Detailed Technical Design

Project Name Hydra – High Availability Solution

Project Number PR00005527 Initial Release Date

Last Revised Date

Author

Approvals

**Use the Approval Repository to obtain and document the required approvals.**

History of Revisions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Revision Date | Version # | Description of Revision | Revision Requested By | Revision Made By |
| 12th Aug 2015 | 0.1 | Initial draft |  | Peter Lord |
| 13th Aug 2015 | 0.2 | Updates following initial internal Tibco review |  | Peter Lord |
| 18th Aug 2015 | 0.3 | Updates following review meeting |  | Peter Lord |
| 20th Aug 2015 | V20150820 | Updates following desk review |  | Peter Lord |

Table of Contents

Overview 3

Problem Statement 3

Scope 3

Known Technology and/or Business Risks 3

Dependencies 3

Assumptions 3

References 3

Technical Approach 4

Current Design 4

Proposed New Design 4

Disaster Recovery 16

Security 16

Processing Logic 16

Program Comments 16

Data 17

Input 17

I/O 17

Triggers 17

Output 17

Audit and Control 17

Audit Practices 17

Control Routines 17

Error Codes 17

Program and Compile Information 18

Source Language 18

Runtime Environment 18

Module List 18

Copy Code 18

Appendixes 20

Appendix A: Screens/Windows 20

Appendix B: Reports 20

Appendix C: Data Access 20

# Overview

## Problem Statement

## Scope

## Known Technology and/or Business Risks

## Dependencies

1. Active Spaces Transactions 2.5.2 (or later)
   1. <https://edelivery.tibco.com/storefront/view-component-download.ep?partNumber=01008042>
2. High Availability Service Framework 2.0.2 (or later)
   1. http://downloads.fluency.kabira.com/sites/haservice/

## Assumptions

1. The network is fault tolerant – ie there are at least two physical network paths from any node to any other node. The application is not aware of any single network fault.
2. The latency between nodes (especially between data centers) is sufficiently low.

## References

1. [Active Spaces Transactions Developers Guide](https://devzone.tibco.com/display/DOC/ActiveSpaces+Transactions+2.5.1+-+Product+Documentation+-+developersguide+-+index.html)
2. [High Availability Service Framework](http://downloads.fluency.kabira.com/sites/haservice/)
3. [Secondary Store](https://devzone.tibco.com/display/DOC/ActiveSpaces+Transactions+2.5.1+-+Product+Documentation+-+developersguide+-+ch10.html)
4. [Transaction isolation level](https://devzone.tibco.com/display/DOC/ActiveSpaces+Transactions+2.5.1+-+Product+Documentation+-+developersguide+-+ch04s02.html)

# Technical Approach

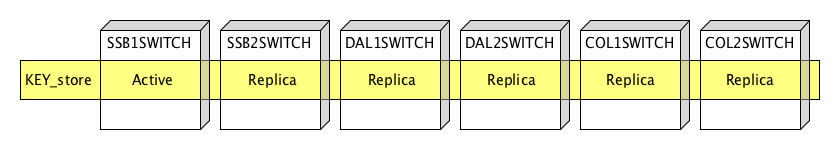
## Current Design

## 

## Proposed New Design

### Replication of Zone PIN keys

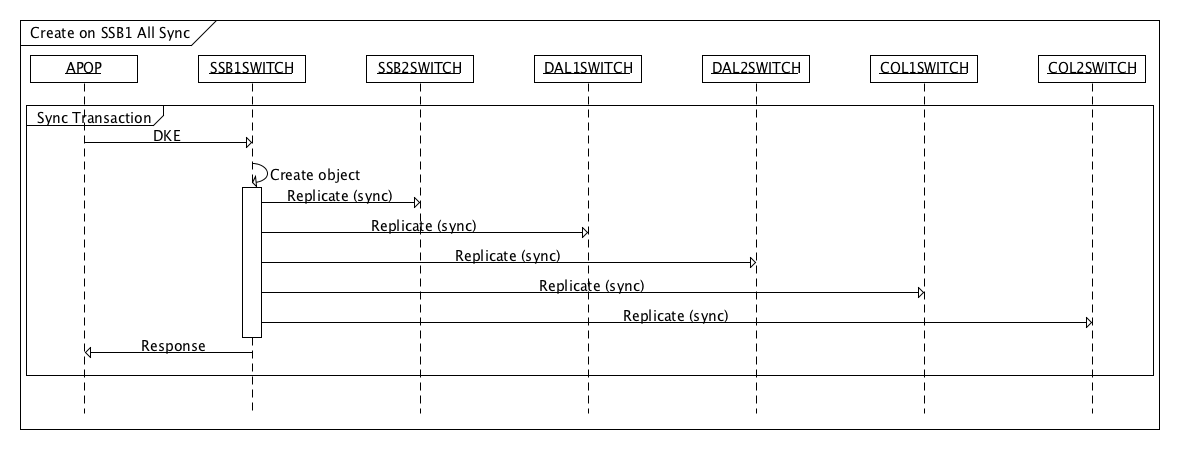
Replication of Zone PIN Keys (ZPK) across all the servers can be achieved using Active Spaces Transaction (AST) partitioning (see [1]) :-



The features of this approach are :-

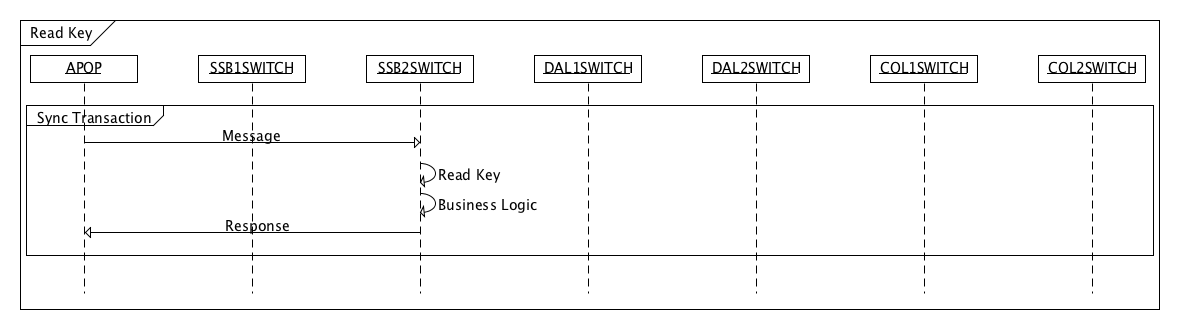
1. When a ZPK managed object is created, it is mapped to a partition and replicated to all nodes
2. When a ZPK managed object is deleted, its automatically deleted from all nodes
3. ZPK reads are always local to the node
4. Replication is configured to be synchronous (updated within the same transaction)

When the ZPK is created on any node, AST will automatically replicate the key to all nodes: -



Note that the object is replicated to all nodes concurrently.

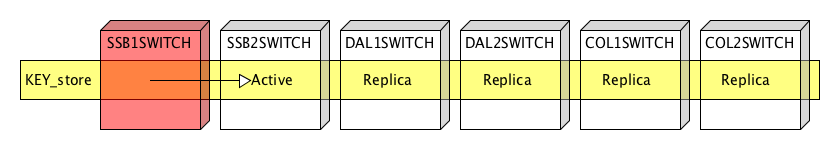
Since the ZPKs are all replicated to all nodes, a read of the ZPK is local :-



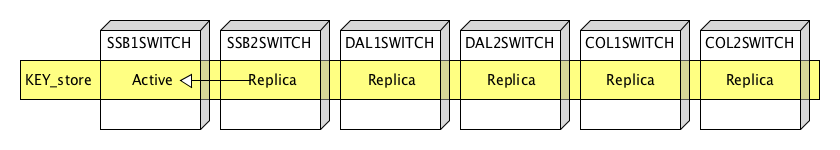
### Server failure

Should an individual server fail and is later recovered (for example hardware failure or planned maintenance), responsibility for the active node of a partition is updated. For example if SSB1SWITCH should fail then: -

1. The first node in the replica list is promoted to become the new active node. In this case the active node for partition KEY\_store becomes SSB2SWITCH
2. Assuming quorumPercentage is set to 99%, then the application quorum notifier is called on the remaining nodes to disable key exchange processing
3. No loss of data



When the SSB1SWITCH node is recovered, partition data is synchronized back from the remaining nodes. There is no need to read any ZPKs from disk. Key exchange processing is resumed automatically.



In the case of all nodes taken offline (so there are no ZPKs in memory), it is expected that an administration command is run to read the ZPKs from disk.

### Proposed haservice configuration

The above behavior can be realized in configuration only. A proposed haservice (see [2]) configuration file is shown below: -

// Overview

// haservice configuration for hydra

//

configuration "hydra" version "1.0" type "com.tibco.haservice"

{

configure com.tibco.haservice

{

HighAvailability

{

// Define list of nodes

//

// A quorum is set so that if one node can't be "seen", then the

// notifier is called. In the notifier, the stopping of DKE can be implemented

//

nodeGroups =

{

{

name = "All\_nodes";

quorum = true;

quorumNotifier = "com.discover.hydra.lifecycle.QuorumNotifier";

quorumPercentage = 99;

nodeList =

{

{

name = "SSB1SWITCH";

},

{

name = "SSB2SWITCH";

},

{

name = "DAL1SWITCH";

},

{

name = "DAL2SWITCH";

},

{

name = "COL1SWITCH";

},

{

name = "COL2SWITCH";

}

};

}

};

// Static partition groups

//

// One partition is sufficient

//

staticPartitionGroups =

{

{

name = "KEYS\_group";

loadOnNodeGroups = { "All\_nodes" };

partitions =

{

{

name = "KEYS\_store";

// Determine how objects are replicated during a migrate of an

// object partition.

//

forceReplication = false;

// Failover performance settings.

//

// numberOfThreads should be set to the number of cores for

// shortest failover time

//

// objectsLockedPerTransaction set low will increase failover time

// but reduce contention with on-going work. Setting it high will

// reduce failover time but increase contention with on-going work.

// Testing should be used to determine optimal settings

//

objectsLockedPerTransaction = 1000;

numberOfThreads = 8;

// Before migrating the partition data, remove any instances that

// exist on the local node.

//

enableAction = JOIN\_CLUSTER\_PURGE;

// Perform any object migrations needed to migrate partitions owned by

// this node to the first order replica node, and remove this node

// from all replica lists. Audits disabled

//

disableAction = LEAVE\_CLUSTER\_FORCE;

activeNode = "SSB1SWITCH";

replicas =

{

{

name = "SSB2SWITCH";

type = SYNCHRONOUS;

},

{

name = "DAL1SWITCH";

type = SYNCHRONOUS;

},

{

name = "DAL2SWITCH";

type = SYNCHRONOUS;

},

{

name = "COL1SWITCH";

type = SYNCHRONOUS;

},

{

name = "COL2SWITCH";

type = SYNCHRONOUS;

}

};

}

};

}

};

// Map the key class to partition group. Partition will be based on the hash of

// the clientNodeId to get a uniform distribution

//

// Note, however, since we only have one partition this mapper will always map

// the key to the single partition. An option is to implement an application

// specific mapper that just statically maps to KEYS\_store

//

partitionMappers =

{

{

partitionGroup = "KEYS\_group";

hashManagedClasses =

{

{

className = "com.discover.hydra.ManagedZPK";

fieldName = "clientNodeId";

}

};

}

};

};

};

};

Mentioned in the above configuration is a quorum notifier – in this case the notifier will be called when quorum fails. Example java code is shown below: -

**package** com.discover.hydra.lifecycle;

**import** com.kabira.platform.annotation.KeyField;

**public** **class** QuorumNotifier **extends** com.tibco.haservice.notifiers.QuorumNotifier {

**protected** QuorumNotifier(@KeyField(fieldName = "m\_nodeGroup") String nodeGroup) {

**super**(nodeGroup);

}

@Override

**public** **void** postDeactivate() {

}

@Override

**public** **boolean** preDeactivate(String reason) {

System.*out*.println("QuorumNotifier: preDeactivate "+reason);

// FIX THIS - add in disable of DKE

// Return false to avoid disabling the partitions

//

**return** **false**;

}

}

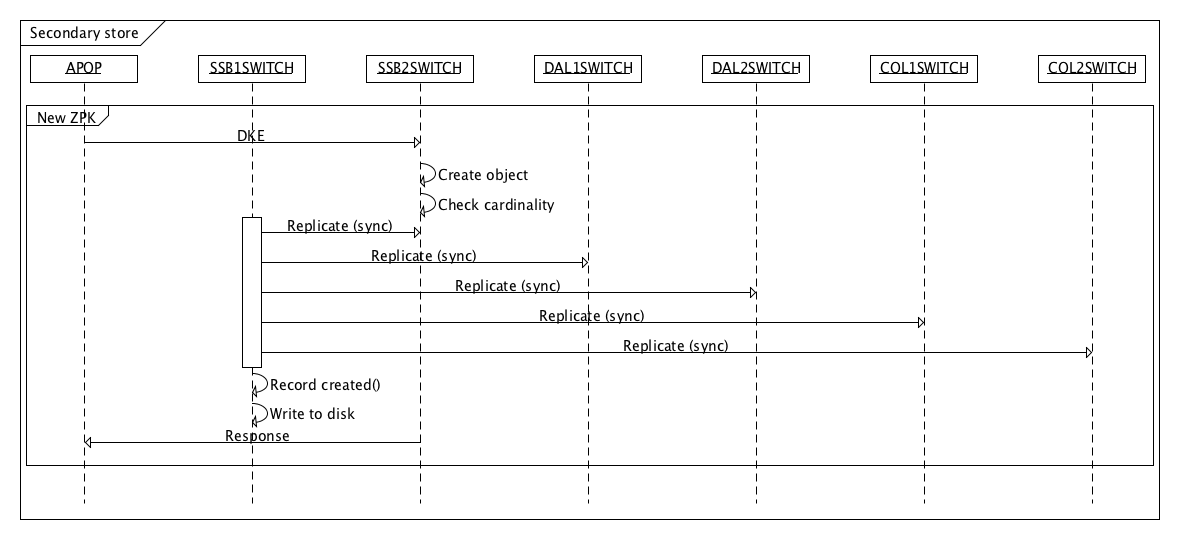
Currently haservice does not support a quorum notifier when the quorum is re-established – indeed failure resolution will usually involve manual steps (for example, testing of the failed network or server) and so automatic resumption of DKE might be problematic.

However, a node notifier can be used to keep track of the state of remote nodes (see later).

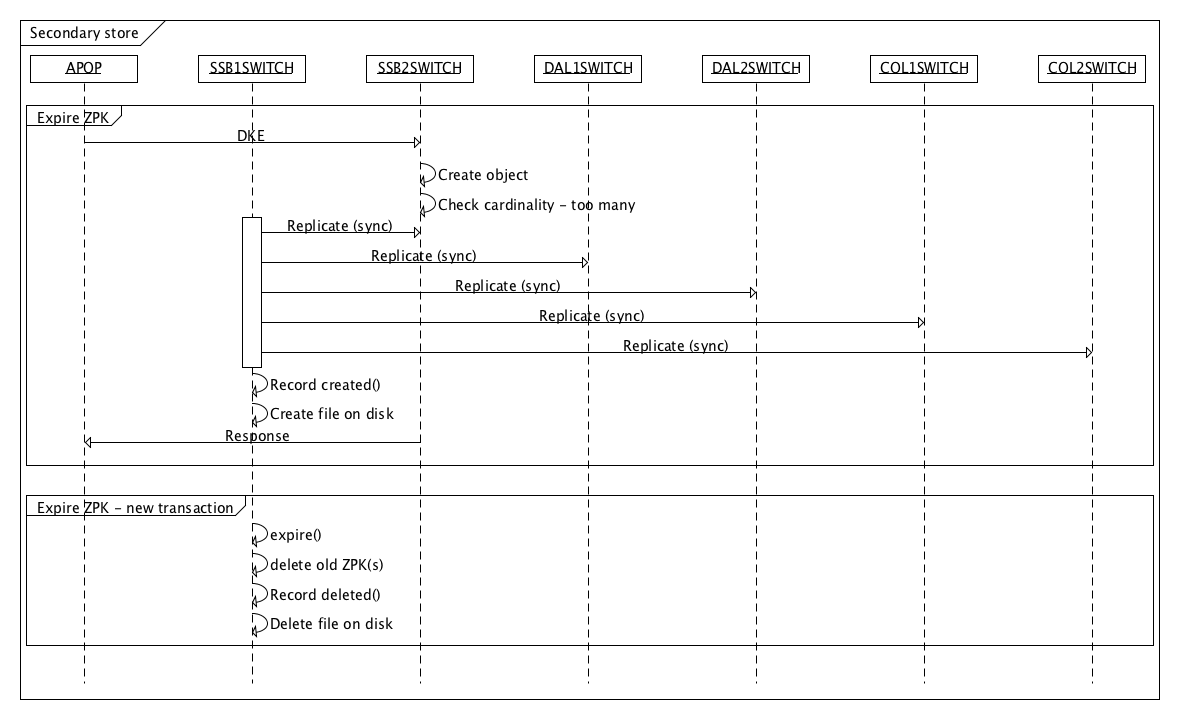
### Secondary store

Each ZPK is required to be written to disk – it is suggested that the secondary store (see [3]) is used for this purpose. The secondary store allows easy encapsulation of mirroring the managed object to disk. Each key held in highly available memory is mapped down to a file.

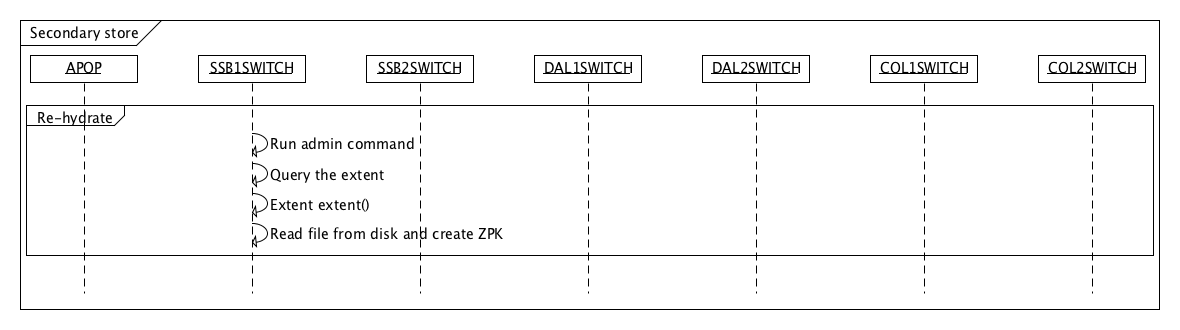
When the object is created, the secondary store is invoked to write to disk: -



Rather than having a scheduled task to find and remove old ZPKs, when a new ZPK is created, any old ZPKs are scheduled for deletion: -



Finally, when the administration command is run to load ZPKs from disk (for example on cold start), the administration command implementation can execute a extent query that triggers the secondary store to fetch from disk: -



See below for a skeleton implementation of the secondary store.

**package** com.discover.hydra;

**import** com.kabira.platform.DeleteTrigger;

**import** com.kabira.platform.LockMode;

**import** com.kabira.platform.ManagedClassError;

**import** com.kabira.platform.ManagedObject;

**import** com.kabira.platform.annotation.Managed;

**import** com.kabira.store.Extent;

**import** com.kabira.store.Record;

@Managed

**public** **class** StoreZPK **implements** DeleteTrigger {

**private** RecordZPK recordZPK;

**private** ExtentZPK extentZPK;

**public** StoreZPK() {

// Create the secondary store objects that we need

//

**this**.recordZPK = **new** RecordZPK();

**this**.extentZPK = **new** ExtentZPK();

}

@Override

**public** **void** uponDelete() {

// tidy-up

//

**if** (**this**.recordZPK != **null**) {

ManagedObject.*delete*(**this**.recordZPK);

}

**if** (**this**.extentZPK != **null**) {

ManagedObject.*delete*(**this**.extentZPK);

}

}

**public** **class** RecordZPK **extends** Record<ManagedZPK> {

**public** RecordZPK() {

**super**(ManagedZPK.**class**);

}

**protected** RecordZPK(Class<ManagedZPK> klass) **throws** ManagedClassError {

**super**(klass);

}

@Override

**public** **void** deleted(ManagedZPK zpk) {

System.*out*.println("StoreZPK: Deleted " + zpk);

// FIX THIS - here the external file can be removed

//

}

@Override

**public** **void** modified(ManagedZPK zpk) {

System.*out*.println("StoreZPK: Modified " + zpk);

// FIX THIS - expect this to be unused since keys arn't modified

//

}

@Override

**public** **void** created(ManagedZPK zpk) {

System.*out*.println("StoreZPK: Created " + zpk);

// FIX THIS - here the zpk object can be written to an external file

//

// There is one key per file, so the filename is based on clientNodeId

//

// If all indexes are required to be written to disk, then the filename

// should also contain the index.

//

}

}

**public** **class** ExtentZPK **extends** Extent<ManagedZPK> {

**public** ExtentZPK() {

**super**(ManagedZPK.**class**);

}

**protected** ExtentZPK(Class<ManagedZPK> klass) **throws** ManagedClassError {

**super**(klass);

}

@Override

**public** **void** extent(Class<? **extends** ManagedZPK> zpk, LockMode lockModes) {

System.*out*.println("StoreZPK: Queried");

// FIX THIS - here the zpk object can be synced to disk. So :-

//

// 1) if object in memory is not on disk, export

// 2) if a file on disk is not in memory, import

//

// The administration command, rehydrate, will perform an extent query which

// will call this operation.

//

// On partition failover, the partition notifier will also perform an extent

// query

//

}

}

}

One feature of the secondary store is that it’s always the currently active node for that particular object that will perform the write to disk. This means that if the ZPKs need to be re-read from disk, then the files are located on the right node – with replicated managed objects, there is no need for each node to load the same ZPK. This does mean that should a node fail, on failover the newly active node will need to export the ZPKs its now active for to disk. This can be achieved with a node notifier: -

**package** com.discover.hydra.lifecycle;

**import** java.util.HashSet;

**import** com.discover.hydra.ManagedZPK;

**import** com.kabira.platform.ManagedObject;

**import** com.kabira.platform.annotation.Asynchronous;

**import** com.kabira.platform.highavailability.Partition;

**import** com.kabira.platform.highavailability.PartitionManager;

**import** com.kabira.platform.property.Status;

**public** **class** NodeNotifier **extends** com.kabira.platform.highavailability.NodeNotifier {

**private** **final** **static** String *nodeName* = System.*getProperty*(Status.*NODE\_NAME*);

**private** **final** **static** HashSet<String> *remoteNodes* = **new** HashSet<String>();

@Override

**protected** **void** active(String node) {

System.*out*.println("NodeNotifier: Node "+node+" is active ");

// Keep track of nodes known about by this node

//

*remoteNodes*.add(node);

// simple check to see if we now have all nodes

//

**if** (*remoteNodes*.size() == expectedRemoteNodes()) {

// FIX THIS - resume key exchange

//

}

System.*out*.println("NodeNotifier: Node "+node+" is active end");

}

@Override

**protected** **void** unavailable(String node) {

System.*out*.println("NodeNotifier: Node "+node+" is unavailable");

// Keep track of nodes known about by this node

//

*remoteNodes*.remove(node);

// at this point, a partition could be active now on this node

//

**this**.checkMigration();

System.*out*.println("NodeNotifier: Node "+node+" is unavailable end");

}

/\*\*

\* Check if a partition has been migrated

\*

\* This is run in a separate transaction to avoid lock contention with any locks

\* held by the node notifier

\*/

@Asynchronous

**private** **void** checkMigration() {

// if a partition is now active on this node, do an extent query

// which will trigger the secondary store to sync to disk

//

**for** (Partition p : PartitionManager.*getPartitions*()) {

**if** (p.getActiveNode().equals(*nodeName*)) {

// This takes a read lock on all ZPK's - this is not expected to be a problem since ZPKs

// are not updated.

//

**for** (ManagedZPK zpk : ManagedObject.*extent*(ManagedZPK.**class**)) {

// FIX THIS - this assumes one partition - if multiple partitions then zpk need to be checked that

// its active on this local node

//

}

}

}

}

/\*\*

\* Find the expected number of remote nodes

\*

\* **@return**

\*/

**private** **int** expectedRemoteNodes() {

**int** expected = 0;

// find expected number of nodes

//

**for** (com.tibco.haservice.config.Partition p : ManagedObject.*extent*(com.tibco.haservice.config.Partition.**class**)) {

**if** (p.replicas.length > expected) {

expected = p.replicas.length;

}

}

**return** expected;

}

}

### Managed object definition

The managed object used to hold the ZPK, ManagedZPK, should have sufficient keys defined to: -

* When reading from disk, avoid creating two objects with the same data
* Query for the latest ZPK
* Expire old ZPKs

The following definition achieves this: -

**package** com.discover.hydra;

**import** java.util.Date;

**import** com.kabira.platform.KeyFieldValueList;

**import** com.kabira.platform.KeyManager;

**import** com.kabira.platform.KeyOrderedBy;

**import** com.kabira.platform.KeyQuery;

**import** com.kabira.platform.LockMode;

**import** com.kabira.platform.ManagedObject;

**import** com.kabira.platform.annotation.Asynchronous;

**import** com.kabira.platform.annotation.KeyField;

**import** com.kabira.platform.annotation.KeyList;

**import** com.kabira.platform.annotation.Key;

**import** com.kabira.platform.annotation.Managed;

@Managed

@KeyList (keys = {

@Key(name = "ByClient", fields = { "clientNodeId" }, unique = **false**),

@Key(name = "ByDate", fields = { "clientNodeId", "index", "activeDate" }, unique = **false**, ordered=**true**),

@Key(name = "ByDistinct", fields = { "clientNodeId", "index", "activeDate" }, unique = **true**),

})

**public** **class** ManagedZPK {

**private** **final** **static** **int** *MAX\_KEYS* = 3;

**public** **final** String clientNodeId;

**public** **final** String index;

**public** String keyValue;

**public** **final** Date activeDate;

**public** ManagedZPK(@KeyField(fieldName = "clientNodeId") String clientNodeId,

@KeyField(fieldName = "index") String index,

@KeyField(fieldName = "activeDate") Date activeDate) {

**this**.clientNodeId = clientNodeId;

**this**.index = index;

**this**.activeDate = activeDate;

}

/\*\*

\* For debug purposes only - note this will be executed on the currently active node

\*/

@Override

**public** String toString() {

**return** "ManagedZPK [clientNodeId=" + clientNodeId + ", index=" + index

+ ", keyValue=" + keyValue + ", activeDate=" + activeDate + "]";

}

/\*\*

\* Get or create an unique ZPK instance. This will be called when a new key is created by

\* the key exchange process and also when the secondary store reads keys from disk (caused

\* by the administration command executing an extent query ). In both cases we need to

\* make sure we don't create two objects representing the same ZPK - hence the use of a

\* unique key

\*

\* **@param** clientNodeId Client Node ID

\* **@param** index Index

\* **@param** activeDate active date

\* **@param** keyValue contents of the key

\* **@return** Instance of ManagedZPK

\*/

**public** **static** ManagedZPK getOrCreateZPK(String clientNodeId, String index, Date activeDate, String keyValue) {

KeyManager<ManagedZPK> km = **new** KeyManager<ManagedZPK>();

KeyQuery<ManagedZPK> keyQuery = km.createKeyQuery(ManagedZPK.**class**, "ByDistinct");

KeyFieldValueList keyList = **new** KeyFieldValueList();

keyList.add("clientNodeId", clientNodeId);

keyList.add("index", index);

keyList.add("activeDate", activeDate);

KeyFieldValueList additionalFields = **new** KeyFieldValueList();

additionalFields.add("keyValue", keyValue);

keyQuery.defineQuery(keyList);

ManagedZPK zpk = keyQuery.getOrCreateSingleResult(LockMode.*READLOCK*, additionalFields);

// check if we should expire this key

//

km = **new** KeyManager<ManagedZPK>();

keyQuery = km.createKeyQuery(ManagedZPK.**class**, "ByDate");

keyList = **new** KeyFieldValueList();

keyList.add("clientNodeId", clientNodeId);

keyList.add("index", index);

keyQuery.defineQuery(keyList);

**if** (zpk != **null** && keyQuery.cardinality() > *MAX\_KEYS*) {

zpk.expire();

}

**return** zpk;

}

/\*\*

\* Get the latest instance of the ZPK. This will be called during normal processing.

\*

\* Note that we believe this code already exists in the current implementation

\*

\* **@param** clientNodeId Client Node ID

\* **@param** index Index

\* **@return** Instance of ManagedZPK, or null if not found

\*/

**public** **static** ManagedZPK getLatestZPK(String clientNodeId, String index) {

KeyManager<ManagedZPK> km = **new** KeyManager<ManagedZPK>();

KeyQuery<ManagedZPK> keyQuery = km.createKeyQuery(ManagedZPK.**class**, "ByDate");

KeyFieldValueList keyList = **new** KeyFieldValueList();

keyList.add("clientNodeId", clientNodeId);

keyList.add("index", index);

keyQuery.defineQuery(keyList);

// date is not included in the query, so the results are sorted by date

//

// Specifically this is a local ( not cluster ) query

//

**return** keyQuery.getMaximumResult(LockMode.*READLOCK*);

}

/\*\*

\* Called on the latest instance of the ZPK to remove older keys

\*

\* This is executed in a separate transaction on the currently active node

\*/

@Asynchronous

**public** **void** expire() {

KeyManager<ManagedZPK> km = **new** KeyManager<ManagedZPK>();

KeyQuery<ManagedZPK> keyQuery = km.createKeyQuery(ManagedZPK.**class**, "ByDate");

KeyFieldValueList keyList = **new** KeyFieldValueList();

keyList.add("clientNodeId", **this**.clientNodeId);

keyList.add("index", **this**.index);

keyQuery.defineQuery(keyList);

**int** i=1;

**for** (ManagedZPK zpk : keyQuery.getResults(KeyOrderedBy.*DESCENDING*, LockMode.*NOLOCK*) ) {

**if** (i++ > *MAX\_KEYS*) {

System.*out*.println("expire: deleting "+zpk);

ManagedObject.*delete*(zpk);

}

}

}

}

### Transaction isolation level

AST supports two transaction isolation levels – the default Serializable and Read Committed (See [4]). Whilst we expect the default, Serializable , to be optimal in this case, its recommended to allow this to be configurable so that performance tests of both options can be executed.

### Administration targets

To support this design, the following administration targets are required: -

* administrator stop keyexchange
  + Used to manually pause the key exchange on this node. Running a stop keyexchange when key exchange is already stopped should have no effect.
* administrator start keyexchange
  + Used to re-start the key exchange on this node. As well as to resume from a manual stop above, this is also used when the HA quorum fails and the switch automatically stop key exchange. Running a start keyexchange when key exchange is already started should have no effect.
* administrator display keyexchange
  + Used to display the status of key exchange (started or stopped) on this node.
* administrator load keys
  + Re-load ZPKs from disk – used for initial load of static keys as well as load of all keys should the whole cluster be stopped. The implementation will do an extent query of the managed object which will cause the secondary store to fetch from disk. Thus this command can be run on any node.

Example code is shown below: -

**package** com.discover.hydra.lifecycle;

**import** com.kabira.platform.management.Command;

**import** com.kabira.platform.management.ManagementTarget;

**import** com.kabira.platform.management.Target;

@ManagementTarget(name = "keyexchange", description = "Key exchange administration")

**public** **class** KeyExchangeTarget **extends** Target {

@Command(description = "Stop key exchange")

**public** **void** stop() {

// FIX THIS - stop the key exchange on this node

//

commandComplete();

}

@Command(description = "Start key exchange")

**public** **void** start() {

// FIX THIS - start the key exchange on this node

//

commandComplete();

}

@Command(description = "Display key exchange")

**public** **void** display() {

// FIX THIS - Display the key exchange status on this node

//

commandComplete();

}

}

**package** com.discover.hydra.lifecycle;

**import** com.kabira.platform.management.Command;

**import** com.kabira.platform.management.ManagementTarget;

**import** com.kabira.platform.management.Target;

@ManagementTarget(name = "keys", description = "Keys administration")

**public** **class** KeysTarget **extends** Target {

@Command(description = "Re-load keys from disk")

**public** **void** load() {

// FIX THIS - Re-load the keys from disk by executing an extent query

//

commandComplete();

}

}

### Lifecycle of notifiers

The notifiers above can be created via a component notifier – so when the application starts these are created automatically: -

**package** com.discover.hydra.lifecycle;

**import** com.discover.hydra.StoreZPK;

**import** com.kabira.platform.ManagedObject;

**import** com.kabira.platform.component.Notifier;

**import** com.kabira.platform.management.Target;

**public** **class** ComponentNotifier **extends** Notifier {

@Override

**protected** **void** preConfigurationInitialize() {

// On startup, create node notifier

//

**new** NodeNotifier();

// create store

//

**new** StoreZPK();

// Register admin target

//

Target.*register*(KeyExchangeTarget.**class**);

Target.*register*(KeysTarget.**class**);

}

@Override

**protected** **void** postConfigurationTerminate() {

// Tidy-up

//

**for** (NodeNotifier notifier : ManagedObject.*extent*(NodeNotifier.**class**)) {

ManagedObject.*delete*(notifier);

}

**for** (StoreZPK store : ManagedObject.*extent*(StoreZPK.**class**)) {

ManagedObject.*delete*(store);

}

Target.*unregister*("keyexchange");

Target.*unregister*("keys");

}

}

However, should these components require configuration, then an alternative is to create these when application configuration is activated.

### Summary

The following gives a summary of the high availability’s behavior: -

#### Cold start

1. All nodes started
2. Load and activate haservice configuration on all nodes
3. Run **administrator rehydrate hydra** on one node – this will trigger the secondary store to load all keys from disk

#### Processing transactions

1. Processing can be sent to any node
2. ZPKs are read from local shared memory

#### Key exchange transaction

1. DKE message sent to any node
2. New ZPK is created and automatically replicated to all nodes
3. Secondary store is triggered to copy ZPK to disk
4. Any old ZPKs are removed (which in turn triggers the secondary store to remove or rename the external file)

#### Node failure (planned or unexpected)

1. If active node fails, then active responsibility is passed to first replica and node is removed from node list. New active node will export all keys to local disk
2. If replica node fails, then replica is removed from node list
3. Quorum notifier called to disable DKE messages
4. Node is restored
5. Haservice configuration loaded and activated – this will re-sync data
6. Key exchange can be manually resumed (or automatically via a node notifier)

### Task list

The following outlines the main implementation tasks required: -

1. Create haservice configuration file
2. Implement secondary store functions: -
   1. Create an external file from shared memory data
   2. Load an external file and create a ZPK in shared memory
   3. Delete (or backup) an external file
   4. Iterate over all external files and create ZPKs in shared memory
3. Implement node notifier: -
   1. Iterate over all ZPKs if this node is now active for the partition
   2. Optionally, keep track of available nodes to automatically re-enable key processing
4. Implement quorum notfier: -
   1. When quorum fails, disable key processing
5. Implement ZPK managed object with suitable keys and queries
6. Implement administration targets
7. Implement lifecycle operations to create the notifiers above

## Disaster Recovery

## Security

.

## Processing Logic

## Program Comments

# Data

}

//Backout method?

}

## Input

## I/O

## Triggers

## Output

## Audit and Control

## Audit Practices

## Control Routines

## Error Codes

| Code | Message | Description | Return Value |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

# Program and Compile Information

## Source Language

## Runtime Environment

## Module List

| Module | Description |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## 

## Copy Code

| Copy/Include | Description |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Appendixes

Appendixes are not required. You can delete appendixes that are not needed for your project.

## Appendix A: Screens/Windows

## Appendix B: Reports

## Appendix C: Data Access