Unleash High Availability Applications with Berkeley DB

1. Introduction

Berkeley DB High Availability (BDB-HA) is a replicated, embedded database management system designed to provide applications fast, reliable, and scalable data management. Due to its implementation as an embedded library, BDB-HA is used in a large range of different configurations and environments. This paper presents the key characteristics of BDB-HA, discussing how best to use it to solve specific data management problems. We discuss the key technical trade-offs that an architect must consider when designing an application.

BDB-HA builds upon the Berkeley DB Transactional Data Store (BDB-TDS) product. BDB-TDS is a library that links directly with an application to provide fast, reliable key-data storage. Applications use BDB-TDS to create indexed databases, using one of hashed, btree, or record-number based structures. Multiple operations on databases and key/data pairs may be wrapped in transactions to provide the conventional transactional ACID properties of atomicity, consistency, isolation, and durability. After an application or system failure, Berkeley DB's recovery mechanism restores the application's data to a consistent state.

Berkeley DB uses write-ahead logging to provide transactional consistency. Before updates are applied directly to data, they are first written to a separate log file. These log files provide the foundation of BDB-HA. As transactions commit, BDB-HA transmits log files from a designated master site to other sites in a replication group, which immediately apply the transaction, providing online data replication.

The BDB-HA architecture supports *single-writer, multiple-reader* replication. This means that all database updates must be performed on a single site (the master), but replicas are available for read activity. Should the master fail, one of the replicas must take over as master, and all writes will then be serviced by the new master.

BDB-HA can be used to address three different application challenges: it provides rapid failover in situations where downtime is unacceptable, it provides read scalability by providing multiple read-only replicas, and it enables more efficient commit processing, allowing applications to provide durability by committing to the (fast) network, rather than to a (slow) disk. BDB-HA, however, does not assist applications that need write-scaling. Such applications will need to partition the data, running a BDB-HA environment per partition. It also does not provide synchronization between BDB-HA and other databases. These last two issues are not addressed in this white paper.

BDB-HA is appropriate in a wide range of environments from replication across a backplane to replication across the WAN. We introduce four sample use cases here that demonstrate the range of applicability of BDB-HA.

- Small-scale LAN-based replication: Consider a data-center based service providing data for a local population, such as a corporate site. Such a service might be implemented by a few servers, all residing in the same data center, communicating over a high-speed LAN.

- World-wide data store: Alternately, consider a world-wide service provider that needs to store account information, accessible anywhere in the world. This service might be implemented by a collection of servers, residing in different data centers, scattered around the globe, communicating via the Internet.

- In-memory replication across a backplane: Imagine a massive network switch containing dozens of boards, each of which independently handles network requests. The data for such a box could be stored in databases replicated across the switch's backplane.

- Master/Slave: Finally, consider a simple installation to provide failover between a master and slave machine. Such a 2-node configuration is possible using BDB, but it introduces interesting design challenges that will be discussed later in this paper.

We will refer back to these use cases as we describe different features of BDB-HA, illustrating how the different scenarios demand different capabilities from the underlying data management system.

Given the wide range of environments across which Berkeley DB replicates data, no single set of data management policies is appropriate. BDB-HA provides applications control over the degree of data consistency between the master and replicas, what constitutes transactional durability, how frequently participants interact, which meta-data structures are maintained in memory and on-disk and a wide range of other design decisions that will dictate the performance, robustness, and availability of the data.

The goal of this paper is to introduce those decisions and provide designers a sufficiently in-depth understanding of the issues, enabling them to make decisions appropriate for a particular application.

2. Terminology

Throughout the rest of this paper, we will use the following terms to describe using Berkeley DB HA.

Key/Data Pair: The basic unit of storage in Berkeley DB. Keys are opaque byte strings used as primary indices. Each key has another opaque byte string, the data, associated with it.

Database: A collection of key/data pairs sharing an indexing structure. This is a table in RDBMS terminology.

Environment: A logical collection of Berkeley DB databases and their associated meta-data and infrastructure files. A directory names an environment. This is a database in RDBMS terminology.

Cursor: Represents a logical position in a database; used for iteration.

Site: A machine or node participating in a replicated environment.

Replication group: The collection of sites replicating a single environment.

Master: The site in a replication group that is currently capable of accepting and processing application write requests.

Replica: A non-master site in a replication group. In the Berkeley DB documentation and APIs, these sites are referred to as clients. In this paper, we use the term replica to clearly differentiate sites running Berkeley DB database code from application agents making requests of the data store.

Electable Peer: A site that is eligible to be elected master. Some applications find it useful to replicate to sites for read access, but due to capacity or connectivity constraints do not want those sites to ever become masters. Such sites are non-electable peers.

Quorum: The number of sites necessary to reach a decision. There are different sized quorums for different purposes, such as master election or acknowledging a successful write.

Internal Initialization: The process of copying log files and databases to a replica when there are not enough logs available to bring the replica up-to-date.

3. Designing your Application Architecture

This section provides an overview of the key features of Berkeley DB that will most directly influence your application architecture. We begin by defining what we mean by "embedded data management" and what benefits this embedding provides to your application. Next, we discuss the two replication APIs available for Berkeley DB.

In most cases, applications will use the Replication Manager API, which provides a high-level, more convenient interface. Next we talk about how the master site is determined. In most cases, we encourage applications to use the Berkeley DB election mechanism, although it is also possible to integrate BDB-HA with other HA substrates. Finally, we discuss how to design applications to integrate with BDB-HA's single-writer replication model.

Embedding data management in your application

We use the term "embedded" when talking about Berkeley DB to refer to the relationship between the application and the data management capabilities. Those capabilities, as realized by the Berkeley DB library, are embedded within the application; application users or administrators need not be aware of the existence of Berkeley DB, as Berkeley DB is used and administered directly from the application. Such an application may itself run on an embedded device, but it need not. It might run on a handheld device, on a desktop machine, or in a datacenter. In fact, Berkeley DB might provide storage-tier

functionality, being used as the storage engine for data management services to a family of applications.

Embedded data management provides several benefits over the more traditional client-server architecture. First, applications can store and retrieve data in their native form, rather than translating to and from a different (e.g., relational) data model. Applications can serialize structures or complex objects into Berkeley DB databases directly.

Second, embedded data management requires no database administrator (DBA). Instead, applications incorporate database configuration and management directly into their own configuration and management, typically resulting in a few new knobs and settings for the application.

Third, the end-user does not observe any separate database recovery phase. Instead, during application startup, the application can open and/or recover its databases by issuing API calls to the Berkeley DB library. Similarly, when the application shuts down, it can shut down the database as well.

Fourth, embedded data management frequently provides performance superior to that of traditional client/server architectures by avoiding interprocess communication overhead.

Picking an API: Replication Manager or Base API

Berkeley DB provides two different APIs for application developers. The Replication Manager is an easier-to-use, higher-level interface for developers, but dictates certain design decisions. The Base API exposes the full power of Berkeley DB, but requires that application developers write more code. If your deployment infrastructure matches the characteristics of the Replication Manager, it is the recommended API, and we strongly encourage you to use it.

The Replication Manager is designed to facilitate quicker and easier application development. It accomplishes these goals in two ways: supplying standard communication infrastructure and thread management implementations and exposing configuration at a semantically higher level than the Base API.

Factors Enabling the use of the Replication Manager

On POSIX platforms, the Replication Manager uses pthreads; on Windows platforms, the Replication Manager uses Windows threads. If applications also use threads, they must use the matching thread mechanism. Furthermore, applications running platforms supporting neither pthreads nor Windows threads will have to use the Base API. The Replication Manager constrains applications to use TCP/IP sockets for the underlying communication infrastructure. The Base API relies on the application to supply a communication infrastructure.

The Replication Manager can provide heartbeat capabilities, so that Berkeley DB automatically detects failed sites. Applications using the Base API that need this capability must provide it themselves. When the master site fails, the Replication Manager by default automatically holds an election to select a

new master, while in the Base API, the application may discover that it can no longer communicate with the master. It is then up to the application to explicitly call for the election.

Applications must manage the resources consumed by the Berkeley DB log by periodically removing older log files that are no longer needed. Berkeley DB has an automatic log reclamation feature to automate this process. The Replication Manager is integrated with this feature but the Base API is not.

Replication Manager and Base API Comparison

	REPLICATION MANAGER	**BASE API**
Threads	Must use pthreads or Windows threads	Any thread package
Communication Protocol	TCP/IP	Any protocol
Heartbeats	Provided	Application implements
Elections	Automatically invoked	Application must invoke
Log Reclamation	Can use Berkeley DB automatic log reclamation	Application must manage

Configuration in Replication Manager versus Base API

In the configuration space, the Replication Manager provides policy-based configuration, while the Base API tends to support configuration through explicit parameter settings. For example, the Replication Manager provides site management operations -- functions that add and remove sites from a group or return the list of sites in a group. The Base API provides no comparable functions -- instead, applications either have a priori knowledge of the participants in a replication group or they become aware of new sites via return codes from BDB-HA methods (e.g., the **`DB_REP_NEWSITE`** return from a call to **`DB_ENV->rep_process_message`**), and it is up to the application to use that information appropriately.

Similarly, the Replication Manager provides configuration at a higher policy level while the Base API provides lower level mechanisms that require the application to implement its own policies. For example, on a transaction commit, applications can decide how many sites need to acknowledge the commit, before considering it durable. In the Replication Manager, that choice is expressed as an "Ack policy" where applications specify one of: ALL (must hear from all sites), ALL_AVAILABLE (must hear from all currently available sites), ALL_PEERS (must hear from all electable peers) NONE (commit immediately without waiting for any acknowledgements), ONE (must hear from a single site), ONE_PEER (must hear from a single electable peer), or QUORUM (must hear from sufficiently many sites to assure durability in the presence of any single failure). For example, in a replication group containing three nodes, the QUORUM policy would require the master to receive at least one acknowledgement to consider a transaction durable. Using the Base API, applications must explicitly know the number of sites participating in the group and the number from which it must receive acknowledgements in order to consider a transaction durable. This decision is encapsulated in the **rep_send** function that the application specifies when it sets up its communication infrastructure.

In general, we encourage applications to use the Replication Manager, resorting to the Base API only when application needs dictate its use.

Master selection: elected or designated?

Both the Replication Manager and the Base API allow applications to explicitly designate a master site. We strongly suggest designers use elections unless your application is running on an HA substrate (e.g., Neverfail's Application Management Framework or IBM's FileNet P8 Systems) that already performs leader election. For example, in the in-memory use case where we are replicating across a backplane, there may be other HA services required, and it may be more appropriate to use a separate HA substrate for both Berkeley DB master election and those other services. However, in the absence of such constraints, Berkeley DB's election algorithm can be tuned to work in concert with master leases (discussed later in this paper) and other BDB techniques that ensure data and application consistency. Designating a new master runs the risk of data or transaction loss if a site that does not have the most up-to-date data is appointed master.

Applications that wish to use elections, but also want control over which sites are selected can use BDB node priorities to achieve this control (**`DB_ENV->rep_set_priority`**). Priorities are relative, so the application can assign values it deems appropriate. The Berkeley DB election algorithm will first select sites eligible to become master given the current state of the system and will then examine the priorities to select the site with the highest priority from those eligible sites. Thus, priorities do not guarantee which site will be elected, but when possible, a higher priority site will be selected over a lower priority site. For example, in either the WAN- or LAN-based scenarios it is common to have machines with different capabilities. In most cases, it is desirable to have the master be one of the most powerful machines in system resources.

Applications that cannot use elections and, instead, designate the master site must be sure to communicate this designation to all participants. A site is started as the designated master via the **`DB_ENV->repmgr_start`** or **`DB_ENV->rep_start`** method with the flags parameter set to **`DB_REP_MASTER`**. Since all write requests must be directed to the master, it is important for any site that might initiate an update to know the identity of the current master. The recommended way to monitor this information is to use Berkeley DB's event notification. Applications should call **`DB_ENV->set_event_notify`** to configure an event callback function. When that callback function receives events of type **`DB_EVENT_REP_NEWMASTER`**, the application will learn the identity of the current master. The application can assume that site is still master until it receives another **`DB_EVENT_REP_NEWMASTER`** event or a **`DB_EVENT_REP_MASTER`** event, indicating that this site is the new master.

Write Shipping

Regardless of how a master site is selected, it is the application's responsibility to submit all database requests that modify the database (key/data pair insert, delete, overwrite and database create, rename, truncate, or remove) to the master.

There are three main ways that an application might direct requests to the master:

1. Every site keeps track of the identity of the master site and directly communicates its requests to the master.

2. The application introduces an intermediary dispatcher that directs requests to the master.

3. Every participant uses the established communication infrastructure to transmit an application message to whatever site is the current master.

There are fundamentally two different ways that applications architect their data management using BDB- HA, the integrated model and the data-tier model.

In the integrated model, all the application logic runs on the same sites as the Berkeley DB logic. Thus, the application logic that generates write requests resides on a site also running BDB-HA and using the event notification mechanism described above, sites can track the identity of the master (thus implementing option 1 above). Let's say that site R is a replica and site M is a master. Now, let's say that the application logic on site R requires a database update. Site R must, using an application communication layer, send a message to site M requesting that the master make the update. If, upon receiving and attempting to process the request, M discovers that it is no longer a master, M must reply to R with an error, indicating that R needs to determine the identity of the new master. Such errors should happen rarely.

In the data-tier model, application logic runs on a set of sites separate from those managing the data and running Berkeley DB. This architecture might arise in a three-tier architecture in the LAN or WAN scenarios where application logic runs in an AppServer running on one or more sites different from the collection of sites implementing the data tier. In this scenario, we assume that BDB-HA runs on the data tier, but not on the application-servers. In this case, the typical solution is to implement a distributor that directs requests from the AppServer(s) to an appropriate data-tier site, thus implementing option 2.

The distributor must keep track of the identity of the master and route all write requests to that master (it might also act as a load balancer, distributing read requests to the different nodes in an efficient manner).

There are two standard ways to build the distributor. The distributor could, itself, be a BDB-HA replica that uses the BDB event notification to keep track of the master. In this scenario, the distributor is frequently implemented as a non-electable replica. Alternately, the sites in the data-tier could explicitly notify the distributor of master changes. In this scenario, the distributor does not need to participate in the replication group.

Option 3 is implemented in the Replication Manager by sharing the Replication Manager's internal message channels. An application does this by using **`DB_ENV->repmgr_channel`** to create a **`DB_CHANNEL`** and implementing a message handler at each site. A **`DB_CHANNEL`** can communicate

with the current master or another site. An application sends its own messages to a **DB_CHANNEL** asynchronously using **DB_ENV->send_msg** or synchronously using **DB_ENV->send_request**.

Integrated applications using the Base API will find option 3 the simplest to implement and use, as described in this paragraph. The Base API replication communication infrastructure is implemented by a function provided by the application in the **DB_ENV->set_rep_transport** method. The **rep_send** function parameter to that method is called by the Berkeley DB library to transmit BDB-HA messages. In doing so, the function can bundle those messages however it sees fit. In particular, it can tag messages as being BDB messages or application messages. Application code is also responsible for receiving messages on this channel and calling **DB_ENV->rep_process_message** on those messages generated by BDB-HA. However, if the received message is an application message, it can be dispatched to application code capable of performing the requested operation and replying appropriately to the originator.

4. Application Nuts and Bolts

Configuring your Application

Replicated applications must take care to configure both the replication system and the underlying transactional data store components, to provide appropriate reliability and durability guarantees. In this section, we focus on those transactional data store configuration options that interact most closely with replication.

Transactions

First, all replicated applications must use the transaction subsystem. Every update operation must be performed in the context of a transaction, although read-only operations need not be transactional. Since replicated applications are using the transaction system, designers should ensure that they have sized the transaction system appropriately. See, **DB_ENV->set_tx_max**, **DB_ENV->set_timeout**, **DB_ENV->set_lk_max_lockers**, **DB_ENV->set_lk_max_locks**, **DB_ENV->set_lk_max_objects**. Each site in the group is responsible for configuration. All sites must specify the same log file size and whether log files are on-disk or in-memory.

Second, since replicated transactions obtain reliability through replication, it is often the case that sites participating in replication groups can weaken their local durability guarantees, letting log writes go to disk asynchronously rather than synchronously. For example, in the in-memory backplane use case, there is no persistent store to which to sync data. But even in the LAN- and WAN-based scenarios, if data is replicated sufficiently many times, it is frequently unnecessary to force data to disk on the master. See the documentation on **DB_TXN_NOSYNC** and **DB_TXN_WRITE_NOSYNC**.

Third, many transactional applications use the Berkeley DB automatic log reclamation feature (using **DB_ENV->log_set_config(DB_LOG_AUTO_REMOVE)**). Replication Manager applications are free to use this feature because the Replication Manager prevents removal of log files still needed by

any site in the replication group. However, most Base API applications should not do so, because the Base API does not prevent removal of log files needed by other sites. This makes it less likely that sites can synchronize with a master without copying all the databases and log files. However, an application that does not perform automatic log reclamation will have to provide its own manual log reclamation process to avoid having log files consume too much disk space. Replicated applications can safely remove log files from the master site when all replicas have received the log records in those logs. Most Base API applications simply enforce a policy to save several most recent log files. Unless transactions are quite long, spanning multiple log files, this approach is usually sufficient to ensure that replicas can restart without having to copy all the databases and log files.

Applications that want finer control over their log files typically have a monitoring process on the master that periodically asks replicas to report the results of running the **`log_archive`** utility, which produces a list of log files eligible for removing. The monitor then instructs all the sites that it is safe to remove the intersection of the set of log files returned by each site.

In-Memory Execution

As described in the backplane use case, another option available to replicated applications is to run with some or all of their data and meta-data in memory. Since replicas keep identical copies of the data, losing a fully in-memory site does not mean losing all the data, because it persists on another site.

There are four different types of data, each of which may be maintained in main memory:

1. replication meta-data,
2. application databases,
3. the shared memory pool, and
4. log files.

When all four of these types of data are configured to reside in memory, the application runs without any persistent storage. However, it is possible to pick and choose which types of data are only in memory and which are on persistent storage.

Applications can configure replication to store its meta-data in-memory using the **`DB_ENV->rep_set_config`** method specifying **`DB_REP_CONF_INMEM`**.

Creating application databases in main memory is easy -- simply use a **`NULL`** file name, with non-null database names.

Single process applications can place the memory pool in-memory by specifying the **`DB_PRIVATE`** value to the flags argument to **`DB_ENV->open`**. Alternatively, an application can specify the **`DB_SYSTEM_MEM`** value to the flags parameter of **`DB_ENV->open`** so multiple processes can share

an in-memory memory pool. **DB_PRIVATE** causes Berkeley DB to allocate regions in heap space, thus restricting the application to a single-process. **DB_SYSTEM_MEM** causes Berkeley DB to allocate regions in System V shared memory segments. Although using **DB_SYSTEM_MEM** requires creation of a Berkeley DB environment "file," this file need not persist across operating system reboots; it can be stored in a memory-based file system. It serves as a mechanism to allow multiple processes to share the same shared memory key, so that if applications crash, but the operating system remains up, applications can still clean up the shared memory segments managed by the operating system.

Finally, applications can use **DB_ENV->log_set_config(DB_LOG_IN_MEMORY)** to tell Berkeley DB to write log files into memory buffers rather than saving them on disk.

Applications using in-memory logs still need to perform routine checkpoints. While checkpoints traditionally cause data to be written to disk, in the context of in-memory logging, checkpointing ensures that we always have a current mapping from database names to log file identifiers, which is crucial for correct operation. Furthermore, the Berkeley DB log buffer, configured via the **DB_ENV->set_lg_bsize**, must be large enough to hold all log records up to and including a checkpoint. Thus, the checkpoint interval and log buffer size are tightly coupled. For this reason, we suggest that applications invoke checkpoints after a specified amount of log data has been written. This is accomplished by specifying a non-0 value for the **kbyte** parameter to the **DB_ENV->txn_checkpoint** method.

Placing each of these types of data in main memory leads to different limitations. When the replication meta-data is stored in-memory, it is possible to lose a recently committed transaction in extremely rare circumstances when sites are repeatedly crashing and restarting. When databases are stored in-memory, a site coming up after a crash will need to copy all the databases from another site.

When database logs are stored in-memory, sites cannot run recovery and will also need to copy databases from another site after failure.

Transactional Consistency

Transactions provide the ACID properties of atomicity, consistency, isolation, and durability. Atomicity means that for a given set of operations grouped in a transaction, either all the operations complete or none of them do. Consistency means that a transaction moves the data from one consistent state to another. Isolation means that each transaction runs as if there were no other transactions running concurrently in the system, and durability means that once a commit returns, the transaction is guaranteed to persist, even in the presence of failure.

On a single machine, these semantics are easy to understand and interpret, but in a replicated environment, things become more complicated. For example, while a transaction might be durable on a single site, if that site crashes, and another site that never saw that transaction takes over, then from the point of view of the replication group, the transaction was not durable. Berkeley DB's transactional data store product allows applications to trade off durability guarantees for performance; BDB-HA also provides applications additional flexibility to make such trade-offs.

The degree of synchronization that an application enforces between a master and the replicas determines the precise consistency semantics that an application will experience. On one extreme, using the Base API, an application could write a transport function that does not return from processing a permanent log record until all the sites in the replication group have acknowledged receipt and application of the record (comparable to the Replication Manager's **DB_REPMGR_ACKS_ALL** policy except that the Replication Manager only waits up to a time limit for acknowledgements). This ensures that the master never considers a transaction committed until it is also committed at all the replicas. This policy might be feasible in the LAN-based or backplane use cases, but is undoubtedly totally unacceptable in the WAN-based scenario. While providing strong consistency, this approach suffers a potentially significant delay during commit processing and means that forward progress stalls in the presence of any site failures. This extreme approach is practically never used in practice.

On the other extreme, a master could consider a transaction successfully committed once it has committed on the master and sent the message to the replicas (comparable to the Replication Manager's **DB_REPMGR_ACKS_NONE** policy). This provides the best responsiveness to the application, but it runs the risk that an update on the master could be lost if it and some of the replicas crash at the wrong point in time.

Most applications choose some point in between these two extremes. In these intermediate cases, the system sets some number of sites from which it insists upon receiving an acknowledgement before considering a transaction durable. The number of sites required to guarantee that a committed transaction is never undone is called a quorum.

Berkeley DB allows applications to specify quorums for different behaviors: write/commit, consistent reads (via master leases) and elections. Write and election quorums provide the guarantee that committed updates are never lost, while read quorums, as implemented via Berkeley DB master leases, provide a way to ensure that reads on a master site never return data that has not been replicated sufficiently to guarantee its durability.

Quorums

First, we discuss write quorums and then discuss the interaction between write quorums and election quorums in the next section. We conclude with a discussion of master leases.

Quorums on Writing

If applications want to make guarantees about the durability of their transactions in the face of node failures, then masters will want to receive acknowledgements from some minimum number of replicas before considering a transaction to be durable. Using the Replication Manager framework, applications can specify **DB_REPMGR_ACKS_QUORUM** to the **DB_ENV->repmgr_set_ack_policy** method to indicate that the master should not consider a transaction to be durably replicated until it has received responses from sufficiently many sites (at least half of the electable peers). If the master receives insufficient acknowledgements, the Replication Manager generates a

DB_EVENT_REP_PERM_FAILED event. Selecting a policy requiring fewer than the quorum runs the risk of losing updates after a master crashes (even after the master comes back up, the update will be undone when the old master synchronizes with a new master that never got the update).

Applications using the Base API implement write quorums by coordinating code in the **rep_send** function specified in the **DB_ENV->set_rep_transport** method and with logic in their application's message processing loop. The send function must block until it knows that the message processing loop has processed sufficiently many acknowledgements from the replicas. In this case, replicas must notify masters when they have successfully applied log records by sending an explicit acknowledgement when a call to **DB_ENV->rep_process_message** returns **DB_REP_ISPERM**. If **DB_ENV->rep_process_message** returns **DB_REP_NOTPERM**, the replica has not applied the current log record and therefore, has not committed the transaction. In this case, the replica should not transmit an acknowledgement to the master, but it should record the Log Sequence Number (LSN) of the log record that was not successfully applied. When the replica later receives a return code of **DB_REP_ISPERM**, this confirms both that the current log record and transaction have been committed and that all transactions with earlier LSNs have also been successfully committed. At this point, the replica can send acknowledgements to the master for all earlier transactions.

Quorums on Elections

To guarantee that we never lose updates, we need to make sure that once we've returned success from transaction commit on the master, under no circumstances will the application continue in a manner where that transaction has not completed. This means that the transaction must persist in the presence of master crashes and network partitions. Ensuring this requires tight coupling of write quorums and election quorums. An election quorum determines how many votes a site needs in order to become master.

In the Replication Manager framework, applications do not explicitly participate in elections; BDB-HA does this automatically. In this case, the election quorum is set to greater than half of the electable peers in the group, unless we have a two-site replication group (the master/slave scenario). In that exceptional condition, there is an option to permit election of a master with a single vote, since we assume that an election happens because the master has failed, and there is only one site remaining. However, this means that in the presence of a network partition between the master and slave, both sites could function independently as masters.

In the Base API, the election quorum is specified as the **nvotes** parameter to the **DB_ENV->rep_elect** method. To guarantee persistence, we must ensure that when the master fails, any site that can be elected has received the latest successfully committed transaction.

The most common approach to ensuring that updates are never lost requires two things:

1. commits be successfully processed by at least half of the electable peers and
2. elections require that more than half of electable peers cast votes.

These two conditions guarantee that at least one site has both received the latest committed transaction and participated in the election.

It is always possible for applications to require more than the minimum number of acknowledgements and in fact, it might simplify coding in cases where acknowledgement delays are not a problem (e.g., the LAN-based case).

Quorums on Reading/Master Leases

So far, we have discussed consistency only with respect to writes. However, some applications require both read consistency and read durability, which is the guarantee that a value read is both up-to-date and will persist, even in the presence of failure. For example, consider a user management service implemented across the WAN. It would be confusing for a user to change his password but then have the new password rejected, because the login attempt was processed by a replica that had not yet received the password modification. The formal way to guarantee consistency on reads is to introduce a read quorum, just like the write quorum, and require that at least half of the sites return the same value. As expected, such an approach can introduce intolerable latencies, especially in a WAN-based scenario.

Instead most systems use a read quorum of 1, obtaining a value from a single site. However, even if you ensure that the read is performed at the master, additional care is required to guarantee that the value read at the master will persist after a master failure. Applications that need such strong read guarantees should use Berkeley DB master leases as well as directing those reads to the master. Applications turn on master leases using the **`DB_ENV->rep_set_config`** method with the **`DB_REP_CONF_LEASE`** argument. A master lease is a promise by a replica not to participate in an election until the lease expires. If the master is directed to honor master leases, then it promises not to respond to a read request unless it holds leases from a majority of the replicas. The result is that the system guarantees that no new master will be elected while another site still believes that it is the master. When used in conjunction with a majority write quorum, master leases guarantee both that data read on the master is current and will persist, even in the presence of failures.

Weaker forms of consistency

Unsurprisingly, consistency and performance trade off against each other. Waiting for multiple sites to acknowledge a commit inevitably introduces a delay, but it strengthens the consistency model. Similarly, using master leases provides strong guarantees, but can introduce delays when reading and before an election. This latter delay can make recovery after a master failure potentially slower.

The default policies implemented by the Replication Manager provide a strong consistency model. Applications that want to optimize for performance above consistency can select a different set of options. When using the Replication Manager, applications enforce weaker semantics by changing the acknowledgement policy via the **`DB_ENV->repmgr_set_ack_policy`** method. When the **`ack_policy`** is set to one of **`DB_REPMGR_ACKS_ALL`**, **`DB_REPMGR_ACKS_ALL_AVAILABLE`**, **`DB_REPMGR_ACKS_ALL_PEERS`**, or **`DB_REPMGR_ACKS_QUORUM`**, Berkeley DB enforces the

strong semantics described above. Weaker policies, such as **DB_REPMGR_ACKS_NONE**, **DB_REPMGR_ACKS_ONE**, and **DB_REPMGR_ACKS_ONE_PEER**, provide transactionally consistent behavior at the replicas as well as in-order updates (i.e., a replica will never reflect the updates of a later transaction without reflecting the updates of all previously committed transactions). However, with these weaker semantics, it is possible for the system to lose transactions after a master failure.

Eventual consistency is a specific, weaker consistency model in which no instantaneous guarantees are made, but which claims that when no updates occur for a long period of time, eventually all the updates will propagate through the system and all the replicas will be consistent. By default, BDB-HA provides eventual consistency with any acknowledgement policy. Even if the system ends up in a state with multiple masters (for example, after a network partition), when the partition is resolved, participants will synchronize to a shared view of the data. However, in this scenario, it is still possible for a master to commit transactions that may later be undone (for example, when a partition is resolved).

When BDB-HA uses master leases an application can never have two masters both applying updates to the database. The leases on an old master that becomes partitioned will eventually expire and no new master will be elected before that expiration time. Once the old master's leases expire, it will be unable to apply updates and when it becomes reconnected with the rest of the group, it will receive a DB_REP_DUPMASTER error. In this configuration, the group will be eventually consistent, and each site will always see updates applied in the same order, which is a slightly stronger guarantee than is typically provided by eventually consistent systems.

Maintaining Group Membership

Enforcing varying forms of consistency requires that an application know the size of its quorums; knowing the size of the quorum requires knowing the number of participants in a replication group. While the Replication Manager tracks group membership automatically, applications using the Base API manage this information manually.

In the Base API, applications that wish to user Berkeley DB to dynamically manage group membership can do so by tracking **DB_REP_NEWSITE** return values from **DB_ENV->rep_process_message**. When an application gets a **DB_REP_NEWSITE** return code, it should compare the identity of the site against its current list of known sites. If the site is already known, no further action is necessary. However, if the application learns of a new site, it should establish a communication channel to that new site and map a locally unique environment ID to that site and communication channel. It should also update the number of sites participating in the replication group and its quorum requirements using the **DB_ENV->rep_set_nsites** method.

When a site determines that it should remove a site from its group, it should also update its participant information accordingly. Such changes in group membership and quorum requirements, in particular, must be handled with great care. If a quorum value is increased prematurely (before a site is fully ready to participate in the system), then the system might block waiting for acknowledgements that

are not yet available. However, if the quorum is not updated when the new site begins participating, Berkeley DB can treat updates as persistent in the face of application failure, without having received sufficiently many acknowledgement.

5. Special Case: Two-Site Replication Groups

Commonly, applications use Berkeley DB to implement conventional master/slave backup. In BDB-HA terminology, this is a 2-site replication group. While such a configuration is possible, there are certain limitations of which developers should be aware.

A failover is when the backup assumes master responsibilities in the event of a master failure. In order to allow failover, the system must ignore election quorums, allowing only a single site (exactly half the total sites) to declare itself master. In this case, a network partition can cause both sites to believe they are master. When the network partition is resolved, new transactions committed on one of these masters are rolled back and are therefore not durable.

By default the Replication Manager enforces the election quorum in a 2-site replication group. This behavior is known as 2SITE_STRICT because it requires both sites to be available to elect a new master. This prevents duplicate masters in the event of a network partition and ensures that committed transactions are durable. However, failover cannot occur if the master site crashes, leaving the replication group unavailable for write operations until the master site restarts.

Master/slave applications that require failover can turn off 2SITE_STRICT behavior using the **DB_ENV->rep_set_config** method with the **which** parameter set to **DB_REP_CONF_2SITE_STRICT** and the **onoff** parameter set to 0. Applications that turn off 2SITE_STRICT behavior are likely to roll back some transactions committed during a network partition.

Preferred master mode is another option for master/slave applications. One site is designated the preferred master and automatically assumes the role of master as much of the time as possible. Preferred master mode provides better availability than 2SITE_STRICT behavior because it allows failover. It provides more predictable durability than turning off 2SITE_STRICT behavior because it never rolls back transactions committed on the preferred master site. To use preferred master mode, the master site specifies **DB_REP_CONF_PREFMAS_MASTER** and the backup site specifies **DB_REP_CONF_PREFMAS_CLIENT** using the **DB_ENV->rep_set_config** method.

6. What is Happening Under the Covers

While developing your application, it is frequently useful to understand the different types of activities that might be happening in a replication group, so you can build your application to respond appropriately. The information in this section will be most helpful to you while you are building your application, but it may not be necessary in early phases as you design your application.

During most of your application's functioning, replication should be reasonably invisible, and the application can largely ignore it. However, there are certain times when your application may experience unexpected stalls or error returns or receive an event notification, and it's useful to understand how those happen and what an application should do about them. In particular, if your application implements any kind of timeout mechanism, it will be important for your application to understand when Berkeley DB internal processing might trigger an application timeout.

We begin by outlining the different activities in which replication applications participate. We present a high level overview of what happens during each of these processes, so that the following error returns, events, and stalling opportunities make sense within a global framework.

Starting up a replication application

Applications make an explicit call to turn on replication; **`DB_ENV->repmgr_start`** for the Replication Manager and **`DB_ENV->rep_start`** for the Base API. Although a site may explicitly designate itself to be either a master or replica (client), we encourage developers to have sites call **`DB_ENV->repmgr_start`** with **`DB_REP_ELECTION`** or **`DB_ENV->rep_start`** with **`DB_REP_CLIENT`**, allowing the normal election mechanism to select a master.

When an environment starts up replication, Berkeley DB will block any other threads using the Berkeley DB library until startup completes. This is necessary to avoid other threads using the library when it is in an inconsistent state, either with respect to its status (master or replica), the state of the database, or any other form of processing. When a site starts up as a replica, it will perform internal cleanup and initialization (aborting prepared but not yet committed transactions and initializing client state temporary databases) and then announce its entry into the system. At that point, the site will not know the identity of the master, until a master announces itself.

If a site calls **`DB_ENV->repmgr_start`** or **`DB_ENV->rep_start`** with **`DB_REP_MASTER`** to declare itself a master, then it will initialize its state and announce itself as a master to any replicas it knows about.

Electing a master

In the absence of a master, the Replication Manager will hold an election to find a master. Applications using the Base API will need to call the **`DB_ENV->rep_elect`** method to trigger an election. The BDB-HA election code implements the Paxos consensus algorithm using the following five phases.

Phase 0: Short Circuit Elections

First, if a site that thinks it is a master is told to hold an election, it attempts to short circuit the election by declaring itself master.

Phase 1: Election Initialization

If the site is not a master, it initializes the election. In doing so, the site determines the expected number of sites from which it should receive votes and it initializes its internal state to be in a new election.

Phase 2: Master Lease Timeout

If master leases are configured, the site has to wait for its current lease to expire before holding an election. This wait will be perceived by the application as a delay, whose duration is dependent upon the lease timeout value specified by the **`DB_ENV->rep_set_timeout`** method using **`DB_REP_LEASE_TIMEOUT`**.

Phase 3: Voting

Next, we transition into a state where we ignore all incoming log records until the election terminates, and we begin participating in the election. We participate by incrementing our election generation number and record and transmit our vote, which consists of our identity, generation number, current end-of-log, and our site election priority. We then wait to receive votes. This wait may also be perceived by the application; its duration is determined by the election timeout value, configured by the **`DB_ENV->rep_set_timeout`** method using **`DB_REP_ELECTION_TIMEOUT`**. While we are waiting, we accumulate votes from other sites. Each time we receive a vote, we record the site that we currently believe should be the next master, according to the contents of each vote. If we receive votes from all participating sites before our timeout expires, we move on to the next phase.

Phase 4: Picking a New Master

If we have received enough votes, then we declare our current master designee the winner, and send a message to that site, indicating that we are voting for it.

If we have not received enough votes to vote for a master, we still transmit a vote to our desired master, but we do not update all our internal state to reflect the new master. In either case, we then wait for a master to declare itself.

During an election, all participating sites might experience delays, because writes will not progress while log records are being ignored (starting at the beginning of phase 3). However, read requests can progress as normal.

Failing over to a new master

When a master fails or the replicas are unable to communicate with the master, the remaining replicas need to select a new master. The Replication Manager will automatically detect master failure and initiate an election. Applications using the Base API must detect master failure and explicitly initiate an election. Each site will go through the steps outlined above, and then, having learned the identity of a new master, each replica will participate in a synchronization exchange with that master before resuming normal processing. This synchronization process requires that the library block all Berkeley DB operations. Any in-progress transactions or operations will be allowed to complete, but no new operations will be allowed to begin until the site is synchronized with the master. Once again, this could be perceived as an application delay, whose duration is determined by how active the system has been (in terms of writes) and the length of time between a master failure and a successful election.

Restarting after a site failure

After a site fails (regardless of whether it was a replica or master), startup is similar to starting up the application. First the application should open its environment and run recovery as it would normally do. Then it should start replication, following the steps outlined in the section on Application startup.

Unexpected Error Returns

In the presence of replication, there are several new error codes that can be returned from any Berkeley DB API call. We list those error returns here, explaining what they mean and what an application should do about them.

DB_REP_HANDLE_DEAD: The DB handle used for this call is no longer valid, due to a change in master and corresponding log synchronization. The application should close the handle immediately and reopen the database.

DB_REP_LEASE_EXPIRED: The system is configured to use master leases and the requested operation could not be performed, because the master's lease had expired. The application should optionally delay and then retry the operation.

DB_REP_LOCKOUT: The replication system is synchronizing with a new master. The application should wait and then retry the operation.

DB_REP_UNAVAIL: When using the Replication Manager, this is returned when a site is unable to join the replication group during startup because a master site is unavailable or there are insufficient replicas available to acknowledge the new site. Replication Manager applications should retry starting up after a delay. When using the Base API, this is returned when a site is unable to communicate with a master. Base API applications should normally call an election.

When using the Base API, the application provides a message processing loop that receives messages from the communication infrastructure, calling **DB_ENV->rep_process_message** on each BDB message. The error returns from **DB_ENV->rep_process_message** provide information about what is happening inside the BDB library, indicating what actions an application must take. These error returns are listed below.

DB_REP_DUPMASTER: BDB has detected that multiple sites are acting as masters. Any master receiving this message should immediately change to a replica, calling **DB_ENV->rep_start** with the **DB_REP_CLIENT** flag.

DB_REP_HOLDELECTION: Some site in the group initiated an election. If a replica receives this return, it too should initiate an election using the **DB_ENV->rep_elect** method. A master should never observe a **DB_REP_HOLDELECTION** return value.

DB_REP_IGNORE: Applications may get this return when they expect specific return values (e.g., **DB_REP_ISPERM**, **DB_REP_NOTPERM**), none of which are appropriate for the call in question, but where the behavior does not constitute an error case. The application should do nothing in response to this message, continuing as if the message had never been received.

DB_REP_ISPERM: This value is returned on replicas, indicating that the records processed by the current call have been persistently applied to the local environment. In most cases, the application need not do anything special. However, this is the only way that Base API applications learn that a transaction has been made persistent, so the replica may need to send an acknowledgement, containing the LSN returned, to the master.

DB_REP_JOIN_FAILURE: This return value indicates that a replication site needed to reinitialize its databases and logs from a master, but had been configured to disallow automatic initialization. In this case, the application should take whatever steps necessary to secure a copy of the database and logs.

DB_REP_NEWSITE: This return indicates that the library has learned of the existence of a (potentially) new member of a replication group. The application should update its group membership data appropriately.

DB_REP_NOTPERM: This return indicates that although a record that needs to be persistent was processed, it has not yet been made persistent. The application may want to record the LSN of the record that was not able to be made permanent, so it knows that the associated transaction is not yet permanently committed. When the site receives a later **DB_REP_ISPERM** return value with a larger LSN, it then knows that this operation has also been made persistent.

Replication events

In addition to error returns, applications are encouraged to configure a callback function for event notification by calling **DB_ENV->set_event_notify**. The following events are the most commonly handled events particular to the replication subsystem.

DB_EVENT_REP_CLIENT: The local site is now a replication client (replica). The application should update its state appropriately.

DB_EVENT_REP_ELECTED: The local site has just won an election. Applications using the Base API should call **DB_ENV->rep_start** to reconfigure the environment as a master. Applications using the Replication Manager need only update their application state to reflect their role as a master.

DB_EVENT_REP_MASTER: This event is received after a site has completed initialization as a master site. The application should ensure that it knows that it is a master, but should not have to make any other calls into the Berkeley DB library.

DB_EVENT_REP_NEWMASTER: This event indicates that a new master has been selected and provides the identity of that new master. The application should update its state to reflect the new master site.

DB_EVENT_REP_PERM_FAILED: This event occurs only when using the Replication Manager. It indicates that the master did not receive enough acknowledgements to ensure a transaction's durability. The operation has been flushed to the master's log, but if the master crashes, the

transaction may not persist. The application may want to store the LSN of the failed operation for later use, so it can keep track of which transactions are not yet permanently durable.

DB_EVENT_REP_STARTUP_DONE: The site has completed synchronization with the master and is now processing live log records. The application may want to update its state to reflect that it has completed synchronization.

Stalls

Applications might also experience unexpectedly long delays in some cases that will be noticed by applications that configure timeouts in some way.

On replicas, stalls can happen during startup, during elections, during synchronization with a master, and while a site is initializing its databases from a master (a special form of synchronization).

If an application experiences an application timeout, it may first wish to determine if BDB-HA processing caused the timeout. If the application has been receiving events, it might already be aware of the potential for a stall. If the application has no expectation that it is in a state that might stall, it can use the **DB_ENV->rep_stat** method and examine the **st_election_status** field to determine if the site is currently involved in an election. It can examine the **st_startup_complete** field to determine if the site is delayed in starting up or synchronizing with a new master.

If none of those conditions explains the delay, then the application should assume that something other than Berkeley DB is responsible for the delay.

In general, the master site should not experience stalls. Individual operations may block awaiting responses from a suitable number of peers, but other threads should continue to make forward progress. Similarly, threads may block due to contention on transactional locks, but this behavior is no different from existing transactional applications. Berkeley DB does delay the completion of checkpoint operations to allow replicas to synchronize their caches without causing long delays; an application might perceive this as a stall if it notices that a checkpoint takes a long time to complete. If this delay is problematic, the application can shorten it by calling **DB_ENV->rep_set_timeout(DB_REP_CHECKPOINT_DELAY)** with a delay value shorter than the default 30 seconds.

Internal initialization

While attempting to synchronize with a master, a replica might learn that the master no longer has the necessary log files. In that case, the replica will attempt to copy the existing databases and subsequent log files from the master. This reinitialization of a replica is a potentially long-running sequence of operations. The replica first requests a list of databases from the master. That list contains a description of each database and the number of pages in that database. The replica then proceeds to request from the master all the pages of each file. Once all the database pages have been transmitted, the replica then requests all newly written log records so that it can apply any transactions that

completed after the backup began. Once all the databases and log files have been transferred, the replica runs recovery and synchronizes with the master.

If applications do not want such database copying to happen automatically (perhaps because the application has a high bandwidth channel it can use to copy the needed data), the application can disable automatic initialization using the **DB_ENV->rep_set_config** method with the **DB_REP_CONF_AUTOINIT** value.

However, if the application permits automatic initialization, it must be prepared to observe potentially long periods of unresponsiveness on a replica while it obtains copies of its database and log files. Internal initializations are most likely after a replica has been down for a long time, or if an application managing its own log reclamation removes a log file before all sites in the replication group is finished with it.

Becoming a master

When a site becomes a master, it becomes responsible for applying all updates. Therefore, when a master receives a **DB_EVENT_REP_MASTER**, the application must assume responsibility for handling database update requests.

7. Conclusion

Berkeley DB High Availability provides fast, reliable, and scalable data management, embedded within an application. It is highly configurable, providing an effective data management solution for devices, data-centers, and wide-area distribution.