TRINITY COLLEGE DUBLIN
COLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH

I

Improvements and Additions to the DTN2 Reference Implementation of the Bundle Protocol and the Oasys Framework Adding SQL Database Storage and Auxiliary Tables

Elwyn Davies

School of Computer Science and Statistics Technical Report TCD-CS-2013-1
Distributed Systems Group **June 14, 2013**

Improvements and Additions to the DTN2 Reference Implementation of the Bundle Protocol and the Oasys Framework Adding SQL Database Storage and Auxiliary Tables

1. Introduction

This document describes some work[1] 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 Bundle Protocol (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 (dtnd) 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.

1.1 Additional Improvements to SQL Schema

As part of the changes documented in [1], persistent storage of the forwarding log for bundles was turned on. This capability of DTN2 had previously not been active as it relies on link names being consistent across restarts of the DTN2 daemon. For statically configured links (those created with the 'link add' command) this problem can be sidestepped by careful management of configurations. However, for OPPORTUNISTIC links this is not the case, as link names are automatically generated and depended only on the order of creation during a particular run of the DTN2 daemon. This was a known problem that had been sidestepped by suppressing the persistent storage of both link data and the bundle forwarding logs.

Unfortunately, the issue of OPPORTUNISTIC links was not addressed when the persistent storage of forwarding logs was turned on and the issue of changed correspondence between link names and remote end

1 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.

Endpoint Identifier (EID) sometimes provoked a panic when a node was using OPPORTUNISTIC links to multiple peers and the daemon was restarted, depending on the order in which opportunistic links were restarted.

To remedy this problem, storage of link data and checking for consistency has now been implemented. While attempting to enable storage of link data it became clear that the initial design of the SQL storage subsystem generally catered only for the tables that used integers as keys with the *globals* table treated as a special case. The *prophet* table as it existed and the 'shadow' system for the *links* table were designed with string keys (respectively the EndpointID – effectively a URI string – of the relevant node for *prophet* and the textual name of the link for *links*). As soon as an attempt was made to test storage in the *links* table, it became clear that a major rethink was needed. In practice the general structure did not require significant alteration but some generalization of the key mechanism was required. The additions and changes in version 1.x of this document document these changes. The result is that the SQL mechanism is now fully general and has removed the special casing needed for the 'globals' table in the previous version.

An additional issue that became clear was that connections to MySQL time out and are automatically closed after being idle for a configurable time, defaulting to 8 hours. For interactive connections this is probably appropriate but for connections to a DTN2 daemon, it is entirely possible that a connection might lie idle for many hours. Accordingly, the additions include a mechanism to execute a simple SQL command periodically even if the daemon is otherwise idle, thereby keeping the connection alive.

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' file-system 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.

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 for 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 local inter-process communications system or 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 then 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 s*qliteodbc* 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. Generalizing the SQL Key Column Solution

In providing the configurable persistent storage system, Oasys passes one or two objects to the generalised *put, get* and *del* routines for each of the tables that have to be implemented in the (templated) storage class that implements each type of storage that can be selected by the user.

These objects represent the (primary) key data and the remainder of the data to be stored in a table row. Each of these objects have to be instances of objects derived from the Oasys::SerializableObject class. For the standard tables that were planned around the capabilities of the Berkeley database (key, value)-tuple storage scheme, the serialize method is called for each object to map between the objects and a pair of opaque binary objects that are then used as key and (where appropriate) value items for the table row. This process is referred to as 'flattening' when going from the extended form to the serialized form.

In the SQL storage class implementation described in [1], the standard tables are created with two columns. In each case the data type for the value column is *blob*, i.e., large undifferentiated binary data item. However, the key column is not fully generalized to cope with arbitrary binary data keys. The *globals table* is specifically recognised and the key is treated specially. All other table keys **are assumed to be 32 bit integers**. Unfortunately, although this is true for the *bundles* and *registrations* tables, that are the main needs of the intended application for the changes documented in [1], it is **not true** for the *prophet* table which would be needed if PRoPHET routing were enabled, or for the *links* table that was not used as of the original release of the SQL capabilities. In both cases, the key is a string: for *prophet* it is an EndpointID URI and for *links* the textual name of the link.

For PRoPHET routing this is disastrous because the per-node information cannot be stored across restarts,.

The changes in [1] also enable another feature in DTN2: previously the forwarding log for bundles was recorded in memory but not added to the persistent state in the *bundles* table. The reason for this was that the names of links were used to record the actions in the forwarding log and link names were not guaranteed to be consistent across restarts. Thus the information in the forwarding log could be unusable and confusing after a restart. This problem can be sidestepped by management control if only configured links are used, since the names of the links are explicitly set by the user. However, with opportunistic links, the names are automatically allocated. Names are constructed using a serial number that is reset to zero when the DTN2 main program (*dtnd*) is restarted. Thus the names of opportunistic links depend on the order in which they are created. This is, in turn, dependent on the order in which discovery announcement beacons are received from peers. This is effectively random, even in the simplest case, because of the random start time of the daemon.

This means that the storage of the forwarding log is incompatible with the use of opportunistic links unless link names are made consistent across restarts. However, not having the forwarding log recreated after restarts results in other problems (which is probably why [1] implemented storage of forwarding logs): for table-based and DTLSR routing, it is assumed that bundles reloaded from the persistent store have not been forwarded at all. This can lead to unnecessary extra forwarding of bundles. The situation is worse for PRoPHET where currently it is assumed that all reloaded bundles *have* been forwarded and they tend to become 'orphaned'.

To fix this problem, a prerequisite is the ability to write link information to persistent storage. Enabling storage in the *links* table revealed the 'compromise' regarding the types of keys that is documented above. A simple solution would be to make all the key columns have SQL type VARBINARY(255), but for the highest volume case (*bundles*) this is fairly inefficient because bundles are keyed by the (integer) bundle id.

The solution adopted, and now implemented, is to add a new static method *shim_length* to the various classes defined in oasys::TypeShims.h that are used to encapsulate objects used as keys when passing them to the storage table routines (see storage/InternalKeyDurableTable.h). This method returns the length of the flattened key value if it is a fixed length (e.g., for integers) or 0 (zero) if the flattened key has a variable length. Because it is static, it can be accessed as a class attribute during creation of the table. The column type can then be set up to be VARBINARY(255) if the *shim_length* returns 0 and a fixed size BINARY column otherwise. The size can be recorded in the table object data for later use when reading and writing records.

With this change, the code in the SQL table access methods in the ODBCDBStore and ODBCDBTable can be rewritten to be completely generic, removing the need for the special cases for the *globals* table and correctly handling the variable length keys used for *links* and *prophet*.

Note that the TypeShim used for EndpointID in the *prophet* table is DTN2 specific and is defined in the DTN2 code rather than the Oasys code.

4. 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.

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.

4.1 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 file-system 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 stored procedures.

4.1.1 Recording Additional State for External Applications

To avoid the necessity for polling or explicit notification mechanisms to allow an external program to be aware of alterations made to the *bundles_aux* table, the external program needs to be able to determine both the additions and deletions from *bundles_aux* when it is accessed as required by the program. Additions are simple because bundles are identified by a monotonically increasing *bundle_id*. The program can maintain a high water mark and access rows with *bundle_id*'s higher than this. [Note that this does not allow for cases where the auxiliary information includes parts of the forwarding log. This information is updated after bundle row creation and would need additional work if the external program was using such information. Current plans avoid the need to know about such changes.]

Deletion of rows can take place in an unrelated order depending on the expiry time and forwarding status of bundles. One way of handling this without the need for any interference by the Oasys framework is to extend the deletion trigger to insert a row into a table recording the deletions, provided with a auto-incrementing index. Garbage collecting this table is problematic if the set of external programs accessing the data is not known.

Finally, in order for the external program to manage the data correctly, it needs to know if the database has been reinitialized. This can be handled by the table creation script recording a row indicating the restart time in a table dedicated to this requirement. Again this can be done without involving the Oasys framework through the database creation scripts.

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

5.1 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.

5.2 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.

5.3 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.

5.4 Con: Process and Resource Overhead

MySQL is a fairly heavyweight package and is likely to consume significantly more resources then 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.

5.5 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).

5.6 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.

6. SQL-related Timers

6.1 Remove Deadlock Timer

This timer did not have any function. When using ODBC, deadlock detection using timers is handled internally in the ODBC driver or at a lower level in the relevant database access library. Deadlock results in a specific error return from the SQLExec and SQLExecDirect routines. In practice, with the trivial queries that are run via Oasys there is a very low probability of deadlock occurring (better not say never!) so this is a very minor issue for DTN2/Oasys. Accordingly the deadlock timer has been removed.

6.2 Add Connection Keep-Alive Timer

Idle database connections to the MySQL database will be shut down automatically by the database after a configurable interval of idleness. The interval is set in the [mysqld] section of the MySQL configuration file *my.cnf*, either by setting *interactive_timeout* (interactive sessions) or *wait_timeout* (non-interactve sessions over TCP networking connections). The defaults for these values are both 8 hours (as seconds).

To avoid MySQL connections being terminated during a long idle period (and such idle periods are quite likely in a delay tolerant networking environment!), a Keep-Alive timer has been added to ODBCMysql.cc only.

The timer period is configurable using

```
   storage set odbc_mysql_keep_alive_interval <interval in minutes>
```

This defaults to 10 minutes (but could be much longer).

7. Additional Bugs Fixed

Two significant problems with the code inherited from Mitre [1] were identified while testing the changes documented here. The analysis and fixes are documented here.

7.1 Bound Parameter State

The *statement handles* used by ODBC (also known as *cursors*) maintain a considerable amount of state within the ODBC driver. This includes:

- the prepared SQL statement last passed to the cursor,
- the set of parameters bound to this cursor,
- the set of result columns bound to this cursor, and
- the last set of results received when the prepared SQL was executed (or SQLExecDirect was executed).

Whilst the permanence of the bindings can have advantages if commands are repeated, it is essential to make sure that bindings are cleared when they are no longer valid. A problem can arise if an earlier statement had more parameters bound than the latest one. It is not sufficient to rebind the parameters that are in use as SQLExec attempts to access data passed through *all* the currently bound parameters. If any of these are 'out of date' and contain stale data pointers, it is likely that an error will occur.

The symptoms of failing to unbind unused parameters include unexpected SQL_NEED_DATA return codes from SQLExec or SQLExecDirect, indicating that the stale data pointer is pointing to a value that indicates the parameter is specified as a 'data-at-execution' parameter (i.e., a negative integer less than -100).

In the code inherited from Mitre, the cursors are 'cleared' before being reused by calling SQLFreeStmt with SQL_CLOSE as the second parameter. However it appears that this does not release the previously bound parameters, and it is necessary to call SQLFreeStmt with SQL_RESET_PARAMS as well. This is now done before any reuse of cursors. It would be less tedious if there was an option to clear everything in one call!

7.2 Serialization Locks

The database access routines may be called from multiple threads. Because of the ODBC driver state documented in the previous section, it is essential that the database access routines are fully serialized.

In principle the code was organized to allow this using the *serialize_all_* flag in ODBCDBStore.

Unfortunately the code as inherited from Mitre contained a systematic bug.

The problem is this code snippet that is used at the start of several routines in ODBCStore.cc:

```
if ( store_->serialize_all_&& !(store_>serialization_lock_.is_locked_by_me()) ) {
    ScopeLock sl(&store_->serialization_lock_, "Access by xxx()");
}
```

Thinking about the way that ScopeLock works it becomes clear that this code is essentially a no-op because the ScopeLock sl is released at the trailing } marking the end of the if statement. It ***does not*** apply a ScopeLock that persists to the end of the routine in which it is placed.

However Maestro Demmer (main begetter of Oasys) has provided us with the required alternative which can look at a flag for optional locking and still provide the required routine wide ScopeLock, i.e., ScopeLockIf. Replacing the above code snippet with

```
ScopeLockIf sl(&store_->serialization_lock_,
               "Access by xxx()",
               store_->serialize_all_);
```

solves the problem. This has now been done.

Symptoms of interleaved calls to the various ODBC routines include SQL parser errors referencing totally garbage code.

8. Summary of Changes to DTN2 and Oasys Implementations

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

8.1 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.

8.2 Changed SQL Type of Key Columns in Tables

Why: To correctly deal with tables which have keys other than integers without resorting to special cases (as was previously done for table *globals*).

Implementation: All keys are now either fixed size BINARY or VARBINARY(255). The size of the key to be used is provided by a static method *shim_length* added to all the TypeShim classes (serialization/TypeShims.h plus EndpointIDShim in DTN2/servlib/storage/ProphetStore.h and explicit setting of flags in the GlobalStore::do_init method in DTN2/servlib/storage/GlobalStore.cc). When setting up tables (in ODBCDBStore::get_table) the SQL column type for the *the_key* column is set according to the result of *shim_length:* If 0, VARBINARY(255), and if a non-zero value, then BINARY(<result of *shim_length>).* When binding the column for *the_key* the SQL type is similarly set accordingly. The result of *shim_length* is passed into the DurableStore::get_table method in bits 16:23 of the *flags* parameter (see DurableStore::KEY_LEN_MASK and DurableStore::KEY_LEN_SHIFT).

The detailed changes to use the different *the_key* column types are made in various methods in storage/ODBCDBStore.cc. The following methods are affected:

- ODBCDBStore::get_table
- ODBCDBTable constructor
- ODBCDBTable::get (both interfaces)

- ODBCDBTable::put
- ODBCDBTable::del
- ODBCDBTable::key_exists
- ODBCDBIterator constructor

In each of these methods fairly wholesale reconstruction has been done to remove specialized code for the *globals* table and modify the key column specification appropriately depending on the *key_size_* value derived from the *shim_length* and set in the ODBCDBTable constructor.

8.3 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 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.

8.4 SQL Timer Modifications

8.4.1 Deadlock Timer Deleted

Why: As noted in Section 6.1 timer-based deadlock protection is not needed for ODBC/SQL database access – a broken deadlock is reported as an error return from SQLExec or SQLExecDirect.

Implementation: Deadlock related code removed from ODBCStore.cc/h, ODBCMySQL.cc and ODBCSqlite.cc.

8.4.2 Keep-Alive Timer Added

Why: As described in Section 6.2 MySQL connections are closed by the server after an extended period of idleness. The timer triggers a non-intrusive query on the database periodically to ensure the connection remains alive.

Implementation: The class ODBCDBMySQL::KeepAliveTimer has been created and an additional storage configuration parameter *odbc_mysql_keep_alive_interval_* has been added to provide the interval between activations (see storage/StorageConfig.h). The timer is created during initialization of the MySQL storage class in storage/ODBCMyQL.cc.

When the timer is triggered, the command `SHOW VARIABLES LIKE 'flush'` is sent to the server and the result read out. To ensure that this does not interfere with normal operations, this has to be serialized as regards access to the server. Serialization uses the *serialize_all_* flag and the lock implemented in ODBCDBStore but requires addition of access routines as the timer class is not embedded in ODBCDBStore [Consider whether this should be altered?]

8.5 Removal of db_type, base_dbenv and Current table_name Concept

The ODBC code was originally cloned from the BerkeleyDB code. In a BerkeleyDB a number of different types of data storage are possible selected from the DBTYPE enumeration. As inherited from Mitre, the ODBCStore code maintained a *db_type_* variable with a type that was synthesized from the DBTYPE definition in the BerkeleyDB 'standard' header db.h. The probable intention was that this should be used in a similar way for ODBC tables. It turns out that this is not really appropriate when using ODBC.

When using ODBC, the abstraction of the extra ODBC layer reduces the ability to control the storage engine used from the application, and any choices are necessarily dependent on the underlying database in use. Thus

- for SQLite there are no choices, and
- for MySQL there are a number of choices. In practice if transactions and recovery are wanted, this limits the choice to the InnoDB engine. The traditional MyISAM engine is slightly faster at the cost of table level locking vs. row level locking and no transactions. The engine used can either be selected by a configuration default that affects all tables created or, on a table-by-table basis, by a non-standard extension to the CREATE TABLE SQL command.

For the current version, the decision has been taken to rely on the default engine selection mechanism in MySQL and not try to affect the engine selection or 'db_type' from within ODBC.

It is recommended to select the InnoDB engine as the default in MySQL, but MyISAM works well also and is possibly marginally faster. This involves setting a configuration variable in the my.cnf file (typically /etc/mysql/my.cnf):

Section: [mysqld]

Variable: default-storage-engine = InnoDB

Note that (apparently) case is important in 'InnoDB'.

The handles for accessing the ODBC connection are stored in an ODBC_dbenv structure. It appears that the original plan was to have a per table instance of this structure but the solution as implemented in the Mitre code has a single instance of this structure that contains a number of ODBC statement handles used for specific purposes and passed to all table instance classes as well as being maintained in the master ODBCDBStore class instance. Previously the master structure was in base_dbenv_ within ODBCDBStore and a pointer to this structure was maintained in dbenv in the same class. This is redundant now that only one structure instance is actually used. The code has therefore been revised to make dbenv_ the actual structure and remove base_dbenv_ completely.

Additionally the ODBC_dbenv structure contained a current_table field that was presumably intended to record the name of the table that the structure was associated with. Since there is now only one structure, this field was not actually used, and has now been removed.

8.6 General Style Clean-up

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 clean-up.

8.7 Added Oasys Auxiliary File Infrastructure

Why: To provide infrastructure for additional auxiliary tables linked to the main database tables as described in Section 6).

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 an 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 8.8)..

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 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 8.3). 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.

8.8 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 8.7). 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.

9. Summary of Code Changes

9.1 DTN2

9.1.1 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**.

9.1.2 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 9.2.16 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).

9.1.3 DTN2/servlib – Makefile

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

9.1.4 DTN2/servlib/bundling – BundleDaemon.cc

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

9.1.5 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.

9.1.6 DTN2/servlib/cmd – StorageCmd.cc

- Add new storage command to set boolean variable **odbc_use_aux_tables**.
- Add new storage command to set string variable **odbc_schema_pre_creation**.
- Add new storage command to set string variable **odbc_schema_post_creation**.
- Add new storage command to set unsigned integer variable **odbc_mysql_keep_alive_interval**.

9.1.7 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.

9.1.8 DTN2/servlib/storage – GlobalStore.cc

- In method **do_init** pass key length indicator for variable length key in **flags** parameter to **Oasys::DurableStoreImpl::get_table** method.

9.1.9 DTN2/servlib/storage – ProphetStore.h

- Add new static method **shim_length,** returning 0 indicating a variable length key shim, to **EndpointIDShim** class.

9.1.10 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**.

9.1.11 DTN2/sqldefs – New directory

- Provided for storing prototype schema creation scripts.

9.1.12 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 DTN2 configuration file (typically) *dtn.conf* using *storage set odbc_schema_pre_creation*
- 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**.

9.1.13 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 DTN2 configuration file (typically) *dtn.conf* using *storage set odbc_schema_post_creation.*
- 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 9.1.12).

9.2 Oasys

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

9.2.1 Makefile

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

9.2.2 oasys/serialize – TypeShims.h

- Added static method **shim_length** to all shim classes to return the length of the serialized item that will be generated when the shimmed item is serialized if this is a fixed quantity or 0 if it is variable. Note that several of the classes in this header are broken and unused. They should have a **shim_length** method defined (correctly) if they are ever pressed into use (outlines have been inserted but will need checking when/if the classes are uncommented).

9.2.3 oasys/storage – BerkeleyDBStore.cc and BerkeleyDBStore.h

- Removed **raw_key** and **raw_data** (not used)
- Style correction: Changed names of routines **beginTransaction**, **endTransaction** and **getUnderlying** to **begin_transaction**, **end_transaction** and **get_underlying**.

9.2.4 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.

9.2.5 oasys/storage – DurableStore.h

- Add additional flag **DS_AUX_TABLE** to indicate that table is an auxiliary one.
- Add constants (macros) **KEY_LEN_MASK** and **KEY_LEN_SHIFT** used for incorporating the key length into the **flags** field passed to **get_table**.
- Add constants (macros) **KEY_VARBINARY_MAX** (value 255, the maximum length of a VARBINARY column in some SQL implementations) and **KEY_LEN_MAX** (251 – allowing for serialization overhead).
- 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.

9.2.6 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.
- Modify fall-back implementations of **begin_transaction** and **end_transaction** so that they generate debug level rather than warn level logging and always return **DS_OK** rather than **DS_ERR** on the assumption that storage initialisation routines will warn the user that transactions don't make sense for the chosen storage subsystem when **auto_commit** is false.
- Modify fall-back **get_underlying** routine to generate a debug level logging message rather then warn level message. Assume that users will test the return value and whinge if it is NULL.

9.2.7 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.

9.2.8 Oases/storage – FileSystemStore.cc and FileSystemStore.h

- Add (virtual) method **end_transaction**. Calls **sync_all** for **fd_cache_ b** (see Section 9.2.18 for effect of this).

9.2.9 oasys/storage – InternalKeyDurableTable.h

- Add templated **do_init_aux** function with same interface as **do_init**.

9.2.10 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.

- Add code to methods **do_init** and **do_init_aux** to call static method **shim_length** (see Sections 9.1.9 and 9.2.2) for templated **ShimType**. Use the result to set bits 16:23 in the **flags** word passed to **DurableStore::get_table**.

9.2.11 oasys/storage – MemoryStore.cc

- Style clean up – add separators between method definitions.

9.2.12 oasys/storage – MemoryStore.h

- Add comment noting **begin_transaction**, **end_transaction** and **get_underlying** are not implemented for memory storage option.

9.2.13 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 roll-back 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.
- Remove usage of **base_dbenv_**. Just use **dbenv_** which is now a structure rather than a pointer.
- Add code to implement MySQL connection keep alive timer:
    - Define private class **ODBCDBMySQL::KeepAliveTimer**.
    - Cancel timer in **ODBCDBMySQL** destructor.
    - Create timer from **ODBCDBMySQL::init** setting timer period from storage configuration variable **odbc_mysql_keep_alive_interval_**.
    - On timeout, call private method **ODBCDBMySQL::do_keepalive_transaction**. This method uses the **idle_hstmt_handle to execute a database command that retrieves the value of a suitable server variable and reads back the answer.**

9.2.14 oasys/storage – ODBCSQLite.cc and ODBCSQLite.h

- New class **ODBCDBSQLite** derived from **ODBCDBStore.**
- Comments explaining configuration of ODBC for SQLite.
- 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.
- Remove usage of **base_dbenv_**. Just use **dbenv_** which is now a structure rather than a pointer.

9.2.15 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.

9.2.16 oasys/storage – ODBCStore.cc and ODBCStore.h

These files (especially **ODBCStore.cc)** have undergone extensive modification to solve a number of issues and remove various redundant features. The set of methods is little changed but there have been a number of internal fixes to implement changes

- ◆ to change the SQL type of the key columns in all tables to either BINARY for fixed length keys or VARBINARY for variable length (typically string) keys.
- ◆ to ensure that any parameters bound to ODBC statement handles are unbound before the statement handle is reused in addition to releasing the result columns and the previously prepared SQL, and
- ◆ to ensure that the **SerializationLock** is actually in effect for the whole of the relevant method.

In detail the changes made were:

- Re-factor **ODBCDBStore** class as a base class for storage modules using ODBC to access a type of SQL database (see Sections 9.2.13 and 9.2.14). Remove **init** and **parse_odbc_ini** methods to derived classes.
- Add extra comments on use of ODBC and configuring it.
- Remove **__my_dbt** structure type (not needed).
- Add **hstmt_idle** to **ODBC_dbenv** structure for use with keep alive transaction.
- Remove **cur_table** form **ODBC_dbenv** structure.
- 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.]
- Replace the ODBCDBStore variable base_dbenv_ with plain dbenv_ and replace all instances of **dbenv_->** with **dbenv_**. Remove all references to **base_dbenv_**.

- 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 o**dbc_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.
- Replace the code snippet at the beginning of all methods that access the database to ensure that only one such routine is accessing the database at once with code that uses **ScopeLockIf(&serialization_lock_, serialize_all_)**. This extends the lock scope to the end of the method as was intended rather than terminating at the end of the if statement previously used. Additionally apply the scope to various extra routines that access the database (**begin_transaction, end_transaction**, **do_keepalive_transaction** and **del).**
- Record the k**ey_size_** passed into **get_table** in the **flags** word when creating the instance of **ODBCDBTable** for each of the database tables.
- Modify the various database access routines to use the appropriate key column type depending on the value of **key_size_** recorded for the table.
- Remove all the special case code inserted to handle the **globals** table (including the name definition) making the code non-specific to DTN2 as it was previously.
- Remove **RealKey** method.
- Wherever **SQLFreeStmt** is called with a second parameter of **SQL_CLOSE**, add an addition call to **SQLFreeStmt** with the same first parameter but with **SQL_RESET_PARAMS** as second parameter.
- 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 class) of **StoreDetail**. Also suppress the row insertion for auxiliary tables because it is carried out by a database trigger.
- Remove the retrieval of the value columns in the constructor of **ODBCDBTable::Iterator**. Only the keys are needed.
- Remove the **raw_key** and **raw_data** methods (not used).
- Remove all reference to and functionality for **DeadlockTimer**.

- Modify the logging statements in **print_error** from **log_debug** to **log_err** as the user needed to see the error messages during normal operation.
- Style correction: Changed names of routines and variables **beginTransaction**, **endTransaction** , **getUnderlying** and **serializeAll** to **begin_transaction**, **end_transaction**, **get_underlying** and **serialize_all**.

9.2.17 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. Default value is **false**.
- Add new string item **odbc_schema_pre_creation**. Set to the pathname of a file containing SQL commands to be actioned by an SQL database during database initialization before the main tables are created. Defaults to the empty string meaning no commands are executed.
- Add new string item **odbc_schema_post_creation**. Set to the pathname of a file containing SQL commands to be actioned by an SQL database during database initialization after the main tables are created. Defaults to the empty string meaning no commands are executed.
- Add new unsigned 16 bit item **odbc_mysql_keep_alive_interval_** to set the interval between MySQL connection keep alive transactions.

9.2.18 oasys/util – OpenFdCache.h

- Add method **sync_all** which calls fsync on all open file descriptors in the cache. Used by **FileSystemStore** to implement a form of **end_transaction**, flushing all data to persistent store.

10. 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.

10.1 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, sqlext.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).

10.2 SQL Database

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

10.2.1 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.

10.2.2 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.

10.3 Compiing 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**).

11. Configuration

.A number of configuration files have to be set up to successfully use SQL storage through ODBC with DTN2 and Oasys. Figure 1 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.

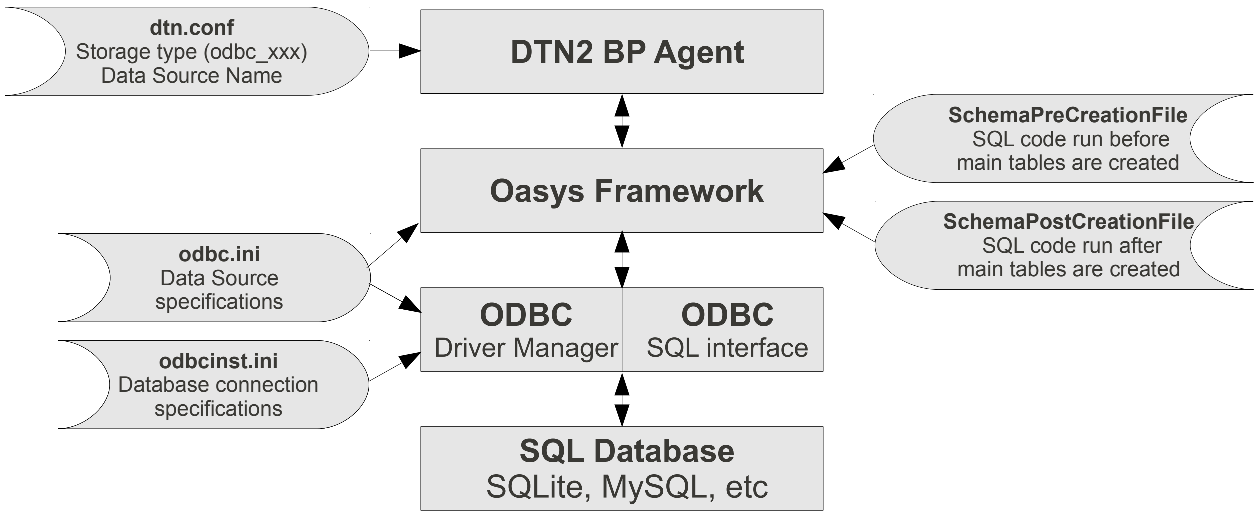

Figure 1: Configuration Files

11.1 Database Configuration

11.1.1 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 '\\' | 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.

11.1.2 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

```
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 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 10.2.2), 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 basic permissions for all tables in the DTN2 database:

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

```
Also allow the user to create tables in the database:
```

```
mysql> GRANT GRANT OPTION 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 tables have 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 tables named *META_DATA_TABLES*, *bundles_aux*, *bundles_del*, and *tidy_occasions* exist. Drop (i.e., delete) all four of these tables using the command:

```
mysql> DROP TABLE <table name>;
```

This shoukld leave an empty database ready for use by DTN2. Exit from **mysql** and proceed with ODBC configuration.

11.2 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 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 of 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.

11.2.1 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 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 **mysql)** 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 to grep the output for ERROR. Another snag is that **isql** has been 'improved' in version 2.3 of unixODBC.]

11.2.2 Header Sections of odbcinst.ini and odbc.ini

The **obdcinst.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.

11.2.3 ODBC Configuration for SQLite3

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

11.2.3.1 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.

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

11.2.3.2 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
```

11.2.4 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..

11.2.4.1 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 10.2.2. 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
```

11.2.4.2 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 11.1.2). The trace file settings are 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>
Option                  = 0
```

11.3 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.

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

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

or

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.

If required, specify the files to be used to add additional items to the database schema before and after creation of the main tables as follows:

```
storage set odbc_schema_pre_creation  <pathname>/DTN2/sqldefs/create_aux.sql
storage set odbc_schema_post_creation <pathname>/DTN2/sqldefs/create_trigger.sql
```

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

Revision History

Version	Date	Author	Comments
0.0 – 0.8	03-09/01/12	Elwyn Davies	Creation
0.9	10/01/12	Elwyn Davies	Documents version released as Oasys Changeset 2290
1.0	20/03/12	Elwyn Davies	Documents additional changes to remove special casing for globals table and make PRoPHET and Links tables work. Modifies keys to BINARY or VARBINARY types.
1.1	03/04/12	Elwyn Davies	Final revisions and changes to location of schema creation configuration parameters.
1.2	15/02/13	Elwyn Davies	Minor correction to MySQL installation instructions – adding GRANT of GRANT OPTION setup.
1.3	14/06/13	Elwyn Davies	Added header page for publication as TCD technical report