

# Improvements and Additions to the DTN2 Reference Implementation of the Bundle Protocol (BP) and the Oasys framework

---

## 1. Introduction

This document describes some work<sup>1</sup> that has been carried out by Elwyn Davies of Trinity College Dublin (TCD) and Folly Consulting Ltd. to improve and extend the DTN2 Reference Implementation of the Bundle Protocol (RFC 5050) [2]. The work builds on the work by MITRE Inc described in [1] and incorporated in the Sourceforge releases of DTN2 (version 2.8.0 *et seq*) and Oasys (version 1.5.0 *et seq*) code.

The work described in [1] is focused on improving the robustness of DTN2 by introducing transaction mechanisms and implementing persistent storage in a relational database using the SQL query language. The implementers chose to use the SQLite (version 3) database which is a lightweight implementation suitable for providing persistent storage for a single process but giving good resilience and transactional support to improve the overall robustness of DTN2 in the face of unexpected termination (e.g., due to power failure).

As part of their contribution to the SAIL project, TCD are implementing a 'Netinf Device' which is intended to demonstrate information centric networking using the BP as a substrate for carrying messages and using the cache of bundles maintained by DTN2 as the information object cache that a NetInf node is expected to maintain. To allow applications that need to 'see' the whole collection of information objects, it is desirable to externalize at least some of the data about the bundles in the cache.

One way of achieving this would be to make the database used by DTN2 to provide persistence across restarts of the DTN2 server daemon (dtn2d) available to other applications. In principle this could be done using SQLite, but SQLite is not optimized for multi-process access. A client-server database such as MySQL is better suited to this kind of (multi-)application usage.

Accordingly these modifications are aimed at allowing MySQL to be used as the persistent storage medium for DTN2 and making the internal information relating to the cached bundles more readily available to external applications.

## 2. Microsoft Open Database Connectivity (ODBC)

The Oasys framework used by DTN2 provides a configurable mechanism allowing the application user to select what package is used to implement the persistent storage. When the Oasys framework was originally designed, it was intended that SQL-based storage systems should be options. Until recently the Berkeley DB system was the default solution among those implemented with a 'raw' filesystem and memory-based solutions as alternatives, but the intended MySQL and PostgreSQL 'native' interfaces have never been implemented.

As documented in [1], when it was decided to provide an SQL-based persistent storage capability for Oasys, it was decided that, rather than using the native API of the chosen database, Oasys should be interfaced to a piece of well-established middleware Open Database Connectivity (ODBC). This component was originally championed by Microsoft as a common interface for their various commercial database products. Over the years ODBC has become the common application programming interface (API) and connection establishment method used by applications and supported by almost all database products. Hence the use of ODBC gives the opportunity to use alternative SQL-capable databases with relatively little effort.

---

<sup>1</sup> This work was partially supported by the Scalable and Adaptable Internet Solutions (SAIL) EU Framework Programme 7 project (Grant # 257448) and by Folly Consulting Ltd.

### 2.1 ODBC Architecture

The ODBC API is implemented by the Driver Manager (DM) component. General purpose DM components are generally operating system and platform specific. Microsoft and some third parties provide DMs for the Windows operating system. The Microsoft DM and a control component are bundled with the operating system. On Apple Macintosh machines, a DM (iODBC, see below) is bundled with the Mac OSx operating system. For Unix and Linux distributions there are a number of commercial and open source DMs. The most generally used open source DMs for Linux are:

- unixODBC (<http://www.unixodbc.org/>); and
- Independent Open Database Connectivity - iODBC (<http://www.iodbc.org>). as also bundled on Mac OSx.

The DTN2/Oasys development has all been carried out on the Linux platform using unixODBC version 2.2.11 or version 2.3.0. Source code and pre-built packages for various Linux distributions are available for both these components. In principle, it should be possible to slot in the iODBC libraries in place of the unixODBC libraries with little trouble, but this has not yet been tried.

The ODBC DM provides a control component for

- selecting the database to be used,
- dynamically loading the relevant ODBC driver component the type of database to be connected, and
- managing the interface between the DM and the loaded driver.

Detailed information about the ODBC API can be found on the Microsoft [MSDN Open Database Connectivity \(ODBC\) web site](#).

Database suppliers or third parties provide ODBC drivers for almost every sort of database available today. After being dynamically loaded, the driver is responsible for setting up the connection to the database selected by the DM configuration and then mediating API calls made through the DM API to the database interface. In the case of client-server databases, such as MySQL the database connection will be over a network transport but ODBC is also implemented for SQLite, both in its native form and for the Berkeley DB SQLite shim layer. In this case the whole ODBC mechanism is integrated into the application process and the database file is accessed directly.

### 2.2 ODBC Drivers

Persistent storage for DTN2/Oasys using SQL databases has so far been implemented using the SQLite3 lightweight file based package and the MySQL client-server package. Both packages are open source components. Unfortunately ODBC is not able to completely hide some of the idiosyncrasies of the different SQL implementations. In particular it is essential to be aware that there are minor variations and optional features in the SQL language used by the databases. Since ODBC does much of its work by passing SQL strings to the connected database's parser as opposed to implementing SQL itself (a wise decision!), any SQL embedded in Oasys or DTN2 needs so far as is possible to use a common subset of the SQL language and will need to be fully tested on each new type of database (e.g., MySQL allows BEGIN or START TRANSACTION as the SQL phrase to initiate a transaction whereas SQLite allows BEGIN TRANSACTION or BEGIN.) There are also some idiosyncrasies in handling transactions when the database is not in auto-commit mode.

There is a third common open source SQL database PostgreSQL (<http://www.postgresql.org>). It should be possible to use this database as an alternative with a small amount of work, as ODBC drivers exist.

### 2.2.1 SQLite ODBC Driver

The SQLite ODBC driver *sqliteodbc* is supplied as a separate package obtainable from <http://www.ch-werner.de/sqliteodbc/>. The software is maintained by Christian Werner. The driver is designed for use with both [SQLite](#) version 2 and version 3. DTN2/Oasys should use version 3.

Pre-built packages are available for various Linux distributions..

There are some idiosyncrasies in the handling of transactions when auto-commit is turned off. By default, *sqliteodbc* starts a new transaction as soon as the previous one is committed without the user having any choice in the matter. This leads to some problems, especially if the database file is being accessed by more than one process, which is allowed by SQLite provided only one process is writing to the file. Inspection of the code identified that setting the NoTxn option (not documented) suppresses this behaviour and allows the user to control transactions explicitly.

Despite the caveats above, *sqliteodbc* appears robust. However the versions of *sqliteodbc* do specify that they support particular minor versions of *sqlite3* (see <http://www.ch-werner.de/sqliteodbc/html/index.html>). It is therefore wise to check that the version of *sqlite3* supported by the version of *sqliteodbc* that is being used is near or higher than the version of *sqlite3* in use. In practice the changes to *sqliteodbc* appear to have been mostly detailed and the undemanding usage in DTN2/Oasys is unlikely to expose the issues solved by recent changes (say post Version 0.80).

This driver can also be used with the SQLite shim available with the recent versions of the Berkeley DB ([Berkeley DB SQL capabilities](#)).

### 2.3 MySQL ODBC Driver

[Oracle MySQL](#) provides the [Connector/ODBC package](#) as a separate component to complement the MySQL database. The source releases can be downloaded from the [Connector/ODBC downloads page](#). Pre-built versions are available for many Linux distributions.

Oracle support two versions of Connector/ODBC. Version 3.51 is the 'traditional' version and Version 5.1 is a more recent partial rewrite. For the purposes of DTN2/Oasys the two versions are more or less equivalent. The main improvements in Version 5.1 are Unicode support and better support for the 64 bit edition of Windows. It is unclear whether Version 5.1 is any better on 64 bit Linux machines [TBD]. More details of the differences can be found at [Connector/ODBC Versions page](#). However should any extra functionality need Unicode, install at least version 2.2.11 of unixODBC to go with Connector/ODBC v5.1, as earlier versions of unixODBC do not have the required functionality.

Note that building the source code version of Connector/ODBC version 5.1 needs the *cmake* toolset rather than the *automake* toolset which most other things use. Do not be fooled by the apparent presence of some of the *automake* scripts: using them will not get you anywhere. They appear to be left over from when *automake* was used.

## 3. Auxiliary Tables

Both the modifications documented in [1] and the current set of modifications envisage that the database used by DTN2/Oasys will have additional tables besides those used for the persistent storage of DTN2's state. The use cases are:

- Additional tables to be used by other parts of the DTN2 code. The 'Army VRL router' is intending to use DTN and needs additional database tables for internal storage.
- External programs can benefit from an overall view of the set of bundles cached in DTN2 rather than just registering for delivery of particular bundles. The NetInf node being implemented by TCD for the SAIL project intends to make use of the BP to transport NetInf protocol messages and maintain the DTN2 cache as its cache of Named Data Objects.

## Adding Auxiliary Tables to DTN2

To provide this functionality, Oasys has to be able to build additional tables into the database schema and potentially provide mechanisms, such as 'triggers' to maintain database integrity.

### 3.1 Exposing Internal Information via Auxiliary Tables

Just getting access to the internal tables used by DTN2/Oasys for persistent storage is not very helpful for external applications. The persistent storage mechanism used by Oasys is focused on 'pickling' (to use the Python term) the internal data structure that have to be persistent, so that they can be reloaded after a system restart. It was also designed with the (key, value) pair mechanism of the Berkeley DB in mind, so that the content of each data structure is serialized into a single opaque data blob before storage in the persistent store. This is ideal for pickling data structures, as it is easy to modify the serialization/deserialization routines when additional items have to be added to the structure or new components are developed.

For an external application, the pickled form of the data is less useful. It requires that the external application accesses and deserializes all of the data into its own internal store before it can decide whether it is actually interested in the data item. Also much if this information is irrelevant or unusable in the external application.

Accordingly, the current set of modifications provide for auxiliary tables to be associated with the main DTN2/Oasys tables with a more elaborate structure that exposes a selected set of the fields in the persistent storage tables. In principle the Oasys mechanism that has been implemented would allow any of the tables in DTN2 to have an auxiliary table. However for the purposes of the NetInf prototype, only the *bundles\_aux* table has been implemented and the remaining description focuses on this single addition.

The schema for an auxiliary table uses the same PRIMARY KEY column type as the corresponding main table. There will be a one-to-one correspondence between rows in the main *bundles* table and the *bundles\_aux* table using the same key for the corresponding rows. The integrity of this referential relationship is maintained by INSERT and DELETE database triggers attached to the *bundles* table. The triggers respectively insert a new row into *bundles\_aux* using the same key as a newly inserted row in *bundles*, and delete the corresponding row from *bundles\_aux* when a row is deleted from *bundles*. The data of the new *auxiliary\_bundles* row is then separately updated, but provided DTN2 does the insert and two table updates in a single transaction, integrity will be maintained.

The corresponding row in *bundles\_aux* will contain selected fields from the *bundle* structure stored in the *bundles* row. Which fields are stored can be relatively easily modified by changing a single class constructor and providing a corresponding database schema construction script. The NetInf usage of this capability envisages that the external applications such as the bundle search and filesystem interface will need to know the bundle identifier, the *name* (a specialized URL for the ni: scheme) and the location of the payload file. Other items may also be needed, but it will be observed that this is a tiny fraction of the total information stored.

It is envisaged that auxiliary tables will be 'write-only' for DTN2/Oasys and 'read-only' for external applications. DTN2/Oasys has the complete set of information in the main table when it needs to read in information on restart and need not be concerned with the information in the auxiliary tables. It is also undesirable for an external application to modify the auxiliary information as it may be used by more than one application. This restriction may be enforceable by user permissions on appropriately capable database packages (e.g., MySQL but not SQLite). However, there might conceivably be cases where additional fields could be included in the auxiliary table to allow external applications to record, for example, state information after processing has occurred. This is probably best enforced by triggers or procedures.

## 4. Reasons for Using MySQL for Auxiliary Table Applications

Although MySQL is a more heavyweight application than SQLite, it is probably desirable to use MySQL for situations where auxiliary tables are being accessed by external applications. There are a number of reasons for doing and for not doing this as discussed in the remainder of this section. For the purposes of the NetInf

## Adding Auxiliary Tables to DTN2

prototype, it is expected that MySQL will be used, but experimentation with the SQLite shim for Berkeley DB should be considered in future.:

### 4.1 Pro: Access Control

Although access to SQLite databases can be controlled coarsely by means of access modes on the underlying file, more fine grained control can be provided by MySQL. Thus different 'users' can be defined for the DTN2/Oasys access and the external application access. The permissions for these users can be defined at the table read/write level preventing external applications from accidentally (or deliberately) corrupting the main DTN2/Oasys tables, and restricting access to auxiliary tables to read-only or even to just some columns.

### 4.2 Pro: Multiple Access Capabilities

SQLite is primarily intended for single process access to a given database although multiple processes can access the database simultaneously. However only one can write or hold a transaction open at a time. For the basic NetInf ideas this may not be a problem, as the external applications may well be read-only. This is potentially restrictive.

### 4.3 Pro: Locking

MySQL offers row level 'fine grained' locking whereas SQLite only provides locking at the database level. Depending on how transactional control is deployed in DTN2 (whether a transaction covers multiple separate writes to the database), this could lock out external applications for extended periods of time, which is not acceptable. Keeping transactions to a single item write (e.g., *bundles* row plus *bundles\_aux* row) would limit the lockout period but would potentially impact on DTN2 performance.

### 4.4 Con: Process and Resource Overhead

MySQL is a fairly heavyweight package and is likely to consume significantly more resources than SQLite. It also requires a little more configuration effort to set up the initial database and the user profiles. This is not very significant as it can be done with a single script.

### 4.5 Con: Inter-Process Communication Overhead

MySQL runs as a separate process. Accordingly each database operation incurs a communication round trip and process context switching overhead. With MySQL running on the same node as the applications, the communication overhead is not so great as it would if a real network round trip was needed. However, the in-process routine calls needed for SQLite are likely to be significantly faster. Experiment is required to see what the penalty is, if any (given a multi-cored processor, for example).

### 4.6 Berkeley DB SQLite Interface Compromise

A possible alternative to using MySQL might be to use the Berkeley DB with its SQLite shim interface. This offers row level locking while avoiding the communication overhead. The resource costs are intermediate because Berkeley DB stores are larger than SQLite ones and the Berkeley DB programme has a significantly higher footprint than SQLite, but significantly less than MySQL.

## 5. Summary of Changes to DTN2 and Oasys Implementations

These changes should be read in conjunction with the change notes in [1].

## 5.1 Modified the Oasys ODBC Storage Class to be a Base Class for Multiple Different ODBC-based SQL Storage Mechanisms

**Why:** The vast majority of the code for ODBC-based access is common to any database connected via ODBC. Factored out the initialisation and ODBC configuration parsing which is generally specialized but provided a number of routines that can be called from initialisation which are again typically common (such as the code for connecting to the database).

**Implementation:** Split up the code in storage/ODBCStore.cc, make a number of fields used by initialisation **protected** rather than **private** and move the **init** and ODBC configuration file parsing to derived classes for different databases. Factor out three chunks of the init code into common (protected) routines in ODBCDBStore class:

- **connect\_to\_database** uses the ODBC Data Source Name (DSN) taken from the Oasys **dbname** configuration parameter to open the selected database connection. The ODBC DM uses the configuration information in the ODBC configuration files to determine the right driver and the database parameters.
- **set\_odbc\_auto\_commit\_mode** is used to turn off auto-commit mode in the database if requested by the user in the Oasys configuration (**storage set auto\_commit false**) – most databases start off in auto-commit mode.
- **create\_aux\_tables** creates any internal auxiliary tables needed by DTN2/Oasys. At present the only such table is META\_DATA TABLES which is used to store the names of the main database tables created for persistent data storage. The **get\_tables** routine uses this table to determine the tables created when tidying etc.

Note that both **begin\_transaction** and **end\_transaction** have been altered to use the common SQL dialect: **BEGIN TRANSACTION** is now just **BEGIN** and **end\_transaction** use **COMMIT** rather than calling **SQLEndTran**.

## 5.2 Added MySQL and SQLite Storage Classes in Oasys

**Why:** To provide the database specific code for ODBC connections to the MySQL and SQLite databases. The classes are derived from ODBCDBStore that provides the majority of the implementation for the classes. The code is primarily concerned with initialising the storage class including parsing the ODBC configuration file, performing database schema initialisation and tidying as requested by the user

**Implementation:** The code is based on the initialisation code for SQLite previously in ODBCStore.cc. The ODBC initialisation file and Oasys configuration files are combined to determine what needs to be done when initialising or tidying the database. A number of changes have been made to the code provided by MITRE as described in [1]:

- More rigorous parsing of the ODBC configuration file.
- Implementation of the database 'tidy' function which can be quite complex for SQLite.
- Removal of the main database table creation code – these tables are created by calls of **get\_table** with appropriate flag settings. The DTN2 code acting as an Oasys client knows the names the tables to create: having the table names explicitly in the SQLite client is inappropriate.
- Factoring out the **connect\_to\_database** functionality as it is common to most databases.
- Factoring out the internal auxiliary table and auto-commit mode setting code which is also common.
- Providing means to execute additional database schema set-up scripts both before and after the main tables are created (the script run afterwards is executed by **create\_finalize**). These scripts are defined

## Adding Auxiliary Tables to DTN2

by parameters in the ODBC DSN specification [Arguably these should be in the Oasys storage configuration rather than `odbc.ini`].

The DurableTable `create_store` function now recognises two new storage types `odbc-mysql` and `odbc-sqlite`. The implementation class instantiated is now one of these two new classes implemented here.

### 5.3 General Style Cleanup

Some of the code generated in the previous modifications did not conform to the Oasys general style. In particular several routine and variable names used Microsoft-style 'Camel Case' rather than essentially all lower case with words separated by underscores. This has been corrected (e.g., `beginTransaction` → `begin_transaction`.) In addition to the files involved in the other changes here, `BerkeleyDB.cc/h` were affected by this cleanup.

### 5.4 Added Oasys Auxiliary File Infrastructure

**Why:** To provide infrastructure for additional auxiliary tables linked to the main database tables as described in Section 3).

**Implementation:** The intention is to create and access auxiliary tables in as similar manner as is possible to how the main tables are accessed. Each table is managed at run time by a singleton instance of a class derived from class `DurableTableImpl`. A new datastore initialisation flag (`DS_AUX_TABLE`) was added to `DurableStoreFlags_t` (in `DurableStore.h`) to signal to the table initialization that an auxiliary table is being managed. The initialization of an auxiliary table manager is handled by a new templated routine in `InternalKeyDurableTableClass` called `do_init_aux`. This is a copy of `do_init` except that additionally the `DS_AUX_TABLE` flag is set in the flags passed into the `get_table` routine. This alters the behaviour of `get_table` in the case that it has to create the table in the database schema because the user has requested databases initialization or tidy. The `get_table` routine is part of the database specific storage implementation: for ODBC this is in `ODBCStore.cc`. [Note: should add tests into `get_table` routines for other storage types to ASSERT that `DS_AUX_TABLE` is not set. No harm is done presently but it is likely that `get_table` would fail because the schema wasn't right.] The storage specific `get_table` routine in turn creates an instance of the storage specific table management implementation. In the case of ODBC, which can handle auxiliary tables, a flag is passed into the constructor for `ODBCDBTable` to indicate whether the table is an auxiliary table or a main table depending on whether the flags passed into `get_table` had `DS_AUX_TABLE` true or false.

Storage classes now have a function `aux_tables_available` which returns a boolean value that indicates if the upper levels can use auxiliary tables. Currently only ODBC storage classes implement auxiliary tables. All other classes return an unconditional false result. ODBC classes return true or false depending on the user configuration setting as to whether to use them or not (see Section 5.5)..

The data to be stored in a `DurableTableImpl` managed table is passed around in an instance of a class derived from `SerializableObject`. For the main classes, the `serialize` routine is used to create the opaque blob stored in the table. For auxiliary tables, the schema is generally completely different. In order to handle auxiliary tables in the same way as main tables, the new `StoreDetail` class has been implemented. This class is nominally derived from `SerializableObject` and can thus be passed to the `InternalKeyDurableTableClass` templates, but for auxiliary tables the `serialize` function is never called and is a dummy implementation.

Instead, the `StoreDetail` carries a vector of `StoreDetailItem` instances that describe the columns and data types that are to be accessed and provide either the data to be put into the column or a pointer to a buffer to hold data got from the column. For each actual auxiliary table a class derived from `StoreDetail` links the actual data and the column names to be accessed. The C++ `dynamic_cast` operation is applied to the instance passed to the `get` and `put` routines in storage modules allowing auxiliary table operations to access the vector of data. The key for the auxiliary table is specified and constructed in the same way as main table keys by serializing a key object.

The implementation of `put` and `get` (for a single type table) in `ODBCStore.cc` have been modified to handle auxiliary tables. Recall that the intention is that rows in auxiliary tables are inserted by triggers in

## Adding Auxiliary Tables to DTN2

synchronism with insertions into the corresponding main table. Thus the put routine never has to actually insert the row, and also should not report an error if the row exists even if asked to create it. At the moment it is assumed that any fields in the auxiliary tables do not require updating if the corresponding main table entry is updated. Accordingly the table update routine has not been modified – this may have to be revisited if more complex updates are required. Also no special action is needed for the del routine as auxiliary table rows are automatically deleted when the corresponding main table row is deleted. [Note that the get functionality exists but is essentially untested as DTN2 does not need the get routine and auxiliary tables are seen as write-only from the point of view of DTN2.]

When a row from an auxiliary table is to be read or written, the vector of StoreDetailItems passed in the StoreDetail instance is used to dynamically create an SQL statement using the specified column names and the ODBC SQLBindParameter API call is used to link the data locations specified in the vector to the parameters in the SQL and specify the size and types of the data to be accessed.

The schema and table creation for auxiliary tables should currently be handled by the schema creation scripts that are fed to the database during initialization (see Section 5.2). In principle it would be possible to use the StoreDetail mechanism to create auxiliary tables but the existing Oasys code does not implement the table prototype mechanism foreseen as a way to achieve this, so this is postponed to a future update. Note that the script mechanism only works at present if the database server is on the same host as the DTN2 daemon. For DTN2 usage it is very unlikely that any other case would be relevant.

### 5.5 DTN2 Usage of Auxiliary Storage for Bundles

**Why:** To allow the SAIL NetInf device prototype to have easy access to information about all the bundles in the DTN2 cache at once.

**Implementation:** The BundleDetail class is derived from the Oasys StoreDetail class (see Section 5.4). The constructor for this class instance calls the AddDetail routine for every item from a bundle that is to be stored in the *bundles\_aux* table.

The corresponding database table schema is inserted into create\_aux.sql.

The user can request whether auxiliary tables are to be used or not by using the `odbc_use_aux_tables` setting in the DTN2 configuration (storage set `odbc_use_aux_tables true/false`). If this flag is set true but the underlying storage class does not support auxiliary tables a warning is given.

During storage initialization (DTNServer.cc and storage/BundleStore.cc) the auxiliary table is made accessible if auxiliary tables are to be used and the table manager class instance for the auxiliary table created and recorded in the BundleStore class instance.

When a new bundle is recorded in the persistent data store and an auxiliary table exists and is in use, a BundleDetail instance for the bundle is created and the detailed data inserted into the auxiliary table immediately after the main table entry has been inserted.

At present it is assumed that get is never used and that update does not affect the auxiliary table. Delete for the auxiliary table is actioned internally in the database by a trigger so no code modification is needed.

An additional database initialization routine has been added that allows post table creation additions (such as triggers) to be fed into the database. DTNServer.cc calls the DurableStore create\_finalize routine as the last stage of database initialization.

## 6. Summary of Code Changes

### 6.1 DTN2

#### 6.1.1 DTN2 – configure.ac

- Add `AC_CONFIG_ODBC` to section checking for third party libraries so that configure provides the `LIBODBC_ENABLED` macro if ODBC is available and configured in. This allows the auxiliary table functionality to be omitted if the deployment does not support ODBC database access. This requires the `build-configure.sh` script to be run to modify `config.h.in`.

#### 6.1.2 DTN2/servlib – DTNServer.cc

- Add check to report warning to user if the configuration setting `odbc_use_aux_tables` is set true but the selected storage module is unable to support auxiliary tables.
- Remove code for ending a transaction across the whole of database initialization (see Section 6.2.12 for a detailed explanation of the reasoning for this).
- Add a call to the storage module `create_finalize` method after all tables have been initialized. This runs any post table creation schema creation commands (such as trigger creation).

#### 6.1.3 DTN2/servlib – Makefile

- Add extra file to compile `bundling/BundleDetail.cc`.

#### 6.1.4 DTN2/servlib/bundling – BundleDaemon.cc

- Style correction: Changed names of routines `beginTransaction` and `endTransaction` to `begin_transaction` and `end_transaction`.

#### 6.1.5 DTN2/servlib/bundling – BundleDetail.cc and BundleDetail.h

- New class `BundleDetail` defined derived from `oasys::StoreDetail` to be used for feeding data to the `bundles_aux` table.
- Normal constructor contains a sequence of calls to method `add_detail` specifying the items that will be written into the `bundles_aux` table.
- The class also has a 'dummy' constructor `BundleDetail(const oasys::Builder&)` that is used by `get_table`.
- It also has a method `durable_key` that is used when constructing the key field for database tables.

#### 6.1.6 DTN2/servlib/cmd – StorageCmd.cc

- Add new storage comand to set boolean variable `odbc_use_aux_tables`.

#### 6.1.7 DTN2/servlib/storage – BundleStore.h

- The `BundleDetailTable` class is defined through the `oasys::InternalKeyDurableTable` template using the `BundleDetail` class as the data type in the prototype.
- The static boolean variable `using_aux_table_` is defined and set true when an auxiliary table is defined and in use.
- The private variable `bundle_details_` has the instance of the `BundleDetailTable` when it is used.

### 6.1.8 DTN2/servlib/bundling – BundleStore.cc

- If LIBODBC\_ENABLED is defined, **bundle\_details\_** is initialised with a new instance of **BundleDetailTable** but **using\_aux\_table** is initially set to **false**.
- When the **BundleStore::init** method is called, if the storage module indicates that auxiliary tables are available (call the method **aux\_tables\_available**), then the **do\_init\_aux** method is called for the **BundleDetailTable** instance, and, assuming success, **using\_aux\_table\_** is set **true**.
- If **using\_aux\_table\_** is true, the **add** method instantiates a new instance of **BundleDetail** passing in the bundle instance that is being inserted into database. This picks out the fields to be stored in the auxiliary table and stores them into the vector of column details stored in the **BundleDetail** instance. This is then passed as a parameter to the **add** method of the **BundleDetailTable**. On completion of the addition, the **BundleDetail** instance is deleted.
- The **get** method is NOT altered. It is assumed that the main table contains all the information needed to reconstruct a bundle so that the auxiliary table need not be accessed when reading bundles back in (the only time this is called).
- The **update** method has some code similar to the **add** code that would update the contents of the auxiliary table, but it is currently disabled as it is assumed that the contents of the auxiliary table are derived from the non-mutable elements of the bundle. If this is changed this code might have to be enabled. It could be enabled immediately but currently would just be an overhead as the auxiliary table row would be rewritten with the same values.
- The **del** method requires no modification as it is assumed that the auxiliary table row will be deleted by a database trigger.
- The **close** method calls the **close** method for the **BundleDetailTable** if **use\_aux\_table\_** is **true**.

### 6.1.9 DTN2/sqldefs – New directory

- Provided for storing prototype schema creation scripts.

### 6.1.10 DTN2/sqldefs – create\_aux.sql

- Contains SQL code to be read into the database as part of the initialisation process. This code is read in before the main tables are created during the first part of database initialisation in the **init** method of **ODBCDBMySQL** and **ODBCDBSQLite**.
- The full path name to this file (or a user modified version) is placed in the ODBC configuration file *odbc.ini* as the value of property **SchemaPreCreationFile**.
- Currently create *bundles\_aux* table with default fields to match stored **BundleDetail** class.
- Contains tricks that allow it to read in either by **sqlite3** or **mysql** client programs:
  - SQL statements that span more than one line have each non-final line terminated by '\'. The backslash characters ('\') are edited out before feeding to **sqlite3** but remain when using **mysql**.
  - When creating triggers or stored procedures, **mysql** requires that the SQL statement delimiter ( normally semi-colon) is temporarily replaced by another character so that the trigger or stored procedure can contain multiple SQL statements. **Sqlite3** appears to use a more intelligent parser that recognizes the start of triggers, etc., and does not require an alternative delimiter. The alternative delimiter statements (currently lines containing 'DELIMITER |' and '|') are entered as comments using the '- -' comment syntax. 'Real' comments are entered using the alternative comment syntax ('/\* ... \*/'). The '-' comment characters are edited out before feeding to **mysql**.

### 6.1.11 DTN2/sqldefs – create\_trigger.sql

- Contains SQL code to be read into the database as part of the initialisation process. This code is read in after the main tables are created at the end of database initialisation in the **create\_finalize** method of **ODBCDBMySQL** and **ODBCDBSQLite**.
- The full path name to this file (or a user modified version) is placed in the ODBC configuration file *odbc.ini* as the value of property **SchemaPostCreationFile**.
- Currently defines triggers to maintain referential integrity between the *bundles* table and the *bundles\_aux* table.
- Use the same tricks as **create\_trigger.sql** to make the same code acceptable to **mysql** and **sqlite3**. (See Section 6.1.10).

## 6.2 Oasys

All changes are in the storage directory apart from the main Makefile

### 6.2.1 Makefile

- Added storage/StoreDetail.cc to the list of files.

### 6.2.2 oasys/storage – BerkeleyDBStore.cc and BerkeleyDBStore.h

- Style correction: Changed names of routines beginTransaction, endTransaction and getUnderlying to begin\_transaction, end\_transaction and get\_underlying.

### 6.2.3 oasys/storage – DurableStore.cc

- Add **create\_finalize** method to be called as last part of database initialization. Calls corresponding function in storage implementation.
- Correct ASSERTs in **get\_table\_names** and **get\_info** by inverting test.
- Add **aux\_tables\_available** method to report to higher levels if auxiliary tables are implemented and in use in the current storage module. Calls corresponding routine in implementation module.
- Change error report on MYSQL\_ENABLED to recommend using ODBC /MySQL.
- Style correction: Changed names of routines and variables beginTransaction, endTransaction, isTransactionOpen, getOpenTransaction and haveSeenTransaction to begin\_transaction, end\_transaction, is\_transaction\_open, get\_open\_transaction and have\_seen\_transaction.

### 6.2.4 oasys/storage – DurableStore.h

- Add additional flag **DS\_AUX\_TABLE** to indicate that table is an auxiliary one.
- Add **create\_finalize** method.
- Add **aux\_tables\_available** method.
- Add transaction counter **tx\_counter\_** for tracking transactions.
- Style correction: Changed names of routines and variables beginTransaction, endTransaction, isTransactionOpen, getOpenTransaction and haveSeenTransaction to begin\_transaction, end\_transaction, is\_transaction\_open, get\_open\_transaction and have\_seen\_transaction.

### 6.2.5 oasys/storage – DurableStoreImpl.cc

- Add default implementation of **aux\_tables\_available** method returning **false** always.
- Style correction: Changed names of routines `beginTransaction`, `endTransaction` and `getUnderlying` to `begin_transaction`, `end_transaction` and `get_underlying`.

### 6.2.6 oasys/storage – DurableStoreImpl.h

- Add virtual **create\_finalize** method with empty default implementation.
- Add virtual **aux\_tables\_available** method.
- Style correction: Changed names of routines `beginTransaction`, `endTransaction` and `getUnderlying` to `begin_transaction`, `end_transaction` and `get_underlying`.

### 6.2.7 oasys/storage – InternalKeyDurableTable.h

- Add templated **do\_init\_aux** function with same interface as **do\_init**.

### 6.2.8 oasys/storage – InternalKeyDurableTable.tcc

- Add templated **do\_init\_aux** function with same interface as **do\_init**. Also has same implementation except always adds flag `DS_AUX_TABLE` to flags passed to storage module **get\_table** function.

### 6.2.9 oasys/storage – ODBCMySQL.cc and ODBCMySQL.h

- New class **ODBCDBMySQL** derived from **ODBCDBStore**.
- Comments explaining configuration of ODBC for MySQL.
- Provides **init**, **create\_finalize** and **get\_info** methods to complete virtual method implementations for **DurableTableImpl** base class.
- Provides **parse\_odbc\_ini\_MySQL** to parse the `odbc.ini` configuration file(s) looking for the DSN section to be used and returning various values from it to use in tidying the database and running the additional schema initialization commands.
- Remove code that purported to enclose the database schema initialization in a single transaction. This causes problems with SQLite when reading in schema configuration files as well as creating tables via ODBC because SQLite cannot allow nested transactions and access from schema creation files is treated as a nested transaction if auto-commit is turned off. There is also no advantage to enclosing the code in a transaction because `DROP TABLE`, `CREATE TABLE` and `CREATE TRIGGER` SQL statements forcibly terminate any in progress transaction. Further, attempting to rollback the hard initialization of the database seems somewhat gratuitous – one might as well rerun the initialization if a problem is encountered since the database is necessarily empty.

### 6.2.10 oasys/storage – ODBCSQLite.cc and ODBCSQLite.h

- New class **ODBCDBSQLite** derived from **ODBCDBStore**.
- Comments explaining configuration of ODBC for SQLiteL.
- Provides **init**, **create\_finalize** and **get\_info** methods to complete virtual method implementations for **DurableTableImpl** base class.
- Provides **parse\_odbc\_ini\_SQLite** to parse the `odbc.ini` configuration file(s) looking for the DSN section to be used and returning various values from it to use in tidying the database and running the additional schema initialization commands.

### 6.2.11 oasys/storage – StoreDetail.cc and StoreDetail.h

- New classes **StoreDetail** and **StoreDetailItem** defined.
- Class **StoreDetail** is intended to be used as the base class for the data passed into the table maintenance function on the table manager implementations (e.g., **ODBCDBTable**) when the table is an auxiliary table. Derived from **SerializableObject** but with a dummy **serialize** method that should never be used. Data for the database is instead derived from the vector of **StoreDetailItem** instances built in the instance on construction.
- Provides an enumerated type **detail\_kind\_t** that provides abstractions of the sorts of data that might be stored in auxiliary table columns.

### 6.2.12 oasys/storage – ODBCStore.cc and ODBCStore.h

- Refactor **ODBCDBStore** class as a base class for storage modules using ODBC to access a type of SQL database (see Sections 6.2.9 and 6.2.10). Remove **init** and **parse\_odbc\_ini** methods to derived classes.
- Add extra comments on use of ODBC and configuring it.
- Provide a set of maps for determining the parameters to be used with ODBC **SQLBindParameter** calls according to the value of the **detail\_kind\_t** defined in **StoreDetail.h**. The maps give the appropriate **SQL\_C\_XXX** type code to use for the *ValueType* parameter and the **SQL\_XXX** value code to use for the *ParameterType* parameter. An additional map indicates the expected SQL column type that would be used to construct the schema for the table(s), but this is not currently used and the schema has to be manually inserted in the schema creation scripts. [TBD It would in principle be possible to create the table definition from the **StoreDetail** instance to avoid this situation which provides a double maintenance burden and potential failure point.]
- Add a protected constant **ODBC\_INI\_FILE\_NAME** with the default name of the ODBC configuration file (odbc.ini).
- Use SQL phrase **BEGIN** rather than **BEGIN TRANSACTION** for starting a transaction in **begin\_transaction**. This phrase is common to SQLite and MySQL. [Warning: it may differ in other databases.]
- Use SQL phrase **COMMIT** to commit a transaction rather than using ODBC routine **SQLEndTran** in **end\_transaction**.
- For auxiliary tables, skip table creation in **get\_table**. It is expected to be done by the schema creation files.
- Add extra boolean parameter to constructor of **ODBCDBTable** class to indicate whether or not the table is an auxiliary table. Constructor records this and uses value to affect various routines in **ODBCDBTable**.
- Add public method **aux\_tables\_available** and associated private variable **aux\_tables\_available** in **ODBCDBStore**. The method returns a boolean value indicating if auxiliary table are available (depending on the user configuration of **odbc\_use\_aux\_tables**).
- Add protected methods **connect\_to\_database**, **set\_odbc\_auto\_commit\_mode**, and **create\_aux\_tables**. These methods contain pieces of commonly used code needed by derived class **init** methods.
- Modify the **get** (for single type table) and **put** methods in **ODBCDBTable** to provide different functionality when the table is an auxiliary table. In each case the row reading and writing has to be done with dynamically constructed SQL derived from the vector of **StoreDetailItem** instances passed in as a parameter in a (derived

## Adding Auxiliary Tables to DTN2

class) of **StoreDetail**. Also suppress the row insertion for auxiliary tables because it is carried out by a database trigger.

- Style correction: Changed names of routines and variables `beginTransaction`, `endTransaction`, `getUnderlying` and `serializeAll` to `begin_transaction`, `end_transaction`, `get_underlying` and `serialize_all`.

### 6.2.13 oasys/storage – StorageConfig.h

- Add new boolean item **`odbc_use_aux_tables`**. Set **`true`** if user wishes to use auxiliary tables if they are available.

## 7. Installation

Before commencing installation of the updated DTN2/Oasys package, two choices have to be made:

1. The ODBC Driver Manager package to be used: Options are
  - unixODBC (<http://www.unixodbc.org/>); and
  - Independent Open Database Connectivity - iODBC (<http://www.iodbc.org/>).
2. The SQL database to be used. Two databases are currently supported:
  - SQLite version 3
  - MySQL versions 5.x

The DM unixODBC has been used during development and will be assumed to be chosen here. IODBC should work also but has not been tested.

### 7.1 ODBC Driver Manager

Source code releases of unixODBC are available from <http://www.unixodbc.org/>.

Ubuntu 10.04 LTS and many other Linux distributions have pre-built packages for unixODBC.

If using pre-built packages both the run time package (Ubuntu *unixodbc*) and the development package (Ubuntu *unixodbc-dev*) need to be installed. The development package contains the SQL header files (*sql.h*, *sqlxext.h*, *sqltypes.h* and *sqlucode.h*) needed to compile the Oasys code when ODBC is used.

It may also be helpful to install the unixODBC graphical tools (*unixodbc-bin*) and the support library for accessing ODBC initialisation files (*odbcinst1debian1*).

### 7.2 SQL Database

In each case both the database and the ODBC driver are needed.

#### 7.2.1 SQLite3

Source code releases of SQLite version 3 are available from <http://www.sqlite.org/>.

Source code releases of the SQLite ODBC driver are available from <http://www.ch-werner.de/sqliteodbc/>.

Ubuntu 10.04 LTS and many other Linux distributions have pre-built packages for SQLite3 (Ubuntu *sqlite3*) and the driver (Ubuntu *libsqliteodbc*).

The SQLite3 development library (Ubuntu package *libsqlite3-dev*) may be needed if installing the ODBC from source.

No further work is needed to install SQLite as the actual databases are just files and access is primarily controlled by file permissions.

### 7.2.2 MySQL

Source code releases of MySQL and the Connector/ODBC ODBC driver are available from <http://www.mysql.com/>. The driver downloads are at <http://dev.mysql.com/downloads/connector/odbc/>.

Ubuntu 10.04 LTS and many other Linux distributions have pre-built packages for MySQL (Ubuntu *mysql-server* and *mysql-client*) and the driver (Ubuntu *libmyodbc*). Both client and server packages are needed. The client is essential both for maintenance and the initialisation of the DTN2 database if additional schema creation files are used. Note that Ubuntu maintainers currently recommend the Version 5.1 of the MySQL server rather than the several more recent releases. This has proved satisfactory for the DTN2 application.

After installing the software, it may be necessary to run the initial configuration script *mysql\_install\_db*. See [Unix Postinstallation Procedures](#) in the MySQL manual for details. Further specific configuration of users and permissions is covered in the next section.

MySQL offers two versions of the Connector/ODBC component, both of which are currently maintained:

- **Version 3.51:** This version does not support Unicode, but that is irrelevant for DTN2/Oasys
- **Version 5:** This version does support Unicode and is Oracle's preferred solution for recent database versions.

Both versions appear to work satisfactorily with Version 5.0 or 5.1 of the database.

### 7.3 Compiling Oasys and DTN2

After installing the selected ODBC Driver Manager, (re-)run the **configure** scripts in turn for Oasys and DTN2. After the modifications described in this document have been installed **configure** will default to attempting to find an ODBC installation and compiling versions of DTN2 and Oasys that can use ODBC, SQL databases and auxiliary tables. If you do not require ODBC and the SQL database storage mechanism, specify `-without-odbc` on the **configure** command line. It may be necessary to specify the location of the ODBC installation if it is not in one of the usual places (`/usr`, `/usr/local`).

Compile and install Oasys and DTN2 in order as usual (**make** and **make install**).

## 8. Configuration

.A number of configuration files have to be set up to successfully use SQL storage through ODBC with DTN2 and Oasys. Figure 2 shows the various software components and the associated configuration files.

For MySQL there is also database creation and user permission configuration to provide.

This section assumes that the configurer has some basic knowledge of SQL and the operation of the selected database.

## Adding Auxiliary Tables to DTN2

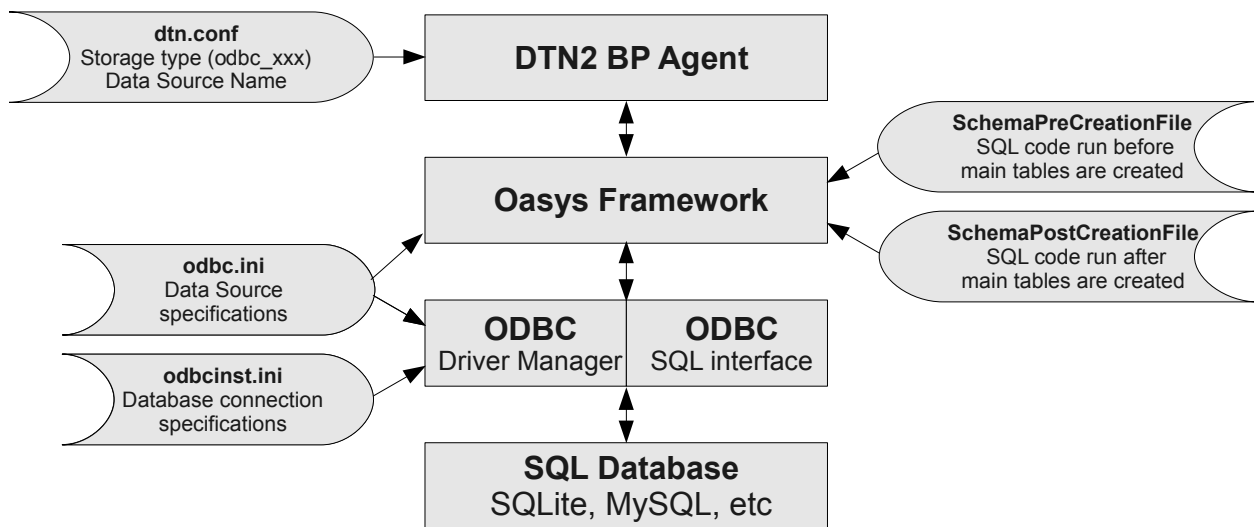


Figure 1: Configuration Files

## 8.1 Database Configuration

### 8.1.1 SQLite3

No explicit configuration is needed. Check that the user that will be running the DTN2 daemon has permission to run the **sqlite3** client program and can create files in the directory selected for holding DTN2 databases (this directory will be configured into the ODBC Data Source entry later). Use **sqlite3** to create a database file and create a simple table. The **create\_aux.sql** can be used as a test: the following should create the **bundles\_aux** table.

```
cat DTN2/sqldefs/create_aux.sql | tr -d '\n' | sqlite3 <database file pathname>
```

Check that this has been successful:

```
shell> sqlite3 <database file pathname>
```

```
sqlite3> .schema
```

*<Examine schema displayed - should have bundles\_aux table.>*

Exit from **sqlite3** by typing **ctrl-D** or **.quit**.

Proceed with ODBC configuration.

### 8.1.2 MySQL

It is assumed here that the MySQL database server runs on the same host as the DTN2 daemon and that this can be referred to as **localhost**. Oasys and the DTN2 daemon would in theory work if this was not the case except for installing the extra schema configuration files. [TBD this could be fixed by passing the hostname to the **mysql** command when running the schema configuration file but is not currently done as it seems that DTN2 would be likely to use isolated hosts in many cases.]

For MySQL it is necessary to create a database and one or more users before initialisation of the tables by DTN2/Oasys.

If transactions are to be used rather than relying on the default auto-commit functionality, it is necessary to select the MySQL InnoDB engine for use with the DTN2 tables. The simplest way to do this is to set InnoDB as the default engine for all tables in the database. This can be done by editing the server configuration file. This is typically **/etc/mysql/my.cnf**. Add (or uncomment) the line

## Adding Auxiliary Tables to DTN2

default-storage-engine=innodb

to the **[mysqld]** section.

Select a user name and a password for the user that will be accessing the database via DTN2. There is no need for the name of this user to match a Linux user name and the password can (and should) be something different from any valuable passwords for the Linux user that is running the DTN2 daemon. The password will be written *en clair* in the `odbc.ini` file later.

Select a name for the DTN2 database. It is advisable to choose a name using the character set `[A-Za-z0-9$_]` to avoid the need for complicated quoting of the name. The Oasys software will warn if the database name does not follow this recommendation, but the existing Oasys software will run with any valid name. Remember that database names are case sensitive.

Using the root user and password set up when MySQL was installed (see Section 7.2.2), run the **mysql** client. Create the database and verify it is present:

```
mysql> CREATE DATABASE <name of database>;
mysql> SHOW DATABASES;
```

Create the user for the localhost domain.

```
mysql> CREATE USER <user name>@localhost IDENTIFIED BY '<password>';
```

Give the user permissions for all tables in the DTN2 database:

```
mysql> GRANT ALL ON <database name>.* TO <user name>@localhost IDENTIFIED BY
'<password>';
```

For earlier versions of MySQL it may also be necessary to separately grant the user SUPER permissions in order to create triggers (ALL covers the TRIGGER option but this was only introduced part way through the Version 5.1 series of releases).

```
mysql> GRANT SUPER ON <database name>.* TO <user name> IDENTIFIED BY
'<password>';
```

Exit from **mysql** (*exit*; or *Ctrl-D*) and check that the new user and database are working as expected. Try running:

```
shell> cat DTN2/sqldefs/create_aux.sql | tr -d '-' | mysql <database name> -u
<user name> -p
```

Enter the password when requested. Assuming this runs successfully, verify that the table has been created.

```
shell> mysql <database name> -u <user name> -p
```

Enter the password when requested, then try the following commands:

```
mysql> SHOW TABLES;
```

This should show that the `bundles_aux` table exists.

Exit from **mysql** and proceed with ODBC configuration.

## 8.2 ODBC Configuration

The ODBC configurations for SQLite and MySQL are significantly different. Note that it is fine to install configurations for several different database types and individual databases. The DTN2 configuration specifies which one to use at run time.

The configuration of ODBC uses two or three files:

- **odbcinst.ini** - contains a list of named database drivers available for ODBC and the names of the shared libraries that implement the drivers. The appropriate library is dynamically loaded when a connection to the selected database is opened. When using unixODBC the property **Threading** can

## Adding Auxiliary Tables to DTN2

take one of the values 0, 1, 2 or 3. This controls the level of isolation between multiple connections through the ODBC driver when these are handled by different threads. Since DTN2 only uses a single connection, the value of threading is currently irrelevant. Level 3 provides maximum protection allowing only one connection to access ODBC at once. It also has some additional configuration and setup information for the driver. This includes a 'plugin' that informs the setup GUI application the properties that can be configured for a data source using this driver. Not all ODBC drivers offer this facility for Linux. [AFAICS this just requires a dynamically loaded library offering a single function ODBCINSTGetProperties but it isn't documented. See `odbcinst/ODBCINSTConstructProperties.c` in `unixODBC` source tree.] The property `UsageCount` is maintained by the automated configuration tools and records the number of data sources currently specified as using the driver. This field is optional. [Unclear if this includes user data sources in `.odbc.ini` as user doesn't have permission to update this file.]

- **odbc.ini** – typically `/etc/odbc.ini`, contains named specifications of Data Sources (Data Source Names - DSNs) that can be connected to using ODBC. Each data source specifies the driver defined in **odbcinst.ini**, the database to be accessed and configuration information that allows the connection to be made and controlled. Data Sources defined in this 'system' `odbc.ini` file are available to any user on the machine.
- **.odbc.ini** – typically in a user's home directory, contains additional named specifications of Data Sources specific to the user. [At present DTN2/Oasys cannot use the user `.odbc.ini` file. Do not put DSNs to be used by DTN2 in this file as they will be ignored or, if duplicated, cause incorrect operation of DTN2. This should be fixed.]

The ODBC configuration files were originally Microsoft Windows ini files and conform to the syntax used in those files. Each file consists of a number of sections introduced by a name in square brackets (e.g., **[DSN for DTN]**). Section names may contain white space. In each section are a number of property definitions of the form

```
<property_name> = <property_value>
```

*Property names* are single words and are not case sensitive. The '=' can be surrounded by as much white space as is convenient to improve the file layout. The *property value* includes any white space after the first non-white space character following the '=' and is case sensitive.

Comment lines start with '#' or ';'. Property lines may contain trailing comments.

Both **odbcinst.ini** and **odbc.ini** are system files and should be only writeable by the *root* user. Accordingly *root* privileges are needed to modify them.

The normal location for the system files is the directory `/etc` but they may also be found in `/etc/odbcinst`. Alternatively, if the environment variable **ODBCINST** is defined, it should contain the name of the directory containing the system ODBC configuration files.

More information about the configuration files can be found at <http://www.unixodbc.org/odbcinst.html>.

### 8.2.1 Relationship of ODBC Configuration to DTN2/Oasys

The DTN2/Oasys storage configuration specifies a Data Source Name (DSN) that will be connected through ODBC to provide DTN2's persistent storage.

In principle, the ODBC and DTN2 storage configuration should only be connected by this single textual name. However, there are currently some additional 'interference' between the two sets of configuration. This could be considered a 'layer violation' and should probably be fixed, although this might be difficult for SQLite.

- **For SQLite** the database is just an ordinary file and needs to be accessible to the user running the DTN2 daemon. There is a potential problem because of the way that the 'tidy' functionality is specified for DTN2. One part of it destroys the directory specified in the DTN2 configuration where

## Adding Auxiliary Tables to DTN2

the **.ds-clean** file is created to indicate that the database was shutdown cleanly when the last daemon execution ended. Since the database file name is buried in the ODBC DSN it is possible that the tidy function would destroy the database file (and others) if the two separately specified directories are the same. This is, at least partly, a hangover from the Berkeley DB storage mechanism. The code as currently implemented for SQLite warns the user that this is happening. If they aren't the same, the database file is left in existence and the tables dropped to tidy up the database. This is all a bit messy. One solution would be to do away with the directory removal and just wipe the **.ds\_clean** file if it exists and drop the database tables.

- **Both SQLite and MySQL** read in additional schema creation files using the standard database client (respectively **sqlite3** and **mysqlb**) rather than ODBC. This means at present that the Oasys code has to use information extracted from the ODBC configuration to invoke these clients. This is messy and also makes assumptions about where the MySQL server is located. [I have just realized that there is a trivial solution to this. Use the ODBC **isql** client which uses the DSN to access the database and is common across the various databases. Minor snag it doesn't return an error code when the SQL doesn't execute in batch mode. Might have grep the output for ERROR. Another snag is that **isql** has been 'improved' in version 2.3 of unixODBC.]

### 8.2.2 Header Sections of **odbcinst.ini** and **odbc.ini**

The **odbcinst.ini** file may contain a header section that lists which drivers are actually installed. The section is optional and doesn't seem to be enforced by any tools currently. The section has the form:

```
[ODBC Drivers]
driver_name1=Installed
driver_name2=Installed
```

The *driver\_name* properties are the names of sections later in the file.

Similarly the **odbc.ini** and **.odbc.ini** files may contain a header section that lists the DSNs in the file. The section is optional and doesn't seem to be enforced by any tools currently. The section has the form:

```
[ODBC Data Sources]
myodbc5           = MyODBC 3.51 Driver DSN test
dtn-test-mysql    = MySQL DTN test database
dtn-test-sqlite   = SQLite3 DTN test database
dtn-prodn-sqlite  = SQLite3 DTN production database
```

The property names are the names of sections in the remainder of the file.

### 8.2.3 ODBC Configuration for SQLite3

The SQLite3 DSN primarily has to specify the pathname of the database file plus a small number of parameters.

#### 8.2.3.1 Driver Linkage in **odbcinst.ini**

For the SQLite3 ODBC driver from <http://www.ch-werner.de/sqliteodbc/> the run time driver library is also used for the setup plugin.

## Adding Auxiliary Tables to DTN2

```
[SQLite3]
Description = SQLite3 ODBC Driver - Version xxx
Driver      = <pathname>/libsqlite3odbc.so
Setup      = <pathname>/libsqlite3odbc.so
Threading  = 2
UsageCount = 0
```

### 8.2.3.2 Data Sources in *odbc.ini*

The key parameters in this configuration are the **Database**, **Timeout** and **NoTxn**. SQLiteodbc will not handle transactions in DTN2 correctly unless **NoTxn** is set to **Yes**. The trace file settings are optional – ODBC tracing is enabled in SQLiteodbc and can be useful in tracking what is going on.

```
[dtn-test-sqlite]
Driver          = SQLite3
Description     = Connection to DTN2 ODBC test database with SQLite
Database       = <full pathname of database file>
Timeout        = 100000
NoTxn          = Yes
StepAPI        = No
NOWCHAR        = No
LongNames      = No
Trace          = Yes
TraceFile      = <pathname>/odbclog
SchemaPreCreationFile = <pathname>/DTN2/sqldefs/create_aux.sql
SchemaPostCreationFile = <pathname>/DTN2/sqldefs/create_trigger.sql
```

[Note: The schema creation file entries should be moved to the DTN2 /Oasys configuration. They are not used by ODBC.]

## 8.2.4 ODBC Configuration for MySQL

The MySQL DSN primarily has to specify the name of the database to be used together with the username and password of the user used by DTN2 to access the MySQL database..

### 8.2.4.1 Driver Linkage in *odbcinst.ini*

For the MySQL ODBC drivers from <http://dev.mysql.com/downloads/connector/odbc/> there is no setup library available. Leave the **Setup** property blank.

MySQL currently has two versions of Connector/ODBC as described in Section 7.2.2. The driver libraries have distinct names:

```
[MyODBC Version 5.1]
Description     = MySQL Connector/ODBC Version 5.1
Driver         = <pathname - typically /usr or /usr/local/>/lib/libmyodbc5.so
Setup          =
Threading      = 2
UsageCount     = 0
```

```
[MyODBC Version 3.51]
Description     = MySQL Connector/ODBC Version 3.51
Driver         = <pathname - typically /usr or /usr/local/>/lib/libmyodbc3.so
Setup          =
Threading      = 2
UsageCount     = 0
```

### 8.2.4.2 Data Sources in *odbc.ini*

The key parameters in this configuration are the **Database**, **Server**, **User** and **Password**. The values used are those setup when the MySQL database was configured (see Section 8.1.2). The trace file settings are

## Adding Auxiliary Tables to DTN2

optional – ODBC tracing is enabled in Connector/ODBC if it is compiled with debugging enabled (not the default) and can be useful in tracking what is going on. The **Option** property is effectively a bit map but has to be expressed as a decimal number. The bit values (there are many of them) are defined in the MySQL Reference Manual [Connector/ODBC Connection Parameters for Server Version 5.1](#). It does not appear that any options need to be set for the DTN2 application. This manual entry also describes the other properties that could be used.

```
[dtn-test-mysql]
Driver           = MyODBC Version 5.1
Description      = Connection to MySQL DTN2 test database with
                  Connector/ODBC Ver 5
Server          = localhost
Socket          =
Port            =
User            = <username>
Password        = <password>
DataBase        = <database name>
Trace           = No
TraceFile       = <full pathname of trace file>
SchemaPreCreationFile = <pathname>/DTN2/sqldefs/create_aux.sql
SchemaPostCreationFile = <pathname>/DTN2/sqldefs/create_trigger.sql
Option          = 0
```

[Note: The schema creation file entries should be moved to the DTN2 /Oasys configuration. They are not used by ODBC.]

### 8.3 DTN2 Configuration – dtn.conf

The usage of the **storage** parameters in the DTN2 configuration file (normally **dtn.conf**) is slightly modified when using ODBC and SQL databases. The configuration needs to specify the type of SQL database used and the DSN to be used to connect to the database via ODBC.

The **storage type** setting is either, for SQLite3:

```
storage set type odbc-sqlite
```

or for MySQL:

```
storage set type odbc-mysql
```

The **storage dbname** parameter is used to identify the DSN to be accessed via ODBC:

```
storage set dbname dtn-test-xxx
```

The **storage dbdir** parameter is used only to identify the directory where the **.ds\_clean** file is created to indicate that the database was shut down cleanly at the end of the previous run of the **dtnd** daemon. For storage types other than ODBC, **dbdir** also contains the database files and the database tidy function (invoked by **dtnd - -tidy**) deleted this entire directory. If this scheme is followed mindlessly for the ODBC case there is chance of inadvertent deletion of database files. In the case of MySQL database, the database storage is typically not owned by the DTN2 user and is hidden from DTN2 by the server so that there is very little chance of confusion between **dbdir** and the location of the database files. This is not so for SQLite. The database file pathname is specified separately in the DSN in **odbc.ini** and there is a risk (or intentional selection) that the two directories are the same. The tidy functionality currently checks and warns if the two directories are the same before clearing **dbdir**. It might be appropriate to alter the functionality so that for the ODBC case, the tidy functionality just deletes **.ds\_clean** if it exists and drops the database tables to avoid risk of problems.

The **storage payloaddir** is still used to hold the bundle payload files, which are not stored in the database.

## Adding Auxiliary Tables to DTN2

The **storage auto-commit** parameter can be set to **true** or **false**.

```
storage set auto-commit true ; All SQL statements are committed immediately
```

```
storage set auto-commit true ; Explicit, larger scope transactions are used
```

If an SQLite database is being used with **auto-commit false**, it is essential that the DSN sets **NoTxn = Yes**. Otherwise sqliteodbc generates transactions automatically resulting in the transactions in DTN2/Oasys causing a 'nested transaction' error.

## References

- [1] MITRE Inc, "Modifications to the DTN2 Reference Implementation of the Bundle Protocol (BP) ",
- [2] K. Scott and S. Burleigh, Bundle Protocol Specification, RFC5050, The Internet Society, November 2007. <http://www.rfc-editor.org/rfc/rfc5050.txt>
- [3] IRTF DTN Research Group, Sourceforge Code Repository, <http://dtn.hg.sourceforge.net/hgweb/dtn>