

Welcome

This documentation introduces the db4o Replication System (dRS).

dRS provides replication functionality to periodically synchronize databases that work disconnected from each other, such as remote autonomous servers or handheld devices synchronizing with central servers.

Before you start, please make sure that you have downloaded the latest dRS distribution from the [download area](#) of the [db4objects website](#).

We invite you to join the db4o community in the public [db4o forums](#) to ask for help at any time. You may also find the [db4o knowledgebase](#) helpful for keyword searches.

1. Roadmap

This section describes the features supported in current release and features planned in upcoming releases.

1.1. Supported Features

- db4o to db4o replication
- db4o to Hibernate replication
- Hibernate to Hibernate replication

- Array Replication
- Inheritance hierarchies
- Standard Primitive Types
- One-To-One Relations
- One-To-Many Relations
- Many-To-One Relations

- UUID and Version generation for db4o and Hibernate
- Queries for changed objects
- Parent-to-Child traversal along changed objects
- Collection Replication
- Replication Event System
- Replication of deleted objects

1.2. Unsupported Features

- Multidimensional Arrays in Hibernate replication

1.3. Planned Features

- Generation Hibernate mapping files
- RDBMS triggers for updating version number
- Instant replication
- Update of UUID generation mechanism, this may break existing data
- JDK5 enum support

2. Requirements

To run the provided build scripts dRS requires Apache Ant 1.6 or later.

<http://ant.apache.org>

dRS is dependant on the JDK 5 version of db4o object database. A db4o-5.x.java5.jar has been included in the "lib" directory of the distribution.

To use the Hibernate Replication functionality, the Hibernate core version 3.1 files are required. These files and their dependancies have been included in the /lib folder for your convenience. If you require more complicated Hibernate mapping configurations, you may wish to download the full Hibernate documentation from the [Hibernate project website](#).

Testing the installation

The build.xml contains targets for running the example and the regression tests:

- run-example
- run-regression

3. First Steps

To get started as simply as possible, we are going to reuse the Pilot class from the db4o tutorial.

```
public class Pilot {  
    String name;  
    int points;  
}
```

3.1. Simple Replication

First, we will also use db4o to db4o replication, as it is the simplest and requires the least initial setup.

When replicating objects to and from a db4o database, we need to enable UUIDs and VersionNumbers. The chapter on [db4o replication](#) explains more about these settings. We need to call these settings before opening our database.

```
//Replication requires UUIDs and VersionNumbers  
Db4o.configure().generateUUIDs(Integer.MAX_VALUE);  
Db4o.configure().generateVersionNumbers(Integer.MAX_VALUE);
```

Open the source db4o database

```
ObjectContainer handheld = Db4o.openFile("handheld.yap");
```

And let's store an object or two:

```
source.set(new Pilot("Scott Felton", 200));  
source.set(new Pilot("Frank Green", 120));
```

Great. Now we need our destination database:

```
//Open the destination db4o database  
ObjectContainer desktop = Db4o.openFile("desktop.yap");
```

Now, it's about time we start replicating some data. Let's set up our replication process to replicate some data from our handheld, to our desktop:

```
ReplicationSession replication = Replication.begin(handheld,  
desktop);
```

The ReplicationSession object contains all the functionality that the replication system requires to be able to comprehensively replicate modified data from one database to another.

Now that we've got the ReplicationSession set up and ready, let's get to the meat of this process:

```
//Query for objects changed from db4o  
ObjectSet changed =  
replication.providerA().objectsChangedSinceLastReplication();  
  
//Iterate changed objects, replicate them  
while (changed.hasNext())  
    replication.replicate(changed.next());
```

This code will get a list of all of the updated objects, and then replicate them to the destination database. Easy!

But wait, there's one important last step:

```
//commit all of our changes to both databases.  
replication.commit();
```

Ok. NOW we're done. Now the data has been copied, and the transaction is committed. All of our changes are safely committed in case of any failures.

3.2. Bi-Directional Replication

Our previous example copied all new or modified objects from our handheld device to our desktop database. What if we want to go the other way? Well, we only have to add one more loop:

```
ObjectSet changed =  
replication.providerB().objectsChangedSinceLastReplication();  
  
while(changed.hasNext())  
    replication.replicate(changed.next());
```

Now our handheld contains all of the new and modified records from our desktop.

Wow, wasn't that easy? Now, if there had been any modifications made to the destination database, the two are now both in sync with each other. Congratulations, you've now synchronized 2 db4o databases!

3.3. Selective Replication

What if our handheld doesn't have enough memory to store a complete set of all of our data objects? Well, then we should check our objects before replicating them by modifying the while loop:

```
ObjectSet changed =
replication.providerB().objectsChangedSinceLastReplication();

while (changed.hasNext()){
    Pilot p = (Pilot)changed.next();
    if(p.name.startsWith("S"))
        replication.replicate(p);
}
```

Now, only Pilots whose name starts with "S" will be replicated to our handheld database.

3.4. db4o-Hibernate Replication

Now let's take it one step further. Let's synchronize our desktop db4o database with a Hibernate-enhanced RDBMS (SQL) database. The first thing we're going to need is a working Hibernate system, so let's start with the Hibernate config xml file:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property
name="hibernate.connection.driver_class">org.hsqldb.jdbcDriver</prope
rty>

        <property
name="hibernate.connection.url">jdbc:hsqldb:mem:replication</property
```

```

>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>

    <!-- JDBC connection pool (use the built-in) -->
    <property name="hibernate.connection.pool_size">1</property>

    <!-- SQL dialect -->
    <property
name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</property>

    <!-- Echo all executed SQL to stdout -->
    <property name="hibernate.show_sql">false</property>

    <!-- Update the database schema if out of date -->
    <property name="hibernate.hbm2ddl.auto">update</property>

    <!-- Specify all your data entity classes here -->
    <mapping resource="f1/chapter4_1/Pilot.hbm.xml"/>
</session-factory>
</hibernate-configuration>

```

Wow, look at all that stuff. At first it looks like a lot, but upon closer inspection, it's really not much. There's a connection, a username, and a password. The rest of it is of little consequence right except the last `<mapping />` tag. We need to add a `<mapping />` tag for each type that we are going to attempt to replicate.

To replicate Pilot using Hibernate, you need to define a mapping file for it.

```

<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-access="field" default-lazy="false"
default-cascade="save-update">
    <class name="f1.singleobject.Pilot">
        <id column="typed_id" type="long">
            <generator class="native"/>

```

```
</id>
<property name="name" />
<property name="points" />
</class>
</hibernate-mapping>
```

Now it's just a simple case of reading our configuration files, and starting our replication session.

```
//Read the Hibernate Config file (in the classpath)
org.hibernate.cfg.Configuration hibernate = new
Configuration().configure("hibernate.cfg.xml");

ReplicationSession replication = HibernateReplication.begin(desktop,
hibernate);
```

That's the only change necessary to replicate your objects into your new SQL database! Depending on which RDBMS you are running, you may need to select different configuration settings in the `Hibernate.config.xml` file. The two most common settings that you may need to change are:

- `hibernate.connection.url`: Set this to the appropriate JDBC driver. For this example, we are using a temporary Hibernate in-memory database.

- `hibernate.dialect`: Make sure that you're talking to your database in the correct version of SQL. Query formats and DDL language varies widely among different database vendors. See the Hibernate documentation for a list of values.

4. db4o Replication

In order to use replication, the following configuration settings have to be called before a database file is created or opened:

```
Db4o.configure().generateUUIDs(Integer.MAX_VALUE);
Db4o.configure().generateVersionNumbers(Integer.MAX_VALUE);
```

UUIDs are object IDs that are unique across all databases created with db4o. That is achieved by having the database's creation timestamp as part of its objects' UUIDs. The db4o UUID contains two parts. The first part, contains an object ID. The second part identifies the database that originally created this ID.

The replication system will use this version number to invisibly tell when an object was last replicated, and if any changes have been made to the object since it was last replicated. An object's version number indicates the last time an object was modified. It is the database version at the moment of the modification.

4.1. Simple Replication

Now suppose we have opened two ObjectContainers from two different databases called "handheld" and "desktop", that we want to replicate. This is how we do it:

```
ObjectContainer handheld = db4o.OpenFile("handheld.yap");
ObjectContainer desktop = db4o.OpenFile("desktop.yap");

ReplicationSession replication = Replication.begin(handheld,
desktop);

ObjectSet changed =
replication.providerA().objectsChangedSinceLastReplication();

while (changed.hasNext())
    replication.replicate(changed.next());

replication.commit();
handheld.close();
desktop.close();
```

We start by opening two ObjectContainers. The next line, creates the ReplicationSession. This object contains all of the replication-related logic.

After creating the session, there is an interesting line:

```
ObjectSet changed =  
replication.providerA().objectsChangedSinceLastReplication();
```

This line of code will get the provider associated with the first of our sources (the handheld ObjectContainer in this case). Then it finds all of the objects that have been updated or created. The new/modified objects will be returned in an enumerable ObjectSet.

After that comes a simple loop where the resulting objects are replicated one at a time.

The replication.commit() call at the end is important. This line will save all of the changes we have made, and end any needed transactions. Forgetting to make this call will result in your replication changes being discarded when your application ends, or your ObjectContainers are closed.

The #commit() calls also mark all objects as replicated. Therefore, changed/new objects that are not replicated in this session will be marked as replicated.

4.2. Replicating Existing Data Files

As we learned in the previous section, Db4o.configure().generateUUIDs() and Db4o.configure().generateVersionNumbers() (or its objectClass counterparts) must be called before storing any objects to a data file because db4o replication needs object versions and UUIDs to work. This implies that objects in existing data files stored without the correct settings can't be replicated.

Fortunately enabling replication for existing data files is a very simple process:

We just need to use the Defragment tool in com.db4o.tools (source code only) after enabling replication:

```
Db4o.configure().objectClass(Task.class).enableReplication(true);  
new Defragment().run(currentFileName(), true);
```

After a successful defragmentation our data files are ready for replication.

5. Hibernate Replication

In this chapter, we will look into several examples that showcase the Hibernate Replication feature.

5.1. Basics

Let's take a look at what you have to do in order to use the Hibernate Replication feature. **5.1.1. hibernate.cfg.xml**

Hibernate requires a xml configuration file (hibernate.cfg.xml) to run. In order to run dRS with Hibernate, the user has to set some parameters in the configuration file.

```
<hibernate-configuration>    <session-factory>
    <!-- Database connection settings -->
    <property
name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDri
ver</property>

    <property
name="hibernate.connection.url">jdbc:oracle:thin:@ws-
peter.v:1521:websys</property>

    <property
name="hibernate.connection.username">db4o</property>
    <property
name="hibernate.connection.password">db4o</property>

    <!-- JDBC connection pool (use the built-in) -->
    <property name="hibernate.connection.pool_size">1</property>

    <!-- SQL dialect -->
    <property
name="hibernate.dialect">org.hibernate.dialect.OracleDialect</propert
y>

    <!-- Echo all executed SQL to stdout -->
    <property name="hibernate.show_sql">>false</property>

    <!-- Update the database schema if out of date -->
    <property name="hibernate.hbm2ddl.auto">update</property>

    <property name="hibernate.jdbc.batch_size">0</property>
    </session-factory>
</hibernate-configuration>
```

For the property "hibernate.connection.pool_size", dRS requires only 1 connection to the RDBMS, increasing it will not have any effect. "hibernate.jdbc.batch_size" is set to 0 is for easier debugging. You may increase it to batch SQL statements to potentially increase performance.

It is a MUST that "hibernate.hbm2ddl.auto" be set to "update" because the system will create some extra tables to store the metadata for replication to work properly.

```
<property name="hibernate.hbm2ddl.auto">update</property>
```

5.1.2. Turning off the automatic creation of dRS tables and columns

In some situations, you may not have the privilege to create or alter tables. You may need to ask your DBA to create the tables for you before using dRS.

You can turn off the automatic creation of dRS tables and columns by changing the "hibernate.hbm2ddl.auto" property to "validate" in hibernate.cfg.xml. By doing so, dRS will not create or alter any tables. Rather, it will check the existence of the dRS tables before starting replication.

If the required dRS tables do not exist, dRS will throw a RuntimeException notifying the user and the replication will halt.

5.1.3. Hibernate mapping files

Persisted Objects are objects that the user wants to store to the database, e.g. cars, pilots, purchase orders.

For dRS to operate properly, for each persisted object, the user MUST declare the primary key column of the database table in the hbm.xml mapping file as follow:

```
<id column="typed_id" type="long">  
  <generator class="native"/>  
</id>
```

- column - The name of the primary key column. The value can be well-formed string . "typed_id" is recommended.
- type - MUST be "long"
- class - MUST be "native"

The "typed_id" column stores the id used by Hibernate. It allows dRS to identify a persisted object by invoking org.hibernate.Session#getIdentifier(Object object).

If you do not define getter/setter for property, make default-access="field". default-lazy="false" default-cascade="save-update" is required for replication to work properly. Note, you should not set the cascade style to "delete", otherwise deletion replication will not work.

5.2. Detecting object changes

This section describes how do you configure dRS to detect object changes.

When using Hibernate normally outside dRS, It is crucial to follow these procedures so that dRS can detect new/changed objects and replicate them during replication sessions.

```
// Read or create the Configuration as usual
Configuration cfg = new Configuration().configure("your-hibernate.cfg.xml");

// Let the ReplicationConfigurator adjust the configuration
ReplicationConfigurator.configure(cfg);

// Create the SessionFactory as usual
SessionFactory sessionFactory = cfg.buildSessionFactory();

// Create the Session as usual
Session session = sessionFactory.openSession();

// Let the ReplicationConfigurator install the listeners to the Session
ReplicationConfigurator.install(session, cfg);

//Update objects as usual
Transaction tx = session.beginTransaction();
Pilot john = (Pilot) session.createCriteria(Pilot.class)
```

5.3. Running dRS

So much preparation. Now it's the time to run dRS to do some replications! Before that, here are some precautions:

- 1 - Do not open more than one dRS replication session against the same RDBMS concurrently. Otherwise data corruption will occur.
- 2 - When dRS is in progress, do not modify the data in the RDBMS by using SQL or Hibernate. Otherwise data corruption will occur.

Ready to go? Let's go to the next section for some real examples!

5.4. Examples 5.4.1. A simple example

A one-to-one association example:

```
public class Pilot {  
    String name;  
    Helmet helmet;  
}
```

```
public class Helmet {  
    String model;  
}
```

A one-to-one association to another persistent class is declared using a one-to-one element.

```
<hibernate-mapping default-access="field" default-lazy="false"  
default-cascade="save-update">  
    <class name="f1.one_to_one.Pilot">  
        <id column="typed_id" type="long">  
            <generator class="native"/>  
        </id>  
        <property name="name"/>  
        <one-to-one name="helmet" lazy="false"/>  
    </class>  
</hibernate-mapping>
```

Remember to add mappings in hibernate.cfg.xml

```
<mapping resource="f1/one_to_one/Pilot.hbm.xml"/>  
<mapping resource="f1/one_to_one/Helmet.hbm.xml"/>
```

Now, create and save the pilot. Helmet is saved automatically.

```
ExtDb4o.configure().generateUUIDs(Integer.MAX_VALUE);
```

```

ExtDb4o.configure().generateVersionNumbers(Integer.MAX_VALUE);

ObjectContainer objectContainer = Db4o.openFile("OneToOne.yap");

Helmet helmet = new Helmet();
helmet.model = "Robuster";

Pilot pilot = new Pilot();
pilot.name = "John";
pilot.helmet = helmet;

objectContainer.set(pilot);
objectContainer.commit();

```

the replication..

Perform the replication.

```

Configuration config = new
Configuration().configure("f1/one_to_one/hibernate.cfg.xml");

ReplicationSession replication =
HibernateReplication.begin(objectContainer, config);

ObjectSet changed =
replication.providerA().objectsChangedSinceLastReplication();

while (changed.hasNext())
    replication.replicate(changed.next());

replication.commit();
replication.close();
objectContainer.close();

```

Here helmet is cascaded from pilot and is replicated automatically.

5.4.2. Collection

This section covers examples on Collection, including array, Set, List and Map.

As an experienced db4o user, you may know that db4o treats Collection as first class object, which means it assigns unique UUID to each Collection. Hence a Collection can be shared among many owners. This is different to

Hibernate's approach, which Collection does not have a unique ID and they cannot be shared among objects.

To bridge this gap, dRS treats Collections as second class objects and does not assign UUIDs to them. When a shared Collection is replicated from db4o to Hibernate using dRS, it is automatically cloned. Each owner of the Collection receives a copy of the Collection. Further modifications to the db4o copy will not be replicated to cloned copies. Therefore, you cannot share Collections if you want to perform RDBMS replications with dRS.

In the following examples, we will use Car as the element in the following examples.

```
public class Car {
    public String brand;
    public String model;
}
```

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-access="field" default-lazy="false"
default-cascade="save-update">
    <class name="f1.collection.Car">
        <id column="typed_id" type="long">
            <generator class="native"/>
        </id>
        <property name="brand"/>
        <property name="model"/>
    </class>
</hibernate-mapping>
```

5.4.2.1. Array

Hibernate supports one dimensional arrays but does not support multidimensional arrays.

```
public class Pilot {
    String name;
    Car[] cars;
}
```

```
}
```

```
<hibernate-mapping default-access="field" default-lazy="false"  
default-cascade="save-update">  
  <class name="f1.collection.array.Pilot">  
    <id column="typed_id" type="long">  
      <generator class="native"/>  
    </id>  
  
    <property name="name"/>  
  
    <array name="cars" table="cars">  
      <key column="pilotId"/>  
      <list-index column="sortOrder"/>  
      <one-to-many class="f1.collection.Car"/>  
    </array>  
  </class>  
</hibernate-mapping>
```

Add Pilot and Car to hibernate.cfg.xml

```
<mapping resource="f1/collection/array/Pilot.hbm.xml"/>  
<mapping resource="f1/collection/Car.hbm.xml"/>
```

Save the pilot as usual and start replication. You are done!

```
ExtDb4o.configure().generateUUIDs(Integer.MAX_VALUE);  
ExtDb4o.configure().generateVersionNumbers(Integer.MAX_VALUE);  
  
ObjectContainer objectContainer = Db4o.openFile("ArrayExample.yap");  
  
Pilot pilot = new Pilot();  
pilot.name = "John";  
  
Car car1 = new Car();  
car1.brand = "BMW";  
car1.model = "M3";
```

```

Car car2 = new Car();
car2.brand = "Mercedes Benz";
car2.model = "S600SL";

pilot.cars = new Car[]{car1, car2};

objectContainer.set(pilot);
objectContainer.commit();

Configuration config = new
Configuration().configure("f1/collection/array/hibernate.cfg.xml");

ReplicationSession replication =
HibernateReplication.begin(objectContainer, config);

ObjectSet changed =
replication.providerA().objectsChangedSinceLastReplication();

while (changed.hasNext())
    replication.replicate(changed.next());

replication.commit();
replication.close();
objectContainer.close();

```

5.4.2.2. List

Similar to array, you can replicate a List of Cars.

```

public class Pilot {
    String name;
    List cars;
}

```

Map the List using the list tag in Pilot.hbm.xml

```

<hibernate-mapping default-access="field" default-lazy="false"
default-cascade="save-update">
    <class name="f1.collection.list.Pilot">

```

```

<id column="typed_id" type="long">
    <generator class="native"/>
</id>

<property name="name"/>

<list name="cars" table="cars">
    <key column="pilotId"/>
    <list-index column="sortOrder"/>
    <one-to-many class="fl.collection.Car"/>
</list>
</class>
</hibernate-mapping>

```

Replicate the pilot

```

ExtDb4o.configure().generateUUIDs(Integer.MAX_VALUE);
ExtDb4o.configure().generateVersionNumbers(Integer.MAX_VALUE);

ObjectContainer objectContainer = Db4o.openFile("ListExample.yap");

Pilot pilot = new Pilot();
pilot.name = "John";

Car car1 = new Car();
car1.brand = "BMW";
car1.model = "M3";

Car car2 = new Car();
car2.brand = "Mercedes Benz";
car2.model = "S600SL";

pilot.cars = new ArrayList();
pilot.cars.add(car1);
pilot.cars.add(car2);

objectContainer.set(pilot);
objectContainer.commit();

```

```

Configuration config = new
Configuration().configure("f1/collection/list/hibernate.cfg.xml");

ReplicationSession replication =
HibernateReplication.begin(objectContainer, config);

ObjectSet changed =
replication.providerA().objectsChangedSinceLastReplication();

while (changed.hasNext())
    replication.replicate(changed.next());

replication.commit();
replication.close();
objectContainer.close();

```

5.4.2.3. Set

Replicating a Set of objects is simple and is similar to the List example.

```

public class Pilot {
    String name;
    Set cars;
}

```

Use the set tag.

```

<hibernate-mapping default-access="field" default-lazy="false"
default-cascade="save-update">
    <class name="f1.collection.set.Pilot">
        <id column="typed_id" type="long">
            <generator class="native"/>
        </id>

        <property name="name"/>

        <set name="cars" table="cars">
            <key column="pilotId"/>

```

```
        <one-to-many class="f1.collection.Car"/>
    </set>
</class>
</hibernate-mapping>
```

Begins the replication.

```
ExtDb4o.configure().generateUUIDs(Integer.MAX_VALUE);
ExtDb4o.configure().generateVersionNumbers(Integer.MAX_VALUE);

ObjectContainer objectContainer = Db4o.openFile("SetExample.yap");

Pilot pilot = new Pilot();
pilot.name = "John";

Car car1 = new Car();
car1.brand = "BMW";
car1.model = "M3";

Car car2 = new Car();
car2.brand = "Mercedes Benz";
car2.model = "S600SL";

pilot.cars = new HashSet();
pilot.cars.add(car1);
pilot.cars.add(car2);

objectContainer.set(pilot);
objectContainer.commit();

Configuration config = new
Configuration().configure("f1/collection/set/hibernate.cfg.xml");

ReplicationSession replication =
HibernateReplication.begin(objectContainer, config);
ObjectSet changed =
replication.providerA().objectsChangedSinceLastReplication();

while (changed.hasNext())
```

```
replication.replicate(changed.next());

replication.commit();
replication.close();
objectContainer.close();
```

As you can see, these examples are very similar to each other.

5.4.2.4. Map

Replication supports replicating a Map of objects.

```
public class Pilot {
    String name;
    Map cars;
}
```

Use the map element to define the Map in the hbm file.

```
<hibernate-mapping default-access="field" default-lazy="false"
default-cascade="save-update">
    <class name="f1.collection.map.Pilot">
        <id column="typed_id" type="long">
            <generator class="native"/>
        </id>

        <property name="name"/>

        <map name="cars" table="cars">
            <key column="pilotId"/>
            <map-key type="string"/>
            <one-to-many class="f1.collection.Car"/>
        </map>
    </class>
</hibernate-mapping>
```

Begins the replication.

```
ExtDb4o.configure().generateUUIDs(Integer.MAX_VALUE);
```

```

ExtDb4o.configure().generateVersionNumbers(Integer.MAX_VALUE);

ObjectContainer objectContainer = Db4o.openFile("MapExample.yap");

Pilot pilot = new Pilot();
pilot.name = "John";

Car car1 = new Car();
car1.brand = "BMW";
car1.model = "M3";

Car car2 = new Car();
car2.brand = "Mercedes Benz";
car2.model = "S600SL";

pilot.cars = new HashMap();
pilot.cars.put("car1", car1);
pilot.cars.put("car2", car2);

objectContainer.set(pilot);
objectContainer.commit();

Configuration config = new
Configuration().configure("f1/collection/map/hibernate.cfg.xml");

ReplicationSession replication =
HibernateReplication.begin(objectContainer, config);
ObjectSet changed =
replication.providerA().objectsChangedSinceLastReplication();

while (changed.hasNext())
    replication.replicate(changed.next());

replication.commit();
replication.close();
objectContainer.close();

```

5.5. Hibernate Replication internals

So far we have seen that dRS allows you to replicate objects between db4o and relational database. You maybe

curious that how dRS keeps track of the identity of objects in relational database and how dRS knows which objects are changed since last round of replication. Read on and you will see how dRS does that.

dRS internal objects keep information used by dRS. Each internal object is associated with a Hibernate mapping file (.hbm.xml). Hibernate reads these files and understands how to store / retrieve these internal objects to / from the RDBMS. Each type of internal object maps to one table in RDBMS. If such table does not exist, Hibernate creates it automatically.

5.5.1. ProviderSignature, MySignature and PeerSignature

ProviderSignature uniquely identifies a ReplicationProvider. MySignature and PeerSignature are the subclasses of ProviderSignature. A HibernateReplicationProvider has a MySignature to serve as its own identity. PeerSignature identifies the peer ReplicationProvider during a ReplicationSession.

5.5.2. Record

Record contains the version of the RDBMS during a ReplicationSession. Near the end of a ReplicationSession, two ReplicationProviders synchronize their versions.

Record allows dRS to detect changed objects. dRS detects changed objects by comparing the version of an object (v) with the maximum version of all Records (m). An object is updated if $v > m$.

5.5.3. UUID

UUID uniquely identifies a persisted object.

As seen in the Persisted Objects section, each persisted object is identified by an "typed_id". This "typed_id" is unique only with its type of that object (i.e. A car has an "typed_id" of 1534, a Pilot can also has an "typed_id" of 1534) and within the current RDBMS.

How do we identify "a Pilot that is originated from Oracle instance pi2763" ? To do so, we need two parameters:

1. an id that is unique across types
2. associates this id with the ProviderSignature of the RDBMS (The RDBMS that owns this object)

```
class UUID {
    long longPart;
    ProviderSignature provider;
}
```

Collectively, 1 and 2 forms the "UUID".

5.5.4. ObjectReference

ObjectReference contains the UUID and the version of a persisted object. It also contains the className and the typed_id of that persisted object.

UUID forms an 1 to 1 relationship with {className, typedId}.

```
class ObjectReference {
    String className;
    long typedId;
    Uuid uuid;
    long version;
}
```

5.5.5. List of dRS tables

5.5.5.1. Table: drs_providers

Column	Type	Function
id	long	synthetic, auto-increment primary key
is_my_sig	char(1)	't' if MySignature, 'f' if PeerSignature
signature	binary	holds the unique identifier - byte array
created	long	legacy field used by pre-dRS db4o replication code

5.5.5.2. Table: drs_history

Column	Type	Function
provider_id	long	primary key, same as the PK of a PeerSignature
time	long	the version of the RDBMS during a ReplicationSession

5.5.5.3. Table: drs_objects

Column	Type	Function
id	long	synthetic, auto-increment primary key
created	long	the UUID long part of this ObjectReference
provider_id	long	specifies the originating ReplicationProvider of this ObjectReference

class_name	varchar	the type of the referenced object
typed_id	long	the id used by Hibernate, which is only unique within its type
modified	long	the version of the referenced object

6. Advanced Topics

In this chapter, we will look into several advanced replication features.

6.1. Events

Imagine you have just passed in an object that contains a huge List of objects. You only want to replicate the root object but not the list. What you can do is to make use of the event system of dRS to stop traversal at the list.

Here is how:

First, you need to implement the ReplicationEventListener interface and pass it to Replication.

```
ReplicationEventListener listener = new ReplicationEventListener() {
    public void onReplicate(ReplicationEvent event) {
        if (event.stateInProviderA().getObject() instanceof List)
            event.stopTraversal();
    }
};

ReplicationSession replication = Replication.begin(objectContainer,
hibernateConfiguration, listener);
```

6.1.1. Resolving conflicts

The event system of dRS can also be used to resolve conflicts. When there is a conflict, You can choose to override the result with either the object from provider A, provider B or null. If you choose null, then the object will not be replicated.

```
ReplicationEventListener listener = new ReplicationEventListener() {
    public void onReplicate(ReplicationEvent event) {
        if (event.isConflict()) {
            ObjectState chosenObjectState = event.stateInProviderA();
            event.overrideWith(chosenObjectState);
        }
    }
};

ReplicationSession replication = Replication.begin(objectContainer,
hibernateConfiguration, listener);
```

6.2. Deleted Objects Replication

In addition to replicating changed/new objects, dRS is able to replicate deletions of objects. When an object is deleted since last replication in provider A and you would like to replicate this changes to provider B, you can make use of the dRS to handle that.

You could replicate deletions as follow:

```
replication.replicateDeletions(Car.class);
```

dRS traverses every Car object in both providers. For instance, if a deletion is found in one provider, the deletion will be replicated to the other provider. During the traversal, ReplicationEvent will be generated as usual, you can listen to them. By default, the deletion will prevail. You can choose the counterpart of the deleted object to prevail if required.

Note that the deletions of a Parent will not be cascaded to child objects. For example, if a Car contains a child object, e.g. Pilot, Pilot will not be traversed and the deletions of Pilot will not be replicated.

7. License

7.1. dRS

[db4objects Inc.](#) supplies the db4o Replication System (dRS) under the General Public License (GPL).

Under the GPL dRS is free to be used:

- for development,
- in-house as long as no deployment to third parties takes place,
- together with works that are placed under the GPL themselves.

You should have received a copy of the GPL in the file dRS.license.txt together with the dRS distribution.

7.2. Bundled 3rd Party Licenses

The dRS distribution comes with several 3rd party libraries, located in the /lib/ subdirectory together with the respective license files.