Oracle Berkeley DB

Getting Started with Replicated Applications for Java

11g Release 2

(Library Version 11.2.5.1)

Legal Notice

Published 1/31/2011

Table of Contents

Preface

This document describes how to write replicated applications for Berkeley DB 11*g* Release 2 (library version 11.2.5.1). 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: "The `Environment()` constructor returns an `Environment` class object."

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:

```
import com.sleepycat.db.DatabaseConfig;

...

// Allow the database to be created.
DatabaseConfig myDbConfig = new DatabaseConfig();
myDbConfig.setAllowCreate(true);
```

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:

```
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;

...

// Allow the database to be created.
DatabaseConfig myDbConfig = new DatabaseConfig();
myDbConfig.setAllowCreate(true);
Database myDb = new Database("mydb.db", null, myDbConfig);
```

Note

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 Java
- Getting Started with Berkeley DB for Java
- Berkeley DB Collections Tutorial
- Berkeley DB Programmer's Reference Guide
- Berkeley DB Javadoc

To download the latest Berkeley DB documentation along with white papers and other collateral, visit http://www.oracle.com/technetwork/indexes/documentation/index.html.

For the latest version of the Oracle Berkeley DB downloads, visit http://www.oracle.com/technetwork/database/berkeleydb/downloads/index.html.

Contact Us

You can post your comments and questions at the Oracle Technology (OTN) forum for Oracle Berkeley DB at: http://forums.oracle.com/forums/forum.jspa?forumID=271, or for Oracle Berkeley DB High Availability at: http://forums.oracle.com/forums/forum.jspa?forumID=272.

For sales or support information, email to: berkeleydb-info_us@oracle.com You can subscribe to a low-volume email announcement list for the Berkeley DB product family by sending email to: bdb-join@oss.oracle.com

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 to 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.

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 Manager does place a limit on the number of environments you can use. See Replication Manager Overview (page 4) 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 `EnvironmentConfig.addDataDir()` 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 when they are 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 Manager 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. Usually masters are selected by holding an *election*, although it is possible to turn elections off and manually select masters (this is not recommended for most replicated applications).

When elections are being used, they 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 5) for details.

For more information on holding and managing elections, see Holding Elections (page 5).

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:

- Improve 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 the portion of your data that you read on a given replica, that replica may need to cache part of your data, decreasing cache misses and reducing I/O on the replica.

- 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 replicas 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 33).

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 Manager that comes with the DB distribution. This framework should be sufficient for most customers.

If for some reason the Replication Manager does not meet your application's technical requirements, you will have to use the Replication Base 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 Manager Overview

DB's pre-packaged Replication Manager exists as a layer on top of the DB library. The Replication Manager is a multi-threaded implementation that allows you to easily add replication to your existing transactional application. You access and manage the Replication Manager using special methods and classes designated for its use. Mostly these are centered around the `Environment` and `EnvironmentConfig` classes.

The Replication Manager:

- 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 Manager 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 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 Manager, you must write implement replication using the DB Replication Base APIs. See the next section for introductory details.

Replication Base API Overview

The Replication Base 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 Base API only if the Replication Manager does not meet your application's technical requirements.

To make use of the Base API, you must write your own networking code. This frees you from the technical constraints imposed by the Replication Manager. 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 Manager uses TCP/IP sockets. While this should be acceptable for the majority of applications, sometimes UDP or even raw sockets might be desired.

For information on writing a replicated application using the Berkeley DB Replication Base 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.

Note

In some rare situations, it is desireable for the application to manually select its master. For these cases, elections can be turned off.

Manually selecting a master is an activity that should be performed infrequently, if ever. You turn elections off by using the `ReplicationConfig` and `ReplicationManagerStartPolicy` classes.

How elections are held depends upon the API that you use to implement replication. For example, if you are using the Replication Manager 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 Manager, 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 up-to-date 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, the winner is determined first by the environment with the most up-to-date log records. But if this is a tie, the the environment's priority is considered. So given two environments with log records that are equally recent, 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 Manager, 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 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 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 Manager 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 Manager sends the master a message acknowledgment.

For the master (again, using the Replication Manager), 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 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 acknowledge them. 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 Manager, 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 peers needed to ensure that the record remains durable if an election is held.

  An *electable peer* is any other site that potentially can be elected master.

  The goal here is to be absolutely sure the record is durable. The master wants to hear from enough electable peer 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 at least one electable peer.
- Require acknowledgments from all electable peers.

Note that the Replication Manager 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 Manager, see Permanent Message Handling (page 33).

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 implement replication using the Base APIs.

When using the Base APIs, 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 and 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.

```
> java db.repquote_gsg.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: RepConfig

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 `RepConfig` and its only purpose at this time is to track the location of our environment home directory.

```
package db.repquote_gsg;

public class RepConfig
{
    // Constant values used in the RepQuote application.
    public static final String progname = "SimpleTxn";
    public static final int CACHESIZE = 10 * 1024 * 1024;

    // member variables containing configuration information
    public String home; // String specifying the home directory for
                        // rep files.

    public RepConfig()
    {
        home = "TESTDIR";
    }

    public java.io.File getHome()
    {
        return new java.io.File(home);
    }

}
```

Class: SimpleTxn

Our transactional example will consist of a class, `SimpleTxn`, that performs all our work for us.

First, we provide the package declaration and then a few import statements that the class needs.

```
package db.repquote_gsg;

import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.UnsupportedEncodingException;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.DatabaseType;
import com.sleepycat.db.Environment;
import com.sleepycat.db.EnvironmentConfig;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;
import db.repquote_gsg.RepConfig;

public class SimpleTxn
{
    private RepConfig repConfig;
    private Environment dbenv;
```

Next, we provide our class constructor. This simply initializes our class data members.

```
    public SimpleTxn()
        throws DatabaseException
    {
        repConfig = null;
        dbenv = null;
    }
```

And then we provide our `usage()` method. At this point, this method has very little to report:

```
    public static void usage()
    {
        System.err.println("usage: " + repConfig.progname);
        System.err.println("-h home");

        System.err.println("\t -h home directory\n");

        System.exit(1);
```

```
    }
```

Method: SimpleTxn.main()

Having implemented our `usage()` method, we can jump directly into our `main()` method. This method begins by instantiating a `RepConfig` object, and then collecting the command line arguments so that it can populate the object with the appropriate data (just the environment home directory, at this time):

```
    public static void main(String[] argv)
        throws Exception
    {
        RepConfig config = new RepConfig();
        // Extract the command line parameters
        for (int i = 0; i < argv.length; i++)
        {
            if (argv[i].compareTo("-h") == 0) {
                // home - a string arg.
                i++;
                config.home = argv[i];
            } else {
                System.err.println("Unrecognized option: " + argv[i]);
                usage();
            }
        }
```

And then perform a little sanity checking on the command line input:

```
        // Error check command line.
        if (config.home.length() == 0)
            usage();
```

Now we perform the class' work. To begin, we initialize the object. The `init()` method actually opens our environment for us (shown in the next section).

```
        SimpleTxn runner = null;
        try {
            runner = new SimpleTxn();
            runner.init(config);
```

And then we call our `doloop()` method. This method is where we perform all our database activity. See Method: SimpleTxn.doloop() (page 16) for it's details.

```
                    runner.doloop();
```

And then, finally terminate the application (which closes our environment handle) and end the method.

```
            runner.terminate();
        } catch (DatabaseException dbe) {
            System.err.println("Caught an exception during " +
                "initialization or processing: " + dbe.toString());
            if (runner != null)
                runner.terminate();
```

```
        }
            System.exit(0);
    } // end main
```

Method: SimpleTxn.init()

The `SimpleTxn.init()` method is used to 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.

The only thing worth noting in this method here is that we relax our transactional durability guarantee for this application. We do this because the application will eventually be replicated and so we don't need a high durability guarantee.

```
    public int init(RepConfig config)
        throws DatabaseException
    {
        int ret = 0;
        repConfig = config;
        EnvironmentConfig envConfig = new EnvironmentConfig();
        envConfig.setErrorStream(System.err);
        envConfig.setErrorPrefix(RepConfig.progname);

        envConfig.setCacheSize(RepConfig.CACHESIZE);
        envConfig.setTxnNoSync(true);

        envConfig.setAllowCreate(true);
        envConfig.setRunRecovery(true);
        envConfig.setInitializeLocking(true);
        envConfig.setInitializeLogging(true);
        envConfig.setInitializeCache(true);
        envConfig.setTransactional(true);
        try {
            dbenv = new Environment(repConfig.getHome(), envConfig);
        } catch(FileNotFoundException e) {
            System.err.println("FileNotFound exception: " + e.toString());
            System.err.println(
                "Ensure that the environment directory is pre-created.");
            ret = 1;
        }

        return ret;
    }
```

Finally, we present the `SimpleTxn.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.

```
    public void terminate()
        throws DatabaseException
    {
```

```
            dbenv.close();
    }
```

Method: SimpleTxn.doloop()

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:

```
    public int doloop()
        throws DatabaseException, , UnsupportedEncodingException
    {
        Database db = null;
```

Next, we begin the loop and we immediately open our database if it has not already been opened.

```
        for (;;)
        {
            if (db == null) {
                DatabaseConfig dbconf = new DatabaseConfig();
                dbconf.setType(DatabaseType.BTREE);
                dbconf.setAllowCreate(true);
                dbconf.setTransactional(true);

                try {
                    db = dbenv.openDatabase(null,           // Txn handle
                                            RepConfig.progname, // db filename
                                            null,               // db name
                                            dbconf);
               } catch (FileNotFoundException fnfe) {
                    System.err.println("File not found exception" +
                                            fnfe.toString());
                    // Get here only if the environment home directory
                    // somehow does not exist.
               }
            }
```

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 `printStocks()` 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.

```
            BufferedReader stdin =
                new BufferedReader(new InputStreamReader(System.in));

            // listen for input, and add it to the database.
            System.out.print("QUOTESERVER> ");
            System.out.flush();
            String nextline = null;
            try {
                nextline = stdin.readLine();
            } catch (IOException ioe) {
                System.err.println("Unable to get data from stdin");
                break;
            }
            String[] words = nextline.split("\\s");

            // A blank line causes the DB to be dumped to stdout.
            if (words.length == 0 ||
                (words.length == 1 && words[0].length() == 0)) {
                try {
                    printStocks(db);
                } catch (DatabaseException e) {
                    System.err.println("Got db exception reading " +
                        "DB: " + e.toString());
                    break;
                }
                continue;
            }

            if (words.length == 1 &&
                (words[0].compareToIgnoreCase("quit") == 0 ||
                words[0].compareToIgnoreCase("exit") == 0)) {
                break;
            } else if (words.length != 2) {
                System.err.println("Format: TICKER VALUE");
                continue;
            }
```

Now we assign data to the `DatabaseEntry` classes that we will use to write the new information to the database.

```
            DatabaseEntry key =
                    new DatabaseEntry(words[0].getBytes("UTF-8"));
            DatabaseEntry data =
                    new DatabaseEntry(words[1].getBytes("UTF-8"));
```

Having done that, we can write the new information to the database. Remember that because a transaction handle is not explicitly used, but we did open the database such that it supports transactions, then autocommit is automatically used for this database write.

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

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 `OperationStatus.KEYEXIST` — which we ignore.

```
            db.put(null, key, data);
```

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

```
        }
        if (db != null)
            db.close(true);
        return 0;
    }
```

Method: SimpleTxn.printStocks()

The `printStocks()` 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.

```
    public void terminate()
        throws DatabaseException
    {
            dbenv.close();
    }

    /*
     * void return type since error conditions are propagated
     * via exceptions.
     */
    private void printStocks(Database db)
        throws DatabaseException
    {
        Cursor dbc = db.openCursor(null, null);

        System.out.println("\tSymbol\tPrice");
        System.out.println("\t======\t=====");

        DatabaseEntry key = new DatabaseEntry();
        DatabaseEntry data = new DatabaseEntry();
        OperationStatus ret;
        for (ret = dbc.getFirst(key, data, LockMode.DEFAULT);
            ret == OperationStatus.SUCCESS;
            ret = dbc.getNext(key, data, LockMode.DEFAULT)) {
            String keystr = new String
```

```
                (key.getData(), key.getOffset(), key.getSize());
            String datastr = new String
                (data.getData(), data.getOffset(), data.getSize());
            System.out.println("\t"+keystr+"\t"+datastr);

        }
        dbc.close();
    }
} // end class
```

Chapter 3. The DB Replication Manager

The easiest way to add replication to your transactional application is to use the Replication Manager. The Replication Manager provides a comprehensive communications layer that enables replication. For a brief listing of the Replication Manager's feature set, see Replication Manager Overview (page 4).

To use the Replication Manager, you make use of special methods off the `Environment` and `EnvironmentConfig` classes. You also use a series of related classes to perform your implementation. For example, in order to detect whether your code is running as a master or a replica, you must implement `com.sleepycat.db.EventHandler`. (see Determining State (page 38)). 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 Manager replication methods to configure the Replication Manager. Using these methods causes DB to know that you are using the Replication Manager.

   Configuring the Replication Manager entails setting the replication environment's priority, setting the TCP/IP address that this replication environment will use for incoming replication messages, identifying 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 `EnvironmentConfig.setInitializeReplication()` when you configure your environment handle. This is in addition to the configuration that you would normally use for a transactional application. This causes replication to be initialized for the application.

5. Start replication by calling `Environment.replicationManagerStart()`.

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.

Note

The Replication Manager 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.

Note

Before you can use the Replication Manager, 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 Manager 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 `Environment.replicationManagerStart()`.

You stop replication by closing your environment cleanly in the same way you would for any DB application.

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

Note

Note that the following code fragment would be part of a larger class that must implement `com.sleepycat.db.EventHandler`. This class is used to track state changes between master and replica. We put off that implementation for the moment, but the point remains that the following code fragment would be contained in a method or two that you would include in your `com.sleepycat.db.EventHandler` implementation.

```
import com.sleepycat.db.Environment;
import com.sleepycat.db.EnvironmentConfig;
import com.sleepycat.db.ReplicationHostAddress;
import com.sleepycat.db.ReplicationManagerStartPolicy;

...
    String progname = "example_replication";
    String envHome = "TESTDIR";
    int cachesize = 10 * 1024 * 1024;

    Environment dbenv;
    ReplicationHostAddress thisHost;
    String listenHost = "mymachine.sleepycat.com";
```

```
    int listenPort = 8080;

    ReplicationHostAddress otherReplica;
    String otherHost = "anothermachine.sleepycat.com";
    int otherPort = 8081;


try {
    // Configure the environment handle
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setErrorStream(System.err);
    envConfig.setErrorPrefix(progname);
    envConfig.setCacheSize(cachesize);
    envConfig.setTxnNoSync(true);

    // Identify the local replication site. 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.
    thisHost = new ReplicationHostAddress(listenHost, listenPort);
    envConfig.setReplicationManagerLocalSite(thisHost);

    // 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.
    envConfig.setReplicationPriority(100);
    // Add a site to the list of replication environments known to
    // this application.
    otherReplica = new ReplicationHostAddress(otherHost, otherPort);
    envConfig.replicationManagerAddRemoteSite(otherReplica);

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

    // Configure the environment's subsystems. Note that we initialize
    // replication. This is required.
    envConfig.setAllowCreate(true);
    envConfig.setRunRecovery(true);
    envConfig.setThreaded(true);
    envConfig.setInitializeReplication(true);
    envConfig.setInitializeLocking(true);
    envConfig.setInitializeLogging(true);
    envConfig.setInitializeCache(true);
    envConfig.setTransactional(true);

    // Missing from this is where we set the event handle and the
```

```
    // acknowledgement policy. We discuss these things later in this
    // book.

    // Open our environment handle.
    try {
        dbenv = new Environment(envHome, envConfig);
    } catch(FileNotFoundException e) {
        System.err.println("FileNotFound exception: " + e.toString());
        System.err.println(
            "Ensure that the environment directory is pre-created.");
    }

    // Start the replication manager such that it has three threads.
    dbenv.replicationManagerStart(3,
        ReplicationManagerStartPolicy.REP_ELECTION);

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

} catch (DatabaseException dbe) {
    // Error handling goes here
}

// Close out your application here.
try {
    // Make sure all your databases are closed.

    // Closing your environment stops replication.
   dbenv.close();
} catch (DatabaseException dbe) {
    // Error handling here.
}

// All done.
```

Managing Election Policies

Before continuing, it is worth taking a look at the startup election options that you can set for replication. You set these using the `ReplicationManagerStartPolicy` class that you pass to the `Environment.replicationManagerStart()` method.

In the previous example, we specified `ReplicationManagerStartPolicy.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 option 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 `EnvironmentConfig.setReplicationNumSites()`. This is always true of elections held using the Replication Manager.
- As always, the environment participating in the election with the most up-to-date log files is selected as master. If an environment with more recent 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 Manager offers the following start-up options:

Option	Description
`ReplicationManagerStartPolicy.REP_MASTER`	The application starts up and declares the environment 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.
`ReplicationManagerStartPolicy.REP_CLIENT`	The application starts up and declares the environment to be a replica without calling for an election. Note that the environment can still become a master if a subsequent application starts up, calls for an election, and this environment is elected master.
`ReplicationManagerStartPolicy.REP_ELECTION`	As described above, the application starts up, looks for a master, and if one is not found calls for an election.

Selecting the Number of Threads

Under the hood, the Replication Manager is threaded and you can control the number of threads used to process messages received from other replicas. The threads that the Replication Manager 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 messages are sent from whatever thread performed a write to the database(s). That is, the thread that called, for example, `Database.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 Manager 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 Manager 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 Manager.
- Minimally configure the Replication Manager.
- 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 make some significant changes to our `RepConfig` class because we will be using it to maintain a lot more information that we needed for our simple transactional example.

We begin by importing a few new classes. `java.util.Vector` is used to organize a list of "other host" definitions (that is, the host and port information for the other replication participants known to this application). We also need a couple of classes used to manage individual host and port information, as well as replication startup policy information.

```
package db.repquote_gsg;

import java.util.Vector;

import com.sleepycat.db.ReplicationHostAddress;
import com.sleepycat.db.ReplicationManagerStartPolicy;

public class RepConfig
{
```

Next we add considerably to the constants and data members used by this class. All of this is used to manage information necessary for replication purposes. We also at this point change the program's name, since we will be doing that to the main class in our application a little later in this description.

```
    // Constant values used in the RepQuote application.
    public static final String progname = "RepQuoteExampleGSG";
    public static final int CACHESIZE = 10 * 1024 * 1024;
    public static final int SLEEPTIME = 5000;

    // member variables containing configuration information
    // String specifying the home directory for rep files.
    public String home;
    // stores an optional set of "other" hosts.
    public Vector<ReplicationHostAddress> otherHosts;
    // priority within the replication group.
    public int priority;
    public ReplicationManagerStartPolicy startPolicy;
    // The host address to listen to.
    public ReplicationHostAddress thisHost;
    // Optional parameter specifying the # of sites in the
    // replication group.
    public int totalSites;

    // member variables used internally.
    private int currOtherHost;
    private boolean gotListenAddress;
```

Now we update our class constructor to initialize all of these new variables:

```
    public RepConfig()
    {
        startPolicy = ReplicationManagerStartPolicy.REP_ELECTION;
```

```
        home = "TESTDIR";
        gotListenAddress = false;
        totalSites = 0;
        priority = 100;
        currOtherHost = 0;
        thisHost = new ReplicationHostAddress();
        otherHosts = new Vector()<ReplicationHostAddress>;
    }
```

Finally, we finish updating this class by providing a series of new getter and setter methods. These are used primarily for setting a retrieving host information of interest to our replicated application:

```
    public java.io.File getHome()
    {
        return new java.io.File(home);
    }

    public void setThisHost(String host, int port)
    {
        gotListenAddress = true;
        thisHost.port = port;
        thisHost.host = host;
    }

    public ReplicationHostAddress getThisHost()
    {
        if (!gotListenAddress) {
            System.err.println("Warning: no host specified.");
            System.err.println("Returning default.");
        }
        return thisHost;
    }

    public boolean gotListenAddress() {
        return gotListenAddress;
    }

    public void addOtherHost(String host, int port, boolean peer)
    {
        ReplicationHostAddress newInfo =
            new ReplicationHostAddress(host, port, peer);
        otherHosts.add(newInfo);
    }

    public ReplicationHostAddress getFirstOtherHost()
    {
        currOtherHost = 0;
        if (otherHosts.size() == 0)
            return null;
```

```
        return (ReplicationHostAddress)otherHosts.get(currOtherHost);
    }

    public ReplicationHostAddress getNextOtherHost()
    {
        currOtherHost++;
        if (currOtherHost >= otherHosts.size())
            return null;
        return (ReplicationHostAddress)otherHosts.get(currOtherHost);
    }

    public ReplicationHostAddress getOtherHost(int i)
    {
        if (i >= otherHosts.size())
            return null;
        return (ReplicationHostAddress)otherHosts.get(i);
    }
}
```

Having completed our update to the `RepConfig` class, we can now start making changes to the main portion of our program. We begin by changing the program's name. (This, of course, means that we copy our `SimpleTxn` code to a file named `RepQuoteExampleGSG.java`.)

```
package db.repquote_gsg;

import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.lang.Thread;
import java.lang.InterruptedException;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.DatabaseType;
import com.sleepycat.db.Environment;
import com.sleepycat.db.EnvironmentConfig;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;
import db.repquote.RepConfig;

public class RepQuoteExampleGSG
{
    private RepConfig repConfig;
    private Environment dbenv;
```

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

- `-l` parameter which allows us to identify the host and port used by this application to listen for replication messages.
- `-r` 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)

```
    public RepQuoteExampleGSG()
        throws DatabaseException
    {
        repConfig = null;
        dbenv = null;
    }

    public static void usage()
    {
        System.err.println("usage: " + repConfig.progname);
        System.err.println("-h home[-r host:port][-l host:port]" +
            "[-n nsites][-p priority]");

        System.err.println("\t -l host:port +
             "(required; l stands for local)\n" +
             "\t -r host:port (optional; r stands for replica; any " +
             "number of these may be specified)\n" +
             "\t -h home directory\n" +
             "\t -n nsites (optional; number of sites in replication " +
             "group; defaults to 0\n" +
             "\t    in which case we try to dynamically compute the " +
             "number of sites in\n" +
             "\t    the replication group)\n" +
             "\t -p priority (optional: defaults to 100)\n");

        System.exit(1);
    }
```

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.

```
    public static void main(String[] argv)
        throws Exception
    {
        RepConfig config = new RepConfig();
```

```
        String tmpHost;
        int tmpPort = 0;
```

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
        for (int i = 0; i < argv.length; i++)
        {
            if (argv[i].compareTo("-h") == 0) {
                // home - a string arg.
                i++;
                config.home = argv[i];
           } else if (argv[i].compareTo("-l") == 0) {
                // "me" should be host:port
                i++;
                String[] words = argv[i].split(":");
                if (words.length != 2) {
                    System.err.println(
                        "Invalid host specification host:port needed.");
                    usage();
                }
                try {
                    tmpPort = Integer.parseInt(words[1]);
                } catch (NumberFormatException nfe) {
                    System.err.println("Invalid host specification, " +
                        "could not parse port number.");
                    usage();
                }
                config.setThisHost(words[0], tmpPort);
            } else if (argv[i].compareTo("-n") == 0) {
                i++;
                config.totalSites = Integer.parseInt(argv[i]);
           } else if (argv[i].compareTo("-p") == 0) {
                i++;
                config.priority = Integer.parseInt(argv[i]);
            } else if (argv[i].compareTo("-r") == 0) {
                i++;
                String[] words = argv[i].split(":");
                if (words.length != 2) {
                    System.err.println(
                        "Invalid host specification host:port needed.");
                    usage();
                }
                try {
                    tmpPort = Integer.parseInt(words[1]);
                } catch (NumberFormatException nfe) {
                    System.err.println("Invalid host specification, " +
                        "could not parse port number.");
                    usage();
```

```
                }
                config.addOtherHost(words[0], tmpPort, isPeer);
            } else {
                System.err.println("Unrecognized option: " + argv[i]);
                usage();
            }
        }

        // Error check command line.
        if ((!config.gotListenAddress()) || config.home.length() == 0)
            usage();
```

Having done that, the remainder of our `main()` function is left unchanged, with the exception of a few name changes required by the new class name:

```
        RepQuoteExampleGSG runner = null;
        try {
            runner = new RepQuoteExampleGSG();
            runner.init(config);

            runner.doloop();
            runner.terminate();
        } catch (DatabaseException dbe) {
            System.err.println("Caught an exception during " +
                "initialization or processing: " + dbe.toString());
            if (runner != null)
                runner.terminate();
        }
            System.exit(0);
    } // end main
```

Now we need to update our `RepQuoteExampleGSG.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 `-l` command line option):

```
    public int init(RepConfig config)
        throws DatabaseException
    {
        int ret = 0;
        appConfig = config;
        EnvironmentConfig envConfig = new EnvironmentConfig();
        envConfig.setErrorStream(System.err);
        envConfig.setErrorPrefix(RepConfig.progname);

        envConfig.setReplicationManagerLocalSite(appConfig.getThisHost());
```

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 (ReplicationHostAddress host = appConfig.getFirstOtherHost();
            host != null; host = appConfig.getNextOtherHost())
        {
            envConfig.replicationManagerAddRemoteSite(host);
        }
```

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 (appConfig.totalSites > 0)
            envConfig.setReplicationNumSites(appConfig.totalSites);
        envConfig.setReplicationPriority(appConfig.priority);
```

We can now open our environment. Note that the options 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 `EnvironmentConfig.setInitializeReplication()` option.

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

```
        envConfig.setCacheSize(RepConfig.CACHESIZE);
        envConfig.setTxnNoSync(true);

        envConfig.setAllowCreate(true);
        envConfig.setRunRecovery(true);
        envConfig.setInitializeReplication(true);
        envConfig.setInitializeLocking(true);
        envConfig.setInitializeLogging(true);
        envConfig.setInitializeCache(true);
        envConfig.setTransactional(true);
        try {
            dbenv = new Environment(appConfig.getHome(), envConfig);
        } catch(FileNotFoundException e) {
            System.err.println("FileNotFound exception: " + e.toString());
            System.err.println(
                "Ensure that the environment directory is pre-created.");
            ret = 1;
        }
```

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

```
        // start Replication Manager
        dbenv.replicationManagerStart(3, appConfig.startPolicy);
        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 replicas. More importantly, the thread performing the transaction commit blocks until it either receives enough acknowledgments, or the acknowledgment timeout expires.

The Replication Manager 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 Manager for you. However, you do have to set some policies that tell the Replication Manager 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 `ReplicationManagerAckPolicy` class which you pass to the environment using the `EnvironmentConfig.setReplicationManagerAckPolicy` method. 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 Manager:

Note

The following list mentions *electable peer* several times. This 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 54) for more information on client to client transfers.

- `ReplicationManagerAckPolicy.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).

- `ReplicationManagerAckPolicy.ONE`

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

- `ReplicationManagerAckPolicy.ONE_PEER`

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

- `ReplicationManagerAckPolicy.ALL`

  All replicas 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 replicas that must respond is determined by the value set for `EnvironmentConfig.setReplicationNumSites()`.

- `ReplicationManagerAckPolicy.ALL_AVAILABLE`

  All currently connected replication clients must acknowledge the message. This policy will invoke the `DB_EVENT_REP_PERM_FAILED` event if fewer than a quorum of clients acknowledged during that time.

- `ReplicationManagerAckPolicy.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.

- `ReplicationManagerAckPolicy.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.

By default, a quorum of electable peers must must acknowledge a permanent message in order for it considered to have been successfully transmitted. The actual number of replicas that must respond is calculated using the value set with `EnvironmentConfig.setReplicationNumSites()`.

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 `Environment.setReplicationTimeout()`. You pass this method the `ReplicationTimeoutType.ACK_TIMEOUT` constant and a timeout value in microseconds.

For example:

```
dbenv.setReplicationTimeout(ReplicationTimeoutType.ACK_TIMEOUT, 100);
```

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

Adding a Permanent Message Policy to RepQuoteExampleGSG

For illustration purposes, we will now update `RepQuoteExampleGSG` 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 `RepQuoteExampleGSG.init()` immediately after we set the application's priority and before we open our environment handle.

```
    public int init(RepConfig config)
        throws DatabaseException
    {
        int ret = 0;
        appConfig = config;
        EnvironmentConfig envConfig = new EnvironmentConfig();
        envConfig.setErrorStream(System.err);
        envConfig.setErrorPrefix(RepConfig.progname);

        envConfig.setReplicationManagerLocalSite(appConfig.getThisHost());
        for (ReplicationHostAddress host = appConfig.getFirstOtherHost();
            host != null; host = appConfig.getNextOtherHost())
        {
            envConfig.replicationManagerAddRemoteSite(host);
        }

        if (appConfig.totalSites > 0)
            envConfig.setReplicationNumSites(appConfig.totalSites);
        envConfig.setReplicationPriority(appConfig.priority);

        envConfig.setReplicationManagerAckPolicy(
                ReplicationManagerAckPolicy.ALL);
        envConfig.setReplicationTimeout(ReplicationTimeoutType.ACK_TIMEOUT,
                500);

        envConfig.setCacheSize(RepConfig.CACHESIZE);
        envConfig.setTxnNoSync(true);
    ...
```

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 an 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 how many replicas have participated in the election. 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, if not enough replicas have participated in the election 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 Manager 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 `Environment.setReplicationTimeout()`. You pass this method the `ReplicationTimeoutType.ELECTION_TIMEOUT` constant and a timeout value in microseconds.

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 election timeout value using `Environment.setReplicationTimeout()`. You pass this method the `ReplicationTimeoutType.ELECTION_RETRY` constant and a retry value in microseconds.

Note that this constant is only valid when you are using the Replication Manager. If you are using the Base APIs, then this constant is ignored.

Managing Connection Retries

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

this method the `ReplicationTimeoutType.CONNECTION_RETRY` constant and a retry value in microseconds.

Managing Heartbeats

If your replicated application experiences few updates, it is possible for the replication group to lose a master without noticing it. This is because normally a replicated application only knows that a master has gone missing when update activity causes messages to be passed between the master and replicas.

To guard against this, you can configure a heartbeat. The heartbeat must be configured for both the master and each of the replicas.

On the master, you configure the application to send a heartbeat on a defined interval when it is otherwise idle. Do this by using the `Environment.setReplicationTimeout()` method. You pass this method the `ReplicationTimeoutType.HEARTBEAT_SEND` constant. You must also provide the method a value representing the frequency of the heartbeat in microseconds. Note that the heartbeat is sent only if the system is idle.

On the replica, you configure the application to listen for a heartbeat. The time that you configure here is the amount of time the replica will wait for some message from the master (either the heartbeat or some other message) before concluding that the connection is lost. You do this using the `Environment.setReplicationTimeout()` method. You pass this method the `ReplicationTimeoutType.HEARTBEAT_MONITOR` constant and a timeout value in microseconds.

For best results, configure the heartbeat monitor for a longer time interval than the heartbeat send interval.

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 quite 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 must write your application as an implementation of `com.sleepycat.db.EventHandler`. This class gives you a series of methods within which you can detect and respond to various events that occur in your DB code. Some, but not all, of these methods have to do with elections:

- `EventHandler.handleRepMasterEvent()`

  The local environment is now a master.

- `EventHandler.handleRepClientEvent()`

  The local environment is now a replica.

- `EventHandler.handleRepStartupDoneEvent()`

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

- `EventHandler.handleRepElectedEvent()`

  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.

- `EventHandler.handleRepPanicEvent()`

  An error has occured in the Berkeley DB library requiring your application to shut down and then run recovery.

- `EventHandler.handleRepPermFailedFailedEvent()`

  The Replication Manager did not receive enough acknowledgements to ensure the transaction's durability within the replicationg group. The Replication Manager has therefore flushed the transaction to the master's local disk for storage.

  How the Replication Manager knows whether the acknowledgements it has received is determined by the ack policy you have set for your applicaton. See Identifying Permanent Message Policies (page 33) for more information.

- `EventHandler.handleWriteFailedEvent()`

  A Berkeley DB write to stable storage failed.

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

The `EventHandler` implementation is fairly simple. First you detect the event, and then you record the state change in some data member maintained in a location that is convenient to you.

For example:

```
package db.repquote;

// We make our main class an EventHandler implementation
...
import com.sleepycat.db.EventHandler;
...

public class MyReplicationClass implements EventHandler
{

...

// Somewhere we provide a data member that is used to track
// whether we are a master server. This could be in our main
// class, or it could be part of a supporting class.
private boolean isMaster;

...

isMaster = false;

...

// In the code where we open our environment and start replication,
// we must identify the class that is the event handler. In this
// example, we are performing this from within the class that
```

```
// implements com.sleepycat.db.EventHandler so we identify
// "this" class as the event handler
envConfig.setEventHandler(this);
```

That done, we still need to implement the methods required for handling replication events. For a simple application like this one, these implementations can be trivial.

```
    public void handleRepClientEvent()
    {
        dbenv.setIsMaster(false);
    }

    public void handleRepMasterEvent()
    {
        dbenv.setIsMaster(true);
    }

    public void handleRepNewMasterEvent(int envId)
    {
        // Ignored for now
    }

    public void handleWriteFailedEvent(int errorCode)
    {
        System.err.println("Write to stable storage failed!" +
            "Operating system error code:" + errorCode);
        System.err.println("Continuing....");
    }

    public void handleRepStartupDoneEvent()
    {
        System.out.println("Replication startup is completed.");
    }

    public void handleRepPermFailedEvent()
    {
        System.out.println("This application failed to receive enough" +
            "acks for a permanent message. The transaction is flushed" +
            "to disk on this master host.");
    }

    public void handleRepElectedEvent()
    {
        // Safely ignored for Replication Manager applications.
    }

    public void handlePanicEvent()
    {
        System.err.println("Panic encountered!");
        System.err.println("Shutting down.");
```

```
        System.err.println("You should restart, running recovery.");
        try {
            terminate();
        } catch (DatabaseException dbe) {
            System.err.println("Caught an exception during " +
                "termination in handlePanicEvent: " + dbe.toString());
        }
        System.exit(-1);
    }
```

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 `ReplicationHandleDeadException` exception 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 close and then 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
public int doloop()
{
    Database db = null;


// 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 (db == null) {
         // Create the handle and then configure it. Before you open
         // it, you have to decide what open flags to use:
         DatabaseConfig dbconf = new DatabaseConfig();
         dbconf.setType(DatabaseType.BTREE);
         if (isMaster) {
            dbconf.setAllowCreate(true);
         }

         // Now you can open your database handle, passing to it the
         // optins selected above.

         try {
            db = dbenv.openDatabase
                 (null, progname, null, dbconf);
         } catch(java.io.FileNotFoundException e) {
                // Put your error handling code here.
         }
    }

     // Now that the databases have been opened, continue with general
     // processing, depending on whether we are a master or a replica.
     if (isMaster) {
         // 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: SimpleTxn.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 must implement a way to track whether our application is running as a master or a client. There are many ways to do this, but in this case what we will do is extend `com.sleepycat.db.Environment` to carry the information. We do this by creating the `RepQuoteEnvironment` class.

```
package db.repquote;

import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.Environment;
import com.sleepycat.db.EnvironmentConfig;

public class RepQuoteEnvironment extends Environment
{
    private boolean isMaster;

    public RepQuoteEnvironment(final java.io.File host,
        EnvironmentConfig config)
        throws DatabaseException, java.io.FileNotFoundException
    {
        super(host, config);
        isMaster = false;
    }

    boolean getIsMaster()
    {
        return isMaster;
    }

    public void setIsMaster(boolean isMaster)
    {
        this.isMaster = isMaster;
    }
}
```

Next, we go to `RepQuoteExampleGSG.java` and we include the `RepQuoteEnvironment` class as well as the `EventHandler` class. We then cause our `RepQuoteExampleGSG` class to implement `EventHandler`. We also change our environment handle to be an instance of `RepQuoteEnvironment` instead of `Environment`.

Note that we also import the `com.sleepycat.db.ReplicationHandleDeadException` class. We will discuss what that exception is used for a little later in this example.

```
package db.repquote;

import java.io.FileNotFoundException;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.lang.Thread;
import java.lang.InterruptedException;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.DatabaseType;
import com.sleepycat.db.EnvironmentConfig;
import com.sleepycat.db.EventHandler;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;
import com.sleepycat.db.ReplicationHandleDeadException;

import db.repquote.RepConfig;
import db.repquote.RepQuoteEnvironment

public class RepQuoteExampleGSG  implements EventHandler
{
    private RepConfig repConfig;
    private RepQuoteEnvironment dbenv;
```

That done, we can skip the `main()` method and our class constructor, because they do not change. Instead, we skip down to our `init()` method where we take care of opening our environment and setting the event handler.

To update our `init()` method, we only need to do a couple of things. First, we identify the current class as the event handler. Then, when we open our environment, we instantiate a `RepQuoteEnvironment` class instead of an `Environment` class.

```
    public int init(RepConfig config)
        throws DatabaseException
    {
        int ret = 0;
        repConfig = config;
        EnvironmentConfig envConfig = new EnvironmentConfig();
        envConfig.setErrorStream(System.err);
        envConfig.setErrorPrefix(RepConfig.progname);

        envConfig.setReplicationManagerLocalSite(repConfig.getThisHost());
        for (ReplicationHostAddress host = repConfig.getFirstOtherHost();
            host != null; host = repConfig.getNextOtherHost())
```

```
        {
            envConfig.replicationManagerAddRemoteSite(host);
        }

        if (appConfig.totalSites > 0)
            envConfig.setReplicationNumSites(repConfig.totalSites);
        envConfig.setReplicationPriority(repConfig.priority);

        envConfig.setReplicationManagerAckPolicy(
            ReplicationManagerAckPolicy.ALL);
        envConfig.setCacheSize(RepConfig.CACHESIZE);
        envConfig.setTxnNoSync(true);

        envConfig.setEventHandler(this);

        envConfig.setAllowCreate(true);
        envConfig.setRunRecovery(true);
        envConfig.setThreaded(true);
        envConfig.setInitializeReplication(true);
        envConfig.setInitializeLocking(true);
        envConfig.setInitializeLogging(true);
        envConfig.setInitializeCache(true);
        envConfig.setTransactional(true);
        envConfig.setVerboseReplication(appConfig.verbose);
        try {
            dbenv = new RepQuoteEnvironment(repConfig.getHome(),
                                                    envConfig);
        } catch(FileNotFoundException e) {
            System.err.println("FileNotFound exception: " + e.toString());
            System.err.println(
                "Ensure that the environment directory is pre-created.");
            ret = 1;
        }

        // start Replication Manager
        dbenv.replicationManagerStart(3, repConfig.startPolicy);
        return ret;
    }
```

That done, we need to implement the methods required for responding to replication events. These methods are required because we are now implementing `com.sleepycat.db.EventHandler`. While we are required to provide an implementation for all of these methods, for our simple application we are really only interested in these because they allow us to track whether a replication instance is a master.

```
    public void handleRepClientEvent()
    {
        dbenv.setIsMaster(false);
    }
```

```
    public void handleRepMasterEvent()
    {
        dbenv.setIsMaster(true);
    }

    public void handleRepNewMasterEvent(int envId)
    {
        // Ignored for now
    }

    public void handleWriteFailedEvent(int errorCode)
    {
        // Ignored for now
    }

    public void handleRepStartupDoneEvent()
    {
        // Ignored for now
    }

    public void handleRepPermFailedEvent()
    {
        // Ignored for now
    }

    public void handleRepElectedEvent()
    {
        // Safely ignored for Replication Manager applications.
    }

    public void handlePanicEvent()
    {
        System.err.println("Panic encountered!");
        System.err.println("Shutting down.");
        System.err.println("You should restart, running recovery.");
        try {
            terminate();
        } catch (DatabaseException dbe) {
            System.err.println("Caught an exception during " +
                "termination in handlePanicEvent: " + dbe.toString());
        }
        System.exit(-1);
    }
```

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

We begin by updating our `DatabaseConfig` instance to determine which options to use, depending on whether the application is running as a master.

```
    public int doloop()
```

```
        throws DatabaseException
    {
        Database db = null;

        for (;;)
        {
            if (db == null) {
                DatabaseConfig dbconf = new DatabaseConfig();
                dbconf.setType(DatabaseType.BTREE);
                if (dbenv.getIsMaster()) {
                    dbconf.setAllowCreate(true);
                }
                dbconf.setTransactional(true);
```

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 Manager 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 Manager a chance to create the database so that our replica can continue operations.

```
                try {
                    db = dbenv.openDatabase
                        (null, RepConfig.progname, null, dbconf);
                } catch (java.io.FileNotFoundException e) {
                    System.err.println("no stock database available yet.");
                    if (db != null) {
                        db.close(true);
                        db = null;
                    }
                    try {
                        Thread.sleep(RepConfig.SLEEPTIME);
                    } catch (InterruptedException ie) {}
                    continue;
                }
            }
```

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.

```
            BufferedReader stdin =
                new BufferedReader(new InputStreamReader(System.in));

            // listen for input, and add it to the database.
            System.out.print("QUOTESERVER");
            if (!dbenv.getIsMaster())
                System.out.print("(read-only)");
            System.out.print("> ");
```

```
            System.out.flush();
            String nextline = null;
            try {
                nextline = stdin.readLine();
            } catch (IOException ioe) {
                System.err.println("Unable to get data from stdin");
                break;
            }
            String[] words = nextline.split("\\s");
```

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: SimpleTxn.printStocks() (page 18)).

When we call `printStocks()`, 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.

```
            // A blank line causes the DB to be dumped to stdout.
            if (words.length == 0 ||
                (words.length == 1 && words[0].length() == 0)) {
                try {
                    printStocks(db);
                } catch (DeadlockException de) {
                    continue;
                // Dead replication handles are caused by an election
                // resulting in a previously committing read becoming
                // invalid. Close the db handle and reopen.
                } catch (ReplicationHandleDeadException rhde) {
                    db.close(true); // close no sync.
                    db = null;
                    continue;
                } catch (DatabaseException e) {
                    System.err.println("Got db exception reading " +
                        "replication DB: " + e.toString());
                    break;
                }
                continue;
            }

            if (words.length == 1 &&
                (words[0].compareToIgnoreCase("quit") == 0 ||
                words[0].compareToIgnoreCase("exit") == 0)) {
                break;
            } else if (words.length != 2) {
```

```
                System.err.println("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 (!dbenv.getIsMaster()) {
                System.err.println("Can't update client.");
                continue;
            }

            DatabaseEntry key = new DatabaseEntry(words[0].getBytes());
            DatabaseEntry data = new DatabaseEntry(words[1].getBytes());

            db.put(null, key, data);
        }
        if (db != null)
            db.close(true);
        return 0;
    }
```

With that completed, we are all done updating our application for replication. The only remaining method, `printStocks()`, is unmodified from when we originally introduced it. For details on that function, see Method: SimpleTxn.printStocks() (page 18).

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 `-l` option to identify the local host and port that this process will use to listen for replication messages, the `-n` option to specify the number of sites in the replication group, and the `-r` 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
> java db.repquote.RepQuoteExampleGSG -h env1 -n 2 -l localhost:8080 \
-r 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 `-l` flag to at least change the port number the process is listening on, and use the `-r` option to identify the host and port of the other replication process:

```
> mkdir env2
> java db.repquote.RepQuoteExampleGSG -h env2 -n 2 -l localhost:8081 \
-r 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. On the master:

```
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. On the replica:

```
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 54)
5. Bulk Transfers (page 55)

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 `ReplicationConfig.DELAYCLIENT` and `true` to `Environment.setReplicationConfig()`. To turn off delayed synchronization, specify `false` for the `ReplicationConfig.DELAYCLIENT` field.

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 `Environment.syncReplication()` on the replica that has been delayed.

Managing Blocking Operations

When a replica is in the process of synchronizing with its master, DB operations are blocked at some points during this process until 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 throwing a

`com.sleepycat.db.ReplicationLockoutException` exception. When this happens, it is up 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 `ReplicationConfig.NOWAIT` and `true` to `Environment.setReplicationConfig()`. To turn off this feature, specify `false` for the `ReplicationConfig.NOWAIT` field.

Stop Auto-Initialization

As stated in the previous section, when a replication replica is synchronizing with its master, it will block DB operations at some points during this process 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 decide 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 completely 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 its 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 `ReplicationConfig.AUTOINIT` and `false` to `Environment.setReplicationConfig()`.

Read-Your-Writes Consistency

In a distributed system, the changes made at the master are not always instantaneously available at every replica, although they eventually will be. In general, replicas not directly involved in contributing to the acknowledgement of a transaction commit will lag behind other replicas because they do not synchronize their commits with the master.

For this reason, you might want to make use of the read-your-writes consistency feature. This feature allows you to ensure that a replica is at least current enough to have the changes made by a specific transaction. Because transactions are applied serially, by ensuring a replica has a specific commit applied to it, you know that all transaction commits occurring prior to the specified transaction have also been applied to the replica.

You determine whether a transaction has been applied to a replica by generating a *commit token* at the master. You then transfer this commit token to the replica, where it is used to determine whether the replica is consistent enough relative to the master.

For example, suppose the you have a web application where a replication group is implemented within a load balanced web server group. Each request to the web server

consists of an update operation followed by read operations (say, from the same client), The read operations naturally expect to see the data from the updates executed by the same request. However, the read operations might have been routed to a replica that did not execute the update.

In such a case, the update request would generate a commit token, which would be resubmitted by the browser, along with subsequent read requests. The read request could be directed at any one of the available web servers by a load balancer. The replica which services the read request would use that commit token to determine whether it can service the read operation. If the replica is current enough, it can immediately execute the transaction and satisfy the request.

What action the replica takes if it is not consistent enough to service the read request is up to you as the application developer. You can do anything from blocking while you wait for the transaction to be applied locally, to rejecting the read request outright.

For more information, see the `Read your writes consistency` section in the `Berkeley DB Replication` chapter of the *Berkeley DB Programmer's Reference Guide*.

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 Manager, you configure replica to replica synchronization by declaring an 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 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.

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 constructing a `ReplicationHostAddress` object that specifies `true` for the `isPeer` parameter, and then providing that object to `EnvironmentConfig.replicationManagerAddRemoteSite()` when you add the remote site to the local replication 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 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 `ReplicationConfig.BULK` and `true` to `Environment.setReplicationConfig()`. To turn off this feature, specify `false` for the `ReplicationConfig.BULK` field.