Repository - Object Schema

Mar 2008
David Barrett-Lennard

Introduction

The document concerns the representation within a ceda repository of the object model schema information.

Identifying a model

A model is uniquely identified by a *fully qualified name* - a ModelName.

It is assumed these are globally unique. Therefore organisations should name their models under a unique namespace to avoid name clashes.

ModelIndex values

The Repository persists a local bijective mapping between ModelIndex and ModelName as follows

```
typedef int32 ModelIndex;

struct ModelIndexMap
{
    vector<ModelName> indexToName;
    map<ModelName,ModelIndex> nameToIndex;
};
```

This allows ModelIndex values to be used instead of ModelNames in the model schema representations.

New models can be added to a repository. Existing models can never be removed. The assigned ModelIndex for a given ModelName never changes for a given repository.

FieldIndex values

We assume there can only be a linear upgrade path for a given model schema.

A field cannot change its type. Once a field has been dropped it cannot be added again.

All the fields of a given model are identified (across all schema) by a FieldIndex. This is a zero based index assigned to the fields in the unique order in which they have been added during the course of the linear upgrade path of schema changes.

The repository never deletes a field.

Field types in the repository

```
// assignment semantics on fixed size opaque types
fixedOpaque :-
        RBC_OPAQUE1 |
        RBC_OPAQUE2 |
        RBC_OPAQUE4 |
        RBC_OPAQUE8 |
```

```
        RBC_OPAQUE16 |
        RBC_OPAQUEn, n32

// assignment on variable sized opaque blobs
varOpaque :-
        RBC_OPAQUEv,   // variable sized opaque blobs

opaque :-
        fixedOpaque |
        varOpaque

// create/delete semantics on fixed size opaque immutable elements.
// Move not supported
vectorOfOpaque :-
        RBC_VECTOR fixedOpaque

// Modulo addition semantics
deltaInt :-
        RBC_DELTA_INT8 |
        RBC_DELTA_INT16 |
        RBC_DELTA_INT32 |
        RBC_DELTA_INT64

// Takes maximum over all assignments
maxiInt :-
        RBC_MAXI_INT8 |
        RBC_MAXI_INT16 |
        RBC_MAXI_INT32 |
        RBC_MAXI_INT64 |
        RBC_MAXI_UINT8 |
        RBC_MAXI_UINT16 |
        RBC_MAXI_UINT32 |
        RBC_MAXI_UINT64 |
        RBC_MAXI_FLOAT32 |
        RBC_MAXI_FLOAT64

// create/delete/move semantics.  All prefs are unique
setOfPrefs :-
        RBC_VECTOR_PREF |      // ordered set of prefs
        RBC_SET_PREF,         // unordered set of prefs

map :-
        RBC_MAP opaque type

set :-
        RBC_SET opaque

bag :-
        RBC_BAG opaque

array :-
        RBC_ARRAY n32 type

// Model referenced using a 32 bit ModelIndex
model :-
        RBC_MODEL n32

type :-
        opaque |
        vectorOfOpaque |
        maxiInt |
        deltaInt |
        setOfPrefs |
        map |
        array |
        model

fieldtype :-
        type
```

Assignment on opaque fields
Assignment on entire models is not supported.

Assignment of fixed as well as variable sized opaque fields is supported.

Vector of opaque elements
Elements are fixed size, opaque and immutable. Operations can create or delete elements. Moves are not supported.

Vector of prefs
The repository records a mapping between Osn and Osid to allow 64 bit OIDs to be mapped to 160 bit GOIDs.

Prefs are not typed. Prefs must appear as either vector<pref> or set<pref> within a model. Prefs are never NULL. All prefs are unique. There is no need to model a bag of prefs. Objects form trees using prefs. Operations can create/delete or move elements. There are no constraints on source or destination for moves. Move operations ensure conservation of matter.

The repository uses the prefs to

- cluster related data together on disk.
- send multiple objects at a time to a client to avoid roundtrip delays

vector<pref> members cannot be accessed in model constructors, or in evolve code.

vector<pref> members can't be dropped from a model.

Sets and Bags
Elements are opaque and immutable. Elements may be fixed size or variable size blobs.

Maps
The keys of a map are opaque and immutable. We support fixed size as well as variable sized opaque keys.

The values of a map are editable.

The repository never deletes map entries.

todo: Do we support a special marker for allowing clients to delete map entries?

Arrays
The size of an array is immutable. Elements of arrays are editable.

Model constructors

All models have a default constructor. Within a ctor, all fields can be initialised except vector<pref> or set<pref> fields. This does not invoke the write barriers (or log changes in a UniversalOp).

Evolve Code
Only new added fields in the schema change can be initialised. vector<pref> and set<pref> fields cannot be initialised. This does not invoke the write barriers (or log changes in a UniversalOp).

Serialising RSN of objects to represent schema
Definition: Let P,C be models. We say C is an *embedded model* of P if the serialised state of an instance of P includes the serialised state of an instance of C.

Definition: Given datasource D, T = Target(D) if T contains the cpp file which contains the generated reflection code and implementation of the Serialise() method for D.

Definition: Given model M, T = Target(M) if T contains the cpp file which contains the generated reflection code and implementation of the Serialise() method for M.

Definition: Given datasource D, model(D) is the root model of D

When a given datasource D is serialised to an archive the following is written

1. CID(D), the 32 bit class identifier of D. This allows for dynamic creation of D.
2. CurrentRSN(Target(D))
3. model(D)

By identifying a particular release of Target(D) the format of all embedded models is determined, allowing the framework to correctly and unambiguously deserialise model(D). Note therefore that embedded models do not require their own schema numbers to be written to the archive.

During deserialisation of datasource D, r = CurrentRSN(Target(D)) is read from the archive and the pointer SubTargetRsnMap(Target(D))[r] is assigned to a special member

```
int* m_subTgtRsnArray;
```

in the archive.

When an embedded model M is deserialised, it can retrieve the pointer subTargetRsnMap(Target(D))[r] from the m_subTgtRsnArray member in the archive. It also can read t = tin(Target(M)) directly from the TIN global variable within Target(M).

This allows it to very quickly calculate the RSN relevant to target(M). ie

```
r' = ar.m_subTgtRsnArray[s_tin]
   = SubTargetRsnMap(Target(D))[r][tin(Target(M))]
```

= SubTargetRsn(Target(D),r,Target(M))

This in turn allows M to be deserialised correctly.

Maxi fields
Consider an assignable data model field of type T on which a total order is defined. We can support assignment operations that yield the maximum across all assignments that have been applied to the field. Here is the corresponding operation:

```
template <class T>
class MaxiOp
{
        void Do()
        {
                if (field < value) field = value;
        }
        T& field;
        T value;
}
```

These operations commute, so there is nothing to do under IT/ET.

In the repository these assignments form a linked list in decreasing order by value. To calculate the current value for a given vector time v we scan left to right until we find the first assignment in X(v) (ie with (s,t) satisfying t < v(s)).

Recording of RSN in the repository
For each object (= data source) in the repository, the RSN is recorded using a maxi field. This makes it possible to calculate the RSN for any given valid vector time (including a vector time that represents the merge of branches). This will give the maximum RSN across concurrent branches.

Lazy loading of objects by a client
A client can lazily load objects from the repository (by specifying an OID and vector time). The repository sends the object in the schema associated with the RSN recorded in a maxi field, calculated for the given vector time. The RSN is sent as part of the object format allowing the Serialise() function (generated by xcpp.exe) to be used on the client to deserialise the object.

Justifications

In the following sections we justify the various design decisions. Separating out the "why" from the "what" makes it easier to understand the exposition.

Repository needs to be concerned with objects

Consider that the repository only deals with fields that are identified by some key, and has no concern with the concept of objects. The problem is that the repository is not able to cluster the data on disk effectively. Furthermore, a layer on top would still be needed that understands objects to allow for retrieval of an entire object from the repository.

Object schema

We need to efficiently store an object schema. The schema specifies all the fields in the object.

The repository has a requirement of never deleting information, and in particular object schema.

There are some important differences to the normal ceda reflection system

- The object schema can be serialised
- Object behaviour is irrelevant. There is no need for method reflection.
- There is no need for fields to have a name, display string or other meta-data
- There is no C++ class associated with the object schema! Therefore field offsets in the C++ object are not required

Note that the type of a field needs to include the style of operations on that field that are supported. Eg a string could have insert/delete or else assignment semantics. An int could have offset or else assignment semantics. This difference is crucial to the repository.

An indexed array of elements can be implemented in C++ in different ways. Eg it can take the form of std::string, std::vector, std::deque or std::list. These differences are all irrelevant to the repository! So the object schema is the same for them all. In that sense the repository provides some degree of *physical data independence*.

The repository doesn't care about the following

- Whether a vector is implemented using std::string, std::vector, std::deque, std::list etc
- Whether a map is implemented using std::map or a hash map etc.
- Where an int is signed or unsigned.

Therefore these variations aren't represented in the object schema.

Granularity of assignment

C++ can allow assignment of entire structs. However this is converted into a set of assignments on the individual fields. This helps for the following...

- As far as the C++ programmer is concerned, we support assignment of individual elements as well entire models. The merging of assignments works correctly.
- This keeps the repository simple and avoids unnecessary coupling with applications.

Vector of prefs

Prefs cannot appear on their own as simple assignable fields because of the need for conservation of matter. Also prefs need to account for OIDhigh translation. Finally prefs provide information to the repository to allow for clustering of data and also to allow for download of multiple objects at a time to avoid roundtrip delays which would be a problem with individual requests for objects. Therefore we see that prefs are treated specially.

The vector<pref> is a special type, with a single byte code! It supports move semantics with any other vector<pref>.

Vector fields
For a field of type vector<T> we assume that the elements T are immutable. In fact the repository doesn't need to know anything about T apart from its size in bytes. Therefore T is a *fixed size opaque type*.

Furthermore we don't support move semantics on a vector<T> where T is an opaque type. This avoids a lot of potential inefficiency in the repository.

Opaque assignable fields
Given that the repository doesn't allow fields to change type, the repository doesn't need to know anything about assignable types apart from the size. Therefore assignable types are "opaque".

Field types in the repository

Notes

- Opaque means that client is responsible for its interpretation.
- The keys of a map are immutable. The values of a map are editable.
- It is important to allow the key of a map to be a variable sized opaque blob (eg to allow a client to key on a string).
- The size of an array is immutable
- The elements of vectors, sets or bags are immutable
- Prefs cannot be NULL. All prefs are unique. The repository allows prefs to be moved around arbitrarily. There is no need to model a bag of prefs.

For better compression of the byte code we could for example define the following

```
vectorOfOpaque :-
        RBC_VECTOR_OPAQUE1 |
        RBC_VECTOR_OPAQUE2 |
        RBC_VECTOR_OPAQUE4 |
        RBC_VECTOR_OPAQUE8 |
        RBC_VECTOR_OPAQUE16 |
        RBC_VECTOR_OPAQUEn, n32

map :-
        RBC_MAP_OPAQUE1 type |
        RBC_MAP_OPAQUE2 type |
        RBC_MAP_OPAQUE4 type |
        RBC_MAP_OPAQUE8 type |
        RBC_MAP_OPAQUE16 type |
```

```
RBC_MAP_OPAQUEn, n32 type |
RBC_MAP_OPAQUEv type |
```

Deltas on numeric fields
Delta operations on floats don't commute because of underflow or overflow. That is, sometimes it is not the case that (a+b)+c = (a+c)+b. Eg for x = 1.0E-100, (-1+x)+1 = 0 whereas (-1+1)+x = x.

Therefore we don't support delta operations on floating point fields. ie delta operations are only available for INT8, INT16, INT32 and INT64.

Higher level concept of numeric field
In order to promote physical independence, consider that the repository has a higher level concept of "numeric field", allowing it to accommodate changes in type of a field like int32 to int64, or float32 to float64.

For assignment semantics we could support schema changes across all types. However, we need to distinguish between unsigned and signed integers. This is needed when type conversions are applied by the repository.

Schema evolution
We allow any model to get new fields or to drop fields over time.

- In subsequent schema a field may be marked as "dropped". This affects what is received by a client, but doesn't affect the way the repository continues to record all changes to all fields across all branches. Dropping a field doesn't affect the way FieldIndex values are allocated to fields. In a sense the repository never regards a field as being dropped. It is only relevant at the point where we serialise an object, or send operations.

- New fields to be added to the schema. It is assumed that the schema of an object is centrally managed so we don't need to support different versions of object schema. It is easy to read out any version of an object in any required schema!

Fields cannot change type
A field can't change type. Instead it is necessary to deprecate the old field and add a new field.

When we want to change the type of a field, we have to introduce a new field. When reading an object from the repository, we may find that the field is not yet present. This makes the client assign a new value to the field. This can be calculated from the old value. The repository is none the wiser!

This approach keeps the repository simple and closed for change. It also allows for quite complex schema changes. It is clear when a schema change is required. A schema change is triggered at most once along a different branch. When merging branches, all

sites agree on the value of the field according to the convergence requirement of operational transform.

The semantic meaning of a field should not change over time. ie a field must have a consistent semantic meaning. Therefore it is always reasonable to accumulate changes to the field despite schema changes to the object.

The repository will not allow a field to change it type. This simplifies the repository. Consider that a field needs to change from float32 to float64. This is achieved by adding a new and independent field. When a schema evolution is triggered on a client, the float64 value can be calculated from the float32 value in the deserialise function. It remains to correctly set the new values of the field in the subsequent check-in into the repository.

Linear upgrade path for a given model schema
We assume there can only be a linear upgrade path for a given model schema.

Consider that company A develops a software component and it exists in many repositories around the world, then the two directors have a fight over how the component will evolve and they split the company into B and C. Only one of B,C is allowed to continue using the same model name for the component! The other company needs to use a model name. Otherwise there are some difficult problems. For example, a user could alternate between software upgrades from B and C on a client machine. We can't expect that the working set object can be read from disk correctly.

FieldIndex values
Fields of a model are identified by a 32 bit FieldIndex. This is a zero based index assigned to the fields in the unique order in which they have been added during the course of the linear upgrade path of schema changes.

The repository will never delete a field because it is not allowed to lose information. Therefore it assumes that new fields can only be appended to the end of the model, and "dropped" fields are only marked as deprecated. This is compatible with the assumption that an object schema follows a linear upgrade path - there is no chance that different fields get assigned the same field index.

Sending the RSN of objects over the wire
We support lazy loading of objects by a client from a repository. A client cannot expect objects in the repository to have the latest schema. Furthermore the repository isn't able to schema evolve an object because it cannot run application specific code. Therefore the repository must be able to send objects in an old schema.

We see therefore that different versions of the object schema are sent over the wire in a single session between client and server. Hence it makes sense for the schema information to be stored as part of the object. Note in any case that we want to use the

standard deserialise function on a client for objects received from the repository. This of course already performs schema evolution for us.

[todo: how does schema evolution of fields make its way back to the repository?]

An object in the repository is associated with a data source model, not a pure model. When an object is sent over the wire from repository to client we only store a single number to indicate schema information for the entire object. This is a *Release Sequence Number* (RSN) and identifies the release of the DLL/EXE target that defines the data source model. With the ability to map RSNs between targets, this uniquely defines the schema for all embedded pure models serialised within the data source.

When a client checks out an object from the repository, it may receive a very old schema. This triggers a schema evolution on the client. Note that the repository doesn't know how to evolve schema. It can only send the old version of the object and let the client do the upgrade.

Later, the same client may check-in changes to the object, bringing it up to a different schema.

Model constructors
A datasource must have a default constructor in order to support dynamic creation.

A pure model must have a default constructor so that

- a dropped model within a parent model can be declared on the frame in the Serialise() function
- vector<> can be used

There are some reasons to be hesitant about the need for non-trivial constructors on models:

- By definition the data members of a data model are public, so it shouldn't be necessary to use special constructor syntax to initialise their values.
- For performance of deserialisation, default constructors should be very fast.
- The model in a datasource is typically an indirect base class and it is not possible for the final datasource class to directly call an indirect base constructor because of the mixin chain. Furthermore we can't expect the template mixins to pass all the constructors through. We conclude that the model in a datasource must have a default constructor and there is no point having any other constructors at all!
- It is assumed we do need movement semantics with prefs. Therefore operations must be able to track the relationship between extractions and insertions, and that can't be achieved by only looking at the end result (when the CedaLock commits). This suggests that data model constructors cannot execute moves on vector<pref> members.

We allow $$() {..} to be used within a model for a constructor. This allows for initialisation of the current members. Note that dropped members cannot be initialised.

A constructor of a model (whether a pure model or a datasource) will initialise the values of members without invoking read/write barriers. Note that this is a bit tricky for a data source because the member names are mangled with '_' characters.

Initialisation of members is rather subtle. We have to consider what it means for the repository.

A universal operation includes the instructions for creating new objects. This is merely recorded as a set of OIDs. The initial state of the fields is recorded in the normal way as part of the universal operation. This simplifies the schema for universal operations and also the implementation of the repository.

A repository has no way of knowing how to initialise fields or how to evolve fields. Rather, the repository simply stores what it's told to store! It will never guess the value of a field that it has never stored. In fact the repository makes no attempt at making sense of what a field means. For example the repository doesn't care about the difference between a float32 and an int32 - both are treated as an opaque 32 bit value.

During a CSpace exclusive write lock a universal operation is used to record all the changes. Consider that an insert is performed on a vector<pref >. The framework can assign an OID and record the OID in the universal operation. During the write lock, subsequent changes to fields on the object are ignored! [A flag in the IPersistable can be used] Instead, the framework waits until the CSpace lock is released (and the "dust has settled") before it records the initial states of the object's fields as deltas in the universal operation. This is achieved using the reflection information available on the data source.

Note that different machines can't initialise objects with the same identity. A machine can only receive (in a single indivisible operation) an object created by a different machine together with its initial state applied as deltas.

Check-in of evolve code member initialisation

During deserialisation, the evolve code initialises new members of a datasource without invoking the write barriers. Therefore the evolve code is not directly recorded by the universal operation. In fact deserialisation may happen during a shared read lock on a CSpace, where there isn't even a universal operation available to record changes. Note as well that Serialise() functions are predefined on pure models and are unable to invoke write barriers on the containing data source.

When a new assignable field is added to the schema, and it is initialised in terms of previous values then no operation is generated! Instead the initialisation happens silently during deserialisation from an old schema.

A client uses a UniversalOp to accumulate changes in preparation for the next check-in to the repository. At the end of an exclusive write lock the system has an opportunity to

record (in the UniversalOp) the initial values of new fields in objects as a result of prior schema evolution.

It is proposed that objects that have been evolved are flagged (using a flag in IPersistable) to disable subsequent operation generation. NO this won't work!

Note that it is not possible for schema evolution to cause matter transfer, because there is no way that a universal operation will be able to record the movement of state correctly.

Construction and evolution : vector<pref> members
prefs must appear as either vector<pref> or set<pref> within a model. Ownership semantics is assumed and conservation of matter is enforced. As far as the repository is concerned there is no limitation on how objects can be moved around.

It is not permissible to initialise a vector<pref> member with one or more newly created objects in a default constructor of a data source because default constructors are suppose to very fast (and a heap allocation is relatively expensive), and it seems silly to have subsequent deserialisation cause the objects to be garbage collected.

It is not permissible for moves to be performed within a default constructor because a default constructor may be called during a shared read lock so there is no universal operation that is recording the transfer of matter. Similarly in the evolve code.

vector<pref> members can't be dropped from a model. If a vector<pref> is added then it can't be initialised during schema evolution.

Update from old schema
It is possible for a client to update from a branch with an older schema.

Only the fields that are relevant to the schema of the client need to be sent. Because the semantic meaning of a field should be fixed, these can be merged without any concern for schema.

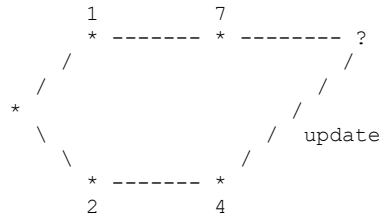
Recording schema number in the repository
The repository must support storage of schemaId. The schemaId has to be versioned like any other field within a struct. ie we can calculate schemaId for a given vector time. So different branches can have different schemaIds.

A schemaId can never decrease along a branch in the repository. When a client performs a check-out or update, it should fail if the repository is more up-to-date than the client. A message box should appear on the client saying that a software upgrade is required.

When a struct contains a schemaId, the schemaId is always stored as part of the struct on disk or when it is sent over the wire. The repository treats the schemaId as an assignable field so that it can keep track of the version of the object schema on different branches.

We assume that a client can never revert to an earlier version of a component dll - causing the schema to revert. Therefore the schemaId may only increase along a branch of the repository.

Consider two branches that get merged, yet have different schemaIds



It is possible that the lower branch dominates the upper branch (depending on site id comparisons). In that case, the schemaId from the lower branch may "win", causing the schema along the top branch to revert! This seems highly undesirable.

It doesn't help to force the lower branch to first evolve to schema 7 (before the merge). For example, suppose that from schema 4 to 7 a field was added that was calculated from a field in 4. The new fields in the lower branch will still "win" over the top branch.

Conclusions

- SchemaId must behave a little differently that assignment, because when we merge we want the *maximum* over all schema.
- We allow the merge to be done under the assumption that each field can be merged independently in the normal way without regard for schema evolution. This assumes there are no class invariants between field members.

Proposal

- Repository supports structs that optionally contain a schemaId.
- When a struct has a schemaId, the repository will ensure that the schema of an object can be found for any given vector time. This is the maximum over all branches. [TODO: This is an interesting semantic for operational transform]

When the struct is sent down the wire, the format implicitly begins with the schemaId.

* The repository keeps track of
  * the set of schema that it knows about
  * the set of fields that are present for each schema.

* The client tells the repository about the schema it is using. If necessary the repository will receive and store the new schema.

* The repository will not allow the schema to revert along a branch

* When updating, the repository will provide the combined result of all the fields, including the schemaId as the maximum over all branches.

Arrays versus Strings

An Array<T> assumes that the elements are editable. Therefore it is forced to store the elements in a form that allows for different versions etc. An Array<char> would be a very expensive way to represent a std::string. Therefore String<T> is an alternative data model type that assumes that the elements are immutable.

Storage of object schema

A naive approach is to defines an abstract base class for a type, and form assemblies using composite types. For example, a StructType could store a vector of pointers to the types for its child members. Each type is allocated on the heap so unfortunately this approach is rather expensive. Furthermore it is not clear how to efficiently serialise the schema to disk.

This implementation uses a rather innovative way to store object schema, that avoids so many heap allocations. It avoids the direct use of polymorphism.

A byte code is used to represent a type. The byte code needs to be "run" in order to read the type.

Let each struct be identified by a 32 bit StructIndex, and a 128 bit StructGuid. Let the byte code be stored using deque<BYTE>.

```
class Struct
{
    StructGuid m_id;
    std::deque<BYTE> m_byteCode;
};

class ReposSchema
{
private:
    std::map<StructGuid, StructIndex> m_structGuidMap;
    std::deque<StructIndex, Struct> m_structs;
};
```

The byte code to represent a struct is assumed to simply list the types of its members. The leaf types only take up one byte each. The composite types are as follows

1. FT_STRUCT --> next 4 bytes form a StructIndex
2. FT_MAP --> Next two types form the key and value types respectively
3. FT_ARRAY, FT_SET, FT_BAG --> Next type is the element type

IDEA: Use reverse polish. ie as we run the byte code we use a stack of types. the FT_ARRAY instruction means pop the element type and push array<element>. The FT_MAP instruction means pop the key and value types and push the map<key,value> type.

Note that running the byte code is extremely fast, and arguably as efficient as recursing through any other data structure.

Note that the repository schema is additive only. Therefore it is reasonable to use a single large deque<BYTE> to store all the byte codes. A special byte code us used to terminate the members of the struct. There byte codes can be appended to this single big deque. The byte code for the sequence begins at a specified index position in the deque<BYTE>.

This leads to the following implementation:-

```
class Struct
{
    StructGuid m_id;
    int m_byteCodeStartPos;
};

class ReposSchema
{
private:
    std::map<StructGuid, StructIndex> m_structGuidMap;
    std::deque<StructIndex, Struct> m_structs;
    std::deque<BYTE> m_byteCode;
};
```

References

[1]	David Barrett-Lennard. Vector Time. Feb 2008