Oracle Berkeley DB

Getting Started with Replicated Applications for C++

Release 4.6

Legal Notice

Published 6/30/2007

Table of Contents

Preface

This document describes how to write replicated Berkeley DB applications. The APIs used to implement replication in your application are described here. This book describes the concepts surrounding replication, the scenarios under which you might choose to use it, and the architectural requirements that a replication application has over a transactional application.

This book is aimed at the software engineer responsible for writing a replicated DB application.

This book assumes that you have already read and understood the concepts contained in the *Berkeley DB Getting Started with Transaction Processing* guide.

Conventions Used in this Book

The following typographical conventions are used within in this manual:

Class names are represented in `monospaced font`, as are `method names`. For example: "`DbEnv::open()` is a `DbEnv` class method."

Variable or non-literal text is presented in *italics*. For example: "Go to your *DB_INSTALL* directory."

Program examples are displayed in a `monospaced font` on a shaded background. For example:

```
typedef struct vendor {
    char name[MAXFIELD];            // Vendor name
    char street[MAXFIELD];          // Street name and number
    char city[MAXFIELD];            // City
    char state[3];                  // Two-digit US state code
    char zipcode[6];                // US zipcode
    char phone_number[13];          // Vendor phone number
} VENDOR;
```

In some situations, programming examples are updated from one chapter to the next. When this occurs, the new code is presented in **`monospaced bold`** font. For example:

```
typedef struct vendor {
    char name[MAXFIELD];            // Vendor name
    char street[MAXFIELD];          // Street name and number
    char city[MAXFIELD];            // City
    char state[3];                  // Two-digit US state code
    char zipcode[6];                // US zipcode
    char phone_number[13];          // Vendor phone number
    char sales_rep[MAXFIELD];       // Name of sales representative
    char sales_rep_phone[MAXFIELD]; // Sales rep's phone number
} VENDOR;
```

Finally, notes of special interest are represented using a note block such as this.

For More Information

Beyond this manual, you may also find the following sources of information useful when building a transactional DB application:

- Getting Started with Transaction Processing for C++
  [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg_txn/CXX/index.html]
- Getting Started with Berkeley DB for C++
  [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg/CXX/index.html]
- Berkeley DB Programmer's Reference Guide
  [http://www.oracle.com/technology/documentation/berkeley-db/db/ref/toc.html]
- Berkeley DB C++ API
  [http://www.oracle.com/technology/documentation/berkeley-db/db/api_cxx/frame.html]

Chapter 1. Introduction

This book provides a thorough introduction and discussion on replication as used with Berkeley DB (DB). It begins by offering a general overview to replication and the benefits it provides. It also describes the APIs that you use to implement replication, and it describes architecturally the things that you need to do to your application code in order to use the replication APIs. Finally, it discusses the differences in backup and restore strategies that you might pursue when using replication, especially where it comes to log file removal.

You should understand the concepts from the *Berkeley DB Getting Started with Transaction Processing* guide before reading this book.

Overview

The DB replication APIs allow you to distribute your database write operations (performed on a read-write master) to one or more read-only *replicas*. For this reason, DB's replication implementation is said to be a *single master, multiple replica* replication strategy.

Note that your database write operations can occur only on the master; any attempt to write to a replica results in an error being raised by the DB API used to perform the write.

A single replication master and all of its replicas are referred to as a *replication group*. While all members of the replication group can reside on the same machine, usually each replication participant is placed on a separate physical machine somewhere on the network.

Note that all replication applications must first be transactional applications. The data that the master transmits its replicas are log records that are generated as records are updated. Upon transactional commit, the master transmits a transaction record which tells the replicas to commit the records they previously received from the master. In order for all of this to work, your replicated application must also be a transactional application. For this reason, it is recommended that you write and debug your DB application as a stand-alone transactional application before introducing the replication layer to your code.

Finally, be aware that all machines participating in the replication group must share identical *endianess*. Endianess is the byte-order strategy used by the computing platform to store multibyte numbers. Berkeley DB is sensitive to byte order and so your replication infrastructure must be consistent in this area. We expect to remove this requirement in some future release of the product.

Replication Environments

The most important requirement for a replication participant is that it must use a unique Berkeley DB database environment independent of all other replication participants. So while multiple replication participants can reside on the same physical machine, no two such participants can share the same environment home directory.

For this reason, technically replication occurs between unique *database environments*. So in the strictest sense, a replication group consists of a *master environment* and one or more *replica environments*. However, the reality is that for production code, each such environment will usually be located on its own unique machine. Consequently, this manual sometimes talks about *replication sites*, meaning the unique combination of environment home directory, host and port that a specific replication application is using.

There is no DB-specified limit to the number of environments which can participate in a replication group. The only limitation here is one of resources — network bandwidth, for example.

(Note, however, that the replication framework does place a limit on the number of environments you can use. See Replication Framework Overview (page 5) for details.)

Also, DB's replication implementation requires all participating environments to be assigned IDs that are locally unique to the given environment. Depending on the replication APIs that you choose to use, you may or may not need to manage this particular detail.

For detailed information on database environments, see the *Berkeley DB Getting Started with Transaction Processing* guide. For more information on environment IDs, see the *Berkeley DB Programmer's Reference Guide*.

Replication Databases

DB's databases are managed and used in exactly the same way as if you were writing a non-replicated application, with a couple of caveats. First, the databases maintained in a replicated environment must reside either in the `ENV_HOME` directory, or in the directory identified by the `DbEnv::set_data_dir()` method. Unlike non-replication applications, you cannot place your databases in a subdirectory below these locations. You should also not use full path names for your databases or environments as these are likely to break as to replicated to other machines.

Communications Layer

In order to transmit database writes to the replication replicas, DB requires a communications layer. DB is agnostic as to what this layer should look like. The only requirement is that it be capable of passing two opaque data objects and an environment ID from the master to its replicas without corruption.

Because replicas are usually placed on different machines on the network, the communications layer is usually some kind of a network-aware implementation. Beyond that, its implementation details are largely up to you. It could use TCP/IP sockets, for example, or it could use raw sockets if they perform better for your particular application.

Note that you may not have to write your own communications layer. DB provides a replication framework that includes a fully-functional TCP/IP-based communications layer. See The Replication APIs (page 4) for more information.

See the *Berkeley DB Programmer's Reference Guide* for a description of how to write your own custom replication communications layer.

Selecting a Master

Every replication group is allowed one and only one master environment. Almost always, masters are selected by holding an *election*. All such elections are performed by the underlying Berkeley DB replication code so you have to do very little to implement them.

When holding an election, replicas "vote" on who should be the master. Among replicas participating in the election, the one with the most up-to-date set of log records will win the election. Note that it's possible for there to be a tie. When this occurs, priorities are used to select the master. See Holding Elections (page 6) for details.

For more information on holding and managing elections, see Holding Elections (page 6).

Replication Benefits

Replication offers your application a number of benefits that can be a tremendous help. Primarily replication's benefits revolve around performance, but there is also a benefit in terms of data durability guarantees.

Briefly, the reasons why you might choose to implement replication in your DB application are:

- Improved application reliability.

  By spreading your data across multiple machines, you can ensure that your application's data continues to be available even in the event of a hardware failure on any given machine in the replication group.

- Improve read performance.

  By using replication you can spread data reads across multiple machines on your network. Doing so allows you to vastly improve your application's read performance. This strategy might be particularly interesting for applications that have readers on remote network nodes; you can push your data to the network's edges thereby improving application data read responsiveness.

  Additionally, depending on how you partition your data across your replicas, any given replica may only need to cache part of your data, decreasing cache misses and reducing I/O on the client.

- Improve transactional commit performance

  In order to commit a transaction and achieve a transactional durability guarantee, the commit must be made *durable*. That is, the commit must be written to disk (usually, but not always, synchronously) before the application's thread of control can continue operations.

  Replication allows you to avoid this disk I/O and still maintain a degree of durability by *committing to the network*. In other words, you relax your transactional durability

guarantees on the master, but by virtue of replicating the data across the network you gain some additional durability guarantees above what is provided locally.

Usually this strategy is implemented using some form of an asynchronous transactional commit on the master. In this way your data writes will eventually be written to disk, but your application will not have to wait for the disk I/O to complete before continuing with its next operation.

Note that it is possible to cause DB's replication implementation to wait to hear from one or more replica's as to whether they have successfully saved the write before continuing. However, in this case you might be trading performance for a even higher durability guarantee (see below).

- Improve data durability guarantee.

  In a traditional transactional application, you commit your transactions such that data modifications are saved to disk. Beyond this, the durability of your data is dependent upon the backup strategy that you choose to implement for your site.

  Replication allows you to increase this durability guarantee by ensuring that data modifications are written to multiple machines. This means that multiple disks, disk controllers, power supplies, and CPUs are used to ensure that your data modification makes it to stable storage. In other words, replication allows you to minimize the problem of a single point of failure by using more hardware to guarantee your data writes.

  If you are using replication for this reason, then you probably will want to configure your application such that it waits to hear about a successful commit from one or more replicas before continuing with the next operation. This will obviously impact your application's write performance to some degree — with the performance penalty being largely dependent upon the speed and stability of the network connecting your replication group.

  For more information, see Permanent Message Handling (page 34).

The Replication APIs

There are two ways that you can choose to implement replication in your transactional application. The first, and preferred, mechanism is to use the pre-packaged replication framework that comes with the DB distribution. This framework should be sufficient for most customers.

If for some reason the Replication Framework does not meet your application's technical requirements, you will have to use the replication APIs available through the Berkeley DB library to write your own custom replication framework.

Both of these approaches are described in slightly greater detail in this section. The bulk of the chapters later in this book are dedicated to these two replication implementation mechanisms.

Replication Framework Overview

DB's pre-packaged replication framework exists as a layer on top of the DB library. The replication framework is a multi-threaded implementation that allows you to easily add replication to your existing transactional application. You access and manage the replication framework using methods that are available off the `DbEnv` class.

The replication framework:

- Provides a multi-threaded communications layer using pthreads (on Unix-style systems and similar derivatives such as Mac OS X), or Windows threads on Microsoft Windows systems.
- Uses TCP/IP sockets. Network traffic is handled via threads that handle inbound and outbound messages. However, each process uses a single socket that is shared using `select()`.

  Note that for this reason, the replication framework is limited to a maximum of 60 replicas (on Windows) and approximately 1000 replicas (on Unix and related systems), depending on how your system is configured.
- Requires a single process for the master replica.
- Requires that only one instance of the environment handle be used.
- Upon application startup, a master can be selected either manually or via elections. After startup time, however, during the course of normal operations it is possible for the replication group to need to locate a new master (due to network or other hardware related problems, for example) and in this scenario elections are always used to select the new master.

If your application has technical requirements that do not conform to the implementation provided by the replication framework, you must write a custom replication framework using the DB replication APIs directly. See the next section for introductory details.

Replication API Overview

The replication API is a series of Berkeley DB library classes and methods that you can use to build your own replication infrastructure. You should use the replication API only if the replication framework does not meet your application's technical requirements.

To make use of the replication API, you must write your own networking code. This frees you from the technical constraints imposed by the replication framework. For example, by writing your own framework, you can:

- Use a threading package other than pthreads (Unix) or Windows threads (Microsoft Windows). This might be interesting to you if you are using a platform whose preferred threading package is something other than (for example) pthreads, such as is the case for Sun Microsystem's Solaris operating systems.

- Implement your own sockets. The replication framework uses TCP/IP sockets. While this should be acceptable for the majority of applications, sometimes UDP or even raw sockets might be desired.

- Write a multi-process master replica.

For information on writing a replicated application using the Berkeley DB replication APIs, see the *Berkeley DB Programmer's Reference Guide*.

Holding Elections

Finding a master environment is one of the fundamental activities that every replication replica must perform. Upon startup, the underlying DB replication code will attempt to locate a master. If a master cannot be found, then the environment should initiate an election.

How elections are held depends upon the API that you use to implement replication. For example, if you are using the replication framework elections are held transparently without any input from your application's code. In this case, DB will determine which environment is the master and which are replicas.

Influencing Elections

If you want to control the election process, you can declare a specific environment to be the master. Note that for the replication framework, it is only possible to do this at application startup. Should the master become unavailable during run-time for any reason, an election is held. The environment that receives the most number of votes, wins the election and becomes the master. A machine receives a vote because it has the most number of log records.

Because ties are possible when elections are held, it is possible to influence which environment will win the election. How you do this depends on which API you are using. In particular, if you are writing a custom replication layer, then there are a great many ways to manually influence elections.

One such mechanism is priorities. When votes are cast during an election, both the number of log records contained in the environment *and* the environment's priority are considered. So given two environments with the same number of log records, votes are cast for the environment with the higher priority.

Therefore, if you have a machine that you prefer to become a master in the event of an election, assign it a high priority. Assuming that the election is held at a time when the preferred machine has up-to-date log records, that machine will win the election.

Winning Elections

To win an election:

1. There cannot currently be a master environment.

2. The environment must have the most recent log records. Part of holding the election is determining which environments have the most recent log records. This process happens automatically; your code does not need to involve itself in this process.

3. The environment must receive the most number of votes from the replication environments that are participating in the election.

If you are using the replication framework, then in the event of a tie vote the environment with the highest priority wins the election. If two or more environments receive the same number of votes and have the same priority, then the underlying replication code picks one of the environments to be the winner. Which winner will be picked by the replication code is unpredictable from the perspective of your application code.

Switching Masters

To switch masters:

1. Start up the environment that you want to be master as normal. At this time it is a replica. Make sure this environment has a higher priority than all the other environments.

2. Allow the new environment to run for a time as a replica. This allows it to obtain the most recent copies of the log files.

3. Shut down the current master. This should force an election. Because the new environment has the highest priority, it will win the election, provided it has had enough time to obtain all the log records.

4. Optionally restart the old master environment. Because there is currently a master environment, an election will not be held and the old master will now run as a replica environment.

Permanent Message Handling

Messages received by a replica may be marked with an special flag that indicates the message is permanent. Custom replicated applications will receive notification of this flag via the `DB_REP_ISPERM` return value from the `DbEnv::rep_process_message()` method. There is no hard requirement that a replication application look for, or respond to, this return code. However, because robust replicated applications typically do manage permanent messages, we introduce the concept here.

A message is marked as being permanent if the message affects transactional integrity. For example, transaction commit messages are an example of a message that is marked permanent. What the application does about the permanent message is driven by the durability guarantees required by the application.

For example, consider what the replication framework does when it has permanent message handling turned on and a transactional commit record is sent to the replicas. First, the replicas must transactional-commit the data modifications identified by the

message. And then, upon a successful commit, the replication framework sends the master a message acknowledgment.

For the master (again, using the replication framework), things are a little more complicated than simple message acknowledgment. Usually in a replicated application, the master commits transactions asynchronously; that is, the commit operation does not block waiting for log data to be flushed to disk before returning. So when a master is managing permanent messages, it typically blocks the committing thread immediately before `commit()` returns. The thread then waits for acknowledgments from its replicas. If it receives enough acknowledgments, it continues to operate as normal.

If the master does not receive message acknowledgments — or, more likely, it does not receive *enough* acknowledgments — the committing thread flushes its log data to disk and then continues operations as normal. The master application can do this because replicas that fail to handle a message, for whatever reason, will eventually catch up to the master. So by flushing the transaction logs to disk, the master is ensuring that the data modifications have made it to stable storage in one location (its own hard drive).

When Not to Manage Permanent Messages

There are two reasons why you might choose to not implement permanent messages. In part, these go to why you are using replication in the first place.

One class of applications uses replication so that the application can improve transaction through-put. Essentially, the application chooses a reduced transactional durability guarantee so as to avoid the overhead forced by the disk I/O required to flush transaction logs to disk. However, the application can then regain that durability guarantee to a certain degree by replicating the commit to some number of replicas.

Using replication to improve an application's transactional commit guarantee is called *replicating to the network*.

In extreme cases where performance is of critical importance to the application, the master might choose to both use asynchronous commits *and* decide not to wait for message acknowledgments. In this case the master is simply broadcasting its commit activities to its replicas without waiting for any sort of a reply. An application like this might also choose to use something other than TCP/IP for its network communications since that protocol involves a fair amount of packet acknowledgment all on its own. Of course, this sort of an application should also be very sure about the reliability of both its network and the machines that are hosting its replicas.

At the other end of the extreme, there is a class of applications that use replication purely to improve read performance. This sort of application might choose to use synchronous commits on the master because write performance there is not of critical performance. In any case, this kind of an application might not care to know whether its replicas have received and successfully handled permanent messages because the primary storage location is assumed to be on the master, not the replicas.

Managing Permanent Messages

With the exception of a rare breed of replicated applications, most masters need some view as to whether commits are occurring on replicas as expected. At a minimum, this is because masters will not flush their log buffers unless they have reason to expect that permanent messages have not been committed on the replicas.

That said, it is important to remember that managing permanent messages involves a fair amount of network traffic. The messages must be sent to the replicas and the replicas must then acknowledge the message. This represents a performance overhead that can be worsened by congested networks or outright outages.

Therefore, when managing permanent messages, you must first decide on how many of your replicas must send acknowledgments before your master decides that all is well and it can continue normal operations. When making this decision, you could decide that *all* replicas must send acknowledgments. But unless you have only one or two replicas, or you are replicating over a very fast and reliable network, this policy could prove very harmful to your application's performance.

Therefore, a common strategy is to wait for an acknowledgment from a simple majority of replicas. This ensures that commit activity has occurred on enough machines that you can be reliably certain that data writes are preserved across your network.

Remember that replicas that do not acknowledge a permanent message are not necessarily unable to perform the commit; it might be that network problems have simply resulted in a delay at the replica. In any case, the underlying DB replication code is written such that a replica that falls behind the master will eventually take action to catch up.

Depending on your application, it may be possible for you to code your permanent message handling such that acknowledgment must come from only one or two replicas. This is a particularly attractive strategy if you are closely managing which machines are eligible to become masters. Assuming that you have one or two machines designated to be a master in the event that the current master goes down, you may only want to receive acknowledgments from those specific machines.

Finally, beyond simple message acknowledgment, you also need to implement an acknowledgment timeout for your application. This timeout value is simply meant to ensure that your master does not hang indefinitely waiting for responses that will never come because a machine or router is down.

Implementing Permanent Message Handling

How you implement permanent message handling depends on which API you are using to implement replication. If you are using the replication framework, then permanent message handling is configured using policies that you specify to the framework. In this case, you can configure your application to:

- Ignore permanent messages (the master does not wait for acknowledgments).

- Require acknowledgments from a quorum. A quorum is reached when acknowledgments are received from the minimum number of electable replicas needed to ensure that the record remains durable if an election is held.

  The goal here is to be absolutely sure the record is durable. The master wants to hear from enough electable replicas that they have committed the record so that if an election is held, the master knows the record will exist even if a new master is selected.

  This is the default policy.

- Require an acknowledgment from at least one replica.
- Require acknowledgments from all replicas.
- Require an acknowledgment from a peer. (The replication framework allows you to designate one environment as a peer of another).
- Require acknowledgments from all peers.

Note that the replication framework simply flushes its transaction logs and moves on if a permanent message is not sufficiently acknowledged.

For details on permanent message handling with the replication framework, see Permanent Message Handling (page 34).

If these policies are not sufficient for your needs, or if you want your application to take more corrective action than simply flushing log buffers in the event of an unsuccessful commit, then you must use write a custom replication implementation.

For custom replication implementation, messages are sent from the master to its replica using a `send()` callback that you implement. Note, however, that DB's replication code automatically sets the permanent flag for you where appropriate.

If the `send()` callback returns with a non-zero status, DB flushes the transaction log buffers for you. Therefore, you must cause your `send()` callback to block waiting for acknowledgments from your replicas. As a part of implementing the `send()` callback, you implement your permanent message handling policies. This means that you identify how many replicas must acknowledge the message before the callback can return `0`. You must also implement the acknowledgment timeout, if any.

Further, message acknowledgments are sent from the replicas to the master using a communications channel that you implement (the replication code does not provide a channel for acknowledgments). So implementing permanent messages means that when you write your replication communications channel, you must also write it in such a way as to also handle permanent message acknowledgments.

For more information on implementing permanent message handling using a custom replication layer, see the *Berkeley DB Programmer's Reference Guide*.

Chapter 2. Transactional Application

In this chapter, we build a simple transaction-protected DB application. Throughout the remainder of this book, we will add replication to this example. We do this to underscore the concepts that we are presenting in this book; the first being that you should start with a working transactional program and then add replication to it.

Note that this book assumes you already know how to write a transaction-protected DB application, so we will not be covering those concepts in this book. To learn how to write a transaction-protected application, see the *Berkeley DB Getting Started with Transaction Processing* guide.

Application Overview

Our application maintains a stock market quotes database. This database contains records whose key is the stock market symbol and whose data is the stock's price.

The application operates by presenting you with a command line prompt. You then enter the stock symbol and its value, separated by a space. The application takes this information, writes it to the database.

To see the contents of the database, simply press `return` at the command prompt.

To quit the application, type 'quit' or 'exit' at the command prompt.

For example, the following illustrates the application's usage. In it, we use entirely fictitious stock market symbols and price values.

```
> ./SimpleTxn -h env_home_dir
QUOTESERVER> stock1 88
QUOTESERVER> stock2 .08
QUOTESERVER>
        Symbol  Price
        ======  =====
        stock1  88

QUOTESERVER> stock1 88.9
QUOTESERVER>
        Symbol  Price
        ======  =====
        stock1  88.9
        stock2  .08

QUOTESERVER> quit
>
```

Program Listing

Our example program is a fairly simple transactional application. At this early stage of its development, the application contains no hint that it must be network-aware so the only command line argument that it takes is one that allows us to specify the environment home directory. (Eventually, we will specify things like host names and ports from the command line).

Note that the application performs all writes under the protection of a transaction; however, multiple database operations are not performed per transaction. Consequently, we simplify things a bit by using autocommit for our database writes.

Also, this application is single-threaded. It is possible to write a multi-threaded or multi-process application that performs replication. That said, the concepts described in this book are applicable to both single threaded and multi-threaded applications so nothing is gained by multi-threading this application other than distracting complexity. This manual does, however, identify where care must be taken when performing replication with a non-single threaded application.

Finally, remember that transaction processing is not described in this manual. Rather, see the *Berkeley DB Getting Started with Transaction Processing* guide for details on that topic.

Class: RepConfigInfo

Before we begin, we present a class that we will use to maintain useful information for us. Under normal circumstances, this class would not be necessary for a simple transactional example such as this. However, this code will grow into a replicated example that needs to track a lot more information for the application, and so we lay the groundwork for it here.

The class that we create is called `RepConfigInfo` and its only purpose at this time is to track the location of our environment home directory.

```
#include <db_cxx.h>
#include <iostream>

class RepConfigInfo {
public:
    RepConfigInfo();
    virtual ~RepConfigInfo();

public:
    char* home;
};


RepConfigInfo::RepConfigInfo()
{
    home = "TESTDIR";
```

```
}

RepConfigInfo::~RepConfigInfo()
{
}
```

Class: RepMgr

Our transactional example will instantiate a class, `RepMgr`, that performs all our work for us. Before we implement our `main()` function, we show the `RepMgr` class declaration.

First, we provide some declarations and definitions that are needed later in our example:

```
using std::cout;
using std::cin;
using std::cerr;
using std::endl;
using std::flush;

#define CACHESIZE   (10 * 1024 * 1024)
#define DATABASE    "quote.db"

const char *progname = "SimpleTxn";
```

And then we define our `RepMgr` class:

```
class RepMgr
{
public:
    // Constructor.
    RepMgr();
    // Initialization method. Creates and opens our environment handle.
    int init(RepConfigInfo* config);
    // The doloop is where all the work is performed.
    int doloop();
    // terminate() provides our shutdown code.
    int terminate();

private:
    // disable copy constructor.
    RepMgr(const RepMgr &);
    void operator = (const RepMgr &);

    // internal data members.
    RepConfigInfo   *app_config;
    DbEnv           dbenv;

    // private methods.
    // print_stocks() is used to display the contents of our database.
```

```
    static int print_stocks(Db *dbp);
};
```

Note that we show the implementation of the various `RepMgr` methods later in this section.

Function: usage()

Our `usage()` is at this stage of development trivial because we only have one command line argument to manage. Still, we show it here for the sake of completeness.

```
static void usage()
{
    cerr << "usage: " << progname << endl
         << "-h home" << endl;

    exit(EXIT_FAILURE);
}
```

Function: main()

Now we provide our `main()` function. This is a trivial function whose only job is to collect command line information, then instantiate a `RepMgr` object, run it, then terminate it.

We begin by declaring some useful variables. Of these, note that we instantiate our `RepConfigInfo` object here. Recall that this is used to store information useful to our code. This class becomes more interesting later in this book.

```
int main(int argc, char **argv)
{
    RepConfigInfo config;
    char ch;
    int ret;
```

Then we collect our command line information. Again, this is at this point fairly trivial:

```
    // Extract the command line parameters
    while ((ch = getopt(argc, argv, "h:")) != EOF) {
        switch (ch) {
        case 'h':
            config.home = optarg;
            break;
        case '?':
        default:
            usage();
        }
    }

    // Error check command line.
    if (config.home == NULL)
        usage();
```

Now we instantiate and initialize our RepMgr class, which is what is responsible for doing all our real work. The RepMgr::init() method creates and opens our environment handle.

```
    RepMgr runner;
    try {
        if((ret = runner.init(&config)) != 0)
            goto err;
```

Then we call the RepMgr::doloop() method, which is where the actual transactional work is performed for this application.

```
        if((ret = runner.doloop()) != 0)
            goto err;
```

Finally, catch exceptions and terminate the program:

```
    } catch (DbException dbe) {
        cerr << "Caught an exception during initialization or"
            << " processing: " << dbe.what() << endl;
    }
err:
    runner.terminate();
    return 0;
}
```

Method: RepMgr::init()

The RepMgr::init() method is used to create and open our environment handle. For readers familiar with writing transactional DB applications, there should be no surprises here. However, we will be adding to this in later chapters as we roll replication into this example.

First, we show the class constructor implementation, which is only used to initialize a few variables:

```
RepMgr::RepMgr() : app_config(0), dbenv(0)
{
}
```

We now provide the init() method implementation. The only thing of interest here is that we specify DB_TXN_NOSYNC to our environment. This causes our transactional commits to become non-durable, which is something that we are doing only because of the nature of our example.

```
int RepMgr::init(RepConfigInfo *config)
{
    int ret = 0;

    app_config = config;

    dbenv.set_errfile(stderr);
    dbenv.set_errpfx(progname);
```

```
    /*
     * We can now open our environment.
     */
    dbenv.set_cachesize(0, CACHESIZE, 0);
    dbenv.set_flags(DB_TXN_NOSYNC, 1);

    try {
        dbenv.open(app_config->home,
            DB_CREATE |
            DB_RECOVER |
            DB_INIT_LOCK |
            DB_INIT_LOG |
            DB_INIT_MPOOL |
            DB_INIT_TXN,
            0);
    } catch(DbException dbe) {
        cerr << "Caught an exception during DB environment open." << endl
            << "Ensure that the home directory is created prior to starting"
            << " the application." << endl;
        ret = ENOENT;
        goto err;
    }

err:
    return ret;
}
```

Finally, we present the `RepMgr::terminate()` method here. All this does is close the environment handle. Again, there should be no surprises here, but we provide the implementation for the sake of completeness anyway.

```
int RepMgr::terminate()
{
    try {
        dbenv.close(0);
    } catch (DbException dbe) {
        cerr << "error closing environment: " << dbe.what() << endl;
    }
    return 0;
}
```

Method: RepMgr::doloop()

Having written our `main()` function and support utility methods, we now implement our application's primary data processing method. This method provides a command prompt at which the user can enter a stock ticker value and a price for that value. This information is then entered to the database.

To display the database, simply enter `return` at the prompt.

To begin, we declare a database pointer, several `Dbt` variables, and the usual assortment of variables used for buffers and return codes. We also initialize all of this.

```
#define BUFSIZE 1024
int RepMgr::doloop()
{
    Db *dbp;
    Dbt key, data;
    char buf[BUFSIZE], *rbuf;
    int ret;

    dbp = NULL;
    memset(&key, 0, sizeof(key));
    memset(&data, 0, sizeof(data));
    ret = 0;
```

Next, we begin the loop and we immediately open our database if it has not already been opened. Notice that we specify autocommit when we open the database. In this case, autocommit is important because we will only ever write to our database using it. There is no need for explicit transaction handles and commit/abort code in this application, because we are not combining multiple database operations together under a single transaction.

Autocommit is described in greater detail in the *Berkeley DB Getting Started with Transaction Processing* guide.

```
    for (;;) {
        if (dbp == NULL) {
            dbp = new Db(&dbenv, 0);

            // Set page size small so page allocation is cheap.
            if ((ret = dbp->set_pagesize(512)) != 0)
                goto err;

            try {
                dbp->open(NULL, DATABASE, NULL, DB_BTREE,
                    DB_CREATE | DB_AUTO_COMMIT, 0);
            } catch(DbException dbe) {
                dbenv.err(ret, "DB->open");
                throw dbe;
            }
        }
```

Now we implement our command prompt. This is a simple and not very robust implementation of a command prompt. If the user enters the keywords `exit` or `quit`, the loop is exited and the application ends. If the user enters nothing and instead simply presses `return`, the entire contents of the database is displayed. We use our `print_stocks()` method to display the database. (That implementation is shown next in this chapter.)

Notice that very little error checking is performed on the data entered at this prompt. If the user fails to enter at least one space in the value string, a simple help message is printed and the prompt is returned to the user. That is the only error checking performed here. In a real-world application, at a minimum the application would probably check to ensure that the price was in fact an integer or float value. However, in order to keep this example code as simple as possible, we refrain from implementing a thorough user interface.

```
        cout << "QUOTESERVER" ;
        cout << "> " << flush;

        if (fgets(buf, sizeof(buf), stdin) == NULL)
            break;
        if (strtok(&buf[0], " \t\n") == NULL) {
            switch ((ret = print_stocks(dbp))) {
            case 0:
                continue;
            default:
                dbp->err(ret, "Error traversing data");
                goto err;
            }
        }
        rbuf = strtok(NULL, " \t\n");
        if (rbuf == NULL || rbuf[0] == '\0') {
            if (strncmp(buf, "exit", 4) == 0 ||
                strncmp(buf, "quit", 4) == 0)
                break;
            dbenv.errx("Format: TICKER VALUE");
            continue;
        }
```

Now we assign data to the `Dbt`s that we will use to write the new information to the database.

```
        key.set_data(buf);
        key.set_size((u_int32_t)strlen(buf));

        data.set_data(rbuf);
        data.set_size((u_int32_t)strlen(rbuf));
```

Having done that, we can write the new information to the database. Remember that this application uses autocommit, so no explicit transaction management is required. Also, the database is not configured for duplicate records, so the data portion of a record is overwritten if the provided key already exists in the database. However, in this case DB returns `DB_KEYEXIST` — which we ignore.

```
        if ((ret = dbp->put(NULL, &key, &data, 0)) != 0)
        {
            dbp->err(ret, "DB->put");
            if (ret != DB_KEYEXIST)
```

```
                goto err;
        }
    }
```

Finally, we close our database before returning from the method.

```
err:    if (dbp != NULL) {
        (void)dbp->close(DB_NOSYNC);
        cout << "database closed" << endl;
        }

    return (ret);
}
```

Method: RepMgr::print_stocks()

The `print_stocks()` method simply takes a database handle, opens a cursor, and uses it to display all the information it finds in a database. This is trivial cursor operation that should hold no surprises for you. We simply provide it here for the sake of completeness.

If you are unfamiliar with basic cursor operations, please see the *Getting Started with Berkeley DB* guide.

```
int RepMgr::print_stocks(Db *dbp)
{
    Dbc *dbc;
    Dbt key, data;
#define MAXKEYSIZE  10
#define MAXDATASIZE 20
    char keybuf[MAXKEYSIZE + 1], databuf[MAXDATASIZE + 1];
    int ret, t_ret;
    u_int32_t keysize, datasize;

    if ((ret = dbp->cursor(NULL, &dbc, 0)) != 0) {
        dbp->err(ret, "can't open cursor");
        return (ret);
    }

    memset(&key, 0, sizeof(key));
    memset(&data, 0, sizeof(data));

    cout << "\tSymbol\tPrice" << endl
        << "\t======\t=====" << endl;

    for (ret = dbc->get(&key, &data, DB_FIRST);
        ret == 0;
        ret = dbc->get(&key, &data, DB_NEXT)) {
        keysize = key.get_size() > MAXKEYSIZE ? MAXKEYSIZE : key.get_size();
        memcpy(keybuf, key.get_data(), keysize);
        keybuf[keysize] = '\0';
```

```
        datasize = data.get_size() >=
            MAXDATASIZE ? MAXDATASIZE : data.get_size();
        memcpy(databuf, data.get_data(), datasize);
        databuf[datasize] = '\0';

        cout << "\t" << keybuf << "\t" << databuf << endl;
    }
    cout << endl << flush;

    if ((t_ret = dbc->close()) != 0 && ret == 0) {
        cout << "closed cursor" << endl;
        ret = t_ret;
    }

    switch (ret) {
    case 0:
    case DB_NOTFOUND:
        return (0);
    default:
        return (ret);
    }
}
```

Chapter 3. The DB Replication Framework

The easiest way to add replication to your transactional application is to use the replication framework. The replication framework provides a comprehensive communications layer that enables replication. For a brief listing of the replication framework's feature set, see Replication Framework Overview (page 5).

To use the replication framework, you make use of special methods off the `DbEnv` class. That is:

1. Create an environment handle as normal.
2. Configure your environment handle as needed (e.g. set the error file and error prefix values, if desired).
3. Use the replication framework replication methods to configure the replication framework. Using these methods causes DB to know that you are using the replication framework.

   Configuring the replication framework entails setting its replication priority, setting the TCP/IP address that this replication environment will use for incoming replication messages, identify TCP/IP addresses of other replication environments, setting the number of replication environments in the replication group, and so forth. These actions are discussed throughout the remainder of this chapter.
4. Open your environment handle. When you do this, be sure to specify `DB_INIT_REP` and `DB_THREAD` to your open flags. (This is in addition to the flags that you normally use for a single-threaded transactional application). The first of these causes replication to be initialized for the application. The second causes your environment handle to be free-threaded (thread safe). Both flags are required for replication framework usage.
5. Start replication by calling `DbEnv::repmgr_start()`.
6. Open your databases as needed. Masters must open their databases for read and write activity. Replicas can open their databases for read-only activity, but doing so means they must re-open the databases if the replica ever becomes a master. Either way, replicas should never attempt to write to the database(s) directly.

The replication framework allows you to only use one environment handle per process.

When you are ready to shut down your application:

1. Close your databases
2. Close your environment. This causes replication to stop as well.

Before you can use the replication framework, you may have to enable it in your DB library. This is *not* a requirement for Microsoft Windows systems, or Unix systems that use pthread mutexes by default. Other systems, notably BSD and BSD-derived systems (such as Mac OS X), must enable the replication framework when you configure the DB build.

You do this by *not* disabling replication and by configuring the library with POSIX threads support. In other words, replication must be turned on in the build (it is by default), and POSIX thread support must be enabled if it is not already by default. To do this, use the `--enable-pthread_api` switch on the configure script.

For example:

```
../dist/configure --enable-pthread-api
```

Starting and Stopping Replication

As described above, you introduce replication to an application by starting with a transactional application, performing some basic replication configuration, and then starting replication using `DbEnv::repmgr_start()`.

You stop replication by closing your environment cleanly, as is normal for an DB application.

For example, the following code fragment initializes, then stops and starts replication. Note that other replication activities are omitted for brevity.

```
#include <db_cxx.h>

/* Use a 10mb cache */
#define CACHESIZE   (10 * 1024 * 1024)

...

    DbEnv *dbenv;            /* Environment handle. */
    const char *progname;     /* Program name. */
    const char *envHome;      /* Environment home directory. */
    const char *listen_host;  /* A TCP/IP hostname. */
    const char *other_host;   /* A TCP/IP hostname. */
    u_int16 listen_port;      /* A TCP/IP port. */
    u_int16 other_port;       /* A TCP/IP port. */

    /* Initialize variables */
    dbenv = NULL;
    progname = "example_replication";
    envHome = "ENVIRONMENT_HOME";
    listen_host = "mymachine.sleepycat.com";
    listen_port = 5001;
    other_host = "anothermachine.sleepycat.com";
    other_port = 4555;

    try {
```

```
        /* Create the environment handle */
        dbenv = new DbEnv(0);

        /*
         * Configure the environment handle. Here we configure
         * asynchronous transactional commits for performance reasons.
         */
        dbenv->set_errfile(stderr);
        dbenv->set_errpfx(progname);
        (void)dbenv->set_cachesize(0, CACHESIZE, 0);
        (void)dbenv->set_flags(DB_TXN_NOSYNC, 1);

        /*
         * Configure the local address. This is the local hostname and
         * port that this replication participant will use to receive
         * incoming replication messages. Note that this can be performed
         * only once for the application. It is required.
         */
       dbenv->repmgr_set_local_site(listen_host, listen_port, 0);

       /*
        * Set this application's priority. This is used for elections.
        *
        * Set this number to a positive integer, or 0 if you do not want
        * this site to be able to become a master.
        */
        dbenv->rep_set_priority(100);

        /*
         * Add a site to the list of replication environments known to
         * this application.
         */
         dbenv->repmgr_add_remote_site(other_host, other_port);

        /*
         * Identify the number of sites in the replication group. This is
         * necessary so that elections and permanent message handling can
         * be performed correctly.
         */
         dbenv->repmgr_add_nsites(2);

        /* Open the environment handle. Note that we add DB_THREAD and
         * DB_INIT_REP to the list of flags. These are required.
         */
         dbenv->open(home, DB_CREATE | DB_RECOVER |
                           DB_INIT_LOCK | DB_INIT_LOG |
                           DB_INIT_MPOOL | DB_INIT_TXN  |
                           DB_THREAD | DB_INIT_REP,
                           0);
```

```
        /* Start the replication framework such that it uses 3 threads. */
         dbenv->repmgr_start(3, DB_REP_ELECTION);

       /* Sleep to give ourselves time to find a master */
        sleep(5);

       /*
        ************************************************************
        *** All other application code goes here, including  *****
        *** database opens                                   *****
        ************************************************************
        */


    } catch (DbException &de) {
        /* Error handling goes here */
    }

    /* Close out the application here.
    try {
        /*
         * Make sure all your database handles are closed
         *  (omitted from this example).
         */

        /* Close the environment */
        if (dbenv != NULL)
            (void)dbenv->close(dbenv, 0);

    } catch (DbException &de) {
        /* Error handling goes here */
    }

    /* All done */
```

Managing Election Policies

Before continuing, it is worth taking a look at the startup election flags accepted by `DbEnv::repgmr_start()`. These flags control how your replication application will behave when it first starts up.

In the previous example, we specified `DB_REP_ELECTION` when we started replication. This causes the application to try to find a master upon startup. If it cannot, it calls for an election. In the event an election is held, the environment receiving the most number of votes will become the master.

There's some important points to make here:

- This flag only requires that other environments in the replication group participate in the vote. There is no requirement that *all* such environments participate. In other words, if an environment starts up, it can call for an election, and select a master, even if all other environment have not yet joined the replication group.
- It only requires a simple majority of participating environments to elect a master. The number of environments used to calculate the simple majority is based on the value set for `DbEnv::rep_set_nsites()`. This is always true of elections held using the replication framework.
- As always, the environment participating in the election with the most up-to-date log files is selected as master. If an environment with better log files has not yet joined the replication group, it may not become the master.

Any one of these points may be enough to cause a less-than-optimum environment to be selected as master. Therefore, to give you a better degree of control over which environment becomes a master at application startup, the replication framework offers the following start-up flags:

Flag	Description
`DB_REP_MASTER`	The application starts up and declares itself to be a master without calling for an election. It is an error for more than one environment to start up using this flag, or for an environment to use this flag when a master already exists. Note that no replication group should *ever* operate with more than one master. In the event that a environment attempts to become a master when a master already exists, the replication code will resolve the problem by holding an election. Note, however, that there is always a possibility of data loss in the face of duplicate masters, because once a master is selected, the environment that loses the election will have to roll back any transactions committed until it is in sync with the "real" master.
`DB_REP_CLIENT`	The application starts up and declares itself to be a replica without calling for an election. Note that the application can still become a master if a subsequent application starts up, calls for an election, and this application is elected master.

Flag	Description
DB_REP_ELECTION	As described above, the application starts up, looks for a master, and if one is not found calls for an election.
DB_REP_FULL_ELECTION	Identical to DB_REP_ELECTION except that the election requires all known members of the replication group to participate. If a given environment has not yet started but it is included in the replication group count (using DbEnv::rep_set_nsites()) then a master can not be elected.

Selecting the Number of Threads

Under the hood, the replication framework is threaded and you can control the number of threads used to process messages received from other replicas. The threads that the replication framework uses are:

- Incoming message thread. This thread receives messages from the site's socket and passes those messages to message processing threads (see below) for handling.
- Outgoing message thread. Outgoing are performed in whatever thread performed a write to the database(s). That is, the thread that called, for example, Db::put() is the thread that writes replication messages about that fact to the socket.

  Note that if this write activity would cause the thread to be blocked due to some condition on the socket, the replication framework will hand the outgoing message to the incoming message thread, and it will then write the message to the socket. This prevents your database write threads from blocking due to abnormal network I/O conditions.
- Message processing threads are responsible for parsing and then responding to incoming replication messages. Typically, a response will include write activity to your database(s), so these threads can be busy performing disk I/O.

Of these threads, the only ones that you have any configuration control over are the message processing threads. In this case, you can determine how many of these threads you want to run.

It is always a bit of an art to decide on a thread count, but the short answer is you probably do not need more than three threads here, and it is likely that one will suffice. That said, the best thing to do is set your thread count to a fairly low number and then increase it if it appears that your application will benefit from the additional threads.

Adding the Replication Framework to SimpleTxn

We now use the methods described above to add partial support to the SimpleTxn example that we presented in Transactional Application (page 11). That is, in this section we will:

- Enhance our command line options to accept information of interest to a replicated application.
- Configure our environment handle to use replication and the replication framework.
- Minimally configure the replication framework.
- Start replication.

Note that when we are done with this section, we will be only partially ready to run the application. Some critical pieces will be missing; specifically, we will not yet be handling the differences between a master and a replica. (We do that in the next chapter).

Also, note that in the following code fragments, additions and changes to the code are marked in **bold**.

To begin, we copy the `SimpleTxn` code to a new file called `RepMgr.cpp`. Having done that, we must make some significant changes to our `RepConfigInfo` class because now we will be using it to maintain a lot more information.

First, we create a new structure, `RepHostInfoObj`, which we use to store host and port information for all "other" servers identified to the application via the `-o` command line option. This structure is chain-able, which makes cleaning up at program shutdown time easier.

```
#include <db_cxx.h>
#include <iostream>

// Chain-able struct used to store host information.
typedef struct RepHostInfoObj{
    char* host;
    int port;
    RepHostInfoObj* next; // used for chaining multiple "other" hosts.
} REP_HOST_INFO;
```

Next, we update our `RepConfigInfo` class definition to manage a lot more information and a new method.

```
class RepConfigInfo {
public:
    RepConfigInfo();
    virtual ~RepConfigInfo();

    void addOtherHost(char* host, int port);
public:
    u_int32_t start_policy;
    char* home;
    bool got_listen_address;
    REP_HOST_INFO this_host;
    int totalsites;
    int priority;
```

```
    // used to store a set of optional other hosts.
    REP_HOST_INFO *other_hosts;
};
```

Then, we update our constructor to initialize our new variables.

```
RepConfigInfo::RepConfigInfo()
{
    start_policy = DB_REP_ELECTION;
    home = "TESTDIR";
    got_listen_address = false;
    totalsites = 0;
    priority = 100;
    other_hosts = NULL;
}
```

Next, we implement our new method, `RepConfigInfo::addOtherHost`, which is used to create `RepHostInfoObj` instances and add them to the chain of "other" hosts.

```
RepConfigInfo::addOtherHost(char* host, int port)
{
    REP_HOST_INFO *newinfo;
    newinfo = (REP_HOST_INFO*)malloc(sizeof(REP_HOST_INFO));
    newinfo->host = host;
    newinfo->port = port;
    if (other_hosts == NULL) {
        other_hosts = newinfo;
        newinfo->next = NULL;
    } else {
        newinfo->next = other_hosts;
        other_hosts = newinfo;
    }
}
```

Having done that, we update our class destructor to release the `RepHostInfoObj` chain of objects at class destruction time.

```
RepConfigInfo::~RepConfigInfo()
{
    // release any other_hosts structs.
    if (other_hosts != NULL) {
        REP_HOST_INFO *CurItem = other_hosts;
        while (CurItem->next != NULL)
        {
            REP_HOST_INFO *TmpItem = CurItem;
            free(CurItem);
            CurItem = TmpItem;
        }
        free(CurItem);
    }
```

```
    other_hosts = NULL;
}
```

Having completed our update to the `RepConfigInfo` class, we can now start making changes to the main portion of our program. We begin by changing the program's name.

```
using std::cout;
using std::cin;
using std::cerr;
using std::endl;
using std::flush;

#define CACHESIZE   (10 * 1024 * 1024)
#define DATABASE    "quote.db"

const char *progname = "RepMgr";
```

Next we update our usage function. The application will continue to accept the `-h` parameter so that we can identify the environment home directory used by this application. However, we also add the

- `-m` parameter which allows us to identify the host and port used by this application to listen for replication messages.
- `-o` parameter which allows us to specify other replicas.
- `-n` parameter which allows us to identify the number of sites in this replication group.
- `-p` option, which is used to identify this replica's priority (recall that the priority is used as a tie breaker for elections)

```
class RepMgr
{
public:
    // Constructor.
    RepMgr();
    // Initialization method. Creates and opens our environment handle.
    int init(RepConfigInfo* config);
    // The doloop is where all the work is performed.
    int doloop();
    // terminate() provides our shutdown code.
    int terminate();

private:
    // disable copy constructor.
    RepMgr(const RepMgr &);
    void operator = (const RepMgr &);

    // internal data members.
    RepConfigInfo   *app_config;
```

```
    DbEnv           dbenv;

    // private methods.
    // print_stocks() is used to display the contents of our database.
    static int print_stocks(Db *dbp);
};

static void usage()
{
    cerr << "usage: " << progname << endl
         << "[-h home][-o host:port][-m host:port]"
         << "[-n nsites][-p priority]" << endl;

    cerr << "\t -m host:port (required; m stands for me)" << endl
         << "\t -o host:port (optional; o stands for other; any "
         << "number of these may be specified)" << endl
         << "\t -h home directory" << endl
         << "\t -n nsites (optional; number of sites in replication "
         << "group; defaults to 0" << endl
         << "\t  in which case we try to dynamically compute the "
         << "number of sites in" << endl
         << "\t  the replication group)" << endl
         << "\t -p priority (optional: defaults to 100)" << endl;

    exit(EXIT_FAILURE);
}
```

Now we can begin working on our `main()` function. We begin by adding a couple of variables that we will use to collect TCP/IP host and port information.

```
int main(int argc, char **argv)
{
    RepConfigInfo config;
    char ch, *portstr, *tmphost;
    int tmpport;
    int ret;
```

Now we collect our command line arguments. As we do so, we will configure host and port information as required, and we will configure the application's election priority if necessary.

```
    // Extract the command line parameters
    while ((ch = getopt(argc, argv, "h:m:n:o:p:")) != EOF) {
        switch (ch) {
        case 'h':
            config.home = optarg;
            break;
        case 'm':
            config.this_host.host = strtok(optarg, ":");
            if ((portstr = strtok(NULL, ":")) == NULL) {
```

```
                cerr << "Bad host specification." << endl;
                usage();
            }
            config.this_host.port = (unsigned short)atoi(portstr);
            config.got_listen_address = true;
            break;
        case 'n':
            config.totalsites = atoi(optarg);
            break;
        case 'o':
            tmphost = strtok(optarg, ":");
            if ((portstr = strtok(NULL, ":")) == NULL) {
                cerr << "Bad host specification." << endl;
                usage();
            }
            tmpport = (unsigned short)atoi(portstr);
            config.addOtherHost(tmphost, tmpport);
            break;
        case 'p':
            config.priority = atoi(optarg);
            break;
        case '?':
        default:
            usage();
        }
    }

    // Error check command line.
    if ((!config.got_listen_address) || config.home == NULL)
        usage();
```

Having done that, the remainder of our `main()` function is left unchanged:

```
    RepMgr runner;
    try {
        if((ret = runner.init(&config)) != 0)
            goto err;
        if((ret = runner.doloop()) != 0)
            goto err;
    } catch (DbException dbe) {
        cerr << "Caught an exception during initialization or"
            << " processing: " << dbe.what() << endl;
    }
err:
    runner.terminate();
    return 0;
}
```

Now we need to update our `RepMgr::init()` method. Our updates are at first related to configuring replication. First, we need to update the method so that we can identify the local site to the environment handle (that is, the site identified by the `-m` command line option):

```
RepMgr::RepMgr() : app_config(0), dbenv(0)
{
}

int RepMgr::init(RepConfigInfo *config)
{
    int ret = 0;

    app_config = config;

    dbenv.set_errfile(stderr);
    dbenv.set_errpfx(progname);

    if ((ret = dbenv.repmgr_set_local_site(app_config->this_host.host,
        app_config->this_host.port, 0)) != 0) {
        cerr << "Could not set listen address to host:port "
             << app_config->this_host.host << ":"
             << app_config->this_host.port
             << "error: " << ret << endl;
    }
```

And we also add code to allow us to identify "other" sites to the environment handle (that is, the sites that we identify using the `-o` command line option). To do this, we iterate over each of the "other" sites provided to us using the `-o` command line option, and we add each one individually in turn:

```
    for ( REP_HOST_INFO *cur = app_config->other_hosts; cur != NULL;
        cur = cur->next) {
        if ((ret = dbenv.repmgr_add_remote_site(cur->host, cur->port,
                                                0)) != 0) {
                cerr << "could not add site." << endl
        }
    }
```

And then we need code to allow us to identify the total number of sites in this replication group, and to set the environment's priority.

```
    if (app_config->totalsites > 0) {
        try {
            if ((ret = dbenv.rep_set_nsites(app_config->totalsites)) != 0)
                dbenv.err(ret, "set_nsites");
        } catch (DbException dbe) {
            cerr << "rep_set_nsites call failed. Continuing." << endl;
        }
```

```
    }
    dbenv.rep_set_priority(app_config->priority);
```

We can now open our environment. Note that the flags we use to open the environment are slightly different for a replicated application than they are for a non-replicated application. Namely, replication requires the `DB_INIT_REP` flag.

Also, because we are using the replication framework, we must prepare our environment for threaded usage. For this reason, we also need the `DB_THREAD` flag.

```
    dbenv.set_cachesize(0, CACHESIZE, 0);
    dbenv.set_flags(DB_TXN_NOSYNC, 1);

    try {
        dbenv.open(app_config->home,
            DB_CREATE |
            DB_RECOVER |
            DB_THREAD |
            DB_INIT_REP |
            DB_INIT_LOCK |
            DB_INIT_LOG |
            DB_INIT_MPOOL |
            DB_INIT_TXN,
            0);
    } catch(DbException dbe) {
        cerr  << "Caught an exception during DB environment open." << endl
              << "Ensure that the home directory is created prior to starting"
              << " the application." << endl;
        ret = ENOENT;
        goto err;
    }
```

Finally, we start replication before we exit this method. Immediately after exiting this method, our application will go into the `RepMgr::doloop()` method, which is where the bulk of our application's work is performed. We update that method in the next chapter.

```
    if ((ret = dbenv.repmgr_start(3, app_config->start_policy)) != 0)
        goto err;

err:
    return ret;
}
```

This completes our replication updates for the moment. We are not as yet ready to actually run this program; there remains a few critical pieces left to add to it. However, the work that we performed in this section represents a solid foundation for the remainder of our replication work.

Permanent Message Handling

As described in Permanent Message Handling (page 7), messages are marked permanent if they contain database modifications that should be committed at the replica. DB's replication code decides if it must flush its transaction logs to disk depending on whether it receives sufficient permanent message acknowledgments from the participating replica. More importantly, the thread performing the transaction commit blocks until it either receives enough acknowledgments, or the acknowledgment timeout expires.

The replication framework is fully capable of managing permanent messages for you if your application requires it (most do). Almost all of the details of this are handled by the replication framework for you. However, you do have to set some policies that tell the replication framework how to handle permanent messages.

There are two things that you have to do:

- Determine how many acknowledgments must be received by the master.
- Identify the amount of time that replicas have to send their acknowledgments.

Identifying Permanent Message Policies

You identify permanent message policies using the Note that you can set permanent message policies at any time during the life of the application.

The following permanent message policies are available when you use the replication framework:

- `DB_REPMGR_ACKS_NONE`

  No permanent message acknowledgments are required. If this policy is selected, permanent message handling is essentially "turned off." That is, the master will never wait for replica acknowledgments. In this case, transaction log data is either flushed or not strictly depending on the type of commit that is being performed (synchronous or asynchronous).

- `DB_REPMGR_ACKS_ONE`

  At least one replica must acknowledge the permanent message within the timeout period.

- `DB_REPMGR_ACKS_ONE_PEER`

  At least one electable peer must acknowledge the permanent message within the timeout period. Note that an *electable peer* is simply another environment that can be elected to be a master (that is, it has a priority greater than 0). Do not confuse this with the concept of a peer as used for client to client transfers. See Client to Client Transfer (page 53) for more information on client to client transfers.

- `DB_REPMGR_ACKS_ALL`

All environments must acknowledge the message within the timeout period. This policy should be selected only if your replication group has a small number of replicas, and those replicas are on extremely reliable networks and servers.

When this flag is used, the actual number of environments that must respond is determined by the value set for `DbEnv::rep_set_nsites()`.

- `DB_REPMGR_ACKS_ALL_PEERS`

  All electable peers must acknowledge the message within the timeout period. This policy should be selected only if your replication group is small, and its various environments are on extremely reliable networks and servers.

  Note that an *electable peer* is simply another environment that can be elected to be a master (that is, it has a priority greater than 0). Do not confuse this with the concept of a peer as used for client to client transfers. See Client to Client Transfer (page 53) for more information on client to client transfers.

- `DB_REPMGR_ACKS_QUORUM`

  A quorum of electable peers must acknowledge the message within the timeout period. A quorum is reached when acknowledgments are received from the minimum number of environments needed to ensure that the record remains durable if an election is held. That is, the master wants to hear from enough electable replicas that they have committed the record so that if an election is held, the master knows the record will exist even if a new master is selected.

  Note that an *electable peer* is simply another environment that can be elected to be a master (that is, it has a priority greater than 0). Do not confuse this with the concept of a peer as used for client to client transfers. See Client to Client Transfer (page 53) for more information on client to client transfers.

By default, a quorum of sites must must acknowledge a permanent message in order for it considered to have been successfully transmitted. The actual number of environments that must respond is calculated using the value set with `DbEnv::rep_set_nsites()`.

Setting the Permanent Message Timeout

The permanent message timeout represents the maximum amount of time the committing thread will block waiting for message acknowledgments. If sufficient acknowledgments arrive before this timeout has expired, the thread continues operations as normal. However, if this timeout expires, the committing thread flushes its transaction log buffer before continuing with normal operations.

You set the timeout value using the `DbEnv::rep_set_timeout()` method. When you do this, you provide the `DB_REP_ACK_TIMEOUT` flag to the `which` parameter, and the timeout value in microseconds to the `timeout` parameter.

For example:

```
    dbenv->rep_set_timeout(DB_REP_ACK_TIMEOUT, 100);
```

This timeout value can be set at anytime during the life of the application.

Adding a Permanent Message Policy to RepMgr

For illustration purposes, we will now update `RepMgr` such that it requires only one acknowledgment from a replica on transactional commits. Also, we will give this acknowledgment a 500 microsecond timeout value. This means that our application's main thread will block for up to 500 microseconds waiting for an acknowledgment. If it does not receive at least one acknowledgment in that amount of time, DB will flush the transaction logs to disk before continuing on.

This is a very simple update. We can perform the entire thing in `RepMgr::init()` immediately after we set the application's priority and before we open our environment handle.

```
int RepMgr::init(RepConfigInfo *config)
{
    int ret = 0;

    app_config = config;

    dbenv.set_errfile(stderr);
    dbenv.set_errpfx(progname);

    if ((ret = dbenv.repmgr_set_local_site(app_config->this_host.host,
        app_config->this_host.port, 0)) != 0) {
        cerr << "Could not set listen address to host:port "
             << app_config->this_host.host << ":"
             << app_config->this_host.port
             << "error: " << ret << endl;
    }

    for ( REP_HOST_INFO *cur = app_config->other_hosts; cur != NULL;
        cur = cur->next) {
        if ((ret = dbenv.repmgr_add_remote_site(cur->host, cur->port,
                                                0)) != 0) {
                cerr << "could not add site." << endl
        }
    }

    if (app_config->totalsites > 0) {
        try {
            if ((ret = dbenv.rep_set_nsites(app_config->totalsites)) != 0)
                dbenv.err(ret, "set_nsites");
        } catch (DbException dbe) {
            cerr << "rep_set_nsites call failed. Continuing." << endl;
        }
    }
```

```
    dbenv.rep_set_priority(app_config->priority);

    /* Permanent messages require at least one ack */
    dbenv.repmgr_set_ack_policy(DB_REPMGR_ACKS_ONE);
    /* Give 500 microseconds to receive the ack */
    dbenv.rep_set_timeout(DB_REP_ACK_TIMEOUT, 500);

    dbenv.set_cachesize(0, CACHESIZE, 0);
    dbenv.set_flags(DB_TXN_NOSYNC, 1);

    ...
```

Managing Election Times

Where it comes to elections, there are two timeout values with which you should be concerned: election timeouts and election retries.

Managing Election Timeouts

When a environment calls for an election, it will wait some amount of time for the other replicas in the replication group to respond. The amount of time that the environment will wait before declaring the election completed is the *election timeout*.

If the environment hears from all other known replicas before the election timeout occurs, the election is considered a success and a master is elected.

If only a subset of replicas respond, then the success or failure of the election is determined by (1) how many replicas have responded and (2) the election policy that is in place at the time. For example, usually it only takes a simple majority of replicas to elect a master. If there are enough votes for a given environment to meet that standard, then the master has been elected and the election is considered a success.

However, upon application startup you can require that all known replicas must participate in the election. Or, it is possible that not enough votes are cast to select a master even with a simple majority. If either of these conditions occur when the election timeout value is reached, the election is considered a failure and a master is not elected. At this point, your replication group is operating without a master, which means that, essentially, your replicated application has been placed in read-only mode.

Note, however, that the replication framework will attempt a new election after a given amount of time has passed. See the next section for details.

You set the election timeout value using `DbEnv::rep_set_timeout()`. To do so, specify the `DB_REP_ELECTION_TIMEOUT` flag to the `which` parameter and then a timeout value in microseconds to the `timeout` parameter.

Managing Election Retry Times

In the event that a election fails (see the previous section), an election will not be attempted again until the election retry timeout value has expired.

You set the retry timeout value using `DbEnv::rep_set_timeout()`. To do so, specify the `DB_REP_ELECTION_RETRY` flag to the `which` parameter and then a retry value in microseconds to the `timeout` parameter.

Managing Connection Retries

In the event that a communication failure occurs between two environments in a replication group, the replication framework will wait a set amount of time before attempting to re-establish the connection. You can configure this wait value using `DbEnv::rep_set_timeout()`. To do so, specify the `DB_REP_CONNECTION_RETRY` flag to the `which` parameter and then a retry value in microseconds to the `timeout` parameter.

Chapter 4. Replica versus Master Processes

Every environment participating in a replicated application must know whether it is a *master* or *replica*. The reason for this is because, simply, the master can modify the database while replicas cannot. As a result, not only will you open databases differently depended on whether the environment is running as a master, but the environment will frequently behave quit a bit differently depending on whether it thinks it is operating as the read/write interface for your database.

Moreover, an environment must also be capable of gracefully switching between master and replica states. This means that the environment must be able to detect when it has switched states.

Not surprisingly, a large part of your application's code will be tied up in knowing which state a given environment is in and then in the logic of how to behave depending on its state.

This chapter shows you how to determine your environment's state, and it then shows you some sample code on how an application might behave depending on whether it is a master or a replica in a replicated application.

Determining State

In order to determine whether your code is running as a master or a replica, you implement a callback whose function it is to respond to events that happen within the DB library.

Note that this callback is usable for events beyond those required for replication purposes. In this section, however, we only discuss the replication-specific events.

The callback is required to determine which event has been passed to it, and then take action depending on the event. For replication, the events that we care about are:

- `DB_EVENT_REP_MASTER`

  The local environment is now a master.

- `DB_EVENT_REP_CLIENT`

  The local environment is now a replica.

- `DB_EVENT_REP_STARTUPDONE`

  The replica has completed startup synchronization and is now processing log records received from the master.

- `DB_EVENT_REP_NEWMASTER`

  An election was held and a new environment was made a master. However, the current environment *is not* the master. This event exists so that you can cause your code to take some unique action in the event that the replication groups switches masters.

Note that these events are raised whenever the state is established. That is, when the current environment becomes a client, and that includes at application startup, the event is raised. Also, when an election is held and a client is elected to be a master, then the event occurs.

The implementation of this callback is fairly simple. First you pass a structure to the environment handle that you can use to record the environment's state, and then you implement a switch statement within the callback that you use to record the current state, depending on the arriving event.

For example:

```
#include <db_cxx.h>
/* Forward declaration */
void *event_callback(DbEnv *, u_int32_t, void *);

...

/* The structure we use to track our environment's state */
typedef struct {
    int is_master;
} APP_DATA;

...

/*
 * Inside our main() function, we declare an APP_DATA variable.
 */
APP_DATA my_app_data;
my_app_data.is_master = 0; /* Assume we start as a replica */

...

/*
 * Now we open our environment handle and set the APP_DATA structure
 * to it's app_private member.
 */
DbEnv *dbenv = new DbEnv(0);
dbenv->set_app_private(&my_app_data);

/* Having done that, register the callback with the
 * Berkeley DB library
 */
dbenv->set_event_notify(event_callback);
```

That done, we still need to implement the callback itself. This implementation can be fairly trivial.

```
/*
 * A callback used to determine whether the local environment is a replica
```

```
 * or a master. This is called by the replication framework
 * when the local environment changes state.
 */
void *
event_callback(DbEnv *dbenv, u_int32_t which, void *info)
{
    APP_DATA *app = dbenv->get_app_private();

    info = NULL;                /* Currently unused. */

    switch (which) {
    case DB_EVENT_REP_MASTER:
        app->is_master = 1;
        break;

    case DB_EVENT_REP_CLIENT:
        app->is_master = 0;
        break;

    case DB_EVENT_REP_STARTUPDONE: /* fallthrough */
    case DB_EVENT_REP_NEWMASTER:
        /* Ignore. */
        break;

    default:
        dbenv->errx(dbenv, "ignoring event %d", which);
    }
}
```

Notice how we access the `APP_DATA` information using the environment handle's `app_private` data member. We also ignore the `DB_EVENT_REP_NEWMASTER` and `DB_EVENT_REP_STARTUPDONE` cases since these are not relevant for simple replicated applications.

Of course, this only gives us the current state of the environment. We still need the code that determines what to do when the environment changes state and how to behave depending on the state (described in the next section).

Processing Loop

Typically the central part of any replication application is some sort of a continuous loop that constantly checks the state of the environment (whether it is a replica or a master), opens and/or closes the databases as is necessary, and performs other useful work. A loop such as this one must of necessity take special care to know whether it is operating on a master or a replica environment because all of its activities are dependent upon that state.

The flow of activities through the loop will generally be as follows:

1. Check whether the environment has changed state. If it has, you might want to reopen your database handles, especially if you opened your replica's database handles as read-only. In this case, you might need to reopen them as read-write. However, if you always open your database handles as read-write, then it is not automatically necessary to reopen the databases due to a state change. Instead, you could check for a `DB_REP_HANDLE_DEAD` return code when you use your database handle(s). If you see this, then you need to reopen your database handle(s).

2. If the databases are closed, create new database handles, configure the handle as is appropriate, and then open the databases. Note that handle configuration will be different, depending on whether the handle is opened as a replica or a master. At a minimum, the master should be opened with database creation privileges, whereas the replica does not need to be. You must also open the master such that its databases are read-write. You *can* open replicas with read-only databases, so long as you are prepared to closed and the reopen the handle in the event the client becomes a master.

   Also, note that if the local environment is a replica, then it is possible that databases do not currently exist. In this case, the database open attempts will fail. Your code will have to take this corner case into account (described below).

3. Once the databases are opened, check to see if the local environment is a master. If it is, do whatever it is a master should do for your application.

   Remember that the code for your master should include some way for you to tell the master to exit gracefully.

4. If the local environment is not a master, then do whatever it is your replica environments should do. Again, like the code for your master environments, you should provide a way for your replicas to exit the processing loop gracefully.

The following code fragment illustrates these points (note that we fill out this fragment with a working example next in this chapter):

```
/* loop to manage replication activities */

Db *dbp;
int ret;
APP_DATA *app_data;
u_int32_t flags;

dbp = NULL;
ret = 0;

/*
 * Remember that for this to work, an APP_DATA struct would have first
 * had to been set to the environment handle's app_private data
 * member. (dbenv is presumable declared and opened in another part of
 * the code.)
 */
```

```
app_data = dbenv->get_app_private();


/*
 * Infinite loop. We exit depending on how the master and replica code
 * is written.
 */
for (;;) {
    /* If dbp is not opened, we need to open it. */
    if (dbp == NULL) {
        /*
         * Create the handle and then configure it. Before you open
         * it, you have to decide what open flags to use:
         */
         flags = DB_AUTO_COMMIT;
         if (app_data->is_master)
            flags |= DB_CREATE
        /*
         * Now you can open your database handle, passing to it the
         * flags selected above.
         *
         * One thing to watch out for is a case where the databases
         * you are trying to open do not yet exist. This can happen
         * for replicas where the databases are being opened
         * read-only. If this happens, ENOENT is returned by the
         * open() call.
         */
         try {
            dbp->open(NULL, DATABASE, NULL, DB_BTREE,
                    app_data->is_master ? DB_CREATE | DB_AUTO_COMMIT :
                    DB_AUTO_COMMIT, 0);
         } catch(DbException dbe) {
            if (dbe.get_errno() == ENOENT) {
                cout << "No stock db available yet - retrying." << endl;
                try {
                    dbp->close(0);
                } catch (DbException dbe2) {
                    cout << "Unexpected error closing after failed" <<
                            " open, message: " << dbe2.what() << endl;
                    dbp = NULL;
                    goto err;
                 }
                 dbp = NULL;
                 sleep(SLEEPTIME);
                 continue;
            } else {
                dbenv.err(ret, "DB->open");
                throw dbe;
            }
```

```
        }
    }

    /*
     * Now that the databases have been opened, continue with general
     * processing, depending on whether we are a master or a replica.
     */
     if (app_data->is_master) {
        /*
         * Do master stuff here. Don't forget to include a way to
         * gracefully exit the loop. */
         */
     } else {
        /*
         * Do replica stuff here. As is the case with the master
         * code, be sure to include a way to gracefully exit the
         * loop.
         */
     }
}
```

Example Processing Loop

In this section we take the example processing loop that we presented in the previous section and we flesh it out to provide a more complete example. We do this by updating the `doloop()` function that our original transaction application used (see Method: RepMgr::doloop() (page 16)) to fully support our replicated application.

In the following example code, code that we add to the original example is presented in **bold**.

To begin, we include a new header file into our application so that we can check for the `ENOENT` return value later in our processing loop. We also define our `APP_DATA` structure, and we define a `sleeptime` value. Finally, we update `RepMgr` to have a new method for our event notification callback, and to add a new data member for our `APP_DATA` data member.

```
#include <db_cxx.h>
#include <iostream>
#include <errno.h>

...
// Skipping all the RepHostInfoObj and RepConfigInfo code, which does not
// change.
...

using std::cout;
using std::cin;
using std::cerr;
```

```
using std::endl;
using std::flush;

#define CACHESIZE   (10 * 1024 * 1024)
#define DATABASE    "quote.db"
#define SLEEPTIME   3

const char *progname = "RepMgr";

// Struct used to store information in Db app_private field.
typedef struct {
    int is_master;
} APP_DATA;

class RepMgr
{
public:
    // Constructor.
    RepMgr();
    // Initialization method. Creates and opens our environment handle.
    int init(RepConfigInfo* config);
    // The doloop is where all the work is performed.
    int doloop();
    // terminate() provides our shutdown code.
    int terminate();

    // event notification callback
    static void
    event_callback(DbEnv * dbenv, u_int32_t which, void *info);

private:
    // disable copy constructor.
    RepMgr(const RepMgr &);
    void operator = (const RepMgr &);

    // internal data members.
    APP_DATA        app_data;
    RepConfigInfo   *app_config;
    DbEnv           dbenv;

    // private methods.
    // print_stocks() is used to display the contents of our database.
    static int print_stocks(Db *dbp);
};
```

That done, we can skip the `main()` method, because it does not change. Instead, we skip down to our `RepMgr` constructor where we initialize our `APP_DATA is_master` data member:

```
RepMgr::RepMgr() : app_config(0), dbenv(0)
{
    app_data.is_master = 0; // assume I start out as client
}
```

That done, we must also update `RepMgr::init()` to do a couple of things. First, we need to register our event callback with the environment handle. We also need to make our `APP_DATA` data member available through our environment handle's `app_private` field. This is a fairly trivial update, and it happens at the top of the method (we skip the rest of the method's listing since it does not change):

```
int RepMgr::init(RepConfigInfo *config)
{
    int ret = 0;

    app_config = config;

    dbenv.set_errfile(stderr);
    dbenv.set_errpfx(progname);
    dbenv.set_app_private(&app_data);
    dbenv.set_event_notify(event_callback);

    ...
```

That done, we need to implement our `event_callback()` callback. Note that what we use here is no different from the callback that we described in the previous section. However, for the sake of completeness we provide the implementation here again.

```
/*
 * A callback used to determine whether the local environment is a replica
 * or a master. This is called by the replication framework
 * when the local replication environment changes state.
 */
void RepMgr::event_callback(DbEnv *dbenv, u_int32_t which, void *info)
{
    APP_DATA *app = dbenv->get_app_private();

    info = NULL;                /* Currently unused. */

    switch (which) {
    case DB_EVENT_REP_MASTER:
        app->is_master = 1;
        break;

    case DB_EVENT_REP_CLIENT:
        app->is_master = 0;
        break;

    case DB_EVENT_REP_STARTUPDONE: /* fallthrough */
```

```
    case DB_EVENT_REP_NEWMASTER:
        /* Ignore. */
        break;

    default:
        dbenv->errx(dbenv, "ignoring event %d", which);
    }
}
```

That done, we need to update our `doloop()` method.

We begin by updating our database handle open flags to determine which flags to use, depending on whether the application is running as a master.

```
#define BUFSIZE 1024
int RepMgr::doloop()
{
    Db *dbp;
    Dbt key, data;
    char buf[BUFSIZE], *rbuf;
    int ret;

    dbp = NULL;
    memset(&key, 0, sizeof(key));
    memset(&data, 0, sizeof(data));
    ret = 0;

    for (;;) {
        if (dbp == NULL) {
            dbp = new Db(&dbenv, 0);

            // Set page size small so page allocation is cheap.
            if ((ret = dbp->set_pagesize(512)) != 0)
                goto err;

            try {
                dbp->open(NULL, DATABASE, NULL, DB_BTREE,
                    app_data.is_master ? DB_CREATE | DB_AUTO_COMMIT :
                    DB_AUTO_COMMIT, 0);
```

When we open the database, we modify our error handling to account for the case where the database does not yet exist. This can happen if our code is running as a replica and the replication framework has not yet had a chance to create the databases for us. Recall that replicas never write to their own databases directly, and so they cannot create databases on their own.

If we detect that the database does not yet exist, we simply close the database handle, sleep for a short period of time and then continue processing. This gives the replication framework a chance to create the database so that our replica can continue operations.

```
            } catch(DbException dbe) {
                /* It is expected that this condition will be triggered
                 * when client sites start up.
                 * It can take a while for the master site to be found
                 * and synced, and no DB will be available until then.
                 */
                if (dbe.get_errno() == ENOENT) {
                    cout << "No stock db available yet - retrying." << endl;
                    try {
                        dbp->close(0);
                    } catch (DbException dbe2) {
                        cout << "Unexpected error closing after failed"
                             << " open, message: " << dbe2.what() << endl;
                        dbp = NULL;
                        goto err;
                    }
                    dbp = NULL;
                    sleep(SLEEPTIME);
                    continue;
                } else {
                    dbenv.err(ret, "DB->open");
                    throw dbe;
                }
            }
        }
```

Next we modify our prompt, so that if the local process is running as a replica, we can tell from the shell that the prompt is for a read-only process.

```
        cout << "QUOTESERVER" ;
        if (!app_data.is_master)
            cout << "(read-only)";
        cout << "> " << flush;
```

When we collect data from the prompt, there is a case that says if no data is entered then show the entire stocks database. This display is performed by our `print_stocks()` method (which has not required a modification since we first introduced it in Method: RepMgr::print_stocks() (page 19)).

When we call `print_stocks()`, we check for a dead replication handle. Dead replication handles happen whenever a replication election results in a previously committed transaction becoming invalid. This is an error scenario caused by a new master having a slightly older version of the data than the original master and so all replicas must modify their database(s) to reflect that of the new master. In this situation, some number of previously committed transactions may have to be unrolled. From the replica's perspective, the database handles should all be closed and then opened again.

```
        if (fgets(buf, sizeof(buf), stdin) == NULL)
            break;
```

```
        if (strtok(&buf[0], " \t\n") == NULL) {
            switch ((ret = print_stocks(dbp))) {
            case 0:
                continue;
            case DB_REP_HANDLE_DEAD:
                (void)dbp->close(DB_NOSYNC);
                cout << "closing db handle due to rep handle dead" << endl;
                dbp = NULL;
                continue;
            default:
                dbp->err(ret, "Error traversing data");
                goto err;
            }
        }
        rbuf = strtok(NULL, " \t\n");
        if (rbuf == NULL || rbuf[0] == '\0') {
            if (strncmp(buf, "exit", 4) == 0 ||
                strncmp(buf, "quit", 4) == 0)
                break;
            dbenv.errx("Format: TICKER VALUE");
            continue;
        }
```

That done, we need to add a little error checking to our command prompt to make sure the user is not attempting to modify the database at a replica. Remember, replicas must never modify their local databases on their own. This guards against that happening due to user input at the prompt.

```
        if (!app_data.is_master) {
            dbenv->errx(dbenv, "Can't update at client");
            continue;
        }

        key.set_data(buf);
        key.set_size((u_int32_t)strlen(buf));

        data.set_data(rbuf);
        data.set_size((u_int32_t)strlen(rbuf));

        if ((ret = dbp->put(NULL, &key, &data, 0)) != 0)
        {
            dbp->err(ret, "DB->put");
            if (ret != DB_KEYEXIST)
                goto err;
        }
    }

err:    if (dbp != NULL)
        (void)dbp->close(dbp, DB_NOSYNC);
```

```
    return (ret);
}
```

With that completed, we are all done updating our application for replication. The only remaining method, `print_stocks()`, is unmodified from when we originally introduced it. For details on that function, see Method: RepMgr::print_stocks() (page 19).

Running It

To run our replicated application, we need to make sure each participating environment has its own unique home directory. We can do this by running each site on a separate networked machine, but that is not strictly necessary; multiple instances of this code can run on the same machine provided the environment home restriction is observed.

To run a process, make sure the environment home exists and then start the process using the `-h` option to specify that directory. You must also use the `-m` option to identify the local host and port that this process will use to listen for replication messages, and the `-o` option to identify the other processes in the replication group. Finally, use the `-p` option to specify a priority. The process that you designate to have the highest priority will become the master.

```
> mkdir env1
> ./RepMgr -h env1 -n 2 -m localhost:8080 -o localhost:8081 -p 10
No stock database yet available.
No stock database yet available.
```

Now, start another process. This time, change the environment home to something else, use the `-m` to at least change the port number the process is listening on, and use the `-o` option to identify the host and port of the other replication process:

```
> mkdir env2
> ./RepMgr -h env2 -n 2 -m localhost:8081 -o localhost:8080 -p 20
```

After a short pause, the second process should display the master prompt:

```
QUOTESERVER >
```

And the first process should display the read-only prompt:

```
QUOTESERVER (read-only)>
```

Now go to the master process and give it a couple of stocks and stock prices:

```
QUOTESERVER> FAKECO 9.87
QUOTESERVER> NOINC .23
QUOTESERVER>
```

Then, go to the replica and hit **`return`** at the prompt to see the new values:

```
QUOTESERVER (read-only)>
        Symbol  Price
        ======  =====
        FAKECO  9.87
        NOINC    .23
QUOTESERVER (read-only)>
```

Doing the same at the master results in the same thing:

```
QUOTESERVER>
        Symbol  Price
        ======  =====
        FAKECO  9.87
        NOINC    .23
QUOTESERVER>
```

You can change a stock by simply entering the stock value and new price at the master's prompt:

```
QUOTESERVER> FAKECO 10.01
QUOTESERVER>
```

Then, go to either the master or the replica to see the updated database:

```
QUOTESERVER>
        Symbol  Price
        ======  =====
        FAKECO  10.01
        NOINC    .23
QUOTESERVER>
```

And on the replica:

```
QUOTESERVER (read-only)>
        Symbol  Price
        ======  =====
        FAKECO  10.01
        NOINC    .23
QUOTESERVER (read-only)>
```

Finally, to quit the applications, simply type `quit` at both prompts:

```
QUOTESERVER (read-only)> quit
>
```

And on the master as well:

```
QUOTESERVER> quit
>
```

Chapter 5. Additional Features

Beyond the basic functionality that we have discussed so far in this book, there are several replication features that you should understand. These are all optional to use, but provide useful functionality under the right circumstances.

These additional features are:

1. Delayed Synchronization (page 52)
2. Managing Blocking Operations (page 52)
3. Stop Auto-Initialization (page 53)
4. Client to Client Transfer (page 53)
5. Bulk Transfers (page 54)

Delayed Synchronization

When a replication group has a new master, all replicas must synchronize with that master. This means they must ensure that the contents of their local database(s) are identical to that contained by the new master.

This synchronization process can result in quite a lot of network activity. It can also put a large strain on the master server, especially if is part of a large replication group or if there is somehow a large difference between the master's database(s) and the contents of its replicas.

It is therefore possible to delay synchronization for any replica that discovers it has a new master. You would do this so as to give the master time to synchronize other replicas before proceeding with the delayed replicas.

To delay synchronization of a replica environment, you specify `DB_REP_CONF_DELAYCLIENT` to `DbEnv::rep_set_config()` and then specify `1` to the `onoff` parameter. (Specify `0` to turn the feature off.)

If you use delayed synchronization, then you must manually synchronize the replica at some future time. Until you do this, the replica is out of sync with the master, and it will ignore all database changes forwarded to it from the master.

You synchronize a delayed replica by calling `DbEnv::rep_sync()` on the replica that has been delayed.

Managing Blocking Operations

When a replica is in the process of synchronizing with its master, all DB operations are blocked until such a time as the synchronization is completed. For replicas with a heavy read load, these blocked operations may represent an unacceptable loss in throughput.

You can configure DB so that it will not block when synchronization is in process. Instead, the DB operation will fail, immediately returning a `DB_REP_LOCKOUT` error. When this happens, it is up to your application to your application to determine what action to take (that is, logging the event, making an appropriate user response, retrying the operation, and so forth).

To turn off blocking on synchronization, specify `DB_REP_CONF_NOWAIT` to `DbEnv::rep_set_config()` and then specify `1` to the `onoff` parameter. (Specify `0` to turn the feature off.)

Stop Auto-Initialization

As stated in the previous section, when a replication replica is synchronizing with its master, it will block all DB operations until the synchronization is completed. You can turn off this behavior (see Managing Blocking Operations (page 52)), but for replicas that have been out of touch from their master for a very long time, this may not be enough.

If a replica has been out of touch from its master long enough, it may find that it is not possible to perform synchronization. When this happens, by default the master and replica internally decided to completely re-initialize the replica. This re-initialization involves discarding the replica's current database(s) and transferring new ones to it from the master. Depending on the size of the master's databases, this can take a long time, during which time the replica will be complete non-responsive when it comes to performing database operations.

It is possible that there is a time of the day when it is better to perform a replica re-initialization. Or, you simply might want to decide to bring the replica up to speed by restoring it's databases using a hot-backup taken from the master. Either way, you can decide to prevent automatic-initialization of your replica. To do this specify `DB_REP_CONF_NOAUTOINIT` to `DbEnv::rep_set_config()` and then specify `1` to the `onoff` parameter. (Specify `0` to turn the feature off.)

Client to Client Transfer

It is possible to use a replica instead of a master to synchronize another replica. This serves to take the request load off a master that might otherwise occur if multiple replicas attempted to synchronize with the master at the same time.

For best results, use this feature combined with the delayed synchronization feature (see Delayed Synchronization (page 52)).

For example, suppose your replication group consists of four environments. Upon application startup, all three replicas will immediately attempt to synchronize with the master. But at the same time, the master itself might be busy with a heavy database write load.

To solve this problem, delay synchronization for two of the three replicas. Allow the third replica to synchronize as normal with the master. Then, start synchronization for each of the delayed replicas (since this is a manual process, you can do them one at a time if

that best suits your application). Assuming you have configured replica to replica synchronization correctly, the delayed replicas will synchronize using the up-to-date replica, rather than using the master.

When you are using the replication framework, you configure replica to replica synchronization by declaring one environment to be a peer of another environment. If an environment is a peer, then it can be used for synchronization purposes.

Identifying Peers

You can designate one replica to be a peer of another, and then use this special status for permanent message acknowledgments, and for replica-to-replica synchronization. You might want to do this if you have machines that you know are on fast, reliable network connections and so you are willing to accept the overhead of waiting for acknowledgments from those specific machines.

An environment is currently allowed to have only one peer.

Note that peers are not required to be a bi-directional. That is, just because machine A declares machine B to be a peer, that does not mean machine B must also declare machine A to be a peer.

You declare a peer for the current environment when you add that environment to the list of known sites. You do this by specifying the `DB_REPMGR_PEER` flag to `DbEnv::repmgr_add_remote_site()`.

Bulk Transfers

By default, messages are sent from the master to replicas as they are generated. This can degrade replication performance because the various participating environments must handle a fair amount of network I/O activity.

You can alleviate this problem by configuring your master environment for bulk transfers. Bulk transfers simply cause replication messages to accumulate in a buffer until a triggering event occurs. When this event occurs, the entire contents of the buffer is sent to the replica, thereby eliminating excessive network I/O.

Note that if you are using replica to replica transfers, then you might want any replica that can service replication requests to also be configured for bulk transfers.

The events that result in a bulk transfer of replication messages to a replica will differ depending on if the transmitting environment is a master or a replica.

If the servicing environment is a master environment, then bulk a bulk transfer occurs when:

1. Bulk transfers are configured for the master environment, and
2. the message buffer is full or

3. a permanent record (for example, a transaction commit or a checkpoint record) is placed in the buffer for the replica.

If the servicing environment is a replica environment (that is, replica to replica transfers are in use), then a bulk transfer occurs when:

1. Bulk transfers are configured for the transmitting replica, and

2. the message buffer is full or

3. the replica servicing the request is able to completely satisfy the request with the contents of the message buffer.

To configure bulk transfers, specify `DB_REP_CONF_BULK` to `DbEnv::rep_set_config()` and then specify `1` to the `onoff` parameter. (Specify `0` to turn the feature off.)