Oracle Berkeley DB, Java Edition

Getting Started with High Availability Applications

Release 4.0

Legal Notice

Published 1/11/2010

Table of Contents

Preface

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

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

This book assumes that you have already read and understood the concepts contained in the *Berkeley DB, Java Edition 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 *JE_HOME* directory."

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

```
import com.sleepycat.je.Environment;

...

// Open the environment. Allow it to be created if it does not already
// exist.
Environment myDbEnv;
```

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.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import java.io.File;

...

// Open the environment. Allow it to be created if it does not already
// exist.
Environment myDbEnv;
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
myDbEnv = new Environment(new File("/export/dbEnv"), envConfig);
```

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 replicated JE application:

- Getting Started with Berkeley DB, Java Edition
- Berkeley DB, Java Edition Javadoc
- Berkeley DB, Java Edition Getting Started with Transaction Processing
- Berkeley DB, Java Edition Collections Tutorial

Chapter 1. Introduction

This book provides a thorough introduction to replication as used with Berkeley DB, Java Edition (JE). 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.

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

Overview

Welcome to the JE High Availability (HA) product. JE HA is a replicated, single-master, embedded database engine based on Berkeley DB, Java Edition. JE HA offers important improvements in application availability, as well as offering improved read scalability and performance. JE HA does this by extending the data guarantees offered by a traditional transactional system to processes running on multiple physical hosts.

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

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, some of the reasons why you might choose to implement replication in your JE application are:

- Improved application availability.

  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.

- 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 batch disk I/O so that it is performed as efficiently as possible while still maintaining a degree of durability by *committing to the network*. In other words, you relax your transactional durability guarantees on the machine where you perform the database write, but by virtue of replicating the data across the network you gain some additional durability guarantees beyond what is provided locally.

- 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.

Replication Group Participants

Processes that participate in a JE HA application are generically called *nodes*. Most nodes participate as a read-only Replica. One node in the HA application can perform database writes. This is the Master node.

The sum totality of all the nodes participating in the replicated application is called the *replication group*. While it is only a logical entity (there is no object that you instantiate and destroy which represents the replication group), the replication group is the first-order element of management for a replicated HA application. It is very important to remember that the replication group is persistent in that it exists regardless of whether its member nodes are currently running. In fact, all nodes that have been added to a replication group will remain in the group until they are manually removed from the group by you or your application's administrator.

Replication groups consist of electable nodes and, optionally, Monitor nodes. Electable nodes are replication group participants that can become the group's Master node through a *replication election*. If an electable node is not a Master, then it participates in the replication group as a read-only Replica. A key defining characteristic of an electable node is that it has access to a JE environment.

Note

Beyond Master and Replica, a node can also be in several other states. See Replication Group Life Cycle (page 7) for more information.

Most of the nodes in a replication group are electable nodes. However, it is also possible to have *Monitor nodes*, which are nodes that do not have access to a JE environment. For this reason, they cannot serve as either a Master or a Replica. Instead, they merely monitor the composition of the replication group as changes are made by adding and removing electable nodes, and as elections are held to select a new Master. Monitor nodes are therefore used by applications external to the JE replicated application to route data requests to the various participants of the replication group.

Note that all nodes in a replication group have a unique group-wide name. Further, all replication groups are also assigned a unique name. This is necessary because it is possible for a single process to have access to multiple replication groups. Further, any given collection of hardware can be running multiple replication groups (a production and a test group, for example.) By uniquely identifying the replication group with a unique name, it is possible for JE HA to internally check that nodes have not been misconfigured and so make sure that messages are being routed to the correct location.

Replicated Environments

All electable nodes must have access to a database environment. Further, no electable node can share a database environment with another electable node.

More to the point, in order to create an electable node in a replication group, you use a specialized form of the environment handle: ReplicatedEnvironment.

There is no JE-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.

We discuss ReplicatedEnvironment handle usage in Using Replicated Environments (page 13). For an introduction to database environments, see the *Getting Started with Berkeley DB, Java Edition* guide.

Selecting a Master

Every replication group is allowed one and only one Master. Masters are selected by holding an *election*. All such elections are performed by the underlying Berkeley DB, Java Edition replication code.

When a node joins a replication group, it attempts to locate the Master. If it is the first node added to the replication group, then it automatically becomes the Master. If it is not the first node to startup in the replication group, and it cannot locate the Master, it calls for an election. Further, if at any time the Master becomes unavailable to the replication group, the individual replicas will call for an election.

When holding an election, replicas vote on who should be the Master. Among replicas participating in the election, the node with the most up-to-date set of logs will win the election. In order to win an election, a node must win a simple majority of the votes.

Usually JE requires a majority of electable nodes to be available to hold an election. If a simple majority is not available, then the replication group will no longer be able to accept write requests as there will be no Master.

Note that an electable node is part of the replication group even if it is currently not running or is otherwise unreachable by the rest of the replication group. Membership in the replication group is persistent; once a node joins the group it remains in the group regardless of its current state. The only way a node leaves a replication group is if you manually remove it from the group (see Adding and Removing Nodes from the Group (page 55) for details). This is a very important point to remember when considering elections. *An election cannot be held if the majority of electable nodes in the group are not running or are otherwise unreachable*.

Note

There are two circumstances under which a majority of nodes need not be available in order to hold an election. The first is for the special circumstance of the two-node group. See Configuring Two-Node Groups (page 24) for details.

The second circumstance is if you explicitly relax the requirement for a majority of nodes to be available in order to hold an election. This is a dangerous thing to do, and your replication group should rarely (if ever) be configured this way. See Managing a Failure of the Majority (page 70) for more information.

Once a node has been elected Master, it remains in that role until the replication group has a reason to hold another election. Currently, the only reason why the group will try to elect a new Master is if the current Master becomes unavailable to the group. This can happen because you shutdown the current Master, the current Master crashes due to bugs in your application code, or a network outage causes the current Master to be unreachable by a majority of the nodes in your replication group.

In the event of a tie in the number of votes, JE's underlying implementation of the election code will pick the Master. Moreover, the election code will always make a consistent choice when settling a tie. That is, all things being even, the same node will always be picked to win a tied election.

Replication Streams

Write transactions can only be performed at the Master. The results of these transactions are replicated to Replicas using a logical replication stream.

Logical replication streams are performed over a TCP/IP connection. The stream contains a description of the logical changes (for example, insert, update or delete) operations that were performed on the database as a result of the transaction commit. Each such replicated change is assigned a group-wide unique identifier called a Virtual Log Sequence Number (VLSN). The VLSN can be used to locate the replicated change in the log files associated with any member of the group. Through the use of the VLSN, each operation described by the replication stream can be replayed at each Replica using an efficient internal replay mechanism.

A consequence of this logical replaying of a transaction is that physical characteristics of the log files contained at the Replicas can be different across the replication group. The data contents of the environments found across the replication group, however, should be identical.

Note that there is a process by which a non-replicated environment can be converted such that it has the log structure and metadata required for replication. See Converting Existing Environments for Replication (page 58) for more information.

Managing Data Guarantees

All replicated applications are first transactional applications. This means that you have the standard data guarantee issues to consider, all of which have to do with how durable and consistent you want your data to be. Of course, considerations of this nature also play a role in your application's performance. These issues are even more important for replicated applications because replication adds additional dimensions to them.

Notably, in a replicated application you must decide how durable your data is, by deciding how careful the Master will be to make sure a data write has been written to disk on its various Replica nodes before completing the transaction.

Consistency also adds an additional dimension in a replicated application, because now you must decide how consistent the various nodes in the replication group will be relative to the Master at any given time. If no writes are being performed on the Master, all Replicas will eventually catch up to the Master and so be completely consistent with it. But for most HA applications, writes are occurring on the Master, and so it is possible for some number of your Replicas to lag behind the Master. What you have to decide, then, is how sensitive your application is to this kind of temporary inconsistency.

Note that your consistency requirements can be gated by your durability requirements. Durability, in turn, can be gated by any concerns you might have on write throughput. At the same time, your consistency requirement can have an affect on the read performance of your Replicas. It is therefore a mistake to think about any one of these requirements in the absence of the others.

Durability

One of the reasons you might be writing a replicated application is to achieve a higher durability guarantee than you can get with a traditional transactional application. In a traditional application, your data's durability is a function of how you perform your transactional commits, and how frequently you perform your backups. For this class of application, the strongest durability guarantee you can have is to use synchronous commits (the commit does not complete until the data is written to disk), coupled with very frequent backups of your environment.

The problem with a stand-alone application in which you are seeking a very high durability guarantee is that your write throughput will suffer. Synchronous commits require disk writes, and disk I/O is one of the most expensive operations you can ask a database to perform.

In order to increase write throughput in your transactional application, you may decide to use asynchronous commits that do not require the disk I/O to complete before the transaction commit completes. The problem with this is that your application can potentially crash before a transaction has been completely written to disk. This represents a loss of data, which is to say the data is not durable.

Replication can help with your data durability in a couple of ways. Most importantly, replication allows you to *commit to the network*. This means that when your Master commits a transaction, the results of that commit are sent to one or more nodes available over the network. Consequently, multiple disks, disk controllers, power supplies, and CPUs are used to ensure the data modification makes it to stable storage.

Usually JE makes the commit operation on the Master wait until it receives acknowledgements from some number of Replicas before returning from the operation. However, if you want to increase write throughput, you can configure your Master to proceed without acknowledgements, and so return immediately from the commit operation (once the commit operation has met the local durability requirement). The price that you pay for this is a reduced durability guarantee. How reduced the guarantee is, is a function of the number of nodes in your replication group (the more nodes you have, the higher your durability guarantee is) and the quality and stability of your network.

Alternatively, you can obtain an extremely high durability guarantee by configuring the Master to wait for all Replicas to acknowledge a commit operation before returning from the operation. The price you pay for this very high guarantee is greatly reduced write throughput.

For information on configuring and managing durability guarantees for your replicated application, see Managing Durability (page 28).

Managing Data Consistency

Data consistency means that the data you thought your wrote to your environment is in fact written to your environment. It also means that you will never find partial records written to your environment.

In a replicated application, consistency also means that data which is available on the Master is also available on the Replicas.

A simple transactional application offers consistency guarantees that are enforced when you commit a transaction. Your replicated application also offers this consistency guarantee (because it is also a transactional application). For this reason, the environment on the Master is always absolutely consistent. But beyond that, you need to manage consistency for data across all the nodes in your replication group.

When you commit a transaction on the Master, your Replica nodes may or may not have the data changes performed by that transaction at the end of the commit. Whether they do depends on how high a durability guarantee you implemented for your Master (see the previous section). If, for example, you configured your Master to required acknowledgements from all nodes before returning from the commit, then the data will be consistently available across all the nodes in the replication group. However, if you configured the Master such that no acknowledgements are necessary, then your data is probably not consistent across the replication group.

To ensure that read transactions on the Replicas see a sufficiently consistent view of the environment, you can set a consistency policy for each transaction. This policy describes how current the Replica must be before a transaction can be initiated on it. If the Replica is not current enough, the start of the transaction is delayed until the Replica has caught up.

There are two possible consistency policies. First, there is a time-based policy that describes how far back in time the Replica is allowed to lag behind the Master. Secondly, you can use a commit-based consistency policy that is based on the commit of a specified transaction. This policy is used to ensure the Replica is at least current enough to have the changes made by a specific transaction, and by all transaction committed prior to the specified transaction. The start of a transaction on a Replica can be delayed until the Replica can meet the consistency policy defined for that transaction.

This means that a stringent consistency policy can affect your Replica's read throughput. Transactions, even read-only transactions, cannot begin until the Replica is consistent *enough*. So if you have a Replica that has lagged far behind the Master, and which is having trouble catching up due to network latency or other issues, then read requests may stall, and perhaps even time out, which will affect the latency of your Replica's read read requests, and perhaps even its overall availability for read requests. For this reason, give careful consideration to how well you want your Replica to perform on reads, versus how consistent you want the Replica to be with other nodes in the replication group.

For more information on managing consistency in your replicated application, see Managing Consistency (page 33).

Replication Group Life Cycle

This section describes how your replication group behaves over the course of the application's lifetime. Startup is described, both for new nodes as well as for existing nodes that are restarting. This section also describes Master failover.

Terminology

Before continuing, it is necessary to define some terms used in this document as they relate to node participation in a replication group.

- Add/Remove

  When we say that a node has been *added* to a replication group, this means that it has become a persistent member of the group. Regardless of whether the node is running or otherwise reachable by the group, once it has been added to the group it remains a member of the group. If the node is an electable node, the group size used during elections, or transaction commit acknowledgements, is increased by one.

  A node that has been added to a replication group remains a member of that group until it is explicitly *removed* from the group. Once a node has been removed from the group, it is no longer a member of the group. If the node that was removed was an electable node, the group size used during elections, or transaction commit acknowledgements, is decreased by one.

- Join/Leave

  We say that a member has *joined* the replication group when it starts up and begins participating in the group as an active node. Electable nodes join a replication group by successfully opening a ReplicatedEnvironment handle.

A member, then, *leaves* a replication group by shutting down, or otherwise ceasing to participate in the group as an active node. When operating normally, electable nodes leave a replication group by closing its last ReplicatedEnvironment handle.

Joining or leaving a group does not change the group size, and so the number of nodes required to hold an election, as well as the number of nodes required to acknowledge transaction commits, does not change.

Node States

Member nodes can be in the following states:

- Master

  When in the Master state, a member node can service read and write requests. At any given time, there can be only one node in the Master state in the replication group.

- Replica

  Member nodes in the Replica state can only service read requests. The majority of nodes in the replication group should be in the Replica state.

- Unknown

  The member node is not aware of a Master and is actively trying to discover or elect a Master. A node in this state is constantly striving to transition to the more productive Master or Replica state.

  A node in the Unknown state can still process read transactions if the node can satisfy its transaction consistency requirements.

- Detached

  The member node has been shutdown (that is, it has left the group, but it has not been removed from the group — see the previous section). It is still a member of the replication group, but is not an active participant.

Note that from time to time this documentation uses the term *active node*. An active node is a member node that is in the Master, Replica or Unknown state. More to the point, an active node is a node that is available to participate in elections.

New Replication Group Startup

The first time you start up a replication group, the group exists (for at least a small time) as a group of size one. At this time, the single node belonging to the group becomes the Master. So long as there is only one node in the replication group, that one node acts behaves as if it is a non-replicated application. There are some differences in the format of the log file that the application maintains, but it otherwise behaves identically to a non-replicated transactional application.

Subsequently, upon startup a new node must be given the contact information for at least one currently active node in the replication group in order for it to be added to the group. The new node contacts this active node who will identify the Master for the new node.

Note

As is the case with elections, a node cannot be added to the replication group unless a simple majority of nodes are active at the time that it starts up. If too many nodes are down or otherwise unavailable, you cannot add a new node to the group.

The new node then contacts the Master, and provides all necessary identification information about itself to the Master. This includes host and port information, the node's unique name, and the replication group name. The Master stores this identifying information about the node. Because this information is stored persistently, the effective size of the replication group has just grown by one.

Note

Note that the node is now a permanent member of the replication group until you manually remove it. This is true even if you shutdown the node for a long time. See Adding and Removing Nodes from the Group (page 55) for details.

Once the new node is an established member of the group, the Master provides the Replica with the logical logs needed to replicate the environment. The sequence of logical log records sent from the Master to the Replica constitutes the *Replication Stream*. At this time, the node is said to have *joined* the group. Once a replication stream is established, it is maintained until either the Replica or the Master goes down.

Subsequent Startups

Each node stores information about other replication group members in its replicated environment so that this information is available to it upon restart.

When a node that is already an established member of a replication group is restarted, the node uses its knowledge of other members of the replication group to locate the Master. It does this by by querying the members of the group to locate the current Master. If it finds a Master, the node joins the group and proceeds to participate in the group as a Replica.

If a Master is not available, the restarting node initiates an election so as to establish one. If a simple majority of nodes are available for the election, a Master is elected. If the restarting node is elected Master, it then waits for Replicas to connect to it so that it can supply them a replication stream.

Replica Startup

Regardless of how it happens, when a node joins a replication group, it contacts the Master and then goes through the following three steps:

1. Handshake

The Replica sends the Master its configuration information, along with the unique name associated with the Replica's environment. This name is a pseudo-randomly generated Universal Unique Identifier (UUID).

This handshake establishes the node as a valid member of the group. It is used both by new nodes joining the group for the first time, and by existing nodes that are simply restarting.

In addition, during this handshake process, the Master and Replica nodes will compare their clocks. If the clocks are too far off from one another, the handshake will fail and the Replica node will fail to start up. See Time Synchronization (page 24) for more information.

2. Replication Stream Sync-Up

   The Replica sends the Master its current position in the replication stream sequence. The Master and Replica then negotiate a point in the replication stream that the Master can use as a starting point to resume the flow of logical records to the Replica.

   Note that normally this sync-up process will be transparent to your application. However, in rare cases the sync-up may require that committed transactions be undone.

   Also, if the Replica has been offline for a long time, it is possible that the Master can no longer supply the Replica with the required contiguous interval of the replication stream. (This can happen due to log cleaning on the Master.) In this case, the log files must be copied to the restarting node from some other up-to-date node in the replication group. See Restoring Log Files (page 57) for details.

3. Steady state replication stream flow

   Once the Replica has successfully started up and joined the group, the Master maintains a flow of log records to the Replica. Beyond that, the Master will request acknowledgements from the Replica whenever the Master needs to meet transaction commit durability requirements.

Master Failover

A Master failing or shutting down causes all of the replication streams between the Master and its various Replicas to terminate. In reaction, the Replicas transition to the Unknown state and initiate an election.

An election can be held if at least a simple majority of the replication group's nodes are active. The node that wins the election transitions to the Master state, and all other active nodes transition to the Replica state.

Upon transitioning to the Replica state, nodes connect to the new Master and proceed through the handshake, sync-up, replication replay process described in the previous section.

If no Master can be elected (because a majority of nodes are not available to participate in the election), then the nodes remain in the Unknown state until such a time as a Master

can be elected. In this state, the nodes might be able to service read-only requests, but the replication group is incapable of servicing write requests. Read requests can be serviced so long as the transaction's consistency requirements can be met (see Managing Consistency (page 33)).

Note that the JE Replication application needs to make provisions for the following state transitions after failover:

- A node that transitions from the Replica state to the Master state as a result of a failover needs to start accepting update requests. There are several ways to determine whether a node can handle update requests. See Managing Write Requests at a Replica (page 21) for more information.
- If a node remains in the Replica state after a failover, the failover should be transparent to the application. However, an application may need to take corrective action in the rare situation where the sync-up process has to roll back committed transactions.

  See Managing Transaction Rollbacks (page 43) for an example of how handle a transaction commit roll back.

Two Node Groups

Replication groups comprised of just two nodes represents a unique corner case for JE replication. In order to elect a master, usually a simple majority of nodes must be available to participate in an election. However, for replication groups of size two, if even one node is unavailable for the election then by default it is impossible to hold an election.

However, for some classes of application, it is desirable for the application to proceed operations with just one node. That is, the application trades off the durability guarantees offered by using two nodes for the higher availability permissible by allowing the application to run with just one node.

JE allows you to do this by designating one of the nodes in a two-node group as a *primary node*. The other node in the group is then, implicitly, the secondary node. When the secondary node is not available, the number of nodes required for a simply majority is reduced from two to one by the primary node. Consequently, the primary node is able to elect itself as the Master. It can then commit transactions that require a simple majority to acknowledge commits. When the secondary becomes available again, the number of nodes required for a simple majority at the primary once again reverts to two.

At any given time, there must be either zero or one nodes designated as the primary node, but it is up to your application to make sure both nodes are not erroneously designated as the primary. Your application must be very careful not to mistakenly designate two nodes as the primary. If this happened, and the two nodes could not communicate with one another (due to a network malfunction of some kind, for example), they could both then consider themselves to be Masters and start accepting write requests. This violates a fundamental requirement that at any given instant in time, there should be exactly one node that is permitted to perform writes on the replicated environment.

Note that the secondary always needs two nodes for a simple majority, so it can never become the Master in the absence of the primary node. If the primary node fails, you can make

provisions to swap the primary and secondary designations so that the surviving node is now the primary. This swap must be performed carefully so as to ensure that both nodes are not concurrently designated the primary. The most important thing is that the failed node comes up as the secondary after it has been repaired.

For more information on using two-node groups, see Configuring Two-Node Groups (page 24).

Chapter 2. Replication API First Steps

From an API point of view, there are two basic requirements that every replicated application must meet:

1. It must be a transactional application.

2. It must use a specific form of the Environment handle, which you get by using the ReplicatedEnvironment class.

Beyond that, there are some additional requirements in terms of exception handling that your application should perform.

The transactional nature of your replicated application is described in Transaction Management (page 27). This chapter discusses replicated environments and the exceptions unique to exceptions in detail.

Using Replicated Environments

Every replication node manages a single replicated JE environment directory. The environment follows the usual regulations governing a JE environment; namely, only a single read/write process can access the environment at a single point in time.

Usually this requirement is met naturally, because usually each node in a replicated application is also operating on a machine that is independent of all the other nodes. However, in some test and development scenarios, this one node to one machine rule might not be met, so the bottom line is that you need to make sure that no two processes are ever attempting to manage the same environment.

Note

An application can access a replicated JE environment directory using a read only Environment handle. The usual semantics of read only non-replicated Environment handles apply in this case. That is, the application can view a snapshot of the replicated environment as of the time the Environment handle was opened, through the Environment handle. An application can therefore open a ReplicatedEnvironment handle in one process, and concurrently open read only Environment handles in other processes. Any changes subsequently made to the replicated environment, either by virtue of the node being a Master, or due to a replay of the replication stream (if the node is a Replica), are not accessible through the read only Environment handles until they are closed and reopened.

Normally you manage your JE environments using the Environment class. However, to provide for the underlying infrastructure needed to implement replication, your JE HA application must instead use the ReplicatedEnvironment class, which is a subclass of Environment. Its constructor accepts the normal environment configuration properties using the EnvironmentConfig class, just as you would normally configure an Environment object. However, the ReplicatedEnvironment class also accepts an ReplicationConfig class object, which allows you to manage the properties specific to replication.

The following is an example of how you instantiate a ReplicatedEnvironment object. Note that there are some differences in how this is used, depending on whether you are starting a brand-new node or you are restarting an existing node. We discuss these differences in the next section.

For a general description of environments and environment configuration, see the *Getting Started with Berkeley DB Java Edition* guide.

```
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
envConfig.setTransactional(true);

// Identify the node
ReplicationConfig repConfig = new ReplicationConfig();
repConfig.setGroupName("PlanetaryRepGroup");
repConfig.setNodeName("Mercury");
repConfig.setNodeHostPort("mercury.acme.com:5001");

// This is the first node, so its helper is itself
repConfig.setHelperHosts("mercury.acme.com:5001");

ReplicatedEnvironment repEnv =
     new ReplicatedEnvironment(envHome, repConfig, envConfig);
```

Configuring Replicated Environments

You configure a JE ReplicatedEnvironment handle using two different configuration classes: EnvironmentConfig and ReplicationConfig. Your usage of EnvironmentConfig is no different than if you were writing a non-replicated application, so we will not describe its usage here. For an introduction to basic environment configuration, see the *Getting Started with Berkeley DB, Java Edition* guide.

The ReplicationConfig class allows you to configure properties that are specific to replicated applications. Some of these properties are important in terms of how how your application will behave and how well it will perform. These properties are discussed in details later in this book.

To an extent, you can get away with ignoring most of the configuration properties until you are ready to tune your application's performance and behavior. However, no matter what, there are four properties you must always configure for a ReplicatedEnvironment before opening it. They are:

1. Group Name

   The group name is a string that uniquely identifies the group to which the node belongs. This name must be unique. It is possible to operate multiple replication groups on the same network. In fact, a single process can even interact with multiple replication groups, so long as it maintains separate replicated environments for each group in which it is participating.

   By using unique group names, the JE replication code can make sure that messages arriving at a given client are actually meant for that client.

You set the group name by using the ReplicationConfig.setGroupName() method. Note that if you do not set a group name, then the default GROUP_NAME value is used.

2. Node Name

   This name must be unique to the replication group. This name plus the node name uniquely identifies a node in your enterprise.

   You set the node name by using the ReplicationConfig.setNodeName() method.

3. Host

   The host property identifies the network name and port where this node can be reached. Other nodes in the replication group will use this host/port pair to establish a TCP/IP connection to this node. This connection is used to transfer data between machines, hold elections, and monitor the status of the replication group.

   You provide the host and port information using a string of the form:

   ```
   host:[port]
   ```

   The port that you provide must be higher than 1023.

   You set the host information by using the ReplicationConfig.setNodeHostPort() method. Note that if you do not set a node host, then the default NODE_HOST_PORT value is used.

4. Helper Host

   The helper host or hosts are used by a node the very first time it starts up to find the Master. Basically, this string should provide one or more host/port pairs for nodes who should know where the Master is.

   One of the nodes that you provide on this string can be the current Master, but that is not required. All that matters is that the hosts identified here can tell a new node where the current Master is.

   If the brand new node cannot find a Master, it will initiate an election. If no other nodes are available to the new node, then it will elect itself as Master. If the current node is truly the very first node starting up in the replication group, then self-electing itself to be the Master is probably what you want it to do.

   However, if the current node *is not* the very first node starting up in the replication group, then a misconfiguration of this property can cause you to end up with multiple replication groups, each with the same group name. This represents an error situation, one that can be very difficult to diagnose by people who are inexperience with managing replication groups. For this reason, it is very important to make sure the hosts identified on this string do NOT identify the local host.

   On subsequent start ups after the very first startup, the node should be able to locate other participants in the replication group using information located in its own database. In that case, the information provided on this string is largely ignored unless the current

node has been down or otherwise out of communication with the rest of the group for so long that its locally cached information has grown stale. In this case, the node will attempt to use the information provided here to locate the current Master.

You set the helper host information by using the ReplicationConfig.setHelperHosts() method.

When configuring and instantiating a ReplicatedEnvironment object, you should usually configure the environment so that a helper host other than the local machine is used:

```
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
envConfig.setTransactional(true);

// Identify the node
ReplicationConfig repConfig = new ReplicationConfig();
repConfig.setGroupName("PlanetaryRepGroup");
repConfig.setNodeName("Jupiter");
repConfig.setNodeHostPort("jupiter.acme.com:5002");

// Use the node at mercury.acme.com:5001 as a helper to find the rest
// of the group.
repConfig.setHelperHosts("mercury.acme.com:5001");

ReplicatedEnvironment repEnv =
   new ReplicatedEnvironment(envHome, repConfig, envConfig);
```

Note that if you are restarting a node that has already been added to the replication group, then you do not have to supply a helper host at all. This is because the node will already have locally stored host and port information about the other nodes in the group.

```
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
envConfig.setTransactional(true);

// Identify the node
ReplicationConfig repConfig =
    new ReplicationConfig("PlanetaryRepGroup",
                          "Jupiter",
                          "jupiter.acme.com:5002");

ReplicatedEnvironment repEnv =
   new ReplicatedEnvironment(envHome, repConfig, envConfig);
```

However, if you are starting the very first node in the replication group for the very first time, then there is no other helper host that the node can use to locate a Master. In this case, identify the current node as the helper host, and it will then go ahead and become a replication group of size 1 with itself as a Master.

Note

Do this ONLY if you are truly starting the very first node in a replication group for the very first time.

```
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
envConfig.setTransactional(true);

// Identify the node
ReplicationConfig repConfig =
    new ReplicationConfig("PlanetaryRepGroup",
                          "Jupiter",
                          "jupiter.acme.com:5002");

// This is the first node, so the helper is itself.
repConfig.setHelperHosts("jupiter.acme.com:5002");

ReplicatedEnvironment repEnv =
   new ReplicatedEnvironment(envHome, repConfig, envConfig);
```

HA Exceptions

JE HA requires you to manage more error situations that you would have to if you were writing a non-replicated application. These error situations translate to additional exceptions that you must contend with in your code. Before continuing with our description of how to write a replicated application, it is useful to review the HA-specific exceptions that your application must manage.

Master-Specific HA Exceptions

There are two exceptions that you can see on a Master node, and which you will not see anywhere else. They are:

- InsufficientReplicasException

  This exception can be raised on a transaction begin or commit. It means that the Master cannot successfully commit a transaction, or begin one, because it is not in contact with enough Replicas. The number of Replicas required to successfully commit the transaction is a function of the durability policy that you have set for the transaction. See Managing Durability (page 28) for more information.

  If raised on a transaction commit operation, this exception means that the transaction has not been committed. Instead, it has been marked as invalid. In response to this exception, your application must at a minimum abort the transaction. It is up to you whether you want to retry the transaction at some later time when more Replicas are in contact with the Master.

  If raised on a transaction begin operation, this exception means that the transaction has not begun. If the application intended to initiate a read-only transaction on a Master,

it can avoid this exception by ensuring that the transaction is configured to not require any acknowledgments. For information on configuring acknowledgments, see Managing Acknowledgements (page 30).

- InsufficientAcksException

  This exception can be raised on a transaction commit. It means that the Master has successfully committed the transaction locally, but it has not received enough acknowledgements from its Replicas in the timeframe allocated for acknowledgements to be received.

  The application should respond to this exception in such a way as to alert the administrator that there might be a problem with the health of the network or the nodes participating in the replication group.

  For information on how to manage acknowledgement policies, see Managing Acknowledgements (page 30).

Replica-Specific HA Exceptions

The exceptions that you can see on a Replica, and nowhere else, are:

- ReplicaConsistencyException

  Indicates that the Replica was unable to meet the defined consistency requirements in the allocated period of time.

  If this exception is encountered frequently, it indicates that the consistency policy requirements are too strict and cannot be met routinely given the load being placed on the system and the hardware resources that are available to service the load. The exception may also indicate that there is a network related issue that is preventing the Replica from communicating with the Master and keeping up with the replication stream.

  In response to this exception, your application can either attempt to retry the transaction, or you can relax your application's consistency requirements until the transaction can successfully complete.

  For information on managing consistency policies, see Managing Consistency (page 33).

- ReplicaWriteException

  An attempt was made to perform a write operation on a Replica. The exception typically indicates an error in the application logic. In some extremely rare cases it could be the result of a transition of the node from Master to Replica, while a transaction was in progress.

  The application must abort the current transaction and redirect all subsequent update operations to the Master. For example code that performs this action, see Example Run Transaction Class (page 44).

- LockPreemptedException

A read lock currently held by a Replica has been preempted by an HA write operation. The Replica should abort and retry the read operation in response to this exception.

Note that your application should attempt to catch the LockConflictException base class rather than this class because all of the locking exceptions are managed in the same way (abort and retry the transaction).

- DatabasePreemptedException

  The database handle on a Replica was forcibly closed due to the replay of an Environment.truncateDatabase(), Environment.removeDatabase() or Environment.renameDatabase() operation in the replication stream.

  When this exception occurs, the application must close any open Cursors and abort any open Transactions that are using the database, and then close the Database handle. If the application wishes, it may reopen the database if it still exists.

- RollbackException

  A new master has been selected, this Replica's log is ahead of the current Master, but the Replica was unable to rollback without a recovery. As a consequence, one or more of the most recently committed transactions may need to be rolled back, before the Replica can synchronize its state with that of the current Master. This exception can happen if the Replica with the most recent log files was unable to participate in the election of the Master, perhaps because the node had been shut down.

  For details on how to handle this exception, see Managing Transaction Rollbacks (page 43).

- InsufficientLogException

  Indicates that the log files constituting the Environment are insufficient and cannot be used as the basis for continuing with the replication stream provided by the current master.

  This exception generally means that the node has been down for a long enough time that it can not be brought up-to-date by the Master. For information on how to respond to this condition, see Restoring Log Files (page 57).

Replicated Environment Handle-Specific Exceptions

In addition to Master- and Replica-specific exceptions, it is possible for a ReplicatedEnvironment handle to throw an UnknownMasterException. This exception indicates that the operation being tried requires communication with a Master, but the Master is not available.

This exception typically indicates that there is a problem with your physical infrastructure. It might mean that an insufficient number of nodes are available to elect a Master, or that the current node is unable to communicate with other nodes due to, for example, network problems.

In response to this exception, your application can try any number of corrective actions, from immediately retrying the operation, to logging the problem and then abandoning the operation, to waiting some predetermined period of time before attempting the operation again. Your application can also use the Monitor or the StateChangeListener to be notified when a Master becomes available. For more information see Writing Monitor Nodes (page 61) or Using the StateChangeListener (page 22).

Opening a Replicated Environment

In the previous two sections we looked at the basics of how to create a replicated environment, and what exceptions you can expect to see in a JE HA application. Now we need to combine these two topics in order to examine how you should open a ReplicatedEnvironment handle to an existing replicated environment.

When you open the handle, the underlying HA code will attempt to open a TCP/IP connection to other nodes in the replication group, based on the node's stored replication group metadata or the helper host information that you provide. In doing so, the node will attempt to locate a Master or, failing that, will hold an election in order to select a new Master.

Due to issues of timing and network performance, the node may or may not be able to:

1. locate the master; and
2. hold an election.

This can happen if there simply are not enough nodes available in order for the current node to start up, find the current master, or hold an election. Remember that a majority of the nodes registered in the replication group must be available in order to hold an election.

If this situation occurs, the ReplicatedEnvironment constructor will throw an UnknownMasterException. Therefore, typically, it is best that you prepare for this situation by performing the handle creation in a retry loop, as shown in the following code snippet.

In addition, if the Replica has been down for a long enough period of time, it might be so far out of date that it cannot be brought up to date using the normal replication stream. In this case, the ReplicatedEnvironment constructor will throw an InsufficientLogException. See Restoring Log Files (page 57) for information on how to handle this exception.

```
private static int REP_HANDLE_RETRY_MAX = 100;

   ...

ReplicatedEnvironment getEnvironment(File envHome, String groupName,
                                     String nodeName, String nodeHost,
                                     String helperHosts)
     throws IllegalStateException, InterruptedException {

   EnvironmentConfig envConfig = new EnvironmentConfig();
   envConfig.setAllowCreate(true);
   envConfig.setTransactional(true);

   // Identify the node
```

```
    ReplicationConfig repConfig =
         new ReplicationConfig();
    repConfig.setGroupName(groupName);
    repConfig.setNodeName(nodeName);
    repConfig.setNodeHostPort(nodeHost);
    repConfig.setHelperHosts(helperHosts);

    for (int i = 0; i < REP_HANDLE_RETRY_MAX; i++) {
         try {
             return new
                 ReplicatedEnvironment(envHome, repConfig, envConfig);
         } catch (UnknownMasterException ume) {
             /*
              * Insert application specific code here to indicate that
              * this problem was encountered, such as writing the
              * condition to a log file.
              */

             Thread.sleep(5 * 1000);
             continue;
         } catch (InsufficientLogException ile) {
             /* A network restore is required, make the necessary calls */
         }

    }
    throw new
         IllegalStateException("getEnvironment: reached max retries");
}
```

Note that for production code, you may want to retry the handle open without any maximum retry limit.

Managing Write Requests at a Replica

For a replicated JE application, read requests can be serviced by any electable node in the replication group, but write requests can only be serviced by the Master node. For this reason, your application must be prepared to deal with the difference in operating behavior between read-only Replicas and read-write Masters.

It is possible to be quite sophisticated in terms of tracking which node is the Master and so which node can service write requests. You can even route write requests to the Master node by writing a special router process. For an example of an application that does this, see RouterDrivenStockQuotes and HARouter, both of which are available in your JE distribution in the `<JE HOME>/examples/je/rep/quote` directory.

However, for our purposes here, we simply want to make sure our Replica nodes can gracefully handle a situation where they receive a write request. The write request should be rejected by the node, with some notification being returned to the requester that the write activity is rejected. While not the most robust solution, this is the simplest thing your JE replicated application can do if it receives a write request at a Replica node.

There are two ways to determine whether a write request can be handled at the local node:

- Use a monitor node to implement request routing. Monitor nodes are described in Writing Monitor Nodes (page 61).
- Use the StateChangeListener to detect when the local node becomes a Master. Otherwise, forward the write request to the Master node instead of attempting to service it locally.

Either way, any code that attempts database writes for an HA application should always be prepared to handle a ReplicaWriteException.

Using the StateChangeListener

You use the StateChangeListener interface to implement a class that is capable of notifying your node when it has changed state. In this way, you can track whether a node is in the Master, Replica or Unknown state, and so know whether the node is capable of handling write requests.

To do this, you must implement StateChangeListener.stateChange(), which receives a StateChangeEvent object whenever it is called.

If the node is not in the Master state, then the node can either reject write requests outright or, more usefully, forward write requests to the Master. For an example of an HA application that forwards write requests and uses the StateChangeListener, see the UpdateForwardingStockQuotes example.

Alternatively, you can write a router based on an HA Monitor. See Writing Monitor Nodes (page 61) for more information.

Briefly, you can implement StateChangeListener as follows. Notice that this partial implementation relies on StateChangeEvent.getState() to determine the state that the node has just transitioned to. It then uses StateChangeEvent.getMasterNodeName() to determine where write requests should be forwarded to in the event that the new state is not MASTER.

```
private class Listener implements StateChangeListener {

    private String currentMaster = null;

    public void stateChange(StateChangeEvent se)
        throws RuntimeException {

        switch (stateChangeEvent.getState()) {

            case MASTER:
                // Do whatever your code needs you to do when the
                // current node is the MASTER.  For example,
                // set a flag to indicate that the local node
                // is in the MASTER state. Here, we just fall
                // through and do the same thing as if we
                // transitioned to the REPLICA state.
```

```
            case REPLICA:
                // Again, do whatever your code needs done when
                // a node is in the REPLICA state. At a minimum,
                // you should probably capture which node is the
                // current Master.
                currentMaster = se.getMasterNodeName();
                break;

            // We get here if we have transitioned to the UNKNOWN
            // state.
            default:
                currentmasterName = null;
                break;
        }
    }

    public String getCurrentMasterName() {
        return currentMaster;
    }
}
```

In order to make use of the new listener, the application must call ReplicatedEnvironment.setStateChangeListener(). Note that this method can be called at any time after the ReplicatedEnvironment handle has been created. Also, the listener is set per environment, not per handle. So if you set different listeners for different ReplicatedEnvironment handles, the last listener configured is used environment-wide.

```
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    envConfig.setTransactional(true);

    // Identify the node
    ReplicationConfig repConfig = new ReplicationConfig();
    repConfig.setGroupName("PlanetaryRepGroup");
    repConfig.setNodeName("Saturn");
    repConfig.setNodeHostPort("saturn.acme.com:5001");

    // Use the node at mars.acme.com:5002 as a helper to find
    // the rest of the group.
    repConfig.setHelperHosts("mars.acme.com:5002");

    ReplicatedEnvironment repEnv =
         new ReplicatedEnvironment(home, repConfig, envConfig);
    StateChangeListener listener = new Listener();
    repEnv.setStateChangeListener(listener);
```

Catching ReplicaWriteException

If you perform a Database write operation on a node that is not in the Master state, a ReplicaWriteException is thrown when you attempt to commit the transaction. Therefore,

whenever performing database write operations in an HA application, you should catch and handle ReplicaWriteException.

For example:

```
Transaction txn = null;
try {
    txn = env.beginTransaction(null, null);
    /*
     * Perform your write operations under the protection
     * of the transaction handle here.
     */
    txn.commit();
} catch (ReplicaWriteException replicaWrite) {
    /*
     * Perform whatever reporting (logging) activies you want
     * to do in order to acknowledge that the write operation(s)
     * failed. Then abort the transaction.
     */

     if (txn != null) {
        txn.abort();
     }
}
```

Time Synchronization

For best results, you should synchronize the clocks used by all machines in a replication group. If you are using a time-based consistency policy, this is an absolute requirement (see Time Consistency Policies (page 35) for more information). Time synchronization is easily achieved using a mechanism like NTPD.

In addition, time synchronization is also a requirement for internal JE HA bookkeeping. For example, JE checks for clock skew between the Master and a Replica when the Replica performs its startup handshake with the Master. This handshake will abort and throw EnvironmentFailureException if the clock skew between the two machines is greater than the value set for the MAX_CLOCK_DELTA property. This property can be set using the ReplicationConfig.setMaxClockDelta() method, or in the JE configuration file using the `je.rep.maxClockDelta` property.

Finally, well synchronized clocks make it easier to correlate events in the logging output from different nodes in the group.

Configuring Two-Node Groups

A group needs at least a simple majority of active nodes in order to elect a Master. This means that for a replication group of size two, the failure of a single node means that the group as a whole is no longer available. In some cases, it may be desirable for the application to proceed anyway. If you are using a two-node group, and you decide you want your application to continue even if one of the nodes is unavailable, then you can trade off some of your

durability guarantees, as well as potentially some of your performance, in exchange for a higher availability guarantee.

JE HA can explicitly relax the requirement for a simple majority of nodes. This is only possible when the replication group size is two. The application does this by designating one of the nodes as a Primary node. The other node in the group is implicitly the Secondary node.

At any given instant in time, exactly one of the two nodes can be designated as the Primary. The application is responsible for ensuring that this is the case.

When the Secondary node is not available, the number of nodes required for a simple majority is reduced to one. As a consequence, the Primary is able to elect itself as the Master and then commit transactions that require a simple majority to commit. The Primary is said to be *active* when it is operating in this state. The transition from a designated Primary to an active Primary happens when the Primary needs to contact the secondary node, but fails to do so for one of the following reasons:

- An election is initiated by the Primary to determine a new Master. This might happen because the Primary is just starting up, or because the Primary has lost contact with the Secondary. If either case, if the election fails to establish a Master, the Primary is activated and it becomes the Master.

  Note that the Primary will attempt to locate a Master until it has hit the retry limit as defined by the ELECTIONS_PRIMARY_RETRIES configuration property. But until the Primary has reached that limit, it will not transition to the active state.

- An Environment.beginTransaction() operation is invoked on the Primary while it is in the Master state, and it cannot establish contact with the Secondary in the time period specified by the INSUFFICIENT_REPLICAS_TIMEOUT configuration property.

- A Transaction.commit() needing a commit acknowledgement is invoked on the Primary while it is in the Master state, and the Primary does not receive the commit acknowledgement within the time period specified by the REPLICA_ACK_TIMEOUT configuration property.

Both the INSUFFICIENT_REPLICAS_TIMEOUT and REPLICA_ACK_TIMEOUT error cases are driven by the durability policy that you are using for your transactions. See Managing Durability (page 28) for more information.

The three properties described above: ELECTIONS_PRIMARY_RETRIES, INSUFFICIENT_REPLICAS_TIMEOUT and REPLICA_ACK_TIMEOUT impact the time taken by the Primary to become active in the absence of the Secondary. Choosing smaller values for the timeouts and election retries will generally result in smaller service disruptions by activating the Primary more rapidly. The downside is that transient network glitches may result in unnecessary transitions to the active state where the Primary is operating with reduced Durability. It's up to the application to make these tradeoffs appropriately based on its operating environment.

When the Secondary becomes available again, the Primary becomes aware of it as part of the Master/Replica handshake (see Replica Startup (page 9)). At that time, the number of nodes required for a simple majority reverts to two. That is, the Primary is no longer in the active state.

Your application must be very careful to not designate two nodes as Primaries. If both nodes are designated as Primaries, and the two nodes cannot communicated with one another for some reason, they could both consider themselves to be Masters and start accepting write transactions. This would violate a fundamental requirement of JE HA that at any given instant in time, there is only one node that is permitted to write to the replicated environment.

The Secondary always needs two nodes for a simple majority, and as a result can never become a Master in the absence of the Primary. If the Primary node fails, you can make provisions to swap the Primary and Secondary designations, so that the surviving node is now the Primary. The swap must be done carefully to ensure that both nodes are not concurrently designated Primaries. In particular, the failed node must come up as a Secondary after it has been repaired.

You designate a node as Primary using the mutable config property DESIGNATED_PRIMARY. You set this property using ReplicationMutableConfig.setDesignatedPrimary(). This property is ignored for groups of size greater than two.

As stated above, this configuration can only be set for one node at a time. This condition is checked during the Master/Replica startup handshake, and if both are designated as Primary then an EnvironmentFailureException is thrown. However, you should not rely on this handshake process to guard against dual Primaries. As stated above, if both nodes are designated Primary at some point after the handshake occurs, and your application experiences a network partition event such that the two nodes can no longer communicate, then both nodes will become Masters. This is error condition that will require you to lose data on at least one of the nodes if writes have occurred on both nodes while the network partition was in progress.

Chapter 3. Transaction Management

A JE HA application is essentially a transactional application that distributes its data across multiple environments for you. The assumption is that these environments are on separate physical hosts, so the distribution of your data is performed over TCP/IP connections.

Because of this distribution activity, several new dimensions are added to your transactional management. In particular, there is more to consider in the areas of durability, consistency and performance than you have to think about for single-environment applications.

Before continuing, some definitions are in order:

1. *Durability* is defined by how likely your data will continue to exist in the presence of hardware breakage or a software crash. The first goal of any durability scheme is to get your data stored onto physical media. After that, to make your data even more durable, you will usually start to consider your backup schemes.

   By its very nature, a JE HA application is offering you more data durability than does a traditional transactional application. This is because your HA application is distributing your data across multiple environments (which we assume are on multiple physical machines), which means that data backups are built into the application. The more backups, the more durable your application is.

2. *Consistency* is defined by how *current* your data is. In a traditional transactional application, consistency is guaranteed by allowing you to group multiple read and write operations in a single atomic unit, which is defined by the transactional handle. This level of consistency continues to exist for your HA application, but in addition you must concern yourself with how consistent (or correct) the data is across the various nodes in the replication group.

   Because the replication group is a collection of differing machines connected by a network, some amount of a delay in data updates is to be naturally expected across the Replicas. The amount of delay that you will see is determined by the number and size of the data updates, the performance of your network, the performance of the hardware on which your nodes are running, and whether your nodes are persistently available on the network (as opposed to being down or offline or otherwise not on the network for some period of time).

   A highly consistent HA application, then, is an application where the data across all nodes in the replication group is identical or very nearly identical all the time. A not very consistent HA application is one where data across the replication group is frequently stale or out of date relative to the data contained on the Master node.

3. *Performance* is simply how fast your HA application is at performing read and write requests. By its very nature, an HA application tends to perform much better than a traditional transactional application at read-only requests. This is because you have multiple machines that are available to service read-only requests. The only tricky thing here is to make sure you load balance your read requests appropriately across all your nodes so that you do not have some nodes that are swamped with requests while others are mostly idle.

Write performance for an HA application is a mixed bag. Depending on your goals, you can make the HA application perform better than a traditional transactional application that is committing writes to the disk synchronously. However, in doing so you will compromise your data's durability and consistency guarantees. This is no different than configuring a traditional transactional application to commit transactions asynchronously to disk, and so lose the guarantee that the write is stored on physical media before the transaction completes. However, the good news is that because of the distributed nature of the HA application, you have a better durability guarantee than the asynchronously committing single-environment transactional application. That is, by "committing to the network" you have a fairly good chance of a write making it to disk somewhere on some node.

Mostly, though, HA applications commit a transaction and then wait for an acknowledgement from some number of nodes before the transaction is complete. An HA application running with quorum acknowledgements and write no sync durability can exhibit equal or better write performance than a single node standalone application, but your write performance will ultimately depend on your application's configuration.

As you design your HA application, remember that each of these characteristics are interdependent. You cannot, for example, configure your application to have extremely high durability without sacrificing some amount of performance. A highly consistent application may have to make sacrifices in durability. A high performance HA application may require you to make trade-offs in both durability and consistency.

Managing Durability

A highly durable application is one where you attempt to make sure you do not lose data, ever. This is frequently (but not always) one of the most pressing design considerations for any application that manages data. After all, data often equals money because the data you are managing could involve billing or inventory information. But even if your application is not managing information that directly relates to money, a loss of data may very well cost your enterprise money in terms of the time and resources necessary to reacquire the information.

HA applications attempt to increase their data durability guarantees by distributing data writes across multiple physical machines on the network. By spreading the data in this way, you are placing it on stable storage on multiple physical hard drives, CPUs and power supplies. Obviously, the more physical resources available to contain your data, the more durable it is.

However, as you increase your data durability, you will probably lower your consistency guarantees and probably your write performance. Read performance may also take a hit, depending on how many physical machines you include in the mix and how high a durability guarantee you want. In order to understand why, you have to understand how JE HA applications handle transactional commits.

Durability Controls

By default, JE HA makes transactional commit operations on the Master wait to return from the operation until they receive acknowledgements from some number of Replicas. Each Replica, in turn, will only return an acknowledgement once the write operation has met whatever durability requirement exists for the Replica. (For example, you can

require the Replicas to successfully flush the write operation to disk before returning an acknowledgement to the Master.)

Note

Be aware that write operations received on the Replica from the Master have lock priority. This means that if the Replica is currently servicing a read request, it might have to retry the read operation should a write from the Master preempt the read lock. For this reason, you can see read performance degradation if you have Replicas that are heavily loaded with read requests at a time when the Master is performing a lot of write activity. The solution to this is to add additional nodes to your replication group and/or better load-balance your read requests across the Replicas.

There are three things to control when you design your durability guarantee:

- Whether the Master synchronously writes the transaction to disk. This is no different from the durability consideration that you have for a stand-alone transactional application.
- Whether the Replica synchronously writes the transaction to disk before returning an acknowledgement to the Master, if any.
- How many, if any, Replicas must acknowledge the transaction commit before the commit operation on the Master can complete.

You can configure your durability policy on a transaction-by-transaction basis using TransactionConfig.setDurability(), or on an environment-wide basis using EnvironmentMutableConfig.setDurability().

Commit File Synchronization

Synchronization policies are described in the *Berkeley DB, Java Edition Getting Started with Transaction Processing* guide. However, for the sake of completeness, we briefly cover this topic here again.

You define your commit synchronization policy by using a Durability class object. For HA applications, the Durability class constructor must define the synchronization policy for both the Master and the Master's replicas. The synchronization policy does not have to be the same for both Master and Replica.

You can use the following constants to define a synchronization policy:

- Durability.SyncPolicy.SYNC

  Write and synchronously flush the log to disk upon transaction commit. This offers the most durable transaction configuration because the commit operation will not return until all of the disk I/O is complete. But, conversely, this offers the worse possible write performance because disk I/O is an expensive and time-consuming operation.

- Durability.SyncPolicy.NO_SYNC

  Do not synchronously flush the log on transaction commit. All of the transaction's write activity is held entirely in memory when the transaction completes. The log will eventually

make it to disk (barring an application hardware crash of some kind). However, the application's thread of control is free to continue operations without waiting for expensive disk I/O to complete.

This represents the least durable configuration that you can provide for your transactions. But it also offers much better write performance than the other options.

- Durability.SyncPolicy.WRITE_NO_SYNC

  Log data is synchronously written to the OS's file system buffers upon transaction commit, but the data is not actually forced to disk. This protects your write activities from an application crash, but not from a hardware failure.

- This policy represents an intermediate durability guarantee. It is not has strong as SYNC, but is also not as weak as NO_SYNC. Conversely, it performs better than NO_SYNC (because your application does not have to wait for actual disk I/O), but it does not perform quite as well as SYNC (because data still must be written to the file system buffers).

Managing Acknowledgements

Whenever a Master commits a transaction, by default it waits for acknowledgements from a majority of its Replicas before the commit operation on the Master completes. By default, Replicas respond with an acknowledgement once they have successfully written the transaction to their local disk.

Acknowledgements are expensive operations. They involve both network traffic, as well as disk I/O at multiple physical machines. So on the one hand, acknowledgements help to increase your durability guarantees. On the other, they hurt your application's performance, and may have a negative impact on your application's consistency guarantee.

For this reason, JE allows you to manage acknowledgements for your HA application. As is the case with synchronization policies, you do this using the Durability class. As a part of this class' constructor, you can provide it with one of the following constants:

- Durability.ReplicaAckPolicy.ALL

  All of the Replicas must acknowledge the transactional commit. This represents the highest possible durability guarantee for your HA application, but it also represents the poorest performance. For best results, do not use this policy unless your replication group contains a very small number of replicas, and those replicas are all on extremely reliable networks and servers.

- Durability.ReplicaAckPolicy.NONE

  The Master will not wait for any acknowledgements from its Replicas. In this case, your durability guarantee is determined entirely by the synchronization policy your Master is using for its transactional commits. This policy also represents the best possible choice for write performance.

- Durability.ReplicaAckPolicy.SIMPLE_MAJORITY

A simple majority of the Replicas must return and acknowledgement before the commit operation returns on the Master. This is the default policy. It should work well for most applications unless you need an extremely high durability guarantee, have a very large number of Replicas, or you otherwise have performance concerns that cause you to want to avoid acknowledgements altogether.

You can configure your synchronization policy on a transaction-by-transaction basis using TransactionConfig.setDurability(), or on an environment-wide basis using EnvironmentMutableConfig.setDurability(). For example:

```
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    envConfig.setTransactional(true);

    // Require no synchronization for transactional commit on the
    // Master, but full synchronization on the Replicas. Also,
    // wait for acknowledgements from a simple majority of Replicas.
    Durability durability =
           new Durability(Durability.SyncPolicy.WRITE_NO_SYNC,
                          Durability.SyncPolicy.NO_SYNC,
                          Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);

    envConfig.setDurability(durability);

    // Identify the node
    ReplicationConfig repConfig =
         new ReplicationConfig("PlanetaryRepGroup",
                               "Jupiter",
                               "jupiter.acme.com:5002");

    // Use the node at mercury.acme.com:5001 as a helper to find
    // the rest of the group.
    repConfig.setHelperHosts("mercury.acme.com:5001");

    ReplicatedEnvironment repEnv =
       new ReplicatedEnvironment(home, repConfig, envConfig);
```

Note that at the time of a transaction commit, if the Master is not in contact with enough Replicas to meet the transaction's durability policy, the transaction commit operation will throw an InsufficientReplicasException. The proper action to take upon encountering this exception is to abort the transaction, wait a small period of time in the hopes that more Replicas will become available, then retry the exception. See Example Run Transaction Class (page 44) for example code that implements this retry loop.

You can also see an InsufficientReplicasException when you begin a transaction if the Master fails to be in contact with enough Replicas to meet the acknowledgement policy. To manage this, you can configure how long the transaction begin operation will wait for enough Replicas before throwing this exception. You use the INSUFFICIENT_REPLICAS_TIMEOUT configuration option, which you can set using the ReplicationConfig.setConfigParameter() method.

Managing Acknowledgement Timeouts

In addition to the acknowledgement policies, you have to also consider your replication acknowledgement timeout value. This value specifies the maximum amount of time that the Master will wait for acknowledgements from its Replicas.

If the Master commits a transaction and the timeout value is exceeded while waiting for enough acknowledgements, the Transaction.commit() method will throw an InsufficientAcksException exception. In this event, the transaction has been committed on the Master, so at least locally the transaction's durability policy has been met. However, the transaction might not have been committed on enough Replicas to guarantee your HA application's overall durability policy.

There can be a lot of reasons why the Master did not get enough acknowledgements before the timeout value, such as a slow network. As a result, whether you consider the transaction to be a failure is up to you. However, given enough time, all of your Replicas should eventually catch up to the Master.

The default value for this timeout is 5 seconds, which should work for most cases where an acknowledgement policy is in use. However, if you have a very large number of Replicas, or if you have a very unreliable network, then you might see a lot of InsufficientAcksException exceptions. In this case, you should either increase this timeout value, relax your acknowledgement policy, or find out why your hardware and/or network is performing so poorly.

You can configure your acknowledgement policy using the ReplicationConfig.setReplicaCommitTimeout() method.

```
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    envConfig.setTransactional(true);

    // Require no synchronization for transactional commit on the
    // Master, but full synchronization on the Replicas. Also,
    // wait for acknowledgements from a simple majority of Replicas.
    Durability durability =
           new Durability(Durability.SyncPolicy.WRITE_NO_SYNC,
                          Durability.SyncPolicy.NO_SYNC,
                          Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);

    envConfig.setDurability(durability);

    // Identify the node
    ReplicationConfig repConfig =
         new ReplicationConfig("PlanetaryRepGroup",
                               "Jupiter",
                               "jupiter.acme.com:5002");

    // Use the node at mercury.acme.com:5001 as a helper to find the rest
    // of the group.
```

```
    repConfig.setHelperHosts("mercury.acme.com:5001");

    // Set a acknowledgement timeout that is slightly longer
    // than the default 5 seconds.
    repConfig.setReplicaCommitTimeout(7, TimeUnit.SECONDS);

    ReplicatedEnvironment repEnv =
       new ReplicatedEnvironment(home, repConfig, envConfig);
```

Managing Consistency

In a traditional stand-alone transactional application, *consistency* means that a transaction takes the database from one consistent state to another. What defines a consistent state is application-specific. This transition is made atomically, that is, either all the operations that constitute the transaction are performed, or none of them are. JE HA supports this type of transactional consistency both on the Master, as well as on the Replicas as the replication stream is replayed. That is, in the absence of failures, the Replicas will see exactly the same sequence of transitions, from one consistent state to another, as the Master.

A JE HA application must additionally concern itself with the data consistency of the Replica with respect to the Master. In a distributed system like JE HA, the changes made at the Master are not always instantaneously available at every Replica, although they eventually will be. For example, consider a three node group where a change is made on the Master and the transaction is committed with a durability policy requiring acknowledgments from a simple majority of nodes. After a successful commit of this transaction, the changes will be available at the Master and at one other Replica, thus satisfying the requirement for a simple majority of acknowledgments. The state of the Master and the acknowledging Replica will be consistent with each other after the transaction has been committed, but the transaction commit makes no guarantees about the state of the third Replica after the commit.

In general, Replicas not directly involved in contributing to the acknowledgment of a transaction commit will lag in the replay of the replication stream because they do not synchronize their commits with the Master. As a consequence, their state, on an instantaneous basis, may not be current with respect to the Master. However, in the absence of further updates, all Replicas will eventually catch up and reflect the instantaneous state of the Master. This means that a Replica which is not consistent with the Master simply reflects an earlier locally consistent state at the Master because transaction updates on the Replica are always applied, atomically and in order. From the application's perspective, the environment on the Replica goes through exactly the same sequence of changes to its persistent state as the Master.

A Replica may similarly lag behind the Master if it has been down for some period of time and was unable to communicate with the Master. Such a Replica will catch up, when it is brought back up and will eventually become consistent with the Master.

Given the distributed nature of a JE HA application, and the fact that some nodes might lag behind the Master, the question you have to ask yourself is how long will it take for that node to be consistent relative to the Master. More to the point: how far behind the Master are you willing to allow the node to lag?

This should be one of your biggest concerns when it comes to architecting a JE HA application.

You define how current the nodes in your replication group must be by defining a *consistency policy*. You define your consistency policy using an implementation of the ReplicaConsistencyPolicy interface. This interface allows you to define how current the Replica must be before a transaction can be started on the Replica. (Remember that all read operations are performed within a transaction.) If the Replica is not current enough, then the start of that transaction is delayed until that level of consistency has been reached. This means that Replicas that are not current enough will block read operations until they are brought up to date.

Obviously your consistency policy can have an affect on your Replica's read performance by increasing the latency experienced by read transactions. This is because transactions may have to wait to either begin or commit until the consistency policy can be satisfied. If the consistency policy is so stringent that it cannot be satisfied using the available resources, the Replica's availability for reads may deteriorate as transactions timeout. A Durability.SyncPolicy.SYNC policy on the Replica can slow down write operations on the Replica, making it harder for the Replica to meet its consistency guarantee. Conversely, a Durability.SyncPolicy.NO_SYNC policy on the Replica makes it easy for the Replica to keep up, which means you can have a stronger consistency guarantee.

One of three interface implementations are available for you to use when defining your consistency policy:

- NoConsistencyRequiredPolicy

  No consistency policy is enforced. This policy allows a transaction on a Replica to proceed regardless of the state of the Replica relative to the Master. This policy can also be used to access a database when the replication node is in a DETACHED state.

- TimeConsistencyPolicy

  Defines how far back in time the Replica is permitted to lag the Master.

- CommitPointConsistencyPolicy

  Defines consistency in terms of a specified commit token. That is, the Replica must be at least as current as the CommitToken provided to this class.

Setting Consistency Policies

You set a consistency policy by using ReplicationConfig.setConsistencyPolicy(). For example:

```
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    envConfig.setTransactional(true);

    // Require no synchronization for transactional commit on the
    // Master, but full synchronization on the Replicas. Also,
    // wait for acknowledgements from a simple majority of Replicas.
    Durability durability =
           new Durability(Durability.SyncPolicy.NO_SYNC,
```

```
                        Durability.SyncPolicy.SYNC,
                        Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);

    envConfig.setDurability(durability);

    // Identify the node
    ReplicationConfig repConfig =
         new ReplicationConfig("PlanetaryRepGroup",
                               "Jupiter",
                               "jupiter.acme.com:5002");

    // Use the node at mercury.acme.com:5001 as a helper to find the rest
    // of the group.
    repConfig.setHelperHosts("mercury.acme.com:5001");

    // Turn off consistency policies. Transactions can occur
    // regardless of how consistent the Replica is relative
    // to the Master.
    NoConsistencyRequiredPolicy ncrp =
         new NoConsistencyRequiredPolicy();
    repConfig.setConsistencyPolicy(ncrp);

    ReplicatedEnvironment repEnv =
       new ReplicatedEnvironment(home, repConfig, envConfig);
```

Note that the consistency policy is set on a node-by-node basis. There is no requirement that you set the same policy for every node in your replication group.

You can also set consistency policies on a transaction-by-transaction basis when you begin the transaction:

```
    // Turn off consistency policies. The transactions can
    // be performed regardless of how consistent the Replica is
    // relative to the Master.
    NoConsistencyRequiredPolicy ncrp =
         new NoConsistencyRequiredPolicy();

    TransactionConfig tc = new TransactionConfig();
    tc.setConsistencyPolicy(ncrp);
    // env is a ReplicatedEnvironment handle
    env.beginTransaction(null, tc);
```

Time Consistency Policies

A time consistency policy is a time-oriented policy that defines how far back in time the Replica is permitted to lag the Master. It does so by comparing the time associated with the latest transaction committed on the Master with the current time. If the Replica lags by an amount greater than the permissible lag, it will hold back the start of the transaction until the Replica has replayed enough of the replication stream to narrow the lag to within the permissible lag.

Use of a time based consistency policy requires that nodes in a replication group have their clocks reasonably synchronized. This can be easily achieved using a daemon like NTPD.

You implement a time-based consistency policy by using the TimeConsistencyPolicy class. To instantiate this class, you provide it with the following:

- A number representing the permissible lag.
- A TimeUnit constant indicating the units of time that the permissible lag represents.
- A number representing the timeout period during which a transaction will wait for the Replica to catch up so that the consistency policy can be met. If the transaction waits more than the timeout period, a ReplicaConsistencyException is thrown.
- A TimeUnit constant indicating the units of time in use for the timeout value.

For example:

```
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
envConfig.setTransactional(true);

// Require no synchronization for transactional commit on the
// Master, but full synchronization on the Replicas. Also,
// wait for acknowledgements from a simple majority of Replicas.
Durability durability =
       new Durability(Durability.SyncPolicy.NO_SYNC,
                      Durability.SyncPolicy.SYNC,
                      Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);

envConfig.setDurability(durability);

// Identify the node
ReplicationConfig repConfig =
     new ReplicationConfig("PlanetaryRepGroup",
                           "Jupiter",
                           "jupiter.acme.com:5002");

// Use the node at mercury.acme.com:5001 as a helper to find the rest
// of the group.
repConfig.setHelperHosts("mercury.acme.com:5001");

// Set consistency policy for replica.
TimeConsistencyPolicy consistencyPolicy = new TimeConsistencyPolicy
    (1, TimeUnit.SECONDS, /* 1 sec of lag */
    10, TimeUnit.SECONDS /* Wait up to 10 sec */);
repConfig.setConsistencyPolicy(consistencyPolicy);

ReplicatedEnvironment repEnv =
   new ReplicatedEnvironment(home, repConfig, envConfig);
```

Commit Point Consistency Policies

A commit point consistency policy defines consistency in terms of the commit of a specific transaction. This policy can be used 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.

As is the case with a time consistency policy, if the Replica is not current enough relative to the Master, all attempts to begin a transaction will be delayed until the Replica has caught up. If the Replica does not catch up within a specified timeout period, the transaction will throw a ReplicaConsistencyException.

In order to specify a commit point consistency policy, you must provide a CommitToken that is used to identify the transaction that the Replica must have in order to be current enough. Because the commit point that you care about will change from transaction to transaction, you do not specify commit point consistency policies on an environment-wide basis. Instead, you specify them when you begin a transaction.

For example, suppose the application is a web application where a replicated 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 node that did not execute the update.

In such a case, the update request would generate a CommitToken, 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 node which executes the read request would create a CommitPointConsistencyPolicy with that CommitToken and use it at transaction begin. If the environment at the web server was already current enough, it could immediately execute the transaction and satisfy the request. If not, the "transaction begin" would stall until the Replica replay had caught up and the change was available at that web server.

You obtain a commit token using the Transaction.getCommitToken() method. Use this method after you have successfully committed the transaction that you want to base a CommitPointConsistencyPolicy upon.

For example:

```
Database myDatabase = null;
Environment myEnv = null;
CommitToken ct = null;
try {
    ...
    // Environment and database setup removed for brevity
    ...

    Transaction txn = myEnv.beginTransaction(null, null);
```

```
    try {
        myDatabase.put(txn, key, data);
        txn.commit();
        ct = txn.getCommitToken();
        if (ct != null) {
            // Do something with the commit token to
            // forward it to the Replica where you
            // want to use it.
        }
    } catch (Exception e) {
        if (txn != null) {
            txn.abort();
            txn = null;
        }
    }

} catch (DatabaseException de) {
    // Exception handling goes here
}
```

To create your commit point token consistency policy, transfer the commit token to the Replica performing a read using whatever mechanism that makes sense for your HA application, and then create the policy for that specific transaction handle: Note that CommitToken implements Serializable, so you can use the standard Java serialization mechanisms when passing the commit token between processes.

```
Database myDatabase = null;
Environment myEnv = null;
CommitToken ct = null;
try {
    ...
    // Environment and database setup removed for brevity
    ...

    CommitPointConsistencyPolicy cpcp =
        new CommitPointConsistencyPolicy(ct,      // The commit token
                         10, TimeUnit.SECONDS); // Timeout value

    TransactionConfig txnConfig = new TransactionConfig();
    txnConfig.setConsistencyPolicy(cpcp);


    Transaction txn = myEnv.beginTransaction(null, txnConfig);

    try {
        // Perform your database read here using the transaction
        // handle, txn.
        txn.commit();
    } catch (Exception e) {
        // There are quite a lot of different exceptions that can be
```

```
        // seen at this level, including the LockConflictException.
        // We just catch Exception for this example for simplicity's
        // sake.
        if (txn != null) {
            txn.abort();
            txn = null;
        }
    }

} catch (ReplicaConsistencyException rce) {
        // Deal with this timeout error here. It is thrown by the
        // beginTransaction operation if the consistency policy
        // cannot be met within the timeout time.
} catch (DatabaseException de) {
    // Database exception handling goes here.
} catch (Exception ee) {
    // General exception handling goes here.
}
```

Availability

A key difference between standalone JE and JE HA is that for standalone JE the environment is available for both reads and writes as long as the application (including the underlying hardware) is functioning correctly. That is, the availability of a standalone JE application is independent of the local durability policy set for the transaction. However, the distributed nature of JE HA, means that availability can be dependent upon the state of other nodes in the replication group. It can also be dependent upon the policies you set for your HA application.

Write Availability

JE HA requires that a simple majority of nodes be available to elect a Master. If a simple majority of nodes is not available, the group is not available for writes because the group is unable to elect a Master.

In the presence of a Master, the availability of a replicated environment (at the Master) for write operations is determined by the durability requirements associated with the transaction:

- If the transaction's durability requirements specify an acknowledgement policy of NONE, the Master is always available for write operations, just as is the case for standalone JE applications.
- If the durability requirements are made more stringent and specify a simple majority for acknowledgements, or if all the group members must acknowledge transaction commits, the environment might not be available for writes when one or more of the Replicas is unable to provide an acknowledgment. This loss of write availability can occur even in the absence of hardware failures.

  Replicas might be unable to provide acknowledgements because a node is down. It could also occur if the Replica is simply lagging too far behind in the replication stream and

so needs to commit earlier transactions before it can commit the current transaction. Note that in the absence of system level failures, the Replica will eventually commit the transaction, it just can not do so in the window of time required to indicate a successful commit of the transaction to the Master.

In other words, a durability policy that calls for commit acknowledgments can result in decreased availability of the system for write operations. It is important for you to keep this tradeoff in mind when choosing a durability policy.

Read Availability

A Master is always available for read operations because the data on it is always absolutely consistent. However, Replica read availability can be affected by the consistency policy that you are using:

- A Replica is always available for read operations that do not have any read consistency requirements. That is, when the Replica is allowed to lag arbitrarily far behind the Master, then the Replica will always be available to service read requests.
- If you are using higher levels of read consistency, then Replicas might not be available for read operations. This occurs when the Replica is forced to wait until it has caught up far enough in the replication stream before it can service a read operation. For example, if you choose a time consistency policy, and the the Replica cannot meet that consistency policy for a specific read operation, then the operation might be delayed or even abandoned entirely until the consistency policy can be met. This represents a loss of read availability.

  There are many reasons why a Replica might not be able to meet a consistency policy. For example, the Master might be very busy and so is unable to supply the Replica with the replication stream fast enough. Or, it could be because the Replica is experiencing very heavy read loads and so the replication stream might not be fast enough to keep up. It is also possible that the Replica has been down and is trying to catch up, which means that it might not be able to meet a consistency policy.

  All of these scenarios represent a loss of read availability, albeit a temporary one.

In other words, a consistency policy that requires the Replica to match the state of the Master to one degree or another can affect the Replica's read availability. It is important for you to keep this tradeoff in mind when choosing a consistency policy.

Consistency and Durability Use Cases

As discussed throughout this chapter, there is an interaction between consistency and durability. This interaction results in design decisions that you will have to make when designing your HA application. To further illustrate this interaction, this section provides several use cases as examples of how durability and consistency policies are used to reach application design goals.

Out on the Town

Out on the Town is a social networking site about restaurants and artistic events. Restaurant locations and an event calendar are available on the site. Members can submit reviews about

restaurants and events, and other members can comment on the reviews. Further, members maintain accounts and profiles.

The site experiences most of its traffic as read-only requests. There is heavy ready traffic from users who are browsing the site. In addition, periodic write traffic occurs as reviews and comments are submitted to the site.

Reading Reviews

Based on the site's usage characteristics, the web developers know that it is critical that the site perform well for read traffic. Listings must be readily available, and the site must be able to adapt to changing read loads. However, the site only needs a low threshold for most reads.

While users should not experience a delay when they access the site, it is okay if read requests do not see the very latest reviews. For this reason, when starting read-only transactions for the purpose of viewing reviews, the application specifies a consistency policy of NoConsistencyRequiredPolicy. This provides the highest possible availability for read requests for the Replica nodes, which is the critical thing for this particular site. (Any other consistency policy might cause the node to delay reads while waiting for the node to meet its consistency policy, which would represent an unacceptable loss of availability as it could cost the site lost readership.)

Writing Reviews

Most write operations are for new user reviews, and for comments on those reviews. For these writes, the application needs only a very lenient durability policy. It is not critical that a new review is immediately available to other users, nor is it critical that they are saved in the event of a catastrophic failure.

Therefore, the application uses the convenience constant Durability.COMMIT_WRITE_NO_SYNC as the system default durability policy. (This is done by specifying the durability policy using EnvironmentMutableConfig.setDurability().) This means:

- Write operations on the Master use Durability.SyncPolicy.WRITE_NO_SYNC.
- When the write operation is forwarded by the Master to the Replicas, those Replicas use Durability.SyncPolicy.NO_SYNC when they internally update their own databases.
- Only a simple majority of the replication nodes need to acknowledge the update.

Updating Events and Restaurant Listings

Periodically, the calendar of events and restaurant locations are updated. These write operations happen fairly infrequently relative to reviews and comments, but the site's operators deem this information to be of more importance (or valuable) than the reviews and comments. Therefore, they want a stronger guarantee that the information is backed up to all nodes, which is the same thing as saying they want a stronger durability guarantee. Nevertheless, they also want this class of writes to consume few resources

To achieve this, for transactions performing these kind of writes, the web engineers choose to override the sites default durability guarantee. Instead, they use a durability guarantee that:

- Uses Durability.SyncPolicy.SYNC for the local synchronization policy. This ensures that the write is fully backed up to the Master's local disk before the transaction commit operation returns.
- Uses Durability.SyncPolicy.WRITE_NO_SYNC for the synchronization policy on the Replica nodes. This causes the updates to be written to the disk controller's buffers, but they are not flushed to disk before the Replicas acknowledge the commit operation.
- Stays with a simply majority for acknowledgements, which is the same as is used for the default durability policy.

That is, for updating events and restaurant locations, the application uses this durability policy:

```
    useForUpdates =
         new Durability(Durability.SyncPolicy.SYNC,
                        Durability.SyncPolicy.WRITE_NO_SYNC,
                        Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);
```

Updating Account Profiles

If a user makes an account profile change as part of a web session, she will naturally expect to see her changes when she next looks at the profile during the same session. From the user's perspective, this is all one operation: she causes her profile to change and then the profile page is refreshed with her new information.

However, from the application's perspective, there are several things going on:

- A write transaction is performed on the Master.
- One or more read transactions are performed on the Replica node in use by the user as she updates her profile and then reads back the changes she just made.

To ensure that the session interaction looks intuitively consistent to the user, the application:

- Performs the write transaction on the Master.
- Saves the CommitToken for the account profile update within the web session.
- The Replica node uses a CommitPointConsistencyPolicy policy for the follow-on account profile read(s). To do this, the application uses the CommitToken stored in the previous step when beginning the read transactions. In this way, the Replica will not serve up the new profile page until it has received the profile updates from the Master. From the user's perspective, there may be a delay in her page refresh when she submits her updates. How long of a delay experienced by the user is a function of how busy the site is with write updates, as well as the performance characteristics of the hardware and networks in use by the site.

Bio Labs, Inc

Bio Labs, Inc is a biotech company that is doing pharmaceutical production which must be audited by government agencies. Production sampling results are logged frequently. All such

updates must be guaranteed to be backed up. (In other words, this application requires a very high durability guarantee.)

In addition, there are frequent application defined sample points that represent phases in the production cycle. The application performs monitoring of the production stream. These reads are time critical, so the data must be no older than a specific point in time.

Logging Sampling Results

Due to the auditing requirement for the sampling results, the application developers want an extremely high data durability guarantee. Therefore, they require the synchronization policy on both the Master and all Replica nodes to be Durability.SyncPolicy.SYNC, which means that the logging data is guaranteed to be written to stable storage before the host returns from its transaction commit.

For an acknowledgement policy, the engineers considered requiring all nodes to acknowledge the commit. This would provide them with the strongest possible durability guarantee. However, they decided against this because it represents a possible loss of write availability for the application; if even one node is shutdown or hidden by a network outage, then the Master would not be able to perform any write operations at all. So instead, the engineers stick with the default acknowledgement policy, which is to require a simple majority of the nodes to acknowledge the commit.

The durability policy, then, looks like this:

```
    resultsDurability =
         new Durability(Durability.SyncPolicy.SYNC,
                        Durability.SyncPolicy.SYNC,
                        Durability.ReplicaAckPolicy.SIMPLE_MAJORITY);
```

Monitoring the Production Stream

The BioLabs application is required to monitor the production stream. All such monitoring must be of data that is no older than a defined age.

This represents a read activity that has a time concurrency policy requirement. Therefore, whenever the application performs a write (that is, logs sampling results), the application creates a CommitToken. Each of the nodes, then, use this commit token to specify a CommitPointConsistencyPolicy policy when the Environment.beginTransaction() method is called. This guarantees that the application's data monitoring activities will be performed on data that is not out of date or stale.

Managing Transaction Rollbacks

In the event that a new Master is elected, it is possible for a Replica to find that some of its logs are ahead of the logs held by the Master. While this is unlikely to occur, your code must still be ready to deal with the situation. When it happens, you must roll back the transactions represented by the logs that are ahead of the Master.

You do this by simply closing all your ReplicatedEnvironment handles, and then reopening. During the handshaking process that occurs when the Replica joins the replication group, the discrepancy in log files are resolved for you.

JE HA lets your application know that a transaction must be rolled back by throwing RollbackException. This exception can by thrown by any operation that is performing routine database access.

```
    ReplicatedEnvironment repEnv = new ReplicatedEnvironment(...);
    boolean doWork = true;

    while doWork {
        try {
            // performSomeDBWork is the method that
            // performs your database access.
            doWork = performSomeDBWork();
        } catch (RollbackException rb) {
            if (repEnv != null) {
                repEnv.close();
                repEnv = new ReplicatedEnvironment(...);
            }
    }
```

Example Run Transaction Class

Usage of JE HA requires you to handle many different HA-specific exceptions. While some of these are Master-specific and others are Replica-specific, your code may still need to handle both. The reason why is that it is not uncommon for HA applications to have standard classes that perform database access, regardless of whether the class is used for a node in the Master state or a node in the Replica state.

The following class is an example class that can be used to perform transactional reads and writes in an HA application. This class is used by the on-disk HA examples that you can find in your JE distribution (see Replication Examples (page 64) for more information). However, we think this particular example class is important enough that we also describe it here.

RunTransaction Class

The `RunTransaction` abstract class is used to implement a utility class that performs database access for HA applications. It provides all the transaction error handling and retry framework that is required for database access in an HA environment.

Because `RunTransaction` is a class that is meant to be used by different example HA applications, it does not actually implement the database operations. Instead, it provides an abstract method that must be implemented by the HA application that uses `RunTransaction`.

We begin by importing the classes that `RunTransaction` uses.

```
package je.rep.quote;

import java.io.PrintStream;

import com.sleepycat.je.EnvironmentFailureException;
import com.sleepycat.je.LockConflictException;
import com.sleepycat.je.OperationFailureException;
```

```
import com.sleepycat.je.Transaction;
import com.sleepycat.je.rep.InsufficientAcksException;
import com.sleepycat.je.rep.InsufficientReplicasException;
import com.sleepycat.je.rep.ReplicaConsistencyException;
import com.sleepycat.je.rep.ReplicaWriteException;
import com.sleepycat.je.rep.ReplicatedEnvironment;
```

Then we define a series of private data members that identify how our HA transactions are going to behave in the event of an error condition.

```
abstract class RunTransaction {

    /* The maximum number of times to retry the transaction. */
    private static final int TRANSACTION_RETRY_MAX = 10;

    /*
     * The number of seconds to wait between retries when a sufficient
     * number of replicas are not available for a transaction.
     */
     private static final int INSUFFICIENT_REPLICA_RETRY_SEC = 1;

    /* Amount of time to wait top let a replica catch up before
     * retrying.
     */
    private static final int CONSISTENCY_RETRY_SEC = 1;

    /* Amount of time to wait after a lock conflict. */
    private static final int LOCK_CONFLICT_RETRY_SEC = 1;

    private final ReplicatedEnvironment env;
    private final PrintStream out;
```

Then we implement our class constructor, which is very simple because all the heavy lifting is done by whatever application calls this utility class.

```
    RunTransaction(ReplicatedEnvironment repEnv,
                   PrintStream printStream) {
        env = repEnv;
        out = printStream;
    }
```

Now we implement our `run()` method. This is what actually performs all the error checking and retry work for the class.

The `run()` method catches the exceptions most likely to occur as we are reading and writing the database, and then handles them, but it will also throw InterruptedException and EnvironmentFailureException.

InterruptedException can be thrown if the thread calling this method is sleeping and some other thread interrupts it. The exception is possible because this method calls Thread.sleep in the retry cycle.

EnvironmentFailureException can occur both when beginning a transaction and also when committing a transaction. It means that there is something significantly wrong with the node's environment.

The `readOnly` parameter for this method is used to indicate that the transaction will only perform database reads. When that happens, the durability guarantee for the transaction is changed to Durability.READ_ONLY_TXN because that policy does not call for any acknowledgements. This eliminates the possiblity of an InsufficientReplicasException being thrown from the Environment.beginTransaction() operation.

```
    public void run(boolean readOnly)
        throws InterruptedException, EnvironmentFailureException {
```

Now we begin our retry loop and define our sleep cycle between retries. Initially, we do not actually sleep before retrying the transaction. However, some of the error conditions caught by this method will cause the thread to sleep before the operation is retried. After every sleep operation, the sleep time is returned to 0 because usually putting the thread to sleep is of no benefit.

```
        OperationFailureException exception = null;
        boolean success = false;
        long sleepMillis = 0;
        final TransactionConfig txnConfig = readOnly ?
         new TransactionConfig().setDurability(Durability.READ_ONLY_TXN) :
         null;

        for (int i = 0; i < TRANSACTION_RETRY_MAX; i++) {
            /* Sleep before retrying. */
            if (sleepMillis != 0) {
                Thread.sleep(sleepMillis);
                sleepMillis = 0;
             }
```

Now we create our transaction, perform the database operations, and then do the work. The `doTransactionWork()` method is an abstract method that must be implemented by the application using this class. Otherwise, this is standard transaction begin/commit code that should hold no surprises for you.

```
            Transaction txn = null;
            try {
                txn = env.beginTransaction(null, null);
                doTransactionWork(txn); /* CALL APP-SPECIFIC CODE */
                txn.commit();
                success = true;
                return;
```

The first error case that we check for is InsufficientReplicasException. This exception means that the Master is not in contact with enough Replicas to successfully commit the transaction. It is possible that Replicas are still starting up after an application restart, so we put the thread to sleep before attempting the transaction again.

InsufficientReplicasException is thrown by Transaction.commit(), so we do have to perform the transaction all over again.

```
            } catch (InsufficientReplicasException insufficientReplicas) {

                /*
                 * Retry the transaction.  Give replicas a chance to
                 * contact this master, in case they have not had a
                 * chance to do so following an election.
                 */
                exception = insufficientReplicas;
                out.printf(insufficientReplicas.toString());
                sleepMillis = INSUFFICIENT_REPLICA_RETRY_SEC * 1000;
                continue;
```

Next we check for InsufficientAcksException. This exception means that the transaction has successfully committed on the Master, but not enough Replicas have acknowledged the commit within the allowed period of time. Whether you consider this to be a successful commit depends on your durability policy.

As provided here, the code handles considers this situation to be an unsuccessful commit. But if you have a lot of Replicas and you have a strong durability guarantee on the Master, then you might be able to still consider this to be a successful commit. If so, you should set `success = true;` before returning from the method.

For more information on this error case, see Managing Acknowledgement Timeouts (page 32).

```
             } catch (InsufficientAcksException insufficientReplicas) {

                /*
                 * Transaction has been committed at this node. The
                 * other acknowledgments may be late in arriving,
                 * or may never arrive because the replica just
                 * went down.
                 */

                /*
                 * INSERT APP-SPECIFIC CODE HERE: For example, repeat
                 * idempotent changes to ensure they went through.
                 *
                 * Note that 'success' is false at this point, although
                 * some applications may consider the transaction to be
                 * complete.
                 */
                out.println(insufficientReplicas.toString());
                txn = null;
                return;
```

Next we check for ReplicaWriteException. This happens when a write operation is attempted on a Replica. In response to this, any number of things can be done, including reporting the

problem to the application attempting the write operation and then aborting, to forwarding the write request to the Master. This particular method responds to this condition based on how the `onReplicaWrite()` method is implemented.

For more information on how to handle this exception, see Managing Write Requests at a Replica (page 21).

```
            } catch (ReplicaWriteException replicaWrite) {

                /*
                 * Attempted a modification while in the Replica
                 * state.
                 *
                 * CALL APP-SPECIFIC CODE HERE: Cannot accomplish
                 * the changes on this node, redirect the write to
                 * the new master and retry the transaction there.
                 * This could be done by forwarding the request to
                 * the master here, or by returning an error to the
                 * requester and retrying the request at a higher
                 * level.
                 */
                onReplicaWrite(replicaWrite);
                return;
```

Now we check for LockConflictException, which is thrown whenever a transaction experiences a lock conflict with another thread. Note that by catching this exception, we are also catching the LockPreemptedException, which happens whenever the underlying HA code "steals" a lock from an application transaction. The most common cause of this is when the HA replication stream is updating a Replica, and the Replica is holding a read lock that the replication stream requires.

Here, it is useful to sleep for a period of time before retrying the transaction.

```
            } catch (LockConflictException lockConflict) {

                /*
                 * Retry the transaction.  Note that LockConflictException
                 * covers the HA LockPreemptedException.
                 */
                exception = lockConflict;
                out.println(lockConflict.toString());
                sleepMillis = LOCK_CONFLICT_RETRY_SEC * 1000;
                continue;
```

The last error we check for is ReplicaConsistencyException. This exception can be thrown when the transaction begins. It means that the `beginTransaction()` method has waited too long for the Replica to catch up relative to the Master. This situation does not really represent a failed transaction because the transaction never had a chance to proceed in the first place.

In any case, the proper thing to do is to put the thread to sleep for a period of time so that the Replica has the chance to meet its consistency requirements. Then we retry the transaction.

Note that at this point in time, the transaction handle is in whatever state it was in when `beginTransaction()` was called. If the handle was in the null state before attempting the operation, then it will still be in the null state. The important thing to realize here is that the transaction does not have to be aborted, because the transaction never began in the first place.

For more information on consistency policies, see Managing Consistency (page 33).

```
            } catch (ReplicaConsistencyException replicaConsistency) {

                /*
                 * Retry the transaction. The timeout associated with
                 * the ReplicaConsistencyPolicy may need to be
                 * relaxed if it's too stringent.
                 */
                exception = replicaConsistency;
                out.println(replicaConsistency.toString());
                sleepMillis = CONSISTENCY_RETRY_SEC * 1000;
                continue;
```

Finally, we abort our transaction and loop again as needed. `onRetryFailure()` is called if the transaction has been retried too many times (as defined by `TRANSACTION_RETRY_MAX`. It provides the option to log the situation.

```
            } finally {

                if (!success) {
                    if (txn != null) {
                        txn.abort();
                    }

                    /*
                     * INSERT APP-SPECIFIC CODE HERE: Perform any
                     * app-specific cleanup.
                     */
                }
            }
        }

        /*
         * CALL APP-SPECIFIC CODE HERE:
         * Transaction failed, despite retries.
         */
        onRetryFailure(exception);
    }
```

Having done that, the class is almost completed. Left to do is to define a couple of methods, one of which is an abstract method that must be implemented by the application that uses this class.

`doTransactionWork()` is an abstract method where the actual database operations are performed.

onReplicaWrite() is a method that should be implemented by the HA application that uses this class. It is used to define whatever action the Replica should take if a write is attempted on it. For examples of how this is used, see the next section.

For this implementation of the class, we simply throw the ReplicaWriteException that got us here in the first place.

```
    abstract void doTransactionWork(Transaction txn);

        void onReplicaWrite(ReplicaWriteException replicaWrite) {
            throw replicaWrite;
        }
```

Finally, we implement onRetryFailure(), which is what this class does if the transaction retry loop goes through too many iterations. Here, we simply print the error to the console. A more robust application should probably write the error to the application logs.

```
    void onRetryFailure(OperationFailureException lastException) {
            out.println("Failed despite retries." +
                                ((lastException == null) ?
                                  "" :
                                  " Encountered exception:" +
                                  lastException));
        }
    }
```

Using RunTransaction

Having implemented the RunTransaction class, it is fairly easy to use. Essentially, you only have to implement the RunTransaction.doTransactionWork() method so that it performs whatever database access you want.

For example, the following method performs a read on an EntityStore used by the StockQuotes example HA application. Notice that the class is instantiated, doTransactionWork() is implemented, and the RunTransaction.run() method are all called in one place. This makes for fairly easy maintenance of the code.

```
    private void printStocks(final PrintStream out)
        throws InterruptedException {

        new RunTransaction(repEnv, out) {

            @Override
            void doTransactionWork(Transaction txn) {

                // dao is a DataAccessor class used to access
                // an entity store.
                final EntityCursor<Quote> quotes =
                    dao.quoteById.entities(txn, null);
                try {
                    out.println("\tSymbol\tPrice");
```

```
                    out.println("\t======\t=====");

                    int count = 0;
                    for (Quote quote : quotes) {
                        out.println("\t" +  quote.stockSymbol +
                                    "\t" + quote.lastTrade);
                        count++;
                    }
                    out.println("\n\t" + count + " stock"
                                + ((count == 1) ? "" : "s") +
                                " listed.\n");
                } finally {
                    quotes.close();
                }
            }
            }.run(true /*readOnly*/);

        /* Output local indication of processing. */
        System.out.println("Processed print request");
    }
```

In the previous example, we do not bother to override the `RunTransaction.onReplicaWrite()` method because this transaction is performing read-only access to the database. Regardless of whether the transaction is run on a Master or a Replica, ReplicaWriteException can not be raised here, so we can safely use the default implementation.

However, if we were running a transaction that performs a database write, then we should probably do something with `onReplicaWrite()` other than merely re-throwing the exception.

The following is an example usage of `RunTransaction` that is also used in the StockQuotes example.

```
    void updateStock(final String line, final PrintStream printStream)
        throws InterruptedException {

        // Quote is a utility class used to parse a line of input
        // obtained from the console.
        final Quote quote = QuoteUtil.parseQuote(line);
        if (quote == null) {
            return;
        }

        new RunTransaction(repEnv, printStream) {

            void doTransactionWork(Transaction txn) {
                // dao is a Data Accessor class used to perform access
                // to the entity store.
                dao.quoteById.put(txn, quote);
                /* Output local indication of processing. */
```

```
                System.out.println("Processed update request: " + line);
            }

            // For this example, we simply log the error condition.
            // For a more robust example, so other action might be
            // taken; for example, log the situation and then route
            // the write request to the Master.
            void onReplicaWrite(ReplicaWriteException replicaWrite) {
                /* Attempted a modification while in the replica state. */
                printStream.println
                    (repEnv.getNodeName() +
                     " is not currently the master.  Perform the update" +
                     " at the node that's currently the master.");
            }
            }.run(false /*not readOnly */);
    }
```

Chapter 4. Utilities

This chapter discusses the APIs that you use to administer and manage your replication group.

Administering the Replication Group

There are a series of administrative activities that an application might want to take relative to a replication group. These activities can be performed by electable nodes in the replication group or by applications that do not have access to a replicated environment (in other words, utilities designed to help administer and monitor the group). All of these functions can be accessed using the ReplicationGroupAdmin class.

You can use the ReplicationGroupAdmin class to:

1. List replication group members.
2. Locate the current Master.
3. Remove nodes from the replication group.

You instantiate an instance of the ReplicationGroupAdmin class by providing it with the name of the replication group that you want to administer, as well as a Set of InetSocketAddress objects. The InetSocketAddress objects are used as a list of helper hosts that the application can use to perform administrative functions. For example:

```
...

    Set<InetSocketAddress> helpers =
        new HashSet<InetSocketAddress>();
    InetSocketAddress helper1 =
        new InetSocketAddress("node1.sleepycat.com", 1550);
    InetSocketAddress helper2 =
        new InetSocketAddress("node2.sleepycat.com", 1550);

    helpers.add(helper1);
    helpers.add(helper2);

    ReplicationGroupAdmin rga =
        new ReplicationGroupAdmin("test_rep_group", helpers);
```

Listing Group Members

To list all the members of a replication group, use the ReplicationGroupAdmin.getGroup() method. This returns an instance of ReplicationGroup. You then can then:

1. use the ReplicationGroup.getNodes() method to locate all the nodes in the replication group.
2. use the ReplicationGroup.getElectableNodes() method to locate all the electable nodes in the replication group.
3. use ReplicationGroup.getMonitorNodes() to locate all the monitor nodes that currently belong to the replication group.

Note

In order to obtain a ReplicationGroup object, the process must be able to discover the current Master. This means that the helper nodes you provide when you instantiate the ReplicationGroupAdmin class must be reachable and able to identify the current Master. If they cannot, then these methods throw an UnknownMasterException.

All of these methods return a set of ReplicationNode objects, which you can then use to query for node information, such as its name, the InetSocketAddress where the node is located, and the node's type.

For example:

```
...

    Set<InetSocketAddress> helpers =
        new HashSet<InetSocketAddress>();
    InetSocketAddress helper1 =
        new InetSocketAddress("node1.sleepycat.com", 1550);
    InetSocketAddress helper2 =
        new InetSocketAddress("node2.sleepycat.com", 1550);

    helpers.add(helper1);
    helpers.add(helper2);

    ReplicationGroupAdmin rga =
        new ReplicationGroupAdmin("test_rep_group", helpers);

    try {
        ReplicationGroup rg = rga.getGroup();
        for (ReplicationNode rn : rg.getElectableNodes()) {
            // Do something with the replication node.
        }
    } catch (UnknownMasterException ume) {
        // Can't find a master
    }
```

Locating the Current Master

You can use the ReplicationGroupAdmin class to locate the current Master in the replication group. This information is available using the ReplicationGroupAdmin.getMasterNodeName() and ReplicationGroupAdmin.getMasterSocket() methods.

ReplicationGroupAdmin.getMasterNodeName() returns a string that holds the node name associated with the Master.

ReplicationGroupAdmin.getMasterSocket() returns an InetSocketAddress class object that represents the host and port where the Master can currently be found.

Both methods will throw an UnknownMasterException if the helper nodes are not able to identify the current Master.

For example:

```
import java.net.InetSocketAddress;
import java.util.HashSet;
import java.util.Set;

import com.sleepycat.je.rep.UnknownMasterException;
import com.sleepycat.je.rep.util.ReplicationGroupAdmin;

...

    Set<InetSocketAddress> helpers =
        new HashSet<InetSocketAddress>();
    InetSocketAddress helper1 =
        new InetSocketAddress("node1.sleepycat.com", 1550);
    InetSocketAddress helper2 =
        new InetSocketAddress("node2.sleepycat.com", 1550);

    helpers.add(helper1);
    helpers.add(helper2);

    ReplicationGroupAdmin rga =
        new ReplicationGroupAdmin("test_rep_group", helpers);

    try {
        InetSocketAddress master = rga.getMasterSocket();
        System.out.println("Master is on host " +
                    master.getHostName() + " at port " +
                    master.getPort());
        }
    } catch (UnknownMasterException ume) {
        // Can't find a master
    }
```

Adding and Removing Nodes from the Group

In order to add nodes to a replication group, you simply start up a node and identify at least one helper node that can identify the current Master to the new node. After the new node has been populated with a current enough copy of the data contained on the Master, the new node is automatically a member of the replication group.

The node's status as a member of the group is persistent. That is, it is a member of the group regardless of whether it is running, and whether other nodes in the group can reach it over the network. This means that for the purposes of elections and message acknowledgements, the node counts toward the total number of nodes that must respond and/or participate in an event.

If, for example, you are using a durability guarantee that requires all nodes in the replication group to acknowledge a transaction commit on the Master, and if a node is down or otherwise unavailable for some reason, then the commit cannot complete on the Master because it will not receive acknowledgements from all the nodes in the replication group.

Similarly, elections for Masters require a bare majority of nodes to participate in the election. If so many nodes are shutdown or unavailable due to a network partition event that a bare majority of nodes cannot be found to hold the election, then your replication group can perform no write activities. This situation persists until at least enough nodes come back online to represent a bare majority of the nodes belonging to the replication group.

For this reason, if you have a node that you intend to shutdown for a long time, then you should remove that node from the replication group. You do this using the ReplicationGroupAdmin.removeMember() method. Note the following rules when using this method:

- For best results, shutdown the node before removing it.
- You use the node's name (not the host/port pair) to identify the node you want to remove from the group. If the node name that you specify is unknown to the replication group, a MemberNotFoundException is thrown.
- Once removed, the node can no longer connect to the Master, nor can it participate in elections. If you want to reconnect the node to the Master (that is, you want to add it back to the replication group), you will have to do so using a different node name than the node was using when it was removed from the group.
- An active Master cannot be removed from the group. To remove the active Master, either shut it down or wait until it transitions to the Replica state. If you attempt to remove an active Master, a MasterStateException is thrown.

For example:

```
...

    Set<InetSocketAddress> helpers =
        new HashSet<InetSocketAddress>();
    InetSocketAddress helper1 =
        new InetSocketAddress("node1.sleepycat.com", 1550);
    InetSocketAddress helper2 =
        new InetSocketAddress("node2.sleepycat.com", 1550);

    helpers.add(helper1);
    helpers.add(helper2);

    ReplicationGroupAdmin rga =
        new ReplicationGroupAdmin("test_rep_group", helpers);

    try {
        rga.removeMember("NODE3");
    } catch (MemberNotFoundException mnfe) {
        // Specified a node name that is not known to the
        // replication group.
    } catch (MasterStateException mse) {
        // Tried to remove an active Master
```

```
    }
```

Restoring Log Files

During normal operations, the nodes in a replication group communicate with one another to ensure that the JE cleaner does not reclaim log files still needed by the group. The tail end of the replication stream may still be needed by a lagging Replica in order to make it current with the Master, and so the replication group makes sure the trailing log files needed to bring lagging Replicas up-to-date are not reclaimed.

However, if a node becomes unavailable for a long enough period of time, then log files needed to bring it up to date will be reclaimed by the cleaner. The length of time that a node is unavailable before necessary log files are reclaimed is defined by REP_STREAM_TIMEOUT property, which you can manage using ReplicationConfig.setConfigParameter(). The default value is 24 hours.

Once log files have been reclaimed by a cleaner, then the Replica can no longer be brought up to date using the normal replication stream. Your application code will know this has happened when the ReplicatedEnvironment constructor throws an InsufficientLogException.

When your code catches an InsufficientLogException, then you must bring the Replica up-to-date using a mechanism other than the normal replication stream. You do this using the NetworkRestore class. A call to NetworkRestore.execute() causes the Replica to copy the missing log files from a member of the replication group who owns the files and seems to be the least busy. Once the Replica has obtained the log files that it requires, it automatically re-establishes its replication stream with the Master so that the Master can finish bringing the Replica up-to-date.

For example:

```
...
 try {
    node = new ReplicatedEnvironment(envDir, repConfig, envConfig);
} catch (InsufficientLogException insufficientLogEx) {

    NetworkRestore restore = new NetworkRestore();
    NetworkRestoreConfig config = new NetworkRestoreConfig();
    config.setRetainLogFiles(false); // delete obsolete log files.

    // Use the members returned by insufficientLogEx.getLogProviders()
    // to select the desired subset of members and pass the resulting
    // list as the argument to config.setLogProviders(), if the
    // default selection of providers is not suitable.

    restore.execute(insufficientLogEx, config);

    // retry
    node = new ReplicatedEnvironment(envDir, repConfig, envConfig);
} ...
```

Backing up a Replicated Application

In a stand-alone, non-replicated JE application, the log is strictly append only. You use the DbBackup class to help applications coordinate while database operations are continuing to add to the log. This helper class does this by defining the log files needed for a consistent backup, and then freezes all changes to those files, including any changes that might be made by JE background operations. The application can copy that defined set of files and finish operation without checking for the ongoing creation of new files. Also, there will be no need to check for a newer version of the last file on the next backup.

When you are using JE HA, however, log files other than the last log file might be modified as part of the HA sync-up operation. Though a rare occurrence, such modifications would invalidate the backup because there is the chance that files are modified after being copied.

If this happens, DbBackup.endBackup() throws a LogOverwriteException. Upon encountering this exception, the backup files should be discarded and a new set of backup files created.

For example:

```
        for (int i=0; i < BACKUP_RETRIES; i++) {
            final ReplicatedEnvironment repEnv = ...;
            final DbBackup backupHelper = new DbBackup(repEnv);

            backupHelper.startBackup();
            String[] filesForBackup =
                backupHelper.getLogFilesInBackupSet();

            /* Copy the files to archival storage. */
            myApplicationCopyMethod(filesForBackup);

            try {
                backupHelper.endBackup();
                break;
            } catch (LogOverwriteException e) {
                /* Remove backed up files. */
                myApplicationCleanupMethod();
                continue;
            } finally {
                repEnv.close();
            }
        }
```

Converting Existing Environments for Replication

JE HA environments log files contain information and data used only by replication. Non-replicated environments are lacking this information, so in order to use a previously-existing non-replicated environment in an HA application, it must undergo a one time conversion.

Note

If you try to open a non-replicated environment as a replicated environment, the operation will throw an UnsupportedOperationException. This is the only way your code can tell if an environment needs to be converted.

You use the DbEnableReplication class to perform this one-time conversion. This class is particularly useful if you want to prototype a standalone transactional application, and then add in replication after the transactional application is working as desired.

The conversion process is one-way; once an environment directory is converted, the rules that govern ReplicatedEnvironment apply. This means the environment can no longer be opened for writes by a standalone Environment handle (however, it still can be opened by a standalone Environment handle in read-only mode).

Note that DbEnableReplication only adds a minimum amount of replication metadata. The conversion process is not in any way dependent on the size of the environment you are converting.

The converted environment can be used to start a new replication group. After conversion, the environment can be opened as a ReplicatedEnvironment. Additional nodes that join the group are then populated with data from the converted environment.

For example:

```
// Create the first node using an existing environment
DbEnableReplication converter =
    new DbEnableReplication(envDirMars,          // env home dir
                            "UniversalRepGroup", // group name
                            "nodeMars",          // node name
                            "mars:5001");        // node host,port
converter.convert();

ReplicatedEnvironment nodeMars =
           new ReplicatedEnvironment(envDirMars, ...);

// Bring up additional nodes, which will be initialized from
// nodeMars.
ReplicationConfig repConfig = new ReplicationConfig();
try {
    repConfig.setGroupName("UniversalRepGroup");
    repConfig.setNodeName("nodeVenus");
    repConfig.setNodeHostPort("venus:5008");
    repConfig.setHelperHosts("mars:5001");

    nodeVenus = new ReplicatedEnvironment(envDirVenus,
                                          repConfig,
                                          envConfig);
} catch (InsufficientLogException insufficientLogEx) {
```

```
    // log files will be copied from another node in the group
    NetworkRestore restore = new NetworkRestore();
    restore.execute(insufficientLogEx, new NetworkRestoreConfig());

    // try opening the node now
    nodeVenus = new ReplicatedEnvironment(envDirVenus,
                                          repConfig,
                                          envConfig);
}
```

Chapter 5. Writing Monitor Nodes

So far in this book we have mostly discussed electable nodes, which are by definition nodes that have access to a JE ReplicatedEnvironment. However, replication groups can include any number of nodes that cannot be elected Master, because they have no access to the JE replicated environment in use by the replication group.

These type of non-electable nodes are usually called *monitor nodes*. The point of a monitor node is to allow a process to have some understanding of the replication group's structure such as which node is the Master and what nodes belong to the group as Replicas. Monitor nodes also have the ability to know when certain events have happened in the replication group, such as when a new Master is elected or when new nodes are added to, or deleted from, the group.

There are many uses for Monitor nodes, starting with the ability to write processes that monitor the current status of your HA application. But another, arguably more interesting, use for Monitor nodes is for request routing. As we have explained earlier in this book, Replicas can only service read-only requests; all write requests must occur on the Master. However, Replicas are only capable of noticing that they have been asked to process a write request. At most, out of the box, they can complain about it by throwing a ReplicaWriteException, and then completely rejecting the request.

One way to handle this problem is by writing an request router that sits on your network between the electable nodes and your clients. This router can send write requests to the Master, and read requests to the Replicas. A robust example of this sort of thing could also perform load balancing across the various Replicas, so that no one Replica becomes swamped by too many read requests.

Monitor Class

You implement Monitor nodes using the Monitor class. The `Monitor` class allows you to obtain information about the replication group, such as its name, where the Master is, and other such information. The `Monitor` class also allows you to run an event listener that can alert you to changes in the composition of the replication group.

You instantiate a `Monitor` class object in much the same way as you instantiate a ReplicatedEnvironment class object. It is necessary to give the node a name, to indicate that it is a Monitor node, to identify the node's host and port information, and to identify helper hosts. You use a ReplicationConfig object to do these things, just as you do when instantiating a ReplicatedEnvironment object.

Once the `Monitor` object has been instantiated, it must be registered at least once with the Master so that the replication group will know to keep the node informed about changes in the group composition. (Subsequent attempts to register the node are simply ignored by the Master.) You use the Monitor.register() method to register a Monitor node with a Master.

For example:

```
// Initialize the monitor node config
ReplicationConfig config = new ReplicationConfig();
```

```
config.setGroupName("MyRepGroupName");
config.setNodeName("mon1");
config.setNodeType(NodeType.MONITOR);
config.setNodeHostPort("monhost1.acme.com:7000");
config.setHelperHosts("node1.acme.com:5000,node2.acme.com:5000");

Monitor monitor = new Monitor(config);

// If the monitor has not been registered as a member of the
// group, register it now. register() returns the current node
// that is the master.
ReplicationNode currentMaster = monitor.register();
```

Listening for Events

One of the things the Monitor class allows you to do is to listen for certain events that occur in the composition of the replication group. Your Monitor can be notified of these events by running an event listener using Monitor.startListener(). For example:

`Monitor.startListener()` takes a single argument, and that is an instance of MonitorChangeListener. `MonitorChangeListener` is an interface that you implement for the purpose of handling replication group events.

There are four events that the change listener can be notified of. Each of these are represented by a unique class:

1. GroupChangeEvent

   A new instance of this event is generated each time a node is added or removed from the replication group.

2. NewMasterEvent

   A new instance of this event is generated each time a new Master is elected.

3. JoinGroupEvent

   A new instance of this event is generated each time a node joins a group. The event is generated on a "best effort" basis. It may not be generated, for example, if the joining node was unable to communicate with the monitor due to a network problem. The application must be resilient in the face of such missing events.

4. LeaveGroupEvent

   A new instance of this event is generated each time a node leaves the group. The event is generated on a "best effort" basis. It may not be generated if the node leaving the group dies (for example, it was killed) before it has a chance to generate the event, or if the node was unable to communicate with the monitor due to a network problem. The application must be resilient in the face of such missing events.

For example, an implementation of the MonitorChangeListener interface might be:

```
class MyChangeListener implements MonitorChangeListener {

    public void notify(NewMasterEvent newMasterEvent) {

     String newNodeName = newMasterEvent.getNodeName();

     InetSocketAddress newMasterAddr =
            newMasterEvent.getSocketAddress();
     String newMasterHostName = newMasterAddr.getHostName();
     int newMasterPort = newMasterAddr.getPort();

     // Do something with this information here.
    }

    public void notify(GroupChangeEvent groupChangeEvent) {
     ReplicationGroup repGroup = groupChangeEvent.getRepGroup();

     // Do something with the new ReplicationGroup composition here.
    }

    ...

}
```

You can then start the Monitor listener as follows:

```
 // Initialize the monitor node config
 ReplicationConfig config =
        new ReplicationConfig("MyRepGroupName",
                              "mon1",
                              "monhost1.acme.com:7000");
 config.setNodeType(NodeType.MONITOR);
 config.setHelperHosts("node1.acme.com:5000,node2.acme.com:5000");

 Monitor monitor = new Monitor(config);

 // If the monitor has not been registered as a member of the
 // group, register it now. register() returns the current node
 // that is the master.
 ReplicationNode currentMaster = monitor.register();

 // Start up the listener, so that it can be used to track changes
 // in the master node, or group composition.
 monitor.startListener(new MyChangeListener());
```

Chapter 6. Replication Examples

JE HA provides three different example programs that illustrate the concepts discussed in this manual. You can find them in your JE distribution in the `<JE HOME>/examples/je/rep/quote` directory, where <JE HOME> is the directory where you installed your JE distribution.

The examples provided for you are each based on a mock stock ticker application which stores stock values in a replicated JE environment. The differences in the three examples have to do with how each example handles requests for database access; in particular, database write requests.

Briefly, each of the examples are:

- StockQuotes: Illustrates the most basic demonstration of a replicated application. It is intended to help you gain an understanding of basic HA concepts. It demonstrates use of the HA APIs to create a replicated environment and issue read and write transactions.

  For this example, no attempt is made to route or forward write requests. Instead, the application blindly attempts any write requests that are made at the node. If the node is in the Replica state, a ReplicaWriteException is raised by the underlying HA code. The example then informs you of the problem by way of rejecting the operation.

- RouterDrivenStockQuotes and HARouter: Illustrates how a software load balancer might be integrated with JE HA, where HARouter plays the role of the load balancer for purposes of the example. It does this by using the Monitor class to direct application requests to the appropriate node. Read-only requests are sent to Replicas, while read-write requests are sent to the replication group's Master.

- UpdateForwardingStockQuotes and SimpleRouter: Illustrates the use of an HA unaware router that load balances read and write requests across the nodes in a replication group. The router is implemented in SimpleRouter, and is meant to illustrate how a load balancer appliance might fit into the JE HA architecture.

  This example is based on RouterDrivenStockQuotes.

Usage of each of these examples is described in the Javadoc page for each example.

Chapter 7. Administration

This chapter describes issues pertaining to running an JE replication application. The topics discussed here have to do with hardware configuration, backups, node configuration, and other management issues that exist once the application has been placed into production.

Hardware

A JE replicated application should run well on typical commodity multi-core hardware, although greater hardware requirements than this may be driven by the architecture of your particular application. Check with the software developers who wrote your JE replicated application for any additional requirements they may have over and above typical multi-core hardware.

That said, keep the following in mind when putting a JE replication application into production:

- Examine the hardware you intend to use, and review it for common points of failure between nodes in the replication groups, such as shared power supplied, routers and so forth.

- The hardware that you use does not have to be identical across the entire production hardware. However, it is important to ensure that the least capable electable node has the resources to function as the Master.

  The Master is typically the node where demand for machine resources is the greatest. It needs to supply the replication streams for each active Replica, in addition to servicing the transaction load.

  Note that JE requires Monitor nodes to have minimal resource consumption (although, again, your application developers may have written your Monitors nodes such that they need resources over and above what JE requires), because Monitor nodes only listen for changes in the replication group.

- Finally, your network is a critical part of your hardware requirements. It is critical that your network be capable of delivering adequate throughput under peak expected production work loads.

  Remember that your replicated application can consume quite a lot of network resources when a Replica starts up for the first time, or starts up after being shutdown for a long time. This is because the Replica must obtain all the data that it needs to operate. Essentially, this is a duplicate of the data contained by the Master node. So however much data the Master node holds, that much data will be transmitted across your network *per node* every time you start a new node.

  For restarting nodes, the amount of data that will cross your network is equal to the delta between the time the Replica last shutdown and the state of your Master node at the time that the Replica is starting up again. If the Replica has been down for a long time (days or weeks), this can be quite a lot of data, depending on your Master node's workload.

Be aware, however, that restarting nodes do not have to get their data from the Master node. It is possible for them to catch up, or nearly catch up, using data obtained from some other currently running Replica. See Restoring Log Files (page 57) for more information.

Time Synchronization

For best results, you are strongly recommended to synchronize the clocks on all the machines participating in your production replication group. Running a time synchronization daemon like NTPD is a simple way to keep time synchronized across your replication machines. Once the clocks are set, they are maintained by ntpd so that they rarely stray more than 128ms away from one another.

Be aware the JE checks for clock skew between the Master and a starting Replica node, when the Replica node performs its startup handshake with the Master. (See Replica Startup (page 9) for information on the startup handshake.) If the clock skew between the two nodes is too large, the handshake is aborted and JE throws an EnvironmentFailureException.

Also, well-synchronized clocks are required for a proper implementation of a time consistency policy (see Time Consistency Policies (page 35)). It is also required for correct internal booking by JE.

Finally, synchronized system clocks make it easier to correlate events in the logging output from different nodes in the group.

Node Configuration

When you place a node into service, there is a set of information that you must provide which will be unique to each and every node. The application development team may or may not have provided defaults for some or all of these values, so you should check with them to see exactly what you need to override.

This information can be provided to the application in two different ways. One is by using JE API calls. Typically you will pass the information to those calls using command line parameters. Again, how you do this is specific to your application.

In addition, you can provide this information to the application using the `je.properties` file. Note that the information provided in this file is handled as if it is a *default* setting. Therefore, if you also provide conflicting information using the JE APIs (again, usually passed to a production application using command line parameters), then the information provided directly to the APIs takes priority over whatever might be found in the `je.properties` file.

No matter how it is done, there are three pieces of information that you must provide every JE replicated application:

- Group Name

  This is the replication's group name. This value must be the same for every node participating in a given replication group. This name must be made up of alpha numeric characters and must not be zero length.

JE developers can provide this information to the application using the `ReplicationConfig.GROUP_NAME` field. In the `je.properties` file, it is defined using the `je.rep.group.name` parameter.

- Node Name

  This is the name of the node. This name must be unique within the group. This name combined with the group name uniquely identifies the node.

  JE developers can provide this information to the application using the `ReplicationConfig.NODE_NAME` field. In the `je.properties` file, it is defined using the `je.rep.node.name` parameter.

- Node Host

  This is the hostname and port pair that is used by other nodes in the replication group to communicate with this node. The node uses this property to establish a TCP/IP socket for communication with other members of the group.

  The string that you provide to this property takes the form:

  ```
  hostname[:port]
  ```

  The hostname provided to this property must be reachable by the other nodes in the replication group.

  The port number is optional for this property because a default port can be defined using the `je.properties` file (you use the `je.rep.defaultPort` property to do this). However, if a port is provided explicitly to this property, then `je.rep.defaultPort` is ignored.

  Be careful to ensure that the port you identify for the node does not conflict with ports used by other applications (including other nodes, if any) currently running on the local machine.

  Note that monitor nodes will use the socket identified by this property so that they can be kept informed of the results of elections, and so they can keep track of changes in group composition.

  Electable nodes use this socket to:

  - Hold elections
  - Establish replication streams between the Master and its Replicas
  - Support network-based JE HA utility services, such as JE's network restore utility. (See Restoring Log Files (page 57) for details on this utility.)

The properties discussed here are simply the bare-bones minimum properties required to configure a JE node. For a complete description of all the replication properties available to a JE application, see the ReplicationConfig and ReplicationMutableConfig class descriptions.

Running Backups

Because JE replication causes a current copy of your environment to be available at every node in the group, the need for frequent backups is greatly reduced. Basically, every time a change is made on the Master, that change is backed up to every Replica node currently running. The result is that for each write operation you get a real-time incremental backup to *n-1* nodes, where *n* are the total number of nodes (including the Master) currently running in your replication group.

For this reason, JE does not currently support formal incremental backups of replicated environments. An application based upon the DbBackup utility class can be written to allow administrators to create full backups. This is useful for creating a backup to be stored on offline media, if your data strategy calls for that level of protection.

Remember that when performing a full backup, you should obtain the backup from a node that is current. Either use the Master node itself, or use a Replica node that must acknowledge a transaction commit before the commit operation can complete on the Master.

Note that DbBackup has some functionality that is specifically useful for replicated environments. See Backing up a Replicated Application (page 58) for details.

Adding and Removing Nodes

As described in Adding and Removing Nodes from the Group (page 55), a node is added to the replication group simply by starting it up and allowing it to perform its start-up handshake with the Master. Once a node has been added to the replication group, it belongs to the replication group forever, or until you explicitly remove it. Also, the node is uniquely identified within the replication group by a name that you must give it when you start up the process.

This is worth remembering, because if you have nodes that have been added to the replication group, but which you then shutdown for a long period of time, your replication group might not be able to perform a lot of tasks, such as:

1. Elect a Master.
2. Add a new node to the replicated group.
3. Delete a node from the replication group.
4. Successfully commit a transaction (this depends on the durability guarantees in place for your application).

All of these actions might be adversely affected by a series of unavailable nodes because in order to do these things the Master must be in contact with a majority of the electable nodes belonging to the replication group (Monitor nodes do not count). So if too many nodes are either shutdown or unavailable due to a network partition event, then these functions can become delayed or even completely unavailable.

For this reason, if you have nodes that you want to shutdown for a long time, then you should remove those nodes from the replication group. JE provides a utility class that allows for node

removal, so your application developer should have provided you with a tool of some kind that allows you to do this as a normal administrative function.

When removing a node from the replication group, remember that:

- for best results, shut down the node first.
- a majority of the nodes must currently be in contact with the Master in order to acknowledge the node removal.

If at some later time you want to restart the node and have it join the replication group, you can do this using the normal procedure that your application uses when starting a node for the first time. Be aware, however, that you cannot reuse the unique name that the node was using when you removed it from the group. Instead, give the node a completely new unique name before having it rejoin the replication group.

Appendix A. Managing a Failure of the Majority

Normal operation of JE HA requires that at least a simple majority of electable nodes be available to form a quorum for election of a new Master, or when committing a transaction with default durability requirements. The number of electable nodes (the Electable Group Size) is obtained from persistent internal metadata that is stored in the environment and replicated across all members. See Replication Group Life Cycle (page 7) for details.

Under exceptional circumstances, a simple majority of nodes may become unavailable for some period of time. With only a minority of nodes available, the overall availability of the group can be adversely affected. For example, the group may be unavailable for writes because a master cannot be elected. Also, the Master may be unable to satisfy the durability requirements for a transaction commit. The group may also be unavailable for reads, because the absence of a Master might cause a Replica to be unable to meet consistency requirements.

To deal with this exceptional circumstance — especially if the situation is likely to persist for an unacceptably long period of time — JE HA provides a mechanism by which you can modify the way in which the number of electable nodes, and consequently the quorum requirements for elections and commit acknowledgments, is calculated. The escape mechanism provides a way to override the normal computation of the Electable Group Size. The override is accomplished by specifying the size using the mutable replication configuration parameter ELECTABLE_GROUP_SIZE_OVERRIDE.

Note

You should use this parameter sparingly, if at all. Overriding your Electable Group Size can have the consequence of allowing your replication participants to elect two Masters simultaneously. This is especially likely to occur if a majority of the nodes are unavailable due to a network partition event, and so all nodes are running but are simply not communicating with one another.

Be very cautious when using this configuration option.

Overriding the Electable Group Size

When you set ELECTABLE_GROUP_SIZE_OVERRIDE to a non-zero value, the number that you provide identifies the number of nodes that are required to meet quorum requirements. This means that the internally stored Electable Group Size value is ignored (but not changed) when this option is non-zero. By setting ELECTABLE_GROUP_SIZE_OVERRIDE to the number of nodes known to be available, the remaining replication group participants can make forward progress, both in terms of electing a new Master (if this is required) and in terms of meeting durability and consistency requirements.

When this option is zero (0), then the node will behave normally, and the internal Electable Group Size is honored by the node. This is the default value and behavior.

Setting the Override

To override the internal Electable Group Size value:

1. Verify that the simple majority of nodes are in fact down and cannot elect their own independent Master.
2. Set ELECTABLE_GROUP_SIZE_OVERRIDE to the number of electable nodes known to be available. For best results, set this override on all available nodes.

   It might be sufficient to set ELECTABLE_GROUP_SIZE_OVERRIDE on just one node in order to hold an election, because the proposer at that one node can conclude the election. However, if the election results in Master that is not configured with this override, it might result in InsufficientAcksExceptions at the Master. So, again, set the override on all available nodes.

Having set the override, the available members of the replication group can now meet quorum requirements.

Restoring the Default State

Having restored the group to a functioning state by use of the ELECTABLE_GROUP_SIZE_OVERRIDE override, it is desirable to return the group to its normal state as soon as possible. The normal operating state is one where the Electable Group Size is maintained by JE HA, and the override is no longer used.

To restore the group to its normal operational state, do one of the following:

- Remove from the group any electable nodes that you know will be down for an extended period of timed. Remove the nodes using the ReplicationGroupAdmin.removeMember() API.
- Bring up electable nodes as they once again come on line, so that they can join the functioning group. This must be done carefully one node at a time in order to avoid the small possibility that a majority of the downed nodes hold an election amongst themselves and elect a second Master.
- Perform some combination of node removal and bringing up nodes which were previously down.

As soon as there is a sufficient number of electable nodes up and running that election quorum requirements can be met in the absence of the override, the override can be removed, and normal HA operations resumed.

Override Example

Consider a group consisting of 5 nodes: `n1-n5`. Suppose a simple majority of nodes (`n3-n5`) have become unavailable.

If one of the nodes, `n3-n5`, was the Master, then nodes `n1` and `n2` will try to hold an election, and fail due to the lack of a quorum. We now carry out the steps described, above:

1. Verify that `n3-n5` are down.

2. Set ELECTABLE_GROUP_SIZE_OVERRIDE to 2. Do this at both `n1` and `n2`. You can do this dynamically using JConsole, or by setting the property in the `je.properties` file and restarting the node.

3. `n1` and `n2` will choose a new Master, say, `n1`. `n1` can now process write operations, and `n2` can acknowledge transaction commits.

4. Suppose that `n3` is now repaired. You can bring it back online and it will automatically locate the new Master and join the group. As is normal, it will catch up to `n1` and `n2` in the replication stream, and then begin acknowledging commits as requested by `n1`.

5. We now have three nodes that are operational. Because we have a true simple majority of nodes available, we can now reset ELECTABLE_GROUP_SIZE_OVERRIDE to 0 (do this on `n1` and `n2`), which causes the replication group to resume normal operations. Note that `n1` remains the Master.

If `n2` was the Master at the time of the failure, then the situation is similar, except that an election is not held. In this case, `n2` will continue to remain the Master throughout the entire process described above. However, `n2` might not be able to meet quorum requirements for transaction commits until step 2 (above) is performed.