove Data Move Library

Return Codes

Value	Name	Description
0	S_OKAY	normal return, okay
-43	S_INVREPHANDLE	Invalid replication handle provided

See Also

d_dbmir_start	Begin execution of an initialized mirroring server thread
d_dbmir_stop	Terminate mirroring thread
d_dbmir_term	Clean up resources allocated by mirroring initialization
d_dbmir_connect	Begin mirroring a database
d_dbmir_disconnect	Terminate a mirroring connection between master and slave

Example

(missing or bad snippet)

d_dbmir_term

d_dbmir_term

Clean up resources allocated by mirroring initialization

Prototype

```
int32_t d_dbmir_term(  
    REP_HANDLE hREP)
```

Arguments

hREP (input) Handle returned from a successful replication or mirroring initialization call.

Description

This function frees resources allocated by the mirroring utility during and after the call to the initialization function. The utility thread must be stopped if it has been started, prior to calling this function.

Reference Manual

[Database Mirroring Utility](#)

Header File

```
#include "mirutils.h"
```

Library

datamove Data Move Library

Return Codes

Value	Name	Description
0	S_OKAY	normal return, okay
-43	S_INVREPHANDLE	Invalid replication handle provided

See Also

[d_dbmir_start](#) Begin execution of an initialized mirroring server thread
[d_dbmir_stop](#) Terminate mirroring thread
[d_dbmir_term](#) Clean up resources allocated by mirroring initialization
[d_dbmir_connect](#) Begin mirroring a database
[d_dbmir_disconnect](#) Terminate a mirroring connection between master and slave

Example

(missing or bad snippet)

HA Synchronous Notifications

Raima's current HA solution includes synchronous mirroring plus an API of functions used to query the TFS and dbmirror utilities. The API can also control a mode called internal asynchronous, which permits transactions to be committed to the master in the absence of a synchronous slave.

The API is designed to be used to poll the utilities, and the utilities do not wait for their status to be read prior to continuing with their normal operation.

This document describes additional HA functionality that can be used together with the existing API. The new functionality is best described as synchronous notifications, because its behavior is to execute a selected external procedure upon certain error conditions and suspend the utility until the procedure returns. Thus the procedures are invoked immediately upon detection of the conditions, and processing does not proceed until the procedure returns with a code indicating how to proceed.

Synchronous Callouts

The following paragraphs refer to procedures that are called by name. These names represent conditions, and the numeric value of the condition will be passed to a user-defined procedure along with the following information.

When a user-defined procedure is called, it will be invoked as a separate process. The procedure can be anything as long as it is executable on the platform; for instance, a binary executable, a shell script, a Perl script, and so on. The name of the procedure is required to be "notify" (with no file extension) regardless of the type of the procedure.

The following table briefly describes the callout conditions.

Condition	Description
HA_SLAVE_DISCONNECT	Mirroring slave has been disconnected from the master.
HA_COMMIT_TIMEOUT	Mirroring slave has timed out during a transaction commit.
HA_MASTER_DISCONNECT	Mirroring master has been inaccessible from the slave.
HA_SLAVE_SYNCING	Mirroring slave is synchronizing (i.e. catching up) with the master.
HA_SLAVE_UPDATED	Mirroring slave is up-to-date (i.e. caught up) with the master.
HA_SLAVE_LOCKED	Mirroring slave is ready to be switched to the Mirroring Master.
HA_MASTER_IASYNC	Master TFS is in internal asynchronous mode.

When a procedure returns, the return status will be received by the RDM utility. A SUCCESS value will be 0 and a FAILURE value will be 1. If the script fails, the default behavior of the utility will be to shut down.

Error Conditions

1. The Master Mirroring Utility detects a disconnection of a Slave. This can result from a networking failure or a failure of the Slave Utility.

ACTION:

- Call the procedure HA_SLAVE_DISCONNECT.
- Switch to internal asynchronous (IASYNC) mode and continue processing.
- Terminate the thread handling the master function because the mirroring connection is gone. If mirroring is re-established, it will be initiated from the Slave Utility, and a new thread will be spawned to handle it.

2. The Slave Mirroring Utility detects a disconnection from the Master. This can result from a networking failure, a failure of the Master Utility, or the Master Utility carrying out an HA_COMMIT_TIMEOUT condition.

ACTION:

- Perform local mirroring cleanup.
- Call the procedure HA_MASTER_DISCONNECT.
- Terminate the thread that performs these actions because the mirroring connection is gone. Another thread will be created to service this mirrored slave as soon as the request is made by the HA Manager.

Note that if the Master fails, it cannot produce a notification of its condition. The same is true if the Slave fails. Detection of a 'live' process is possible through the HA API.

3. The Master Mirroring Utility receives no response from a Slave during a commit for a configurable period of time. This will be called a timeout condition. The Slave will be treated as though it has been disconnected (above), except that the procedure HA_COMMIT_TIMEOUT will be called instead of HA_SLAVE_DISCONNECT. The timeout value can be specified with the HAtimeout INI file parameter as follows. If not specified, the default timeout value is 30 seconds (30,000 milliseconds). The minimum value that can be specified is 1 second (1,000 milliseconds).

```
[configuration]
HAtimeout=10000 ; commit timeout value in milliseconds. Default is 30000
```

Non-error (Status) Conditions

1. 1. If the Master Mirroring Utility has been in internal asynchronous mode and the slave has reconnected, a synchronization of the logs will automatically begin. During this synchronization process, the TFS will behave as though the internal asynchronous mode has been turned off, that is, it will not return from a transaction commit until it receives notification from the slave that the transaction has been committed on the slave. Since this may involve a noticeable delay during the synchronization, this status will allow the proper notice to be posted.

ACTION:

- Turn off the internal asynchronous mode.
 - Call the procedure HA_SLAVE_SYNCING.
 - Continue normal processing.
2. When the Master Mirroring Utility is notified that the slave has caught up with it (i.e. again synchronized), the normal synchronous operation may begin again.

ACTION:

- Call the procedure HA_SLAVE_UPDATED.
- Continue normal processing.

Switchover

Graceful recoverable switchover is accomplished through the following three steps:

1. Quiesce the Master database. A new API function, `ha_quiesce()`, can be called to block new transactions on the TFS Master. To make sure all the existing transactions have been committed or rolled back, call `ha_isTransActive()`.

ACTION:

- HA Manager calls the function against the Mirroring Master Utility to quiesce the TFS. It returns immediately. The function succeeds if the TFS knows that the database's slave is being mirrored. It returns `HA_NO_DB` if it determines that there is no slave, or `HA_MASTER_IASYNC` if the TFS is in internal asynchronous mode.
 - HA Manager polls the Mirroring Master Utility to verify a status of `HA_SLAVE_UPDATED` (if Slave was active).
 - HA Manager calls `ha_isTransActive()` against the Mirroring Master Utility to make sure there is no active transaction on the Master TFS.
2. Activate the slave. Instruct the Slave TFS and Mirroring Utility to assume the role of Master.

ACTION:

- HA Manager calls `"dbget -e <dbname>"` to terminate any ongoing mirroring process. This will close the connection if it is open and clean up the mirroring activity. It succeeds even if there was no open connection. During this call, the `dbmirror` utility calls the procedure `HA_SLAVE_LOCKED`.
 - HA Manager calls `d_dbmove()` against the Slave TFS to move the position (change the role) of the database from slave to master.
3. Reconfigure the Master.

ACTION:

- HA Manager calls `d_dbmove()` against the Master TFS to move the position (change the role) of the database from master to slave.
4. Wake up the Master database. A new API function, `ha_wakeup()`, can be called to allow new transactions to be processed on the TFS.
 5. Begin mirroring in the reverse direction.

ACTION:

- HA Manager calls the equivalent of `'dbget -b -sync <dbname>'`.

HA API Reference

<code>ha_login</code>	Log into an HA utility.
<code>ha_logout</code>	Log out of an HA utility.
<code>ha_quiesce</code>	Quiesce the Master TFS.
<code>ha_status</code>	Obtain the current HA status from an HA utility.
<code>ha_wakeup</code>	Wake up the Master TFS.
<code>ha_isTransActive</code>	Checks whether or not the Master TFS is processing active transactions.

ha_login

ha_login

Log into an HA utility.

Prototype

```
HA_DIAG_EXTERNAL_FCN ha_login(  
    const char *nodeName,  
    uint16_t    port,  
    HA_USER     *userID)
```

Arguments

nodeName	(input)	Domain name of the computer running the utility. Default is "localhost".
port	(input)	Port number the utility is listening on. Default is 21553 for TFS.
userID	(output)	The unique HA user ID to be used in the other ha calls.

Description

This function is used to log into an HA utility (tfserver or dbmirror) to establish a connection and make calls to obtain HA-related status information from the utility.

To use the default nodeName ("localhost"), NULL may be provided. To use the default port, a 0 or 1 may be provided. The 0 will address the tfserver utility on the default port, and the 1 will address the dbmirror on its default port.

When a port other than the default is used, the dbmirror port must always be one more than the tfserver port.

When the session with this utility is complete, use the ha_logout function.

Header File

hafcns.h

Library

harpc High Availability RPC library

Return Codes

Value	Name	Description
0	HA_DIAG_OKAY	The function successfully called the user-defined file.
2	HA_DIAG_FAILED	Unable to log into an HA utility.
3	HA_DIAG_NOMEMORY	Attempt to allocate memory failed.
7	HA_DIAG_INVNULL	Invalid use of a NULL pointer.
8	HA_DIAG_RPCERR	Network communication error occurred.

Comments

The HAstatus parameter under [configuration] must be set to 1 in TFS.INI for the login to succeed.

ha_logout

ha_logout

Log out of an HA utility.

Prototype

```
HA_DIAG_EXTERNAL_FCN ha_logout(  
    HA_USER *userID)
```

Arguments

userID (input) The unique HA user ID obtained with ha_login().

Description

This function is used to log out of an HA utility (tfserver or dbmirror).

Header File

hafcns.h

Library

harpc High Availability RPC library

Return Codes

Value	Name	Description
0	HA_DIAG_OKAY	The function successfully called the user-defined file.
2	HA_DIAG_FAILED	Unable to log into an HA utility.
3	HA_DIAG_NOMEMORY	Attempt to allocate memory failed.
7	HA_DIAG_INVNULL	Invalid use of a NULL pointer.
8	HA_DIAG_RPCERR	Network communication error occurred.

ha_status

ha_status

Obtain the current HA status from an HA utility.

Prototype

```
HA_DIAG_EXTERNAL_FCN ha_status(  
    HA_USER      userID,  
    const char *dbName,  
    HA_STATUS *hstat,  
    int32_t     *more)
```

Arguments

userID	(input)	The unique HA user ID obtained with <code>ha_login()</code> .
dbName	(input)	Request status of this database. Use NULL for utility status
hstat	(output)	Current HA status returned from an HA utility.
more	(output)	The count of additional HA statuses queued after the current (oldest) one. There is one queue per database, and another for the utility.

Description

This function can be used to obtain the current status of an HA utility. If it returns `HA_DIAG_OKAY`, the current HA status will be returned through the `hstat` parameter. The number of additional statuses, if any, will be returned through the `more` parameter. If the function returns `HA_DIAG_EOQ`, no status is available on the HA utility. The values of `hstat` and `more` are undefined if the function does not return `HA_DIAG_OKAY`.

Header File

hafcns.h

Library

harpc High Availability RPC library

Return Codes

Value	Name	Description
0	HA_DIAG_OKAY	The function successfully called the user-defined file.
1	HA_DIAG_EOQ	The HA utility has no status stored.
2	HA_DIAG_FAILED	Unable to log into an HA utility.
3	HA_DIAG_NOMEMORY	Attempt to allocate memory failed.
7	HA_DIAG_INVNULL	Invalid use of a NULL pointer.
8	HA_DIAG_RPCERR	Network communication error occurred.

ha_quiesce

ha_quiesce

Quiesce the Master TFS.

Prototype

```
HA_DIAG_EXTERNAL_FCN ha_quiesce(  
    HA_USER      userID,  
    const char *dbname)
```

Arguments

userID	(input)	The unique HA user ID obtained with ha_login().
dbname	(input)	Name of the database being mirrored.

Description

This function sets the target TFS in quiescent mode. It means that the TFS will block any further transactions from starting while finishing up the currently active ones. Any transaction, whether or not it affects the mirrored database, will be blocked. The Master TFS cannot be quiesced while it is in internal asynchronous mode.

Header File

hafcns.h

Library

harpc High Availability RPC library

Return Codes

Value	Name	Description
0	HA_DIAG_OKAY	The function successfully called the user-defined file.
2	HA_DIAG_FAILED	Unable to log into an HA utility.
3	HA_DIAG_NOMEMORY	Attempt to allocate memory failed.
4	HA_DIAG_INVOP	An invalid operation has been performed.
7	HA_DIAG_INVNULL	Invalid use of a NULL pointer.
8	HA_DIAG_RPCERR	Network communication error occurred.
9	HA_DIAG_IASYNC	Master TFS is currently in internal asynchronous mode.

Comments

This function is defined to be called against the Mirroring Master Utility (dbmirror master). If called against any other process (such as the Master TFS), it will return HA_DIAG_INVOP.

ha_wakeup

ha_wakeup

Wake up the Master TFS.

Prototype

```
HA_DIAG_EXTERNAL_FCN ha_wakeup(  
    HA_USER      userID,  
    const char *dbname)
```

Arguments

userID	(input)	The unique HA user ID obtained with ha_login.
dbname	(input)	Name of the database being mirrored.

Description

This function clears the quiescent mode flag on the target (Master) TFS, allowing it to process new incoming transactions. HA Manager should call this function once the Master and Slave TFSs have been switched over.

Header File

hafcns.h

Library

harpc High Availability RPC library

Return Codes

Value	Name	Description
0	HA_DIAG_OKAY	The function successfully called the user-defined file.
2	HA_DIAG_FAILED	Unable to log into an HA utility.
3	HA_DIAG_NOMEMORY	Attempt to allocate memory failed.
4	HA_DIAG_INVOP	An invalid operation has been performed.
7	HA_DIAG_INVNULL	Invalid use of a NULL pointer.
8	HA_DIAG_RPCERR	Network communication error occurred.

See Also

ha_login	Log into an HA utility.
ha_logout	Log out of an HA utility.
ha_quiesce	Quiesce the Master TFS.
ha_status	Obtain the current HA status from an HA utility.
ha_wakeup	Wake up the Master TFS.
ha_isTransActive	Checks whether or not the Master TFS is processing active transactions.

ha_wakeup

Comments

This function is defined to be called against the Mirroring Master Utility (dbmirror master). If called against any other process (such as the Master TFS), it will return HA_DIAG_INVOP.

ha_isTransActive

Checks whether or not the Master TFS is processing active transactions.

Prototype

```
HA_DIAG_EXTERNAL_FCN ha_quiesce(
    HA_USER      userID,
    const char *dbname,
    uint16_t     *active)
```

Arguments

userID	(input)	The unique HA user ID obtained with ha_login().
dbname	(input)	Name of the database being mirrored.
Active	(output)	Flag to indicate whether or not the Master TFS is processing active transactions. 1 indicates yes.

Description

This function retrieves the transaction processing status of the Master TFS. If the TFS has one or more active transactions to process, the function will return 1 through the active parameter. 0 will be returned if the TFS is clear of any transactions. HA Manager should use this function in order to determine whether the TFS is ready to be switched over.

Header File

hafcns.h

Library

harpc High Availability RPC library

Return Codes

Value	Name	Description
0	HA_DIAG_OKAY	The function successfully called the user-defined file.
2	HA_DIAG_FAILED	Unable to log into an HA utility.
3	HA_DIAG_NOMEMORY	Attempt to allocate memory failed.
4	HA_DIAG_INVOP	An invalid operation has been performed.
7	HA_DIAG_INVNULL	Invalid use of a NULL pointer.
8	HA_DIAG_RPCERR	Network communication error occurred.

Comments

HA Manager should call ha_quiesce() before calling qa_isTransActive() in order to make sure no new transactions are allowed. Otherwise, the Master TFS may never finish processing all the active transactions.

ha_isTransActive

This function is defined to be called against the Mirroring Master Utility (dbmirror master). If called against any other process (such as the Master TFS), it will return HA_DIAG_INVOP.

RDM Mirroring Example

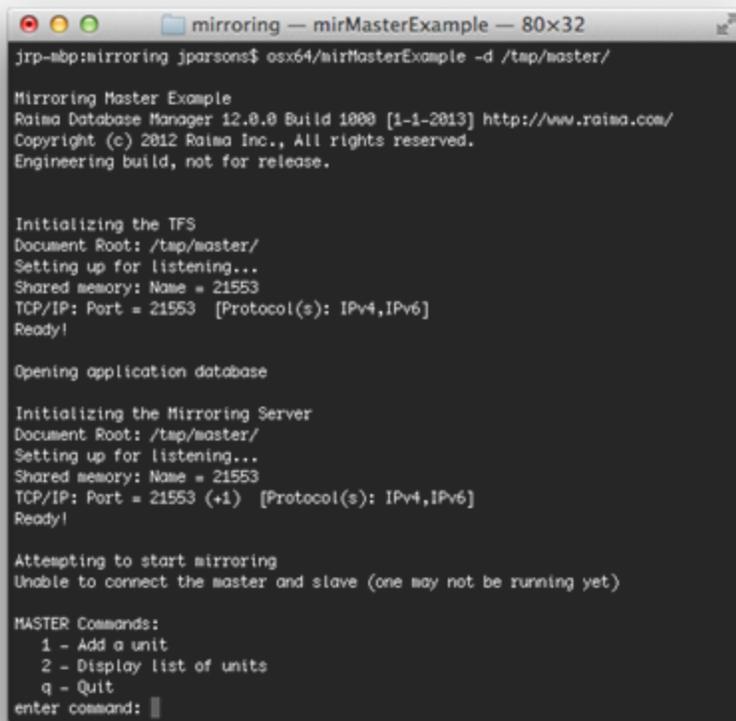
The Mirroring example is a simple application designed to illustrate how to setup the RDM mirroring infrastructure completely within an application by utilizing the RDM Replication APIs. The use case for this example is where an application, the TFS, and the Mirroring server can all be run in the same process space. When running in one process space the configuration, initialization, and execution of RDM Mirroring can be completely embedded within an application

Application Components

The example is implemented using two application components. The first component is the `mirMasterExample` application that serves as the primary database. The second component is the `mirSlaveExample` application that serves as the secondary/backup database. The example allows you to add data to the master and then view the data from either the slave or the master.

mirMasterExample

The `mirMasterExample` hosts a TFS that contains the primary (or ‘master’) database. It also hosts a mirroring server that will communicate to the slave to mirror transaction from the primary database to the secondary. The application utilizes a simple console interface that allows the user to enter a new record or view all of the rows in the unit table.



```
jrp-mbp:mirroring jparsons$ osx64/mirMasterExample -d /tap/master/

Mirroring Master Example
Raima Database Manager 12.0.0 Build 1000 [1-1-2013] http://www.raima.com/
Copyright (c) 2012 Raima Inc., All rights reserved.
Engineering build, not for release.

Initializing the TFS
Document Root: /tap/master/
Setting up for listening...
Shared memory: Name = 21553
TCP/IP: Port = 21553 [Protocol(s): IPv4,IPv6]
Ready!

Opening application database

Initializing the Mirroring Server
Document Root: /tap/master/
Setting up for listening...
Shared memory: Name = 21553
TCP/IP: Port = 21553 (+1) [Protocol(s): IPv4,IPv6]
Ready!

Attempting to start mirroring
Unable to connect the master and slave (one may not be running yet)

MASTER Commands:
 1 - Add a unit
 2 - Display list of units
 q - Quit
enter command: █
```

When the application is first run a database is initialized with a couple of default unit records. These default records can be displayed by command ‘2’ (Display list of units)

```
mirroring — mirMasterExample — 80x32
Document Root: /tmp/master/
Setting up for listening...
Shared memory: Name = 21553
TCP/IP: Port = 21553 (+1) [Protocol(s): IPv4,IPv6]
Ready!

Attempting to start mirroring
Unable to connect the master and slave (one may not be running yet)

MASTER Commands:
 1 - Add a unit
 2 - Display list of units
 q - Quit
enter command: 2
units:
-----
name:      Raisa HQ
unit number: 720
phone:    +1.206.748.5300
fax:      +1.206.748.5200

name:      Raisa Idaho
unit number: 3822
phone:    +1.206.748.5200
fax:      +1.206.748.5200

MASTER Commands:
 1 - Add a unit
 2 - Display list of units
 q - Quit
enter command: |
```

A new unit can be added with command number '1' (Add a unit)

```

mirroring -- mirMasterExample -- 80x32
1 - Add a unit
2 - Display list of units
q - Quit
enter command: 2
units:
-----
name:      Raiwa HQ
unit number: 720
phone:     +1.206.748.5300
fax:       +1.206.748.5200

name:      Raiwa Idaho
unit number: 3822
phone:     +1.206.748.5200
fax:       +1.206.748.5200

MASTER Commands:
1 - Add a unit
2 - Display list of units
q - Quit
enter command: 1
name : Raiwa UK
unitno: 807
phone : +44.1628.826.800
fax   : +44.1628.825.343

MASTER Commands:
1 - Add a unit
2 - Display list of units
q - Quit
enter command:

```

mirSlaveExample

The `mirSlaveExample` hosts a TFS that contains the backup (or 'slave') database. It also hosts a mirroring server that will communicate to the master to receive transactions from the primary database. The application utilizes a simple console interface that allows the user to display all of the rows in the unit table. A slave database must be opened in 'read-only' mode so it is not possible to directly add rows through the slave application. However all rows added to the primary database will be visible to the backup database. This is true even if the `mirSlaveExample` application is not running at the time a new row is added via the `mirMasterExample`.

```

jrp-mbp:mirroring jparsons$ osx64/mirSlaveExample -d /tmp/slave

Mirroring Slave Example
Raima Database Manager 12.0.0 Build 1000 [1-1-2013] http://www.raima.com/
Copyright (c) 2012 Raima Inc., All rights reserved.
Engineering build, not for release.

Initializing the TFS
Document Root: /tmp/slave/
Setting up for listening...
Shared memory: Name = 22553
TCP/IP: Port = 22553 [Protocol(s): IPv4,IPv6]
Ready!

Initializing the Mirroring Server
Document Root: /tmp/slave/
Setting up for listening...
Shared memory: Name = 22553
TCP/IP: Port = 22553 (+1) [Protocol(s): IPv4,IPv6]
Ready!

Attempting to start mirroring
Slave server: Connected to local TFS
Slave server: Connected to master server

Opening application database

SLAVE Commands:
 1 - Display list of units
 q - Quit
enter command: 

```

When the application is first run a database is initialized with and able to connect to the master database, the contents of the master will be copied to the slave. As long as both the master and slave applications are running all modifications made to the master database will be mirrored to the slave. To view the current contents of the slave database menu item '1' (Display list of units) can be chosen.

```
mirroring — mirSlaveExample — 80x32
Attempting to start mirroring
Slave server: Connected to local TFS
Slave server: Connected to master server

Opening application database

SLAVE Commands:
  1 - Display list of units
  q - Quit
enter command: 1
units:
-----
name:      Raisa HQ
unit number: 728
phone:     +1.206.748.5200
fax:       +1.206.748.5200

name:      Raisa Idaho
unit number: 3822
phone:     +1.206.748.5208
fax:       +1.206.748.5208

name:      Raisa UK
unit number: 807
phone:     +44.1628.826.800
fax:       +44.1628.825.343

SLAVE Commands:
  1 - Display list of units
  q - Quit
enter command: █
```

Application Architecture

The Mirroring example utilizes the model-view-controller (MVC) architecture pattern. While this pattern is primarily used in GUI applications the View implementation in this instance utilizes console I/O. The Model is most important part of the example it demonstrates the use of the RDM APIs to configure and utilize database mirroring. The Controller portion of the application ties together the Model and the View.

Model

The Model handles all of the interaction with the RDM database. The methods comprising the model show how to configure an RDM application to utilize database mirroring. In addition there are methods that insert and retrieve data from an RDM database through the RDM C++ API.

CMirModel.h

The `CMirModel.h` file defines the abstract base class for the Mirroring example. Many of the methods in the model are common between both the Master and the Slave and are implemented by the base class. However there are several methods that require specific implementations for the Master and the Model.

CMirModel.cpp

This file implements the functionality common to Master and Slave applications. All of the mirroring specific code is in the source file. The methods to look at include

- InitializeTFS
- InitializeMirServer
- StartMirroring
- StopMirroring
- CleanupMirServer
- CleanupTFS

CMirMasterModel.h

The file defines the model class that is specific the Master application

CMirMasterModel.cpp

The file implements the model class that is specific the Master application

CMirSlaveModel.h

The file defines the model class that is specific the Slave application

CMirSlaveModel.cpp

The file implements the model class that is specific the Slave application

View

The View handles all of the interaction with the application user. In this case the model implemented utilizes console I/O for user interaction. It is possible to implement models that support GUI displays or non-interactive sessions. However for this example we have only included a console implementation.

The master and the slave have slightly different requirements so we have created an abstract definition of the View class and specific implementation for both the Master and the Slave.

CMirView.h

This file contains the definition of the abstract class the view is based on. This is a pure abstract class and does not contain any implementation

CMirViewConsole.h

This file contains the definition for the Console implementation of the View. This class implements view methods that are common to both the Master and Slave applications.

CMirViewConsole.cpp

This file contains the implementation for the Console view methods that are common for the Master and Slave applications.

CMirMasterViewConsole.h

This file contains the definition for the Console view methods specific to the Master application.

CMirMasterViewConsole.cpp

This file contains the implementation for the Console view methods specific to the Master application.

CMirSlaveViewConsole.h

This file contains the definition for the Console view methods specific to the Slave application.

CMirSlaveViewConsole.cpp

This file contains the implementation for the Console view methods specific to the Slave application.

Controller

The Controller handles all the integration between the Model and the View. Using a Controller it is possible to have the same Model work with different Views.

CMirController.h

This file contains the definition of the class the Controller is based on. This is an abstract class that has implementation for methods that are common for both the Master and Slave

CMirController.h

This file contains the implementation of Control methods that a common for both the Master and Slave.

CMirMasterController.h

This file contains the definition of the Control class that has methods for the Master application

CMirMasterController.cpp

This file contains the implementation of the Control class that has methods for the Master application

CMirSlaveController.h

This file contains the definition of the Control class that has methods for the Slave application

CMirSlaveController.cpp

This file contains the implementation of the Control class that has methods for the Slave application

Application Database

This example utilizes a very simple database that contains a single table

```
#define NAME_LEN 32
#define UNIT_LEN 8
#define PHONE_LEN 20

database ia_db
{
    record unit {
        unique key char name[NAME_LEN];
        char unitno[UNIT_LEN];
        char phone[PHONE_LEN];
        char fax[PHONE_LEN];
    }
}
```

Mirroring Utilities

dbmirror

Database Mirroring Server

Prototype

```
dbmirror [-h] [-B] [-V] [-q|[-stdout file]] [-stderr file] [-tfs type]
          [-docroot path] [-trans trans_list] [-p port/name] [-v] [-notify]
          [-start|-stop|-query|-install exepath|-uninstall]
          [-serviceuser username] [-servicepw password] [-nodisk]
```

Description

The `dbmirror` utility acts as either a *master* or *slave* database mirroring server. The same `dbmirror` instance may operate in both master and slave roles at the same time (using separate threads), and may operate on any number of databases, but each database is either a master or a slave database. It always runs on the same computer as a TFS (the `tfserver` utility or a customized TFS created by the programmer). The TFS/`dbmirror` partnership is established by using successive TCP/IP ports (e.g. if the TFS uses port 2335, then `dbmirror` will use 2336).

In the master role, `dbmirror` responds to requests from counterpart `dbmirror` slaves. For the initial request from the `dbmirror` slave, the `dbmirror` master will create a thread to service the database being requested. The thread will be dedicated to serving log files to the slave until the slave terminates the connection. The master thread will receive log file requests from the slave, and search the local directory for each request. If the log file exists, it will send it to the slave. If it doesn't exist, and the master hasn't already done so, it will send the entire database to the slave, after which the slave will begin requesting each log file that has been created after the database image was generated.

In the slave role, `dbmirror` may partner with a TFS or not. If there is a TFS, the changes are applied to the local database. If there is no TFS, then `dbmirror` will save change log files in the database's subdirectory (below the document root), which may be processed at a later time by a TFS. It is the slave that initiates the connection with the master `dbmirror`. The slave initiates the connection when the `dbget` utility is run with command-line options that identify this `dbmirror` and a database that is hosted by another TFS. In response to `dbget`, `dbmirror` will begin a slave thread which connects to the master `dbmirror` and requests updates from the database identified in the `dbget` command-line.

By convention, the slave database mirror is placed within a directory that is named after the host of the master database. For example, if the database named `tims` is mirrored from a host identified as `tfs.raima.com`, then the mirror of `tims` will be stored in:

```
documentRoot/tfs.raima.com-master_port/tims
```

All databases mirrored to this document root from the same host will be stored in the same subdirectory. Databases stored in subdirectories are the products of mirroring or replication, and thus cannot be opened for updating. See [Opening Slave Databases from Programs](#) for more information about using read-only databases from programs.

When this utility is run as a foreground process, it may be cleanly terminated by issuing `SIGINT` (normally `^C` from the keyboard). `SIGTERM` has the same effect. If the utility does not terminate immediately it will be terminated immediately (cleanly or not) upon the third signal.

Options

Specifies the server port (TCP/IP) or name (non-TCP-IP) for connections. The default port is 21553 and the default name is "21553". If a number is specified it will be used for the TCP/IP port and the server name. If a string or invalid port number is specified it will be used as the server name and the TCP/IP transport will not be initialized.

-h -?	Display this usage information
-B	Do not display the banner
-V	Display the version information
-q	Quiet mode. No information will be displayed
-stdout <i>filespec</i>	Redirect <code>stdout</code> to the specified <i>filespec</i>
-stderr <i>filespec</i>	Redirect <code>stderr</code> to the specified <i>filespec</i>
-tfs <i>type</i>	Specify the TFS <i>type</i> to use: tfss standalone s Standalone TFS type tfsr rpc r Client/Server TFS type tfst tfs t Direct-Link TFS type (Default)
-trans <i>trans_list</i>	Specifies a comma separated list of transports to listen on. Options are "tcp" and "shm" (default is "shm,tcp")
-p <i>port/name</i>	Specifies the server port (TCP/IP) or name (non-TCP-IP) for connections. The default port is 21553 and the default name is "21553". If a number is specified it will be used for the TCP/IP port and the server name. If a string or invalid port number is specified it will be used as the server name and the TCP/IP transport will not be initialized.
-v	Verbose output.
-start	Start the server (run as a daemon on Unix)
-stop	Stop the server
-query	Query the execution status of the server
-nodisk	Don't use any disk I/O.
-notify	Enable HA notification
-notfslisten	Do not allow direct connection to TFS
-nopageref	Disable <i>page referencing</i> ¹ read optimization
Windows Only	
-install <i>path</i>	Install the server as a service at the specified path.
-uninstall	Uninstall the server as a service.
-serviceuser	User account to install service as (only valid with <code>-install</code> option)
-servicepw	User account password to install service as (only valid with <code>-install</code> option and required with <code>-serviceuser</code> option)

¹In a TFSR configuration - where the application runs in a separate process from the TFS (which would be running in a `tfserver` or other application), the system normally attempts to determine if the application is on the same physical machine as the TFS. If it is determined that the two processes are on the same machine, the code will use a shortcut - those pages that will be read from the database files and sent over to the application via TCP/IP or other connection are instead read directly from the disk by the user application. This is all handled internally by the software - no separate options are required. Some users, however may not wish for this to take place because of file permission issues or other reasons. In that case, this option can be specified and doing so will disable this shortcut - thus acting as if the user application and the TFS are on different machines and send all pages through the connection between the two.

dbget

Database Mirroring Launch utility

Prototype

```
dbget [-h] [-B] [-V] [-q] [-stdout file] [-stderr file] [-key key]
      [-u DBUSERID] [-s SLAVE_HOSTNAME] [-trans trans_list] [-n name]
      [-p N] [-alias alias] [-id numeric_id] [-nocopy] [-sync]
      [-unsync] [-b] [-e] [-override_inmem] [-dsn dsn;user;pswd]
      [-oracle|-mssql|-mysql] db_namespec
```

Description

The `dbget` utility notifies a *replication or mirroring slave process* that it should begin mirroring or replicating a database. A replication or mirroring slave process is one of the following:

- `dbmirror` - the database will be *mirrored*.
- `dbrep` - the database will be *replicated* to a slave RDM database.
- `dbrepsql` - the database will be *replicated* to a slave ODBC data source.

Regardless of which replication utility is running, it is identified through the `-s SLAVE_HOSTNAME` and `-p PORT` or `-n NAME`.

`dbget` is used to obtain the original copy of the database, which is then continuously updated via mirroring or replication processes.

When the `-b` option is used, mirroring/replication will be started if it is not already active. To discontinue mirroring/replication, use `dbget` with `-e`. At least one of `-b` or `-e` must be specified.

Synchronous mirroring is started by including the `-sync`. This option is invalid for replication (`dbrep` or `dbrepsql`). Synchronous mirroring is persistent, meaning that if the mirroring client stops or is disconnected, the mirroring will remain synchronous the next time `dbget` is used. Beware that a synchronous mirror slave can block transactions applied to the master if the slave is not active.

When `dbrepsql` is the replication utility being notified, it is necessary to identify the type of database. RDM Server is the default, and no option is needed. For Oracle, MySQL, or Microsoft SQL Server, use options `-oracle`, `-mysql`, or `-mssql`, respectively.

Options

<code>-h -?</code>	Display this usage information
<code>-B</code>	Do not display the banner
<code>-V</code>	Display the version information
<code>-q</code>	Quiet mode. No information will be displayed
<code>-stdout filespec</code>	Redirect <code>stdout</code> to the specified <i>filespec</i>
<code>-stderr filespec</code>	Redirect <code>stderr</code> to the specified <i>filespec</i>
<code>-key enckey</code>	Specify the encryption key for the database. The <i>enckey</i> is specified as: [algorithm:] passcode. The valid algorithm are:
	<code>xor</code> XOR encryption key
	<code>aes128</code> 128 bit AES encryption key

dbget

	aes192	192 bit AES encryption key
	aes256	256 bit AES encryption key
-u <i>DBUSERID</i>		DBUSERID to use during transactions
-s <i>SLAVE_HOSTNAME</i>		SLAVE_HOSTNAME of slave DBMIRROR (default is localhost)
-trans <i>trans_list</i>		Specifies a comma separated list of transports to listen on. Options are "tcp" and "shm" (default is "shm,tcp")
-n <i>name</i>		Specifies the anchor name of the slave TFS for non-TCP/IP connections (default is "21553")
-p <i>port</i>		Specifies the anchor port of the slave TFS for TCP/IP connections (default is 21553)
-alias <i>alias_name</i>		Specifies the database alias to use on the slave
-id <i>numeric_id</i>		[Required] Specifies the unique id for this database connection
-nocopy		Don't ever copy the database (data can be lost)
-sync		Valid only when notifying dbmirror. Mirrored database is synchronous.
-unsync		Valid only when notifying dbmirror. End persistent synchronous mirroring.
-b		Begin mirroring or replication.
-e		End mirroring or replication.
-override_inmem		If the master database is in-memory, force slave to be written to disk.
-dsn <i>dsn:user:pswd</i>		Specify DSN for ODBC connection to slave SQL server (dbrepsql only).
-oracle		Slave dbrepsql connects to Oracle server.
-mssql		Slave dbrepsql connects to Microsoft SQL server.
-mysql		Slave dbrepsql connects to MySQL server.
db_namespec		Name and location of master database TFS. Default domain is localhost. Default port is 21553.

dbrep

Database Replication Slave Utility

Prototype

```
dbrep [-h] [-B] [-V] [-q|[-stdout file]] [-stderr file] [-tfs type]
      [-docroot path] [-trans trans_list] [-p port/name] [-v] [-notify]
      [-start|-stop|-query|-install exepath|-uninstall]
      [-serviceuser username] [-servicepw password] [-nodisk]
```

Description

The `dbrep` utility is a replication utility that performs replication only. `dbrep` may be in master or slave roles. When in the master role, it can serve files to `dbmirror`, `dbrep` or `dbrep_sql` slaves. The same `dbrep` utility may act as both master and slave at the same time. The same `dbrep` slave may operate on any number of databases from any number of different master database locations. It always runs on the same computer as a TFS (the `tfserver` utility or a customized TFS created by the programmer). The TFS/`dbrep` partnership is established by using successive TCP/IP ports (e.g. if the TFS uses port 2335, then `dbrep` is using 2336).

`dbrep` requires a TFS to be running on the same computer. The `dbrep` utility initiates a connection to the master `dbmirror` when the `dbget` utility is run with command-line options that identify this `dbrep` and a database that is hosted by another TFS. In response to `dbget`, `dbrep` will begin a slave thread which connects to the master `dbmirror` and requests updates from the database identified in the `dbget` command-line.

By convention, the slave database replicate created by `dbrep` is placed within a directory that is named after the host of the master database. For example, if the database named `tims` is mirrored from a host identified as `tfs.raima.com`, then the replicate of `tims` will be stored in:

```
documentRoot/tfs.raima.com-master_port/tims
```

All databases replicated to this document root from the same host will be stored in the same subdirectory. Databases stored in subdirectories are the products of mirroring or replication, and thus cannot be opened for updating. See *Opening Slave Databases from Programs* for more information about using read-only databases from programs.

When this utility is run as a foreground process, it may be cleanly terminated by issuing `SIGINT` (normally `^C` from the keyboard). `SIGTERM` has the same effect. If the utility does not terminate immediately it will be terminated immediately (cleanly or not) upon the third signal.

Options

Specifies the server port (TCP/IP) or name (non-TCP/IP) for connections. The default port is 21553 and the default name is "21553". If a number is specified it will be used for the TCP/IP port and the server name. If a string or invalid port number is specified it will be used as the server name and the TCP/IP transport will not be initialized.

<code>-h</code> <code>-?</code>	Display this usage information
<code>-B</code>	Do not display the banner
<code>-V</code>	Display the version information
<code>-q</code>	Quiet mode. No information will be displayed
<code>-stdout filespec</code>	Redirect <code>stdout</code> to the specified <i>filespec</i>
<code>-stderr filespec</code>	Redirect <code>stderr</code> to the specified <i>filespec</i>

dbget

<code>-tfs type</code>	Specify the TFS <i>type</i> to use: tfss standalone s Standalone TFS type tfsr rpc r Client/Server TFS type tfst tfs t Direct-Link TFS type (Default)
<code>-docroot path</code>	Document root under which to place database files.
<code>-trans trans_list</code>	Specifies a comma separated list of transports to listen on. Options are "tcp" and "shm" (default is "shm,tcp")
<code>-p port/name</code>	Specifies the server port (TCP/IP) or name (non-TCP-IP) for connections. The default port is 21553 and the default name is "21553". If a number is specified it will be used for the TCP/IP port and the server name. If a string or invalid port number is specified it will be used as the server name and the TCP/IP transport will not be initialized.
<code>-v</code>	Verbose output.
<code>-start</code>	Start the server (run as a daemon on Unix)
<code>-stop</code>	Stop the server
<code>-query</code>	Query the execution status of the server
<code>-nodisk</code>	Don't use any disk I/O.
<code>-notify</code>	Enable HA notification
<code>-notfslisten</code>	Do not allow direct connection to TFS
<code>-nopageref</code>	Disable <i>page referencing</i> ¹ read optimization
Windows Only	
<code>-install path</code>	Install the server as a service at the specified path.
<code>-uninstall</code>	Uninstall the server as a service.
<code>-serviceuser</code>	User account to install service as (only valid with <code>-install</code> option)
<code>-servicepw</code>	User account password to install service as (only valid with <code>-install</code> option and required with <code>-serviceuser</code> option)

¹In a TFSR configuration - where the application runs in a separate process from the TFS (which would be running in a `tfserver` or other application), the system normally attempts to determine if the application is on the same physical machine as the TFS. If it is determined that the two processes are on the same machine, the code will use a shortcut - those pages that will be read from the database files and sent over to the application via TCP/IP or other connection are instead read directly from the disk by the user application. This is all handled internally by the software - no separate options are required. Some users, however may not wish for this to take place because of file permission issues or other reasons. In that case, this option can be specified and doing so will disable this shortcut - thus acting as if the user application and the TFS are on different machines and send all pages through the connection between the two.

Appendix

Document Root (DOCROOT)

The document root (DOCROOT) is a directory (a folder) which is designated for holding databases available to a Transaction File Server (TFS). The concept is similar to a web server document root for storing web pages. Important notes about the TFS are:

- A TFS has exclusive access to a document root. Multiple simultaneous TFS access to a DOCROOT is not allowed;
- Within the domain of one TFS, no files outside the DOCROOT can be accessed;
- A database, by default, will be a sub-directory of the DOCROOT of the same name;
- The DOCROOT path cannot be the root directory of a files system.

The default Direct-Link (TFST) includes an embedded `tfserver` in the process which means that only one Direct-Link application process at a time can access the document root. Multiple process access to a document root requires the use of the Client-Server (TFSR) configuration.

Database Name Specification (db_namespec)

The database name specification uses a Uniform Resource Identifier (URI) format to identify the TFS and database name. The syntax for the naming convention is:

```
dbname_spec:
    <tfs_spec>db_name

tfs_spec:
    'tfs://'
    | 'tfs-tcp://' tcp_addr '/'
    | 'tfs-shm://' shm_name '/'

tcp_addr:
    {tcp_id | hostname}':' portnum
    | ':' portnum

tcp_id:
    <[> <IPv4 address> <]>
    | '[' <IPv6 address> ']'
```

The elements above enclosed with chevron characters (< and >) are optional elements in the specification grammar.

Supported Transport Types (tfs_spec)

tfs-shm://shm_name/	The shared-memory communications transport is the default transport a RDM client running on the same machine as the TFS.
tfs-tcp://tcp_addr/	RDM supports IPv4 and IPv6 TCP/IP connections. Since the colon character (':') is used to separate the hostname and port parts of a TCP/IP address and the numeric notation for IPv6 addresses also contains colons, square brackets are required around an IPv6 address in numeric notation. Square brackets can also be used around IPv4 addresses in numeric notation, but are not required.
tfs:///	Uses the default transport type available. The RDM client will first attempt to connect to the TFS using the shared-memory communication transport followed by the TCP/IP communication transport.

Default TFS Addresses

Specifies the server port (TCP/IP) or name (non-TCP-IP) for connections. The default port is 21553 and the default name is RDM_21553. If a number is specified it will be used for the TCP/IP port and the server name will be RDM_port. If a string (non-number) is specified it will be used as the server name and the TCP/IP transport will not be initialized.

21553	Default port number (used in default shared-memory name and default TCP address below)
21553	Default shared-memory name.
localhost:21553	Default TCP host and port address

The tfs_spec should not be used for TFS types: TFSS and TFST. Only TFSR currently supports the tfs_spec.

Database Name Specifications

db_namespec	Description
sales	"sales" on default TFS using default transport.
tfs:///sales	"sales" on default TFS using default transport.
tfs-tcp://www.raima.com/sales	"sales" using TCP/IP with the default port number at hostname "www.raima.com"
tfs-shm://partition-01/sales	"sales" using shared-memory to TFS named "partition-01"
tfs-tcp://[::1]:1530/sales	"sales" using TCP/IP IPv6 using the loopback address on port 1530
tfs-tcp://[fe80::40da:bf3f:ae9f:fe87]:2000/sales	"sales" using TCP/IP IPv6 to a specified machine on port 2000.
tfs-tcp://192.168.101.139:2000/sales	"sales" using TCP/IP IPv4 to a specified machine on port 2000.

Union Database Name Specifications

A database union is a unified view of the data in more than one identically structured database. It makes the multiple databases appear as one. A union of multiple databases differs from having multiple databases open in that:

- unioned databases must have identical dictionaries (DBD files)
- the union is viewed as one database, using one database number
- it is read-only

Database unions are intended to be used with distributed databases or database mirrors to create a single, merged view of data that is owned and updated in separate locations or by separate entities. Database specifications are separated by a vertical bar (|) in the db_namespec.

Note that a database union is a union of different instances of the same database schema (i.e., definition) contained on separate TFSs. This is not to be confused with the standard SQL **union** of **select** statements operation.

Example Union Database Open Specifications

```
d_open("tfs-shm://partition-01/sales|tfs-tcp://www.raima.com/sales", "r", task);
rsqlOpenDB(hdbc, "tfs-tcp://localhost:21553/sales|tfs-tcp://www.raima.com/sales:1530", "r");
open nsfawards as union of "tfs-shm://partiiion-01", "tfs-tcp://www.raima.com:1650":
```

Antiquarian Bookshop Database

Our fictional bookshop is located in Hertford, England (a very real and charming town north of London). It is located in a building constructed around 1735 and has two rather smallish rooms on two floors with floor-to-ceiling bookshelves throughout. Upon entering, one is immediately transported to a much earlier era being quite overwhelmed by the wonderful sight and odor of the ancient mahogany wood in which the entire interior is lined along with the rare and ancient books that reside on them. There is a little bell that announces one's entrance into the shop but it is not really needed, as the delightfully squeaky floor boards quite clearly makes your presence known.

In spite of the ancient setting and very old and rare books, this bookshop has a very modern Internet storefront through which it sells and auctions off its expensive inventory. A computer system contains a database describing the inventory and manages the sales and auction processes. The database schema for our bookshop is given below.

```
create database bookshop;

create table acctmgr(
  mgrid      char(7) primary key,
  name       char(24),
  hire_date  date,
  commission decimal(4,3)
);

create table author(
  last_name  char(13) primary key,
  full_name  char(35),
  gender     char(1) distinct values = 2,
  yr_born    smallint,
  yr_died    smallint,
  short_bio  varchar(250)
);

create table genres(
  text      char(31) primary key
);

create table subjects(
  text      char(51) primary key
);

create table book(
  bookid     char(14) primary key,
  last_name  char(13)
    references author on delete cascade on update cascade,
  title      varchar(255),
  descr     char(61),
  publisher  char(136),
  publ_year  smallint key,
  lc_class   char(33),
  date_acqd  date,
  date_sold  date,
  price     decimal(10,2),
  cost      decimal(10,2)
);

create table related_name(
  bookid     char(14)
```

```

        references book on delete cascade on update cascade,
        name          char(61)
    );

create table genres_books(
    bookid          char(14)
        references book on delete cascade on update cascade,
    genre          char(31)
        references genres
);

create table subjects_books(
    bookid          char(14)
        references book on delete cascade on update cascade,
    subject        char(51)
        references subjects
);

create table patron(
    patid          char(3) primary key,
    name          char(30),
    street         char(30),
    city          char(17),
    state         char(2),
    country       char(2),
    pc            char(10),
    email         char(63),
    phone         char(15),
    mgrid         char(7)
        references acctmgr
);

create table note(
    noteid        integer primary key,
    bookid        char(14)
        references book on delete cascade on update cascade,
    patid         char(3)
        references patron on delete cascade on update cascade
);

create table note_line(
    noteid        integer
        references note on delete cascade on update cascade,
    text          char(61)
);

create table sale(
    bookid        char(14)
        references book on delete cascade on update cascade,
    patid         char(3)
        references patron on delete cascade on update cascade
);

create table auction(
    aucid         integer primary key,
    bookid        char(14)
        references book on delete cascade on update cascade,
    mgrid         char(7)
        references acctmgr,
    start_date    date,

```

```

    end_date    date,
    reserve    decimal(10,2),
    curr_bid    decimal(10,2)
);

create table bid(
    aucid      integer
               references auction on delete cascade on update cascade,
    patid      char(3)
               references patron on delete cascade on update cascade,
    offer      decimal(10,2),
    bid_ts     timestamp
);

```

Descriptions for each of the above tables are given below.

Table 3. Bookshop Database Table Descriptions

Table Name	Description
author	Each row contains biographical information about a renowned author.
book	Contains information about each book in the bookshop inventory. The last_name column associates the book with its author. Books with a non null date_sold are no longer available.
genres	Table of genre names (e.g., "Historical fiction") with which particular books are associated via the genres_books table.
subjects	Table of subject names (e.g., "Cape Cod") with which particular books are associated via the subjects_books table.
related_name	Related names are names of individuals associated with a particular book. The names are usually hand-written in the book's front matter or on separate pages that were included with the book (e.g., letters) and identify the book's provenance (owners). Only a few books have related names. However, their presence can significantly increase the value of the book.
genres_books	Used to create a many-to-many relationship between genres and books.
subjects_books	Used to create a many-to-many relationship between subjects and books.
note	Connects each note_line to its associated book. Notes include edition info and other comments (often coded) relating to its condition.
note_line	One row for each line of text in a particular note.
acctmgr	Account manager are the bookshop employees responsible for servicing the patrons and managing auctions.
patron	Bookshop customers and their contact info. Connected to their purchases/bids through their relationship with the sale and auction tables.
sale	Contains one row for each book that has been sold. Connects the book with the patron who acquired through the bookid and patid columns.
auction	Some books are auctioned. Those that have been (or currently being) auctioned have a row in this table that identifies the account manager who oversees the auction. The reserve column specifies the minimum acceptable bid, curr_bid contains the current amount bid.
bid	Each row provides the bid history for a particular auction.

Foreign keys are declared using the **references** clause. Many are specified with the **on delete/update cascade** option indicating that deletions or updates to the referenced rows will cause the referencing row to automatically be deleted or updated as well.

A schema diagram depicting the inter-table relationships is shown below. As was mentioned above for the NSF awards database, the arrows represent a one-to-many relationship between the source and target tables and labels on the arrows identify the foreign key in the target table on which the relationship is formed.

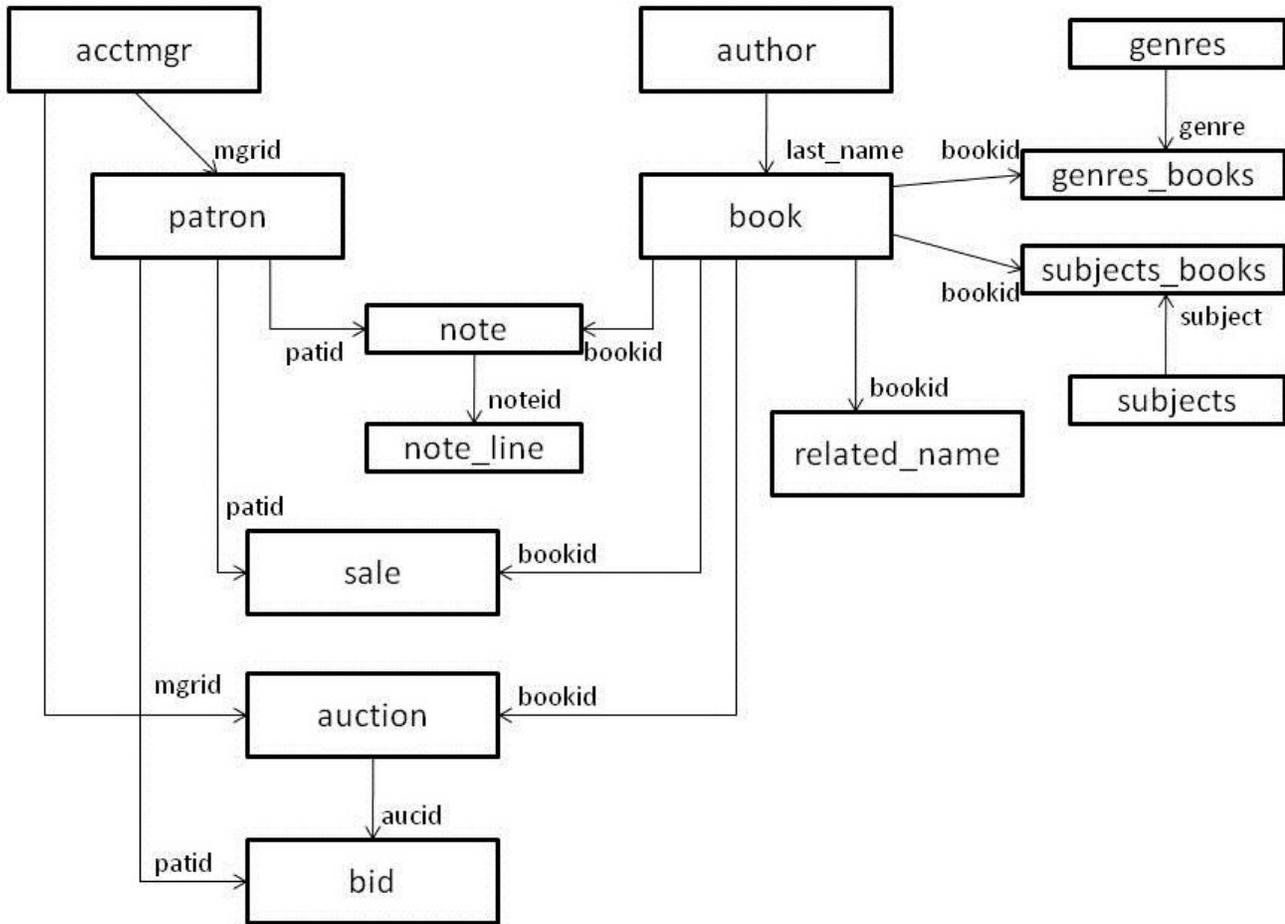


Figure 6 - Bookshop Database Schema Diagram

The sample data that is included with this example contains book descriptions that were obtained from the United States Library of Congress online card catalog: <http://catalog.loc.gov>. The short biographical sketches included with each author entry are condensed descriptions from information about each author contained on Wikipedia: <http://www.wikipedia.org>. The use of the Wikipedia information is governed by the Creative Commons Attribution-ShareAlike license: <http://creativecommons.org/licenses/by-sa/3.0/>. Pricing information and the JPEG files of photographs of some of the books in the database were obtained from the website for Peter Harrington Antiquarian Bookseller in Chelsea London, <http://www.peterharrington.co.uk>, which is a perfect real-world example of the kind of bookshop depicted in this example.

Replication/Mirroring Parameters (DBREP_CONNECT_PARAMS)**DBREP_CONNECT_PARAMS typedef**

```
typedef struct _dbrep_connect_params {
    DB_BOOLEAN        quiet;
    const char        *alias;
    const char        *dburl;
    const char        *dbuserid;
    const char        *id;
    const char        *hostname;
    const char        *dsn;
    int16_t           synchronize;
    uint16_t          port;
    REP_SLAVE_TYPE    sql_slave_type;
    uint16_t          override_inmem;
    uint32_t          nocopy;
    SLAVE_DONE_FCN    slave_done_notifier;
    const char        *stdout_file;
    const char        *name;
    TX_CONN_TYPE      transportBitmap;
    psp_thread_t      TID;
} DBREP_CONNECT_PARAMS;
```

DBREP_CONNECT_PARAMS_W typedef

```
typedef struct _dbrep_connect_paramsW {
    DB_BOOLEAN        quiet;
    const wchar_t     *alias;
    const wchar_t     *dburl;
    const wchar_t     *dbuserid;
    const wchar_t     *id;
    const wchar_t     *hostname;
    const wchar_t     *dsn;
    int16_t           synchronize;
    uint16_t          port;
    REP_SLAVE_TYPE    sql_slave_type;
    uint16_t          override_inmem;
    uint32_t          nocopy;
    SLAVE_DONE_FCN    slave_done_notifier;
    const wchar_t     *stdout_file;
    const wchar_t     *name;
    TX_CONN_TYPE      transportBitmap;
    psp_thread_t      TID;
} DBREP_CONNECT_PARAMS_W;
```

DBREP_CONNECT_PARAMS(W) Elements

Element	Declaration	Description
quiet	DB_BOOLEAN	
alias	const char * const wchar_t *	
dburl	const char * const wchar_t *	Database name URI (db_namespec)

Element	Declaration	Description
dbuserid	const char * const wchar_t *	userid to use when logging in
id	const char * const wchar_t *	optional unique id
hostname	const char * const wchar_t *	hostname of slave dbmirror
dsn	const char * const wchar_t *	dbrepsql data source name: dsn;user;pswd
synchronize	int16_t	1 if this replication is synchronous. -1 if persistent sync should be removed.
port	uint16_t	port of slave dbmirror
sql_slave_type	REP_SLAVE_TYPE	target server(RST_RDMS,RST_MYSQL..)
override_inmem	uint16_t	force inmemory master db files to disk on slave
nocopy	uint32_t	
slave_done_notifier	SLAVE_DONE_FCN	called when slave thread exits
psp_thread_t	TID	thread id of the connect thread
no_page_ref	uint16_t	Disable <i>page referencing</i> ¹ . TRUE or FALSE.
stdout_file	const char * const wchar_t *	The name of a file to write informational messages instead of displaying them to stdout. If the value is NULL or "stdout" the messages will be sent to stdout. If the value is an empty string ("") no messages will be displayed.
name	const char * const wchar_t *	ASCII Name of shared-memory transport. If NULL, the default transport name will be used (see Database Name Specification (db_namespec)),
transportBitmap	TX_CONN_TYPE	Bit map of transports to enable. TX_NONE Enable all available transports TX_TCP Enable TCP/IP communications TX_SHM Enable shared-memory communications

SLAVE_DONE_FCN Prototype

```
void my_slave_done_fcn(
    int32_t    slave_return)
```

SLAVE_DONE_FCN Parameters

slave_return (input) Return code from slave thread.

¹In a TFSR configuration - where the application runs in a separate process from the TFS (which would be running in a tfserver or other application), the system normally attempts to determine if the application is on the same physical machine as the TFS. If it is determined that the two processes are on the same machine, the code will use a shortcut - those pages that will be read from the database files and sent over to the application via TCP/IP or other connection are instead read directly from the disk by the user application. This is all handled internally by the software - no separate options are required. Some users, however may not wish for this to take place because of file permission issues or other reasons. In that case, this option can be specified and doing so will disable this shortcut - thus acting as if the user application and the TFS are on different machines and send all pages through the connection between the two.

SLAVE_DONE_FCN Description

When `slave_done_notifier` is not NULL, it is a callback function. The slave is started in `dbmirror/dbrep/dbrepsql` in a thread. When the slave thread exits, either by success or failure, the function specified in `slave_done_notifier` is called with the return code from the slave thread, `S_OKAY` or some error code. The `slave_done_notifier` only needs to be specified if an application needs to know when a slave ends. If you specify a callback function in `slave_done_notifier`, the value returned in `slaveId` will always be 0.

Replication/Mirroring Initialization Parameters (DBREP_INIT_PARAMS)**DBREP_INIT_PARAMS typedef**

```
typedef struct _dbrep_init_params {
    uint16_t      port;
    uint16_t      verbose;
    uint16_t      diskless;
    uint16_t      no_page_ref;
    uint16_t      ha_on;
    TFS_TYPE      tfs_type;
    const char    *stdout_file;
    const char    *name;
    TX_CONN_TYPE  transportBitmap;
} DBREP_INIT_PARAMS;
```

DBREP_INIT_PARAMS_W typedef

```
typedef struct _dbrep_init_paramsW {
    uint16_t      port;
    uint16_t      verbose;
    uint16_t      diskless;
    uint16_t      no_page_ref;
    uint16_t      ha_on;
    TFS_TYPE      tfs_type;
    const wchar_t *stdout_file;
    const wchar_t *name;
    TX_CONN_TYPE  transportBitmap;
} DBREP_INIT_PARAMS_W;
```

DBREP_INIT_PARAMS(W) Elements

Element	Declaration	Description
port	uint16_t	Server TPC/IP listen port number. The default port number (21553) will be used if set to zero (0).
verbose	uint16_t	Verbose mode, TRUE or FALSE.
diskless	uint16_t	Diskless mode, TRUE or FALSE.
no_page_ref	uint16_t	Disable <i>page referencing</i> ¹ . TRUE or FALSE.
ha_on	uint16_t	HA Synchronous Notifications, ON or OFF

¹In a TFSR configuration - where the application runs in a separate process from the TFS (which would be running in a tfs server or other application), the system normally attempts to determine if the application is on the same physical machine as the TFS. If it is determined that the two processes are on the same machine, the code will use a shortcut - those pages that will be read from the database files and sent over to the application via TCP/IP or other connection are instead read directly from the disk by the user application. This is all handled internally by the software - no separate options are required. Some users, however may not wish for this to take place because of file permission issues or other reasons. In that case, this option can be specified and doing so will disable this shortcut - thus acting as if the user application and the TFS are on different machines and send all pages through the connection between the two.

Replication/Mirroring Initialization Parameters (DBREP_INIT_PARAMS)

no_page_ref	uint16_t	Disable <i>page referencing</i> ¹ . TRUE or FALSE.
stdout_file	const char * const wchar_t *	The name of a file to write informational messages instead of displaying them to stdout. If the value is NULL or "stdout" the messages will be sent to stdout. If the value is an empty string ("") no messages will be displayed.
name	const char * const wchar_t *	ASCII Name of shared-memory transport. If NULL, the default transport name will be used (see Database Name Specification (db_namespec)),
transportBitmap	TX_CONN_TYPE	Bit map of transports to enable. TX_NONE Enable all available transports TX_TCP Enable TCP/IP communications TX_SHM Enable shared-memory communications

¹In a TFSR configuration - where the application runs in a separate process from the TFS (which would be running in a tfsrvr or other application), the system normally attempts to determine if the application is on the same physical machine as the TFS. If it is determined that the two processes are on the same machine, the code will use a shortcut - those pages that will be read from the database files and sent over to the application via TCP/IP or other connection are instead read directly from the disk by the user application. This is all handled internally by the software - no separate options are required. Some users, however may not wish for this to take place because of file permission issues or other reasons. In that case, this option can be specified and doing so will disable this shortcut - thus acting as if the user application and the TFS are on different machines and send all pages through the connection between the two.

Shared Memory Transport:

The shared memory transport is an alternative to TCP/IP communication when both the server and the client communicating with it reside on the same machine. Instead of creating a socket for communications, a simple shared memory buffer is used.

On Windows this uses a file mapping object that is backed by the system paging file.

On non-Windows systems this uses a memory mapped file which will be created in '/tmp' by default (this can be changed using the RDMTEMP environment variable if /tmp is not desired - but it must be the same location for both the server process and each client accessing that server).

Data being sent between the two processes will be copied into this buffer and then the other side will be notified and it will copy the data out. In order to implement this efficiently, certain atomic functions are required and they are not present on every system. Those systems that do not support these atomic functions will not have the shared memory transport available to them.

The benefit of the shared memory transport for same machine communication to a server is performance. The share memory transport is faster than TCP/IP as there is less overhead to go through to get data back and forth. Both shared memory and TCP/IP can be available at the same time so it is possible to have remote clients communicate with the server via TCP/IP and local clients communicate via shared memory.

When communicating via TCP/IP, a hostname and a port are required to identify which server to communicate with. When using shared memory, the hostname is not required (as the client will always be on the same machine as the server), and instead of a port, a name is used. This name is string of characters and needs to follow the guidelines for a filename on the system being used (i.e. don't use a directory character ('/' or '\') in the name).

Opening Slave Databases from Programs

The `d_open` or `d_iopen` functions are able to open a database mirror for reading. Referring to the example above, an application running on `RLM-lptp` can open the database mirrored from `tfs.raima.com` with the following statement:

```
d_open("tfs.raima.com-21553/sales", "r", task);
```

Or, if the application is not necessarily running on `RLM-lptp` (although it may be), the following reference to the sales database will work from anywhere:

```
d_open("tfs.raima.com-21553/sales@RLM-lptp.raima.com", "r", task);
```

Generally, the advantage of opening a mirror for reading is because it is local and therefore faster.

Another reason to mirror a database to the local computer is to view it in a union with other identically structured databases. For example, suppose that `acctg-main` needs to create a report on `inventory`, which is maintained independently at two sites. The first is on `tfs.raima.com` (as in the example above), and the second is on `sf.raima.uk`. A mirror of `inventory` from both sites is mirrored to the local computer. Then the following statement will be able to open a merged view of the inventory:

```
d_open("tfs.raima.com-21553/inventory|sf.raima.uk-21553/inventory", "r", task);
```

Index

H

HA-API

ha_isTransActive 40

ha_login 34

ha_logout 35

ha_quiesce 37

ha_status 36

ha_wakeup 38

M

MIRRORING-API

d_dbmir_connect 23

d_dbmir_connectW 23

d_dbmir_disconnect 25

d_dbmir_disconnectW 25

d_dbmir_init 21

d_dbmir_initW 21

d_dbmir_start 27

d_dbmir_stop 29

d_dbmir_term 30

U

Utility

dbget 55

dbmirror 53

dbrep 57