ORACLE®
BERKELEY DB

Oracle Berkeley DB High Availability

ORACLE®

Table of Contents

Executive Overview

Oracle Berkeley DB Java Edition High Availability (JE HA) is a replicated transactional data management system that provides high availability along with improved read scalability and performance.

This paper presents the key concepts and characteristics of JE HA, which enable developers and application designers to understand how to best use it to solve specific data management problems.

This paper also discusses the key technical trade-offs that an architect must consider when designing an application based on JE HA.

Introduction

Oracle Berkeley DB Java Edition includes a high availability feature. Oracle Berkeley DB Java Edition High Availability (JE HA) extends Oracle Berkeley DB Java Edition (JE), a transactional data management system. You can use JE's Direct Persistence Layer API in applications to create indexed databases that store Java objects. Multiple operations can be wrapped in transactions to provide the conventional transactional properties of atomicity, consistency, isolation, and durability (ACID). If an application or system fails, JE's recovery mechanism restores the application's data to a consistent state.

JE HA is an embedded data management system designed to provide applications with JE's capabilities along with a "master read-write, replica read-only" replication paradigm. This means that all database writes/updates are performed on a single master node, and replica nodes are available only for read activity. If the master fails, one of the replica nodes automatically takes over as a master and you can perform all the write operations on the new master. Due to its implementation as an embedded library, JE HA can be used in a wide range of configurations and environments.

JE HA addresses three different application challenges:

» Provides rapid failover in situations where downtime is unacceptable
» Provides read scalability by providing multiple read-only replicas

» Enables more efficient commit processing, allowing applications to provide durability by committing to the (fast) network, rather than to a (slow) disk.

JE HA, however, does not assist applications that need write-scaling. Such applications will need to partition the data, running a JE HA environment per partition. It also does not provide synchronization between JE HA and other databases. These last two issues are not addressed in this white paper.

JE HA can be used in a wide range of environments. Here are some sample use cases where JE HA is applicable:

» **Small-scale LAN-based replication**: A data-center based service that provides data for a local population, such as a corporate site. This service is implemented by a few servers, all of which reside in the same data center, communicating over a high-speed LAN.
» **Wide-area data store**: A world-wide service provider who stores account information that is accessible anywhere in the world. This service is implemented by a collection of servers, residing in different data centers, scattered around the globe, communicating via high-speed data-center to data-center connections.
» **Master/Slave**: A simple installation to provide failover between a master and slave machine.

Given the wide range of environments across which JE HA replicates data, no single set of data management and replication policies is appropriate. JE HA lets the application control the degree of data consistency between the master and replicas, transactional durability, and a wide range of other design choices that dictate the performance, robustness, and availability of the data. The goal of this paper is to introduce those options and provide a sufficiently detailed understanding of the issues, so you are able to make the right decisions and trade-offs for your application.

JE HA Overview

This section provides an overview of the organization and key concepts underlying JE HA.

In the following discussion the term Database is used to denote a collection of key/data pairs sharing an indexing structure. This is a Table in RDBMS terminology. The term Environment denotes a logical collection of Oracle Berkeley DB databases and their associated meta-data and infrastructure files. A directory names an Environment. This is a database in RDBMS terminology.

JE HA enables replication of an environment across the nodes in a Replication Group. An environment is the unit of replication. All the databases within a replicated environment are themselves replicated and transactional.

Replication Group

A Replication Group is the collection of nodes involved in the replication of a single environment. A Replication Group consists of electable nodes.

An Electable node has its own replicated copy of the environment and can serve as a master or a replica. The choice of a master is made using a distributed two-phase voting protocol, which ensures that a unique master is always elected. Elections require that at least a simple majority of the Electable nodes participate in the process. The participating Electable node which has the most up-to-date state of the environment is elected the master.

An Electable node can be in one of the following states:

» **Master**: the node was chosen by a simple majority of electable nodes. The node can process both read and write transactions while in this state.

» **Replica**: the node is in communication with a master via a replication stream that is used to keep track of changes being made at the master. The replication stream is described in greater detail in subsequent sections. The replica only supports read transactions.

» **Unknown**: the node is not aware of a master and is actively trying to discover, or elect a master by contacting other members of the group. A node in this state is constantly trying to transition to the more productive Master or Replica state. An Unknown node can still process read transactions if it can satisfy the consistency requirements of the transaction as described in subsequent sections.

» **Detached**: the node has been shutdown. It is still a member of the Replication group, but is not an active participant. A node that is not in the detached state is also referred to as being active.

Note: The terms master and replica are used as a shorthand in this discussion to denote a replication group member that is in one of these states.

The replication nodes in a group are typically distributed across different physical machines to ensure that a single machine failure does not impact multiple nodes in the same group. It is not uncommon to have nodes from different replication groups running on the same physical machine. Such configurations are useful, for example, when implementing data partitioning and are discussed in greater detail in subsequent sections.

The following figure highlights the key elements of a replication group G that consists of three electable nodes (N1, N2 and N3).
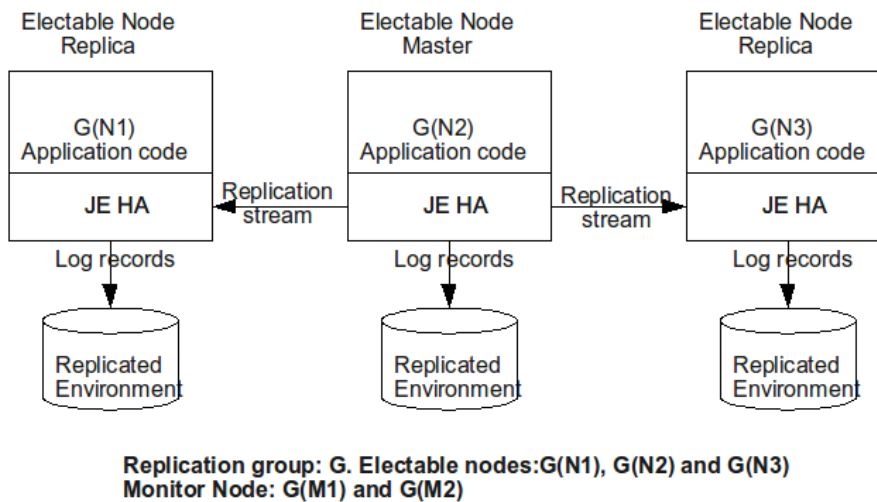

Replication group: G. Electable nodes:G(N1), G(N2) and G(N3)
Monitor Node: G(M1) and G(M2)

Figure 1 - Replication Groups

Replication Group Lifecycle

A Replication Group always starts out with a single member which acts as a master. The replicated environment can either be empty, or it can be an existing standalone JE environment that has been converted to the replicated environment format. New electable nodes can then be added to the group, increasing both the size of the group and potentially the quorum requirements for elections and durability.

A new electable node typically goes through the following sequence before it is ready to participate in the Replication Group:

- » The new node locates the current master and supplies it with its configuration information.
- » The master makes a persistent record of this node in an internal, replicated, group membership database. Other electable nodes are made aware of this node through the replication stream. The node is now considered a member of the group.
- » The member now initializes its copy of the replicated environment. If some of the log files underlying the environment have been reclaimed by JE's log cleaner, it uses an operation called a Network Restore to physically copy the log files from one of the members of the group and use this copy as a starting point from which it can replay the replication stream.
- » The member replays the replication stream until it meets its replica consistency requirements.

Having become consistent, the member is fully functional and can start operating as a Replica. It is available for read operations to the application and is ready to serve as a master in the event of a failover.

If the machine hosting the member fails, the group may not be able to function if it cannot meet quorum requirements. Such a failed member can be explicitly removed from the group, via the HA API, so that the remaining members can satisfy quorum requirements and the group can resume normal operations with reduced durability.

Arbiters

An arbiter node tracks the state of the electable nodes in a Replication Group. The arbiter node does not contain any data, but is kept informed of changes in the structure of the group resulting from new nodes being added or removed from it. An arbiter cannot become a primary node. It is also kept informed of the dynamic status of electable nodes, that is, whether an electable node is actively participating in the Replication Group.

Replication Stream

Write transactions executed at the master are replicated to the replicas using a logical Replication Stream established over a TCP/IP connection. There is a dedicated connection between the master and each replica. The stream contains a description of the logical changes, that is operations such as insert, update, delete, commit, abort, and so on, that were executed on the environment. The logical operations are computed from the log entries contained in the logs on the current master and are replayed at each replica using an efficient internal replay mechanism.

For a replica to be able to successfully replay the replication stream, the log entries must be contiguous. It must not be missing entries that were removed because JE's log cleaning had deleted the files containing those entries. To ensure this, JE HA divides the log into two parts:

- » The head of the log, representing the older section of the log in which the log cleaner can delete files.
- » The tail of the log, representing the part of the log in which all the log files are still present. Only the tail of the log can be used to construct the replication stream. The tail expands as new entries are added to the log and shrinks (and the head expands) as replicas catch up with the master and no longer need that portion of the log.

The figure below represents a three node group with node 1 being the master. Node 2 is the farthest behind in its replay of the logs and therefore determines the composition of the head and tail sections of the log. All the log entries that have not yet been replayed by it constitute the tail of the log. In this example, log files 5 and 6 at node 1, and log files 4 and 5 at node 3, constitute the tail portion of the log cannot be reclaimed by either node until node 2 makes more progress.
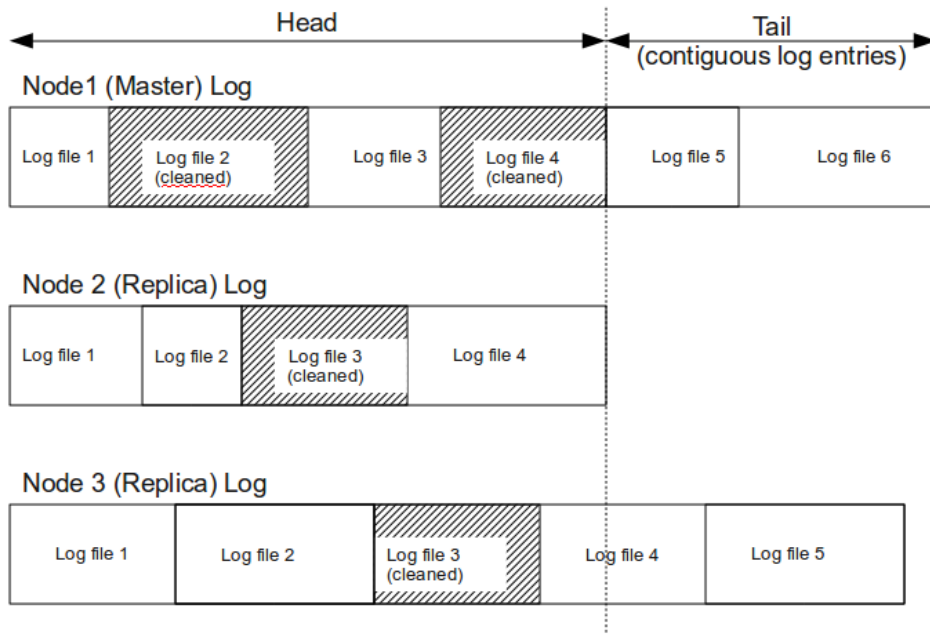
Figure 2 - Replicated Log Head and Tail

.

The master sends changes down to the replica asynchronously, as fast as the replica can process them. Changes are streamed to the replica eagerly, that is, even while the transaction is in progress and before the application attempts to commit the transaction on the master. If the transaction is subsequently aborted on the master, the replica will abort the transaction as well. An application thread may wait for one or more replicas to catch up to a specific transaction if the durability requirements of the transaction require it. The replica responds with explicit acknowledgment responses for such transactions, but this does not inhibit forward progress on the master, as it is free to proceed with other transactional work.

The write operations to the replication stream by the master and the read (and replay) operations on the replica are asynchronous with respect to each other. The master can write to the replication stream as soon as the application creates a new log entry. The replica reads from the replication stream and replays changes as soon as it reads them, additionally responding with acknowledgments to the replay of a commit operation if requested by the master.

Heartbeat

In addition to the logical change operations, the master also sends periodic heartbeat requests to the replica. The replica uses the absence of a heartbeat as an indication that there is a potential problem with the master or the network and starts an election to help resolve the problem. At any given point in time, a replica may lag the master in its replay of the replication stream, so that the state of the replicated environment may not be identical to that on the master at that instant in time, although it eventually will be, as the replica catches up. The replica uses information in the heartbeat to determine how far behind it is in its replay of the log, so that it can satisfy the read consistency requirements (which places limits on the amount of the lag that is acceptable) on the replica.

Application Architecture

JE HA is built upon JE and shares much of the JE API. However, a JE HA application is a distributed application and brings in additional considerations that are best addressed up front as part of the initial application architecture design for new applications. This section provides an overview of the key features of JE HA that will have a direct bearing on the application's architecture.

Embedding Data Management in Your Application

This section starts by describing what it means to "embed" data management in the context of JE and then extends the discussion to cover JE HA.

The term "embedded" is used when talking about JE to refer to the relationship between the application and the data management capabilities. Those capabilities, as realized by the JE library, are embedded within the application. Application users or administrators need not be aware of the existence of JE, as JE is used and administered directly from the application. Such an application may run on an embedded device such as a handheld device, a desktop machine, or in a datacenter. In fact, JE 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:

- » Data is stored and retrieved in its native form, rather than translating to and from a different (for example, relational) data model. Applications can serialize structures or complex objects directly into JE databases.
- » No database administrator (DBA) is required. 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.
- » Recovery is automatic. The end-user does not observe any separate database recovery phase. Instead, if necessary JE automatically and transparently recovers a database when the application opens it. Similarly, when the application shuts down, it can shut down the database as well.
- » Performance is superior to that of traditional client/server architectures because it eliminates interprocess communication overhead, between the application and the data management service.

In JE HA, the same application typically runs at each electable node in the replicated group, with a copy of the replicated environment being maintained at each node. The application embeds data management in exactly the same way as does JE. The application running on the node that is currently the master can process both read and write requests, while the applications running on the replicas can process read requests.

Managing Write Requests

It is the application's responsibility to submit all application level requests that modify the database (for example, via JE insert, delete, overwrite and database create, rename, truncate, or remove) to the master.

Replica-based forwarding: Each application on a replica keeps track of the identity of the master and forwards any write requests it receives to the current master.

Some of the factors that should be considered when choosing between them:

- » **Integration with existing load balancing schemes**. In some cases, for example, a hardware based load balancer, it may not be easy, or even possible, to integrate an intermediate router into the load balancer. In such cases, replica based forwarding is the preferred solution, because it does not perturb the existing load balancing support.
- » **Write request latency**. Forwarding a write request when using replica-based routing involves an additional network hop, thus increasing the latency associated with write requests. This may not be desirable in applications that require low write latency and response variability.

- » **Network usage**. If the payload associated with write requests is large on average, the request forwarding mechanism may compete for network resources with the replication streams, thus adversely impacting replication performance. Applications with large write request payloads would benefit from monitor-based routing.
- » **Request formats**. If using monitor-based routing, the application request formats may need to be designed to permit easy determination of whether the request is a write request. For example, webapplications may make provisions for URL formatting conventions that the router understands to distinguish between read and write requests. The design of these formats is an additional design consideration that must be kept in mind when using monitor-based routing.
- » **Router availability**. For high availability, the application designer needs to plan for multiple router instances, so that the router does not represent a single point of failure. This represents an additional level of complexity when using monitor-based routing.

The two approaches trade off simplicity and performance. monitor-based routing has the potential for providing higher performance, but brings with it an additional level of complexity. Replica-based forwarding is simpler to implement, but may not perform as well.

Because request formats and conventions vary widely across applications, JE HA does not provide mechanisms to transport application requests between members, or between a router and a member. The application must provide this functionality. JE HA does provide the interfaces needed to determine the members of a replication group, and to track the status of the member that is currently serving as the master within the replication group.

The following sections describe the support that is available in JE HA to help implement the above approaches.
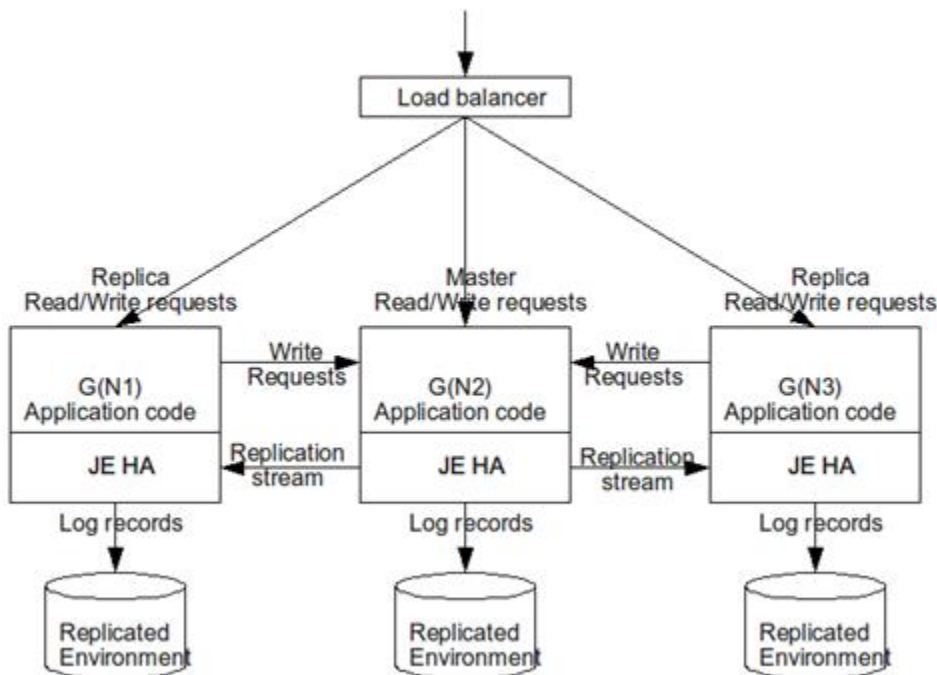

Figure 3 - Replica Based Write Forwarding

Replica-Based Forwarding

The JE HA application accesses its replicated environment via a replicated environment handle. The handl API makes provisions for the application to:

» Determine its current state. For example, the node can be queried as to whether it is currently the master or the replica.
» Associate a state change listener with the handle. The state change listener then receives events that can keep it informed of the current member that is serving as the master.

With these APIs the application can thus determine whether it is currently the master. If it is not the master, it can arrange to forward the write request to the node that is currently the master, as illustrated in the above figure. The actual mechanism used to forward the request to an appropriate node is application-specific and does not involve JE.

Handling Failover at Electable Nodes

While the election process itself is transparent to the running application, the application must allow for the state transitions that result from an election, so that it can continue operating across the state transition. The following transitions could arise as a result of a failover:

» An electable node may transition from a replica to a master, if the current master fails. In an application using a monitor-based routing scheme this means that a node that had read requests routed to it starts accepting write requests as well and must start processing them. In an application using the replica-based forwarding scheme, the master node must stop forwarding write requests and start accepting write requests directed at it from other replicas.
» A member which is a master may transition to a replica. This is possible in rare circumstances involving a temporary network partitioning, when the master is unable to participate in an election. As a result, the remaining nodes may elect a different master. In this case, the replica must forward any write requests that it receives, or reject them, because transactions that attempt to update the environment at the replica node fail with an exception.
» A replica may, as a result of a failover, switch over from the failed master node to the one that was newly elected. This transition is typically transparent to the application. JE HA simply re-establishes its replication stream with the new master. In very rare circumstances, JE HA may need to perform a recovery and undo transactions that were previously committed. In such cases, the replicated environment handle is invalidated. The application must recreate the environment and database handles, and recompute any cached transient state that was obtained by accessing the previous environment handle.

JE HA Transaction Configuration Options

Replicated environments are transactional. A replicated transaction provides additional options for transaction configuration, which have a direct bearing on the performance of the application. In choosing these settings you are making tradeoffs in the areas of durability, write availability, replica read consistency, and performance. This section describes these configuration options and highlights the tradeoffs.

Durability Options

Standalone JE provides three levels of durability, expressed in terms of the degree to which committed data is synchronized with respect to its persistent state on disk:

» **SYNC**: Committed changes written to the disk are considered durable. This level protects committed data in the event that the machine itself crashes. It does not protect the data from media failures.
» **WRITE_NO_SYNC**: Committed change written to the file systems buffers are considered durable. This level only protects committed data from application level failures. In the absence of a machine failure the committed data will eventually be written to disk.

» **NO_SYNC**: Committed changes written to JE's internal buffers are considered durable. At this level committed data can be lost in the event of an application level failure.

JE HA extends JE's notion of durability, by allowing transactions to persist changes by committing them to a fast network instead of a slow disk. So, in addition to permitting configuration of the level of synchronization on both the master and the replica as above, JE HA makes provisions for a replica to send back a commit acknowledgment, after it has finished replaying the transaction. Not all transaction commits result in acknowledgment responses from the replica. The master explicitly requests acknowledgments when required. If a replica has fallen behind the master and is trying to catch up, the master does not request acknowledgments.

JE HA durability thus has three components associated with it:

» The degree of synchronization on the master.

» The number of acknowledgments required from replicas as expressed by the Acknowledgment Policy (described below) associated with the transaction.

» The degree of synchronization on the replica, before it can acknowledge a commit.

Acknowledgment Policy Configuration

The acknowledgment policy provides three possible choices: ALL, SIMPLE_MAJORITY and NONE.

» **ALL**: The ALL policy requires that all electable members acknowledge that they have committed the transaction before it can be considered to be durable by the master. It results in high durability because a copy of the changes exists at every replica before the transaction is considered to be durable. Read consistency on the replicas is high, because the replicas are keeping up with the master and there is no lag in the replica's processing of the replication stream at a transaction commit. However, this setting lowers the availability of the group for write operations, especially in large replication groups, because the transaction commit operation fails if even a single node becomes unavailable. Another downside is that it increases the latency associated with commits, because the time required for the commit is determined by the performance of the slowest replica.

» **SIMPLE_MAJORITY**: A simple majority of electable members must acknowledge that they have committed the transaction. This setting also provides durability in the medium to high range, because a copy of the changes exists at a simple majority of the replicas at the time of the commit and is eventually propagated to the remaining nodes as they catch up in the replication stream. In the case of a failure of the master after a successful commit, the election process ensures that the change is not lost. It can make this assurance because it requires the participation of a simple majority of electable nodes in the election, thus ensuring that at least one of the nodes that participated in the commit acknowledgment participates in the elections as well. The election algorithm further ensures that the node with the most current logs is elected the master, thus retaining the change even after the failure of the master where the transaction originated.

Replica read consistency is not as high as with a policy of ALL, because the changes may not yet have been replicated to the replicas that did not participate in the acknowledgment of the transaction. Under normal conditions, this lag is small and the lagging replicas rapidly catch up with the transaction. As a result, transactions with strict consistency requirements may stall at any lagging replicas until they have had a chance to catch up.

Write availability with SIMPLE_MAJORITY is in the medium to high range, because the group can survive node failures and continue to make write transactions requiring a SIMPLE_MAJORITY durable as long as a simple majority of electable nodes continues to remain available.

Performance with SIMPLE_MAJORITY is better than that available with ALL, because the master needs to wait for a smaller number of nodes. Also, while it may request acknowledgments from all nodes with which it is in contact, it only needs to wait for a simple majority to respond. That is, commit latency is not determined by the slowest node in the group.

» **NONE**: The transaction does not rely on any acknowledgments for durability. The transaction is considered to be durable as soon as it satisfies the sync policy associated with the master. This policy results in low durability (comparable to standalone JE), because there are no guarantees that the changes have been replicated to other members at the time of the commit. If the master fails after a successful commit, the changes could be lost. Replica read consistency may be low as well, especially if the master is under heavy load and does not have the

resources needed to maintain the replication streams. As a result replicas may lag substantially behind the master. This policy does offer the potential for high write performance, while offering low durability.

The trade-offs involved with each acknowledgment policy are summarized in the table below:

	Durability	Write Availability	Replica Read Consistency	Write Perf
ALL	High	Low	High	Low
SIMPLE_MAJORITY	Medium-High	Medium-High	Medium-High	Medium
NONE	Low	High	Low	High

Replica Read Consistency

The replay of the replication stream results in the replica moving from one transactionally consistent state to another, because the replication stream represents the serialized JE log. The replica replay mechanism ensures that the transactional changes are committed as an atomic unit and that intermediate changes are isolated from other ongoing read transactions in accordance with their isolation requirements. The behavior of the replicas, when viewed from a local node perspective is entirely transactional and satisfies the usual ACID properties as in standalone JE. This section addresses the issues relating to consistency of the replicas with respect to the master.

JE HA supports Eventual Consistency, a consistency model in which no instantaneous guarantees are made about the state of all the replicas at the time of a transaction commit. It ensures however, that when no updates occur for a sufficiently long period of time, all the updates will propagate through all the members of the group and all the replicas will eventually share a single consistent view of the environment with the master.

All read operations at the master are absolutely consistent, so read operations that require absolute consistency should be directed to the master. The application can use the techniques described under write forwarding to forward such read requests to the master. However, in the case where such absolute consistency is not required, JE HA replica read transactions can specify the maximum acceptable amount by which a replica can lag the master at the start of the read transaction. There are two ways of quantifying this lag:

» In terms of time - expressed in the API as TimeConsistencyPolicy
» In terms of the position of the replica in the replication stream - expressed in the API as CommitPointConsistencyPolicy.

The size of the lag is primarily dependent upon the load on the master and replica, and the communication delays in conveying the replication stream. It may also be the case that a node was not in contact with a master for some period of time, and it needs to catch up with the changes made on the master during this time period.

The choice of a particular consistency policy is dependent upon the nature of the operation being undertaken by the application. The configuration of the chosen policy has a bearing on the performance of read operations at a replica. The following sections describe the policies and their configuration in greater detail.

Time Consistency Policy

This policy describes the amount of time by which the replica can lag the Master when the transaction is initiated. If t0 is the instantaneous time, all transactions that were committed before time (t0 - lag) must be replayed at the replica before the read transaction at the replica is allowed to proceed. It's worth emphasizing that t0 is the instantaneous time, and not the fixed time that the transaction was first initiated, so the replica is trying to catch up to

a moving target. The lag therefore represents a fixed size time window that itself moves in time. The start of the transaction waits until the replica is sufficiently caught up with its replay of the replication stream, that is, until the replica replays a transaction that falls within the window covered by the lag, or the heartbeat from the master indicates that the replica has caught up with the master.

If the replica does not catch up within the timeout period associated with the policy, the start of the transaction is aborted.

The time consistency policy may be suitable for some web sites when the lag can be made smaller than the human scale interaction time as a user visits a sequence of web pages.

Setting a lag period that is too small, given the load and available hardware resources, could result in frequent timeout exceptions and reduce a replica's availability for read operations. It could also increase the latency associated with read requests, as the replica makes the read transaction wait so that it can catch up in the replication stream. The application designer needs to determine the largest possible lag that's acceptable from the application's point of view and use it as the setting for this policy.

Commit Point Consistency Policy

This policy defines the lag in terms of the position of the replica, relative to the commit of a specific transaction (the commit point), in the replication stream. It guarantees that all transactions appearing earlier in the serialized replication stream, including the transaction at the commit point are replayed by the replica, before the read transaction on the replica is allowed to proceed. The lag in this case represents an expanding window of changes that's pinned at the low end by the commit point. The window expands at the upper end, as new transactions are committed on the master. JE HA extends the transaction API so that the application can obtain the commit point associated with a committed transaction.

As with the Time Consistency Policy, if the replica has not replayed the transaction identified by the commit point within a policy-specified timeout period, the start of the transaction is aborted.

The Commit Point Consistency policy may be used as the basis for providing session consistency in web applications. This is accomplished by associating a commit point with a user's session and updating the commit point with each write operation performed in the context of the session. Read transactions associated with the session then use a commit point consistency policy to ensure that they always see data that was written by the session, no matter which replica is used for the read operation.

The Commit Point Consistency policy is more complicated to use than the Time Based Consistency policy because it requires that the commit point state be maintained by the application across read/write operations. The Time Based Consistency policy is simpler to use, but its reliance on time means that it may not be as well-behaved when the master, replica or network are operating under unexpectedly high loads.

Performance

JE HA is built on JE. All the tuning considerations that apply to a JE application apply to JE HA as well. In addition, there are other considerations that apply specifically to JE HA. The following sections describe these considerations in greater detail.

Durability

The configuration of durability is one of the most important knobs at the application's disposal when tuning write operations for good performance. The default durability setting for standalone JE applications is SYNC, to ensure that committed transactions are not lost in the event of a machine failure. The use of SYNC typically results in a substantial performance penalty on commodity disks, but is essential in order to ensure that the data has been persisted to disk at the time of a commit.

A good starting point for tuning a JE HA application is the durability setting of (NO_SYNC, SIMPLE_MAJORITY, NO_SYNC). This setting permits the master and the replica to write changes to the file system and disk asynchronously at some later point in time after the commit operation, rather than synchronously at the time of the commit operation. In addition, JE is able to batch changes made across multiple transactions, and use a smaller number of large, but efficient, sequential write operations when the data is ultimately persisted to the disk.

The use of SIMPLE_MAJORITY for commit acknowledgments reduced the latency incurred when the master waits for commit acknowledgments, because the latency is not determined by the performance of the slowest replica. For large group sizes, latency will increase with increasing group size. The use of SIMPLE_MAJORITY for commit acknowledgments ensures that the data is available at other nodes in case the current master fails. This setting does not guard against a scenario where multiple simultaneous application failures result in the simple majority of the nodes which supplied a commit acknowledgments all going down after the commit, but before they have a chance to persist the data to disk.

From this starting point, the designer can choose the level of durability that best fits the overall needs of the application. Note that durability is configurable at the transaction level. This lets different components of an application use different levels of durability based upon their local requirements.

Read Consistency

To make effective use of the read throughput capacity available at the replicas and to minimize the load on the master (because it is the only node servicing write requests) all read requests that do not require absolute consistency should be directed by the application to the replicas. Further, to minimize latency, use the most relaxed form of consistency that meets the requirements of the application. Like durability, consistency can be configured at the level of an individual transaction. The details of how consistency policies are chosen and configured have already been covered in earlier sections.

Replication Stream Considerations

JE HA uses TCP connections to transmit replication streams from the master to the replicas. In a group of size N, N-1 copies of each log entry must (at least eventually) be transmitted over the network. Giving careful consideration to optimizing the size of the values associated with keys can help with performance. This is particularly important for write-intensive applications.

It is worth noting here that just as sequential disk write throughput is often the limiting resource for write performance, network throughput can be the limiting resource for replication performance. Because of that, it is important to ensure that the available network bandwidth is capable of handling the expected write load.

Lagging Replicas

The organization of the caches on the master makes it possible for the replay stream to be computed most efficiently when the replica is already reasonably current and is therefore being sent the latest log entries. This is because the latest entries are likely to be present in master's caches and do not require access to the disk. If disk access is required to fetch the older log items, it will result in disk contention with any ongoing disk writes required to persist new log entries, which disrupts the sequential write pattern that is essential for high JE write performance.

For this reason, it is best to keep nodes up and running, so that replicas can stay consistent with the master. If a replica goes down, it is best to bring it back online as rapidly as possible.

If a node becomes inactive, JE HA will prevent the other members in the group from deleting log files that have been reclaimed by the cleaner if the inactive node needs them for its replication stream. JE HA only holds back this file deletion for a configurable period of time. During this withholding period, log files will begin to accumulate across the nodes in the group and the active members risk running out of disk storage during this time period. If the replica comes back after this withholding period, a contiguous replication stream may not be available at any of the active nodes. As a result the replica will need to perform a Network Restore operation, which copies over the log files from one of the active nodes and then resumes replaying the replication stream from that point forwards. The Network Restore operation can be disk I/O and network resource intensive if the environment is large.

JE HA Scalability

The JE HA architecture supports single-writer, multiple-reader replication. This scaling of read operations and write operations is addressed separately in the sections below.

Write Scalability

With a single-writer, the scalability of write operations is determined primarily by the capacity of the master to handle write operations, which in turn is a function of the hardware resources, available at the master. The master may be able to decrease some of the I/O load that would typically be associated with a standalone JE application, by using durability policies that commit to the network, and by off-loading read operations that did not require absolute consistency to the replicas.

In the current implementation, the master is responsible for maintaining the replication streams to all the other members. Each replication stream places an incremental CPU load at the writer. To help ameliorate this issue, each replication stream is maintained by a dedicated feeder thread that can take advantage of multi-core machines to maintain parallel replication streams. Each replication stream also places an incremental network load on the subnet where the master resides and write-intensive applications may exceed the available network bandwidth on that subnet.

Write Partitioning

Horizontal write scalability can be achieved by partitioning the data across multiple replicated environments, if the application requirements permit it. JE HA does not currently provide explicit API support for such partitioning.

The following figure illustrates how data can be partitioned using three replication environments: RE1, RE2 and RE3, which are, associated with groups G1, G2 and G3. The data could either be uniformly distributed across the three groups or could be partitioned based upon expected write/update frequency. Each group in this example consists of three electable nodes and a monitor node. The groups are distributed across three machines MC1, MC2 and MC3.

HA Write Partitioning
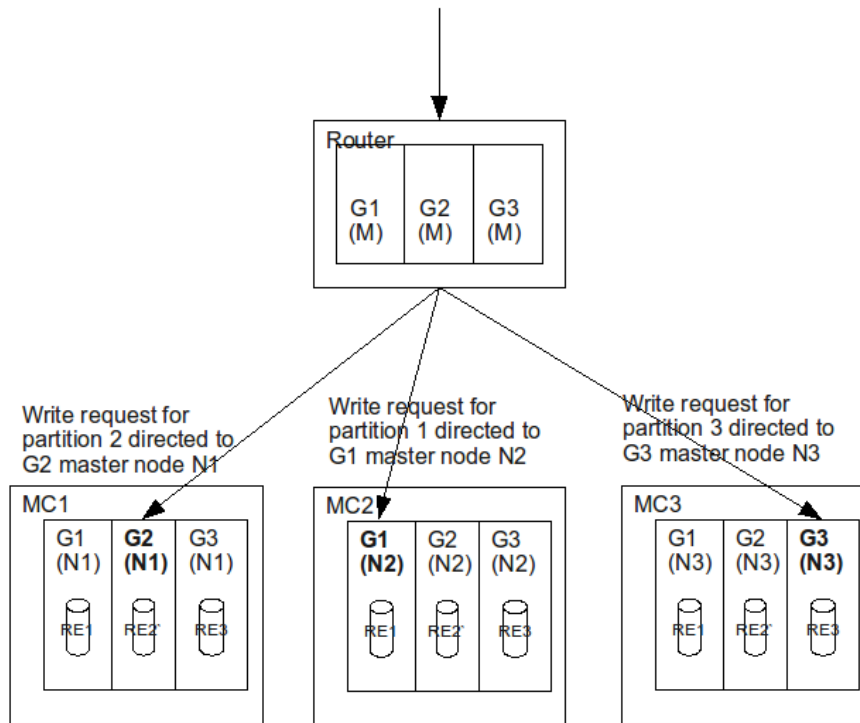(3 partitions, 3 groups with 3 electable nodes/group)

Figure 4 - HA Write Partitioning

The router in this case, contains three monitor instances: G1(M), G2(M), G3(M), and one monitor for each group. The router must be aware of the data partitioning scheme and routes data requests to a specific node based on the data that will be accessed by the request, as well as the type of the request: read versus write.

Read Scalability

Read scalability is achieved in JE HA through multiple read-only replicas. There is no implementation-specific limit to the number of nodes that can be part of a replication group. The addition of a new replica does not increase the resource requirements on the existing replicas. There is incremental load on the writer and the network as described in the preceding section.

The use of additional nodes to improve read scalability is completely transparent to the JE HA application. A new node can be added to a running replication group and can start serving requests as soon as it has a sufficiently current local copy of the replicated environment.

Conclusion

JE HA extends the JE API in a straightforward way to provide fast, reliable, and scalable data management, embedded within an application.

JE HA extends the JE API in a straightforward way to provide fast, reliable, and scalable data management, embedded within an application. It is highly configurable, permitting you to make durability, consistency and performance trade-offs, as appropriate for a wide range of applications.

You can download Oracle Berkeley DB Java Edition at:
http://www.oracle.com/technology/software/products/berkeley-db/je/index.html

You can post your comments and questions at the Oracle Technology Network (OTN) forum for Oracle Berkeley DB Java Edition at:

http://forums.oracle.com/forums/forum.jspa?forumID=273

For sales or support information email: berkeleydb-info_us@oracle.com

Find out about new product releases by sending an email to: bdb-join@oss.oracle.com

ORACLE®

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

Integrated Cloud Applications & Platform Services

Oracle Berkeley DB Java Edition High Availability
November 2016
Authors: AJ,MS,MB

Oracle is committed to developing practices and products that help protect the environment