Collaboration and Repositories

Feb 2008
David Barrett-Lennard

Introduction

This document describes the approach for how to implement the repositories and interactive collaboration in ceda.

The following are relevant to this discussion

- Synchronous versus asynchronous prefs
- prefs and the dependency system
- API on the repositories
- Partitions
- The unit of download
- Protocol for communication with a repository
- Protocol for interactive collaboration
- Support for lazy download of objects over the wire
- How to subscribe to a subset of all operations on the network

Orthogonality of branches and data

It would be preferable to support complete orthogonality between branches and data, meaning that there are no restrictions on what data may be edited on a given branch.

A vector time specifies a particular set of operations that have been executed, and hence a well defined set of objects that have been created and edited up to that time. The end of a branch has an associated vector time, but we don't want to impose any restrictions on the merging of branches.

Similarly, during a collaborative session we don't want to impose any constraints on the data that can be edited as part of the session.

Note that this can be compared to ceda1 where operations and branches were localised to partitions and contexts. In ceda2 we want to avoid the whole idea of associating operations to the data being edited.

It is not clear to what extent this principle can be upheld.

Partitions

It is assumed that a given persistent data source objects belongs to exactly one partition. Objects cannot be moved between partitions. Therefore each partition can be treated completely independently - in terms of the repository and interactive sessions.

Each partition supports its own independent control over various policies such as:

- The connection topology
- Ports to listen on
- Server(s) to connect to

- Whether to push/pull all objects, or to download objects on demand
- Security policies

We assume all data in a partition forms a course level tree, according to the support for tree structures defined in cxPersistStore.

A single partition may become very large (eg many gigabytes). Therefore we must support download on demand.

Given the coarse granularity of partitions, the admin overheads are reduced. Most of the time, *all* data will simply go into one partition.

Synchronous versus asynchronous prefs
We want to support asynchronous prefs. This means that the application need to be written to interpret a null pref as meaning that it is being loaded from disk or over the wire asynchronously.

For example, a folder could store a vector<pref<T> > for its children, and display a special icon for the children that haven't been brought into memory yet. This could allow the application to be very responsive.

A GUI that tests the pref against null ends up creating a dependency link and that in turn causes the binding to take place asynchronously. When the binding is complete the dependency graph system marks the relevant nodes as dirty so that the GUI can be redrawn.

This approach is well suited to the multi-res image; as tiles are loaded into memory the image will "refine".

Alternatively at other times we want to support synchronous prefs. For example, a web server using a thread pool to handle incoming requests. A thread handling a request will want to synchronously bind prefs.

There are two ways we could support this distinction

1. Control the policy for binding a pref<T>, by using a pointer to a bind function in thread local storage.
2. Use alternative versions of the pref<T> class.

Do we need to distinguish two types of prefs?
In ceda1 we sometimes used prefs that could be null. For example, a solid backdrop document used a pref to its child that could be null.

It is important to distinguish two types of null!

1. The pref is truly null (both m_ptr and m_oid) and we know that irrespective of the network

2. The pref is not bound (m_ptr is null, but m_oid is not).

In the latter case, it is always conceivable for a worker thread thread to block until it becomes bound, or for the dependency graph to do what it needs to cause the pref to successfully bind asynchronously.

Conclusion: there is only one type of pref required. We will set the binding policy using a pointer to a function in thread local storage.

Unit of download
In ceda1 there was a concept of a context that defined the unit of download. In ceda2 we have no concept of partitions. Therefore, we instead have a concept of having the send build up a list of objects that are sent all at once. Policies can control the size of the cluster.

Note that a repository can only send a cluster of objects if it understands how to deal with OIDs specially (rather than just opaque, assignable values). This maybe needed anyway so it can deal with OSIDs/OSNs.

Downloading objects on demand during a collaboration
A client could potentially download an object from a repository or another client. The latter may not make sense however, because a remote client won't be able to provide an appropriate version of the object.

Conclusion: We only support download of objects from a repository. This is based on going to the repository with a *base vector time*.

This suggests that we should implement the repository first!

Creation of new objects during a collaboration
As part of an operation we will support the creation of new objects. The serialised state of these new objects are serialised as part of the operation. We need to make sure embedded OIDs are handled correctly by the Osid conversion map when an operation is deserialised over the wire.

Navigating to the associated partition
A single process should support multiple partitions, and therefore multiple collaborative sessions that are all taking place at the same time - perhaps in different windows. Therefore we can't use thread local storage for the main thread to set a single partition that it is associated with.

Now objects store a pointer to their containing CSpace. Perhaps we could bind to a particular partition if we assume a simple relationship between CSpaces and partitions. No, actually we expect a single GUI to be able to show objects from different partitions. Therefore CSpaces can be more coarsely grained than partitions.

It is proposed that we avoid any such assumptions and simply assume the thread has a partition set in thread local storage, and leave it up to the application for how to set this correctly. In practice there are various options for setting this. For example, during the recurse down the scenegraph, a partition object could set this as ambient state. Note therefore that it has an opportunity to intercept mouse events.

Repository

Repository
It should be possible for an application to easily create a repository and directly call synchronous methods to perform checkins, updates, create branches, retrieve diffs. The support for IPC is in a separate layer. This makes it easier to unit test.

ReposGraph

ResposField
Represents the fields that the repository supports. Initially we will support the following types of fields

- Assignable fields
- Vector<T> without move semantics
- Vector<T> with move semantics

IPC for Repository

ReposMsgProtocol
Defines the IPC protocol between repository and client. Preferably this protocol is fully asynchronous. Not so clear because functions on the repository tend to have error codes!

ReposServer
Listens on a port for connections form clients. Uses the ReposMsgProtocol to support communication between client and repository.

ReposClient
A ReposClient represents the state used by a site in order to connect as a client to a repository. The following is the persistent state

- The base vector time
- The name of the current branch being worked on
- The siteid for the current branch
- The server/port to connect to for the associated repository
- Pointer to the current ResposClientSession (if any)

Note: coedit uses a map from branch name to siteid, to allow a site to have distinct siteids on each branch. This is important!

ReposClientSession

Operations on ReposClient

- Update() : retrieve vector time v2 for the current branch from the repository. UpdateWithVectorTime(v2 union `m_baseVectorTime`).

- UpdateWithVectorTime(v2) : retrieve o = diff from `m_baseVectorTime` to v2. DualIT entire HB with o. Execute transformed o. Set `m_baseVectorTime` = v2.

- Checkin() : Merge entire HB into a single operation. Checkin operation, for specified branch and base vector time.

Interactive collaboration

Site versus Partition
We can probably eliminate the class 'Site' and instead rename it 'Partition'.

A Partition is a persistent object. Each Partition persists the following state

1. The SiteId - a 128 bit GUID. It is assumed SiteIds are unique and there is a total ordering. Question: should we use strings for siteids?
2. The UserId.
3. The GSN to be assigned to the next locally generated operation, for the collaboration.
4. A History buffer. This records operations in a PDeque, and hv - the vector time associated with all operations in the history buffer
5. The local port to listen on as a server for an interactive collaboration
6. The server/port to connect to for an interactive collaboration

A partition has the following transient state
1. The set of sessions
2. The server end point for creating interactive collaborations
3. The client end point for creating interactive collaborations
4. The client end point to connect to the repository

Session
A Session hosts two threads. One to pump incoming messages and another to send outgoing messages.

A session records a suffix - a list of operations used to transform incoming operations.