XADisk 1.1 User Guide

http://xadisk.java.net/

This document is the User Guide for XADisk 1.1 and describes the features, usages and example codes.

Copyright © 2010, 2011 Nitin Verma (project owner for XADisk). All rights reserved.

The software XADisk and its associated -

a) **Binaries and other files**
b) **Source code**
c) **Java API documentation**
d) **User Guide (this document)**

are all being made available to the public under the terms specified in the license "Eclipse Public License 1.0" located at http://www.opensource.org/licenses/eclipse-1.0.php.

Anyone making use of any of the above XADisk artifacts (files, source codes, documentation etc) must go through the license text first.

Table of Contents

Prerequisites

A good general understanding of Java, File Systems and Transactions should be enough for using XADisk for most of the common cases. In addition, if you are planning to use XADisk as a JCA Resource Adapter, you would need to be familiar with deployment/use of JCA Resource Adapters.

Additional Resources

You can keep the following as a reference as you work through this guide.

a) XADisk example codes

b) XADisk JavaDoc

These are available from the XADisk project page.

Introduction

XADisk is an open-source project within the Enterprise-Java community of java.net (http://java.net/). All information related to this project can be accessed from the project page, http://xadisk.java.net.

The name *XADisk* derives from *XA* and *Disk*. XA refers to an open standard from The Open Group (http://www.opengroup.org/), and is about distributed transaction processing. XA, with little modifications, has been brought into JavaEE world as JTA. If you are going to use XADisk in local transactions only, you need not worry about XA/JTA (but for the sake of pronouncing XADisk, you can speak it as "x-a-disk"). XADisk acts as a wrapper around existing file-systems (e.g. NTFS, FAT, ext3) to provide a "transactional view" of them to the applications. This enables applications, written using Java or JavaEE technologies, to perform various kinds of I/O operations on these file-systems as if all the operations are part of a single transaction. Most of the time, these file-systems reside on Hard Disks; the word *Disk* in *XADisk* comes from there!

XADisk is all about interacting with the existing file-systems in a transactional manner. XADisk is not a file-system implementation in itself. The operating system on your machine is already having one or more file-systems installed on it. XADisk does not add another file-system to your machine; its job is to simply "wrap" them so that you get the transaction benefits (ACID, as they are known).

XADisk has been written in Java and uses some of the features introduced by Java5. So, it can be compiled and run using Java 5 or above. XADisk can be used by any application written using Java or JavaEE technologies. These include all kinds of Java applications, web-applications running in Java servers like Apache Tomcat, JavaEE applications running in JavaEE servers etc.

[As of its current implementation, XADisk is usable only for Java or JavaEE applications. As a future enhancement, we can write a thin adapter to enable non-Java applications also to work with XADisk running on a JVM.]

For a simple example of XADisk usage, take a Java application. This application wants to create a new file F1, write some data to it from some other file F2, and delete F2. These three operations can be performed using XADisk and with all those guarantees you would expect from a *transaction*: atomicity, consistency, isolation and durability. XADisk would maintain these *guarantees* even if the JVM crashes anywhere in the middle.

To give you another example where the *XA* kicks in, consider a JavaEE application. Among a couple of services, the JavaEE server would provide transaction related services to the application. The application wants to:

a) modify some data inside a typical database
b) create and write 15 new files in the local file system
c) send 10 messages to JMS queues
d) delete 2 files from a remote file system

But this application wants to have some guarantee; it wants all of the four kinds of operations done *inside* a single transaction, and seeks to get benefits of the ACID properties. If the database and the JMS provider support XA, the application can achieve its guarantees if it performs its I/O operations on the local and remote file-systems using XADisk (there will be two XADisk instances involved here; one running on the local machine, other running on the remote machine serving remote calls). This is because XADisk supports XA transactions in addition to resource-local transactions. XADisk follows the two-phase commit protocol of XA to ensure that it commits if-and-only-if all of the other resources (i.e. the database provider, the JMS provider and the remote XADisk instance) involved in the same transaction commit.

Before we go into more details, you may like to have a quick view of what XADisk can provide you:

a) Ensures necessary locking over files and directories to provide consistency among concurrent transactions. XADisk uses its own locks for such locking, not the native file-systems locks.

b) Can survive JVM crashes, while maintaining the required properties of local and global transactions.

c) Provides a configurable transaction time-out to prevent wasting of resources in case of erroneous client application.

d) Allows remote applications to connect and do file-system operations on the host machine; in the same way as applications running on the same JVM can.

e) Detects and remedies deadlocks.

f) Provides a consistent view of the file-system inside a transaction. This view also includes the changes (though not yet committed) done by the current transaction. For example, the files and directories created/deleted/modified in the transaction will be reflected in calls like *listFiles, read, fileExists* etc of the same transaction.

g) Can plug into any JavaEE 5 (or above) server via JCA 1.5 contracts. JavaEE applications can then view it as a JCA Resource Adapter.

h) Implements the XAResource contract of JTA/XA, which enables it to participate in global transactions.

i) The XA support of XADisk can be combined with the remote connection capabilities to have a single distributed transaction working on files/directories of multiple machines (including any other XA resource like Databases in the same distributed transaction).

j) Enables Message Driven Beans (MDB) in a JavaEE server to listen for changes in files/directories of local or remote file-systems, e.g.

creation of a new file can trigger an event to a registered MDB. Such event deliveries also happen inside an XA transaction.

XADisk Fundamentals

This chapter will provide some general information about XADisk which you should know before you go further deep.

Let us begin with defining an XADisk instance. You can think of an XADisk instance as a software module (XADisk being the software) which

a) has been loaded into memory
b) has been initialized
c) maintains a few threads to perform its operations
d) maintains some transient state in the runtime memory; JVM's memory to be precise
e) maintains some persistent state in transaction-logs etc

To start using XADisk, one needs to boot one or more XADisk instances. Booting is the process of bringing up the XADisk instance. XADisk API defines the methods that can be used for booting an XADisk instance. The booting process includes crash recovery if it is not a new XADisk instance. During crash recovery, XADisk will look through the transaction log files to collect transaction statuses during last crash/shutdown and will take appropriate actions.

Further, each XADisk instance is associated with exactly one System Directory. This directory should be accessible from the XADisk instance so that it can maintain its transaction logs and other few artifacts needed for its operation. Each such System Directory should not be tampered with any other software and should not be shared among multiple XADisk instances.

As mentioned in the Introduction, XADisk is a *wrapper* around the existing file-systems. Any kind of file-system that you can access from normal Java programs can also be wrapped by XADisk. To achieve the transactional behavior, the file system operations are performed *via* XADisk. It means, your

applications interact with XADisk APIs and XADisk in-turn interacts with the underlying file-system. So, in a way, XADisk sits between your applications and the underlying file-system. It can be noted that an XADisk instance is never explicitly informed of the file-system it is wrapping; the XADisk APIs simply accept the file/directory paths.

Now, we come to the multiplicity of XADisk instances. More than one XADisk instances can be alive within the same JVM. Further, more than one XADisk instances can be *wrapping* the same file-system. But there are restrictions. Two different XADisk instances must not be used for *wrapping* any common set of files/directories. So, for example, if an XADisk instance *xad1* is used for operations on all files/directories under */root/home*, then another instance *xad2* should not be used for operations on any of the files/directories under */root/home*.

The above paragraph reminds us of scalability. If your applications want to operate heavily on files/directories inside /root/data/, and if one XADisk instance is not enough to scale to the heavy load; one can use multiple XADisk instances and the load can be distributed among them. This can be done by using first instance for sub-tree /root/data/records1/, second instance for /root/data/records2 and like that.

XADisk can run inside any JVM in general. This implies that it can be booted from simple Java programs or from applications within any server running on a JVM (e.g. Apache's Tomcat). You can architect an approach of using XADisk in such server environments, as per your needs. For example, inside Apache's Tomcat, a web-application can be written which is dedicated to booting the required XADisk instances. Then, the other web-applications for your business logic can simply obtain a reference to those XADisk instances, start a new session, perform operations and commit the transaction of session.

An XADisk instance can also reside in a JavaEE Server. Here, it can be booted in the usual way, or can be deployed as a JCA Resource Adapter. JCA has its advantages like transaction, connection and life-cycle management, support for XA transaction etc. Deploying XADisk as a JCA Resource Adapter also brings up one instance of XADisk automatically.

So, how do applications interact with an XADisk instance to perform operations? To make calls to an XADisk instance, applications need not be running inside the same JVM as the XADisk instance. In case applications are on a different JVM, XADisk uses remote communication over TCP/IP to interact with the applications. In case the XADisk is known to be running in the same JVM, an application can use the more efficient approach of obtaining an object reference to invoke the APIs directly.

How does the life cycle of an XADisk instance terminate? It can be done by any application running in the same JVM using the shutdown method of *XAFileSystem* interface. It results in releasing of all resources held by the XADisk instance and no further methods can be now invoked on it. If the *XAFileSystem* reference is pointing to a remote XADisk instance, the effect of calling *shutdown* is simply to disconnect from the remote XADisk instance.

In case XADisk is deployed as a JCA Resource Adapter, the XADisk instance automatically gets shutdown during un-deployment of the Resource Adapter.

Booting XADisk

Booting is the process of bringing an XADisk instance up so that it can become useable. This chapter will explain how to boot an XADisk instance.

For booting an XADisk instance, you would need to have the following in your system environment:

a) Java5 or above
b) XADisk.jar
c) JCA 1.5 Jar
d) xadisk.dll (for Windows) / libxadisk.so for Unix.

Once you have prepared the system, you can start writing code to boot XADisk. It may be convenient for you if you keep a separate method for achieving this.

The main step for booting XADisk is setting various configuration properties inside a configuration object, as we will see below. Most of the configurations fall-back to default values if you don't set them; and the remaining few are mandatory. Below is a quick example for configuring and initializing an XADisk instance:

String XADiskSystemDirectory = "/home/systems/XADiskSystem1";

StandaloneFileSystemConfiguration configuration = new StandaloneFileSystemConfiguration(XADiskSystemDirectory, "instance-1");

configuration.setDeadLockDetectorInterval(15);

configuration.setLockTimeOut(30000);

configuration.setTransactionLogFileMaxSize(50000000L);

configuration.setTransactionTimeout(50);

configuration.setServerAddress("10.30.9.200");

configuration.setServerPort(5151);

configuration.setEnableRemoteInvocations(true);

XAFileSystem xaf = XAFileSystemProxy.bootNativeXAFileSystem(configuration);

xaf.waitForBootup(10000L);

The above sequence of steps completes the configuration of an XADisk instance and goes into a wait for it to complete the boot up process.

Note that there are few more optional configurations for performance tuning etc. The only mandatory fields you need to set on the configuration object are the *XADiskSystemDirectory* and *instanceId;* that is the reason they have been enforced in the constructor itself. Rest other optional fields have default values, which you can skip if you are aware of the default value and are comfortable with the default. Please see Appendix A for description of all available configuration properties.

Once you are done with setting properties in the configuration object, you boot the XADisk instance using the static method *bootNativeXAFileSystem(…)* of class *XAFileSystemProxy*. Now, what you get is a reference of type *XAFileSystem*. This will act as an entry point to the corresponding XADisk instance.

The final step is to wait for the XADisk instance to complete its boot up: *waitForBootup(-1 for waiting without timeout, else number of milliseconds)*.

Once the booting completes this XADisk instance is up and ready to use; both from the applications running on this JVM and applications running on remote JVMs.

Using XADisk in Java Applications

This chapter describes how any Java application can perform operations on an XADisk instance.

There are two cases to be considered for this. The first one is when the Java application is running on the same JVM as the XADisk instance. In this case, it can simply use the *XAFileSystem* reference returned during the boot method:

```
XAFileSystem xaf = XAFileSystemProxy.bootNativeXAFileSystem(configuration);
```

One can also retrieve this reference any time later using the following method, specifying the instance-id of the XADisk instance:

```
XAFileSystem xaf =
XAFileSystemProxy.getNativeXAFileSystemReference("instance-1");
```

The second case is when the Java application is running on a different JVM than the XADisk instance. Here, the communication happens via TCP/IP protocols underneath; your application code remains unaware of the details of the communication and remote invocation. In this case, the application simply grabs a reference to the remote *XAFileSystem* object using the following method:

```
XAFileSystem xaf = XAFileSystemProxy.
                        getRemoteXAFileSystemReference("10.30.9.200", 5151);

/*the parameters are the serverAddress and serverPort configuration properties
used during booting of the remote XADisk instance.*/
```

Once you have obtained the *XAFileSystem* reference, the way you invoke XADisk APIs remains same whether the target XADisk instance is running on the same JVM or a remote one.

When applications operate on files /directories through XADisk, they always need to be inside a transaction. Every such transaction is always bound to a *Session* object, and these Session objects are useable only for a single transaction. The code snippet below can give you a feel of how a typical transaction will flow (please also refer to the XADisk JavaDoc):

```
XAFileSystem xaf = … (a Native or Remote reference).

Session session = xaf.createSessionForLocalTransaction();

File f = new File("/testAPIs/test.txt");

 if(session.fileExists(f)) {

        XAFileInputStream xis = session.createXAFileInputStream(f);

        for (int i = 0; i < 100; i++) {
                byte a = (byte) xis.read();
                if( a== -1) {
                        break;
                }
                System.out.print(a);
```

```
            }

            xis.close();

            session.moveFile(f, new File("/testAPIs/test.txt___" +
        System.currentTimeMillis()));

        } else {
            session.createFile(f, false);

            XAFileOutputStream xafos = session.createXAFileOutputStream(f, false);

            byte[] buffer = new byte[100];

            for (int i = 0; i < 100; i++) {
                    buffer[i] = i*i;
            }

             xafos.write(buffer);

            xafos.close();
        }

          / * You can do more file operations here, by calling Session APIs for
    reading/writing/creating/deleting/updating/copying/moving files and directories.*/

        session.commit();
```

We first get a new *Session* object from the *XAFileSystem* object, which would also start and associate a new transaction with the *Session*. This *Session* object provides you with all of the APIs for file/directory operations. Two more entities for file operations are *XAFileInputStream* and *XAFileOutputStream,* which are also obtained using the *Session* object.

You can do any number of operations on this *Session* object and when you are done, you can call *session.commit()* to commit the associated transaction.

Enlisting XADisk into JTA Transactions

This chapter will describe how applications running in non-JCA environments can also benefit from the XA/JTA support from XADisk. For applications running in a JCA environment (e.g. JavaEE applications), the approach mentioned in this chapter should not be preferable.

Before we start, please note that the techniques discussed in this chapter make use of standard JTA APIs *programmatically*. Some application frameworks may provide abstraction on top of JTA for ease of programming, e.g. declarative transaction management and wrappers around standard interfaces like JTA *TransactionManager* and JTA *Transaction*. We will not go into specific application frameworks. One must be able to use XA support from XADisk with any framework supporting JTA.

For using XADisk inside XA transactions, one needs a JTA Transaction Manager (e.g. Atomikos). With JTA, a usual sequence to start a transaction is this:

---

*TransactionManager tm = /*the JTA vendor specific implementation of the Transaction Manager interface*/*

Transaction tx1 = tm.getTransaction();

In the above code, both *TransactionManager* and *Transaction* are standard JTA interfaces.

XA transactions can involve multiple resources, and the Transaction Manager acts as a coordinator to perform an XA transaction among these resources. There is another JTA interface called *XAResource,* which is implemented by each of these resources. To be able to participate in XA transaction, XADisk also implements the *XAResource* interface.

The *Session* object, which we discussed in the last chapter, has its own transaction associated with it. But *Session*'s transaction is different from XA as it can only involve a single resource, i.e. a single XADisk instance, and no other resource. To be able to include XADisk in XA transactions, you need a different XADisk object called *XASession*. You obtain *XASession* like this:

XAFileSystem xafs = … (a Native or Remote reference).

XASession xaSession = xafs.createSessionForXATransaction();

From this *XASession* object, we retrieve the XADisk implementation of *XAResource*, and then enlist this *XAResource* object with the XA transaction. If you are working with other XA-enabled resources (like Database) and want to involve them in the same XA transaction, you will also have to enlist their *XAResource* objects also:

XAResource xarXADisk = xaSession.getXAResource();
tx1.enlistResource(xarXADisk);

XAResource xarOracle = …….. ;
tx1.enlistResource(xarOracle);

XAResource xarMQ = …….. ;
tx1.enlistResource(xarMQ);

After enlisting the XADisk *XAResource* object, we can invoke various I/O operations on the *XASession* object in the same way as on the normal *Session* object. This is because both *Session* and *XASession* implement a common interface, called *XADiskBasicIOOperations*, meant for XADisk I/O operations.

```
File f = new File("/testAPIs/test.txt");

 if(xaSession.fileExists(f)) {

        xaSession.moveFile(f, new File("/testAPIs/test.txt___" +
        System.currentTimeMillis()));

} else {

        xaSession.createFile(f, false);

}

    / * You can do more file operations here, by calling XASession APIs for
reading/writing/creating/deleting/updating/copying/moving files and
directories.*/
```

Once we are done with operations over all involved resources (including XADisk), we can commit the XA transaction:

```
tx1.commit();
```

This completes the most simple and common usage of XADisk *XAResource* object. Though we did not discuss, XADisk implements the *XAResource* interface completely and hence supports features like

suspend/resume, one-phase commit optimization, crash recovery, transaction time-out etc.

Booting XADisk as JCA Resource Adapter

This chapter will describe the process of booting XADisk as a JCA Resource Adapter. As of JavaEE 5, all JavaEE servers provide support for JCA 1.5 specification. Note that, inside a JavaEE Server also, one is free to boot and use XADisk as a normal XADisk instance, not necessarily as a JCA Resource Adapter.

Before you start, you would need to have the following in your system environment:

a) A JavaEE Server compliant to JavaEE 5 or above,
b) XADisk.rar

The XADisk.rar file is the Resource Adapter Archive and can be deployed as a JCA Resource Adapter. You can think of this RAR as containing both of:

a) *XADisk Core* – this part implements the core functionality of XADisk, which is to provide a transactional wrapper around existing file systems.

b) *XADisk Connector* – this part implements the "Resource Adapter" piece of the JCA 1.5 contract and acts as a bridge between the *XADisk Core* and JavaEE Server. Implementing this contract enables XADisk to plug into any JavaEE (5 or above) Server via JCA. JCA provides XADisk various services from the JavaEE Server; transaction management, message inflow, connection management and lifecycle management are such services.

As JCA's transaction management feature also supports JTA, all JavaEE applications can leverage the XA support from XADisk if the XADisk is deployed as a JCA Resource Adapter. This means that you can (for example) operate on

databases, JMS Queues and XADisk all inside a single global (XA) transaction. So, effectively, your operations on the files and directories will also become part of the same global transaction, and will also preserve the ACID properties of transactions.

Please consult the documentation of your JavaEE Server on deploying JCA Resource Adapters so that you can deploy XADisk.rar in the same manner.

While you are deploying the XADisk.rar, your JavaEE server would provide you some mechanism to read and set the Resource Adapter configuration properties. These properties are actually the "Resource Adapter Bean" properties, and are all applicable at the XADisk instance level. These are the same set of properties set inside *FileSystemConfiguration* while booting XADisk in the normal way. Note that these are not the Connection Factory properties which are applicable only at the connection factory level.

Only two configuration properties, called *xaDiskHome* and *instanceId*, are mandatory. *xaDiskHome* is the *XADisk System Directory* and *instanceId* is the instance-id to be assigned to this XADisk instance. All other configuration properties have their default values in the XADisk.rar/META-INF/ra.xml file, and you can override them as per your requirements. Please refer to Appendix A for description of all configuration properties.

When you deploy the XADisk.rar on the JavaEE Server, an instance of XADisk starts booting automatically. For this automatic booting, the various configuration properties are read from the "Resource Adapter Bean" properties, converted to a *FileSystemConfiguration* object, and used for booting the XADisk instance. This XADisk instance is as good as a normal XADisk instance booted directly; it can also serve remote and local applications in the same way. But, as you have planned to deploy XADisk as a JCA Resource Adapter, you are likely to invoke it in the JCA way. Next chapter will describe the same.

Once the deployment of the Resource Adapter completes, the XADisk instance may not be available immediately. It may take further time for its crash recovery process (which is part of the boot up process) to complete the transactions which were open during the last shutdown/crash.

When the XADisk.rar is un-deployed, the XADisk instance automatically gets shutdown and becomes unavailable for use by any application.

Invoking XADisk JCA Adapter

As mentioned in the last chapter, deploying XADisk as a JCA Resource Adapter also boots an XADisk instance underneath. This XADisk instance can be invoked like any other XADisk instance by local or remote applications. Another approach is via JCA, which this chapter will talk about.

To invoke the XADisk JCA Adapter, one uses XADisk Connection Factories. These are of two types: Local and Remote.

The Local connection factories are used to create connections to XADisk instances running in the same JVM as *this* JavaEE Server. The Connection Factory class is *org.xadisk.connector.outbound.XADiskConnectionFactory* which you would need to specify while creating this local connection factory. There is one mandatory configuration property associated with this connection factory: *instanceId.* Please refer to Appendix B for details on the connection factories' properties.

A Remote connection factory is used to connect to an XADisk instance running remotely. This XADisk instance could either be a normal XADisk instance or the one booted underneath an XADisk JCA Resource Adapter deployment. The Connection Factory class is *org.xadisk.bridge.proxies.interfaces.XADiskRemoteConnectionFactory*. There are three mandatory configuration properties for this connection factory: *serverAddress*, *serverPort* and *instanceId*. Please refer to Appendix B for details on the connection factories' properties.

Once you have created the connection factories, local or remote as you needed, and given them some JNDI name, you can start coding your JavaEE Application (Session Beans, Servlets). For the below code snippet, I assume that a local connection factory named *xadiskcfLocal* and a remote connection factory

named *xadiskcfRemote* have been created. Let's take the case of a bean managed transaction…

UserTransaction utx = obtain the User Transaction object;

utx.begin();

XADiskConnectionFactory cfLocal = (XADiskConnectionFactory) new InitialContext().lookup("xadiskcfLocal");

XADiskConnectionFactory cfRemote = (XADiskConnectionFactory) new InitialContext().lookup("xadiskcfRemote");

XADiskConnection connectionLocal = cfLocal.getConnection();

XADiskConnection connectionRemote = cfRemote.getConnection();

File f1 = new File("/home/testing/test.txt");

File f2 = new File("/app/data.txt");

connectionLocal.createFile(f1, false);

connectioRemote.createFile(f2, false);

/ Between the utx.begin and utx.commit, you can also do some operations on databases, JMS queues, other XADisk instances (using appropriate connection factories pointing to them) or any other XA supporting transactional-system. Then, the single global transaction pointed to by the "utx" object will either be committed on all of these involve transactional systems (including all XADisk instances), or will be rolled-back on all of these transactional systems. */*

utx.commit();

connectionLocal.close();

connectionRemote.close();

In brief, we first started a global transaction using the JavaEE standard *UserTransaction* object. Then we looked-up the XADisk Connection Factories using their JNDI names. We then obtained the *XADiskConnection* objects required to invoke operations on the target XADisk instances. This connection is the main object on which you are going to work most of the time. Most of the methods exposed by the *XADiskConnection* interface are the APIs for working on the file-system.

Once you are done with your file system operations and all operations on other transactional systems also (for example, database, JMS Queues), you can commit the global transaction.

After doing that, and when you think you won't need the *XADiskConnection* object anymore, you should close the connection by calling the *close* method.

As you can see, all the code starting from connection factory lookup to the operations on the connection object uses the XADisk APIs in the same way; whether the connection factory was local or remote, and whether the remote XADisk instance is deployed inside a JavaEE Server or just any JVM.

Receiving XADisk events in Message Driven Beans

In the earlier chapter, we only talked about "outbound" interaction from the client applications to XADisk. These applications were either simple Java applications or JavaEE Applications. We call such invocations, from applications to XADisk, "outbound" because in all such cases the XADisk instance takes some action only when asked by the client application to do so.

In addition to such "outbound" interactions, XADisk also features "inbound" message flows. This functionality enables a registered Message Driven Bean (MDB) to get triggered whenever a file system event of its "interest" takes place. For example, an MDB can be deployed specifying that it should be "informed" whenever a file *F1* or file *F2* or directory *D* are modified through a specified XADisk instance (the XADisk instance is specified using activation spec properties like *isRemote* and *serverAddress/serverPort*). If a transaction inside that XADisk instance modifies any of *F1, F2* or *D*, then an Event object is generated and raised to this MDB.

Note that the XADisk instance which the MDB specifies can be any XADisk instance; it could be running in the same JVM or a remote JVM and need not be deployed as JCA Resource Adapter. Though, in all of these cases, there should be an XADisk JCA Resource Adapter deployed on the same JavaEE server as this MDB.

The MDB that you want to register for receiving events from XADisk should implement an interface called *org.xadisk.connector.inbound.FileSystemEventListener*.

Let's walk through an example. The following is a sample MDB class.

```
public class FileSystemEventsListener implements FileSystemEventListener {

   public void onFileSystemEvent(FileStateChangeEvent event) {
      System.out.println("RECEIVED AN EVENT FOR FILE : " + event.getFile());
   }
}
```

When you deploy this MDB, you need to set a few properties for the activation spec class *org.xadisk.connector.inbound.XADiskActivationSpecImpl.* The below snippet is from an MDB class which sets these activation-spec properties using annotations:

```
@MessageDriven(name = "XADiskListenerMDB1",

activationConfig = {

   @ActivationConfigProperty(propertyName = "fileNamesAndEventInterests",
propertyValue = “/home/admin/guest-records.txt::011|/home/admin/requests::001”),

   @ActivationConfigProperty(propertyName = "areFilesRemote", propertyValue
= "true"),

   @ActivationConfigProperty(propertyName = "remoteServerAddress",
propertyValue = "10.40.15.243"),

   @ActivationConfigProperty(propertyName = "remoteServerPort",
propertyValue = "5151")

})
```

The property value *fileNamesAndEventInterests* above mentions that the MDB should be informed whenever:

a) the *guest-records.txt* file is deleted or modified (011), or

b) the directory contents of */home/admin/requests* are modified (001).

*[A shortcut to remembering this Event Interests bitmap notation is "cdm" (**c**reated/**d**eleted/**m**odified). Same bitmap applies for both files and directories. A directory is called modified when children files/directories are added or removed from it.]*

The properties *areFilesRemote* and *remoteServerAddress/remoteServerPort* are used to "locate" the XADisk instance from which events will be dispatched to this MDB. If the XADisk instance is remote, the address and port specify how the remote XADisk instance can be contacted.

For further details on these properties, please see Appendix C.

Once you have configured and deployed your MDB, there is one more thing required for this MDB to receive the events. As management of such events is costly, these events are not raised whenever an arbitrary transaction touches the files/directories of interest to the MDB. It is upon the transaction doing the respective modifications whether it wants to raise the events upon commit. A transaction is provided with a switch to control this behavior. For Java applications using a *Session* object, it is set using:

```
session.setPublishFileStateChangeEventsOnCommit(true);
```

For JavaEE applications using a *XADiskConnection* object, it is set using:

```
connection.setPublishFileStateChangeEventsOnCommit(true);
```

You can set the value of this property any time before the transaction is committed. During transaction commit time, this property is read and if it is true, all the modifications by the transaction are considered for raising events; independent of when this property was set.

Delivery of such events from XADisk to the MDBs takes places inside an XA Transaction and is *guaranteed*. This means that one event, (for example) representing a modification in file *F1*, is delivered *exactly once* to exactly one of the interested MDBs (if at least one such interested MDB exists during the time of commit). And this premise remains valid even in case the JVM of the XADisk instance or the JavaEE Server crashes.

Global Configuration Properties

Sections 1 and 2 describe the configuration properties applicable for both normal XADisk instances and XADisk deployed as a JCA Resource Adapter. Section 3 describes the properties applicable only to normal XADisk instances.

For deployments as a JCA Resource Adapter, the configuration properties actually correspond to the Resource Adapter bean properties of the JCA 1.5 Specification and are specified in the deployment descriptor named as *ra.xml*.

For normal boot up of XADisk instance, the configuration properties are specified via the object of class *StandaloneFileSystemConfiguration* during the boot up.

1. Mandatory Configuration Properties

a) *xaDiskHome – This is also referred to as XADisk System Directory. This is a directory where XADisk keeps all of its artifacts required for its functioning, including the transaction logs. The directory that you specify may or may not exist. If it exists, it must not be anything other than the one you used for XADisk sometime earlier. If it does not exist, XADisk creates it during initialization. You should not do any kind of modifications inside this directory; as it may lead to failure of XADisk. No two XADisk instances should use the same system directory.*

b) *instanceId – An instance-id uniquely identifies an XADisk instance within a JVM (as multiple XADisk instances can be running within the same JVM). The new XADisk instance (booted directly or via deployment of XADisk JCA Resource Adapter) will be linked to the instanceId specified in the configuration.*

2. Optional Configuration Properties

a) *lockTimeOut – This is the time duration (in milliseconds) for which a request to acquire a lock (over a file/directory) will wait if the lock is not immediately available. If the wait time exceeds this value, a LockingTimedOutException is thrown to the application to let it know. Default value is 10000 (10 seconds).*

b) *transactionTimeout – This is maximum time (in seconds) for which any transaction in the system will be allowed to remain "open". If a transaction times out, it will be rolled-back by the transaction timeout mechanism. Default value is 60 seconds.*

c) *transactionLogFileMaxSize – This is the maximum size of a transaction log file (in bytes). XADisk maintains a transaction log and rotates the current transaction log if its size exceeds this value. You should set this value according to the maximum file-size allowed by the file-system (in which the XADisk System Directory resides). Default value is 1000000000 (1 GB).*

d) *deadLockDetectorInterval – This is the time interval (in seconds) in which the deadlock detection mechanism triggers to detect deadlocks in the system and to take appropriate action to remedy them. Default value is 30 seconds.*

e) *serverAddress/serverPort – An XADisk instance listens (if enableRemoteInvocations is true) on a network port to receive calls from remote application clients. The two properties called ServerAddress and ServerPort are used by the XADisk instance to facilitate this interaction with remote applications. They are also used by the XADisk JCA Resource Adapter to facilitate inbound messaging from remote XADisk instances to MDBs deployed in the same JavaEE Server. The value for ServerAddress should be set such that the applications running on the remote JVMs can contact this XADisk instance using this ServerAddress; i.e. the ServerAddress should be reachable (as in "network-reachability") to them. Default value is '127.0.0.1', which will allow the XADisk instance to serve only those*

applications running on the same machine as the XADisk instance. Default value for ServerPort is 9999.

f) *enableRemoteInvocations – This flag can be used to specify whether the XADisk instance should listen for remote invocations. See the description for ServerAddress/ServerPort. Its default value is false.*

g) *maximumConcurrentEventDeliveries – this is the maximum number of Message Driven Beans invoked by XADisk concurrently to process the file system events from XADisk. Default value is 20.*

h) *directBufferPoolSize - a performance tuning property. It specifies the pool size for the 'direct' (see java.nio.ByteBuffer) byte buffers. Pooled buffers (direct or indirect) are used by i/o streams (XAFileInputStream and XAFileOutputStream) for holding file's contents. Default value is 1000 (means, at most 1000 direct buffers can exist in the pool).*

i) *nonDirectBufferPoolSize - a performance tuning property. It specifies the pool size for the 'nonDirect' (see java.nio.ByteBuffer) byte buffers. Pooled buffers (direct or indirect) are used by i/o streams (XAFileInputStream and XAFileOutputStream) for holding file's contents. Default value is 1000 (means, at most 1000 nonDirect buffers can exist in the pool).*

j) *maxNonPooledBufferSize – a performance tuning property. XADisk tries to hold its ongoing transactions' logs in memory using byte-buffers. Similarly, such byte-buffers are also used by i/o streams (XAFileInputStream and XAFileOutputStream) for holding file's contents if the pooled buffers are all exhausted. As these byte-buffers add to the total memory consumption, but at the same time do boost performance, this property can be used to put an upper limit on the total size of these buffers. This property is specified in bytes and has default value of 1000000 (1 MB).*

k) *bufferSize - a performance tuning property. The i/o streams of XADisk (XAFileInputStream and XAFileOutputStream) use byte-buffers for holding file's contents. These byte-buffers are either from buffer pool or normally allocated (if the pool is exhausted). This property decides*

the size of these byte-buffers, both for pooled and normal cases. This property is specified in bytes and has default value of 4096.

l) *cumulativeBufferSizeForDiskWrite - a performance tuning property. XADisk doesn't write its transaction logs one-by-one separately, to the disk; it does so in big enough batches. This property mentions total size of transaction logs, in bytes, when such a disk write takes place. Default value is 1000000 bytes (1 MB).*

m) *directBufferIdleTime/nonDirectBufferIdleTime - a performance tuning property. This is the number of seconds after which any 'direct'/'nonDirect' pooled buffer is considered idle if not in use. Default value is 100 seconds. An idle buffer is freed by a background thread which runs periodically. The frequency of this thread is decided by another property called bufferPoolRelieverInterval.*

n) *bufferPoolRelieverInterval – see the above paragraph. Default value is 60 seconds.*

3. More Properties for XADisk instances (not deployed as JCA Resource Adapter)

The following are all optional configuration properties and can be set by calling appropriate setter methods on *StandaloneFileSystemConfiguration* object.

a) *workManagerCorePoolSize/workManagerMaxPoolSize/ workManagerKeepAliveTime – performance tuning properties. When running XADisk (not as a JCA Resource Adapter), XADisk uses its own thin implementation of WorkManager (which is otherwise available from the JavaEE Server due to JCA contract). This WorkManager implementation relies on a JDK utility class java.util.concurrent.ThreadPoolExecutor. The three tuning properties mentioned here are used as-is to set the properties corePoolSize, maximumPoolSize and keepAliveTime of the ThreadPoolExecutor. Default values for these properties are 10/Integer.MAX_VALUE/60 respectively.*

Connection Factory Properties

There are two types of Connection Factories in XADisk, with each represented by its own Connection Factory interface as below.

a) *org.xadisk.bridge.proxies.interfaces.XADiskRemoteConnectionFactory: There are three mandatory configuration properties for this connection factory:*
    a. *serverAddress/serverPort – these specify the address/port on which the remote XADisk instance is listening for remote calls.*

    b. *instanceId – this is the instance-id of the locally (not the remote one) deployed XADisk JCA Resource Adapter. While deploying a XADisk JCA Resource Adapter, the instanceId is specified via deployment descriptor (ra.xml). You need to specify that same value for this connection factory property also. Note that this is not the instanceId of the remote XADisk instance you want to invoke via this connection factory.*

b) *org.xadisk.connector.outbound.XADiskConnectionFactory: There is one configuration property associated with this connection factor:*
    a. *instanceId - this is the instance-id of the XADisk instance running in the same JVM which you want to invoke via this connection factory. Note that there can be multiple XADisk instances running within the JavaEE Server. The target XADisk instance (which you will invoke from this connection factory) could have booted directly or underneath a deployment of XADisk JCA Resource Adapter.*

Activation Spec Properties

These properties are used when deploying a Message Driven Bean that would receive file system events from XADisk. These properties actually correspond to the Activation Spec properties of the JCA 1.5 Specification. All of the below properties are mandatory.

a) *areFilesRemote – this specifies whether the XADisk instance is deployed on the same JVM or not. If "true", the remoteServerAddress/remoteServerPort properties are consulted for connecting to the remote XADisk instance.*

b) *remoteServerAddress/ remoteServerPort – if property areFilesRemote is "true", these properties are used to connect to the specified remote XADisk instance and register for receiving events (according to interest) from there.*

c) *fileNamesAndEventInterests – this properties specifies what kind of events this MDB should receive and on which files/directories. The format of this string is*

"file_or_directory_path::xyz|
file_or_directory_path::xyz|
file_or_directory_path::xyz|
…".

The "xyz" represents a 3 bit set with value 0 or 1 for each. Set first bit to 1 for knowing when the specific file/directory is "created"; second bit to 1 for receiving "deletion" event for the file/directory; third bit to 1 for receiving event whenever the specific file/directory is modified. We call a file is modified when its contents are changed by some transaction.

For a directory, modification means addition/removal of some child file/directory of the directory.

Support

If you find something confusing, can't track back an error or seek any kind of clarification, please feel free to discuss at the Discussion Forums.

If you have found a bug, let us chase that. Please raise an issue or write to nitin_verma@java.net.

Your suggestions, feedbacks and criticisms about anything related to XADisk can be great for improvements in XADisk. Please feel free to write to nitin_verma@java.net.