Configuration Management

Sep 1 2005
David Barrett-Lennard

Abstract
This paper discusses options for configuration management in ceda

[Comment: Feb 2008. Most of this document is no longer relevant because we have chosen to use the approach described towards the end of this document - where a repository basically stores the effects document as a total ordered sequence of intervals]

Introduction

In this paper Configuration management relates to the management of versions, branches and tags of an OODB which uses operational transform as the basis for storing deltas between versions.

We will start by using terminology and concepts used with optimistic version control systems like CVS and Subversion. For example

- The *repository* refers to a storage system of a number of active branches, and tagged versions. The repository understands what users have made what check ins (and when)
- The *working set* is associated with a set of objects that have been *checked out* from the repository by a user. A user can directly edit his working set.
- The user can manually request an *update* to the working set from the repository.
- The user can manually request a *check-in* of the working set into the repository.

The working set is associated with

- Storing the current document state for each object (with given OID). As far as the working set is concerned, there is no concept of multiple versions of an object.
- Operations form a linear history
- User edits generate fine grained operations.
- Operations need to undergo log compression.
- Collaborative sessions are supported. This involves automatic upload or download of operations made to working sets on computers that comprise the collaboration.

By contrast Configuration management is associated with

- Keeping track of multiple versions of objects on all the active branches
- Storing each check in as a single operation
- Tagging a branch, associated with a vector time
- Supporting (atomic) update and check-in from a client

Note that lossy compression of check-ins isn't appropriate for configuration management. Updates and check-ins are manual processes, whereas real-time collaboration involves automatic interchange of operations.

These concepts have very little in common, and therefore it is proposed that we avoid blurring the distinction between them. Instead we embrace their differences!

Requirements for configuration management

Firstly, let's assume we have a single repository, and many clients that store working sets. Each client process can only store a *single working set* (because within the process an OID must map to a single object).

The repository is not interested in allowing objects to be viewed or edited (or in fact to have any behavior at all). This protects the repository - because there is no risk that it can run some malicious script or code. Instead it is only interested in storing the data. These requirements are quite distinct from a working set, where objects have behavior.

Furthermore the repository is interested in using copy on write so it can efficiently branch and tag. Clearly the repository will need to store objects identified by their OID in order to store separate versions of objects. However it will need to use index structures to relate a logical OID to a physical OID used to store a particular rendition of an object.

The repository needs to store the graph relating to all the branches and tags. Note that branches can split from a given node then merge back into a node. So we are dealing with a graph, not simply a tree structure.

Interesting idea: Universal ops support linear time compression. When an object is created by an operation then subsequently modified, then under log compression we simply initialise the object with a different initial state. Therefore a universal op from time zero actually provides renditions of all the objects! Therefore is it possible that the configuration management system has no need to explicitly store the current versions of objects. Instead that is implicit in a universal operation that begins from time zero.

The attic, or storing renditions of objects can be seen as merely a caching mechanism. It is certain that the repository should use universal operations to store deltas for check ins, and these allow it to reproduce the rendition of an object at any time.

Subversion uses a global tag number over the whole repository. This seems like a good approach.

Proposal: As a first implementation of a repository, we could disregard the requirements for performance by avoiding any caching. ie there is no attic. However we probably want to store the active rendition at the end of each active branch.

Copy on write: This can be achieved using an envelope for a given object. The envelope manages each version of the object. Hmmm but don't want to be too concerned with caching renditions of objects!

Storing of the version configuration information: Let this be done independently for each context. This is a graph composed from (universal) operations, (linear) chains, branches and nodes.

- A linear sequence of Operations can be stored in a *chain* using a linked list of pages of operations.
- A chain points at its parent node and child node
- A node points at the set of incoming chains and the set of outgoing chains.

A node can be tagged. A leaf node can be associated with an active branch.

Perhaps a chain is nothing more than a branch, and we don't need separate terminology. It depends on whether it is useful for a single branch to be composed from multiple chains.

On a linear HB we store hv (the vector time corresponding to the end of the HB). This is redundant but useful information. Perhaps the repository should cache the vector time at every *node*. In particular this will be at the end of every active branch, and at every tag.

Functions required on the repository:-

- Check in. This simply involves receipt of a single universal op (for each context). This is added to the chain associated with the (active) branch on which the check-in is performed. The system will ensure that the client was up to date so there is no need to transform the provided operation.
- Update: The relevant operations are sent to the client
- Branch: Create a node in order to define a new active branch
- Merge branch. Merge two branches into a single node.
- Tag: Add a node on the end of an active branch in order to tag it.
- Check-out: We send renditions of objects at the end of the requested branch, or for the requested tag. We can't assume we independently render copies of objects on each branch or else branching would be expensive. However we also can't expect to calculate the rendition by compressing the entire log.
- Send diffs: Create a universal op that represents the diffs between two given vector times. All operations between these two vector times need to be compressed. If necessary adjacent swap is needed to reorder some operations.

It is interesting to think of document state as purely a question of caching information. Note that a form of caching may be to compress chains of operations into a single universal operation. This may be more useful than storing renditions which are really diffs since time zero.

Note that from the point of view of the above data structure to represent a graph of branches and nodes, tagging and branching will be instantaneous in practise.

Proposal: Is there some sense in which we can

1. replicate the repository
2. allow for global agreement of tags and branches (at least at quiescence)
3. Allow tagging etc to itself be under configuration management

Problem 1

Consider that we limit ourselves to the following sub-problem

- There is only a single branch, possibly with thousands of check ins along the branch, on large numbers of objects
- In reasonable time we can check-out from *any* position along the branch. This generates a copy of all objects at that point in time.
- In reasonable time we can calculate a difference between any two points along the branch. This generates a single universal operation!
- We are not interested in contexts and context boundaries
- We are not interested in clustering concerns

Proposal : Operations don't live in contexts with the objects that they apply to

At present we think of operations as being localised to the context that they apply to. However there are some serious limitations with this approach

- We don't support move across context boundaries
- A "task" is local to a context.
- We can't merge operations performed by the user into a single delta representing a check in

It is proposed that it is wrong to localise operations with the contexts (which are used to cluster the objects in the data model). Instead, operations belong to a *task*. This is a (long running) transaction that places no limitations on what objects can be changed as part of that task.

Conclusion: *The configuration management system will deal with branches and operations at a global level. Operations are localised by task, not by context*.

Problem: If we allow objects to move accross context boundaries then per context GC can't be achieved because we lose the idea of a context managing its own OID space, which is used to support a sweep.

IDEA for GC

Configuration management is only interested in the management of course level operations and branches, and not with the document state. Therefore it has no need to GC the document state. Normally operations or branches will never be deleted. So configuration management doesn't even need a GC!

By contrast a working set is only interested in document state arranged into contexts. It is not interested in history, apart from the current session. The history for the current session is compressed on the fly so it will be quite econmical on space. The part that is compressed no longer allows for undo. Only the uncompressed tail of the HB allows for undo. Therefore this small section needs to provide roots for GC of a given context, to make sure objects reachable from the uncompressed tail of the HB are not deleted.

The working set is very much interested in deleting objects where possible, so GC's on a context are common and useful.

Ideas to investigate

- Consider two big branches that merge, and we need to recreate the state from this merge. We don't want to IT one branch past the other directly because that is $O(n^2)$ which is prohibitive. Instead it is important to compress each branch down into a single operation, and then only IT a single op past a single op. This makes the branches contextually serialised, so we can then compress them down into a single delta. That is the basis for recreating the state from the merge. From this we conclude that it is indeed useful to calculate and perhaps cache compressed chains/branches.
- There is a cost with caching a compressed version of a chain. Eg we may have some large immutable object added by an operation that is stored within every delta containing that operation. Or can deltas share state!
- It is not clear whether an instruction to merge two branches on the repository is useful. Perhaps it is something that is done on a client working copy.
- It seems simpler to never support merging in only part of a branch. Instead, that subset should be on its own branch. However it is still not clear what that means in terms of the graph.
- The repository can never permanently discard operations. This simplifies garbage collection concerns.
- It is interesting that a universal operation contains the instructions to *create* objects as well as modify objects. This suggests that IPartitionSession doesn't need a separate method to download a context, because a universal op can do that!
- It is not clear whether config mgmt should be global to all contexts, or the branches etc are at the level of each context. It's obviously useful and important to be able to tag huge amounts of data in the LSS. Perhaps operations are not thought of as part of a context. Instead operations belong to a branch or session. A session may operate on any number of contexts. The session is localised (ie clustered on disk) because it relates to a task being done by the user. It is clustered automatically because the operations in the task tend to be written in a short amount of time. Note that operations use OIDs to identify objects and have no problems with being able to identify objects in different contexts. A significant benefit of this approach is that we will support moves across context boundaries!!!! A problem is that GC is no longer easy.

- A problem with the idea of merging operations is that they can't deal with creation of child contexts very well. Merging a task that is across context boundaries can't be done if operations are limited to the context that they apply to.

The repository

The repository is an independent process that manages versions and branches of an OODB. It is not interested in implementing object behaviors and in fact only needs to know about object schema (in the form of datadefs) in order to implement all its required functionality. By never executing application defined behaviors the repository is immune from most forms of attack. Even though it may store objects that contain malicious code, the repository will never execute that code. Ultimately the repository is only interested in registering operations generated by a different process.

The repository doesn't treat the data objects as opaque blobs. Instead it needs complete knowledge of the schema so that it can validate and more generally deal with operations on the (data) objects. Interestingly, it is reasonable to treat a data def as serialisable data. This would allow a repository to be sent schema from clients over the wire eliminating the need to install potentially malicious dlls on the repository machine.

The Subversion model is used for the repository. Clients can *check-out* any version, *check-in* changes made to a version they checked out, perform an *update* from the repository, *tag* a version, create a new *branch* and *merge* a branch into another branch. As a general rule, operations are sent over the wire rather than versions of objects. Unlike Subversion there is no need to compute a difference - because that is implicit in the operations.

Vector times are the basis for keeping track of the context of operations being sent to or from the repository.

The repository is minimalistic in the sense that it avoids higher level concepts such as

- Associating sets of operations with a task
- Giving tags or branches a friendly name
- Associating siteids with a user?
- Project management capabilities, such as entering tasks yet to be done and assigning them to users.

Instead, these higher level project management functions are provided by application dlls that store the additional information in the database like any other information. This allows the project management data to itself to be under configuration management.

Apart from caching concerns, the repository stores nothing but operations corresponding to (course level) check-ins by users. These operations tend to form a graph. Operations are themselves immutable - because they are meant to be a reliable record of what happened over time. Therefore the repository itself only contains *immutable* data.

Any two repositories can be easily synchronised - even if they mostly work on unrelated data (ie in unrelated partitions). This involves exchange of operations so that they take on the union of all operations across both repositories. This assumes the following

- Operations (associated with check-ins) are immutable
- A user can only check-in his/her changes to a single repository. Once the check-in has been performed successfully, a subsequent check-in is disallowed. [This sounds like it requires multi-phase commit. However, the effect could be achieved more simply and reliably by assigning the operation its (s,t) values before the check-in is performed]

Note that synchronisation of repositories doesn't itself require operational transform.

It is interesting to note that such a system allows for atomicity of transactions on a global basis even though each computer only stores a small subset of all the data in the world. Vector times and operational transform make this possible! Atomicity of the LSS is only needed to protect the integrity of a single database. There is no need for atomicity of distributed transactions (cf multiphase commit), and yet the system can behave like that facility is available.

Representation of operations on the repository

Consider that most check-ins are performed sequentially along a single branch in the repository, and the operations are stored in a linear chain on this basis. Over time this could result in a total working set of many gigabytes of data. This is associated with accumulating all the operations into one single big operation. There are clearly some problems with this approach

- The repository could run out of memory while trying to build such a large operation
- It would be unreasonable to have to send all this data to a new user performing a check-out. The user may only be interested in a small subset of this data

We need to support localisation of operations based on the underlying localisation of the data. The partition of data into a hierarchy of contexts forms the basis of this approach. Despite this we would rather not impose restrictions on what data an operation is allowed to work on.

For a given operation we can determine the set of contexts that are involved. This could be used to *index* the operations in the repository. This contrasts with the naive linear data structure described above. As an example, consider an object that is edited on day 1, then left alone, until it is edited again on day 100. The linear operation log would show these edits far apart. However, by indexing on context the edits would be stored near each other. This allows the repository to accumulate those edits in order to send a copy to a user during a check-out.

Consider that the repository is made aware of various nested levels of tasks, allowing it to form hierarchical data structures for storing the operations. There could be a major task which is to rewrite a complex module, and this takes many weeks. This could involve many check-ins over time. These check-ins could be interleaved in time with an independent major task - perhaps being performed by a different user.

Inevitably we need to capture the information inherent in these interleaved check-ins. However, not by using a linear data structure to store all the operations. Instead, a vector time captures this information.

Consider a more extreme approach which is to index operations on both OID and fieldId. A single operation is implicitly broken up into pieces! Let's call these Operets. This is ideal for accumulating deltas on a field of an object.

So for a given field on a given object we see all changes due to check-ins. These must form a connected graph. Each operet stores (s,t) values and a list of incoming and outgoing operets. Clearly it will be straightforward to calculate any required version of the field. This is achieved as follows : Given a vector time x , find all operets on the field that are relevant according to their (s,t) values (ie in X(v)). Then merge all these operets into a single state difference.

The upshot is that a user can check-out parts of the repository on demand. The act of accessing a new context may cause a context to be downloaded. The repository is able to efficiently provide the right version of the context (as if the user had actually checked it out in its original form).

Working sets

A working set is used to perform the edits on the objects in the database, corresponding to a task in the configuration management system.

Are completing a task, it is useful to begin another one, possibly on a different branch. To allow this, while making good use of calculated dependents, it is proposed that the repository can provide a single operation which represents a delta that can be applied to the working set to taking it to a new branch. Only the dependents that are affected need to be recalculated.

The user can also copy the LSS file to quickly create a new and independent working set. The important this is to avoid checking in changes more than once.

Summary of approach

- It is assumed that all changes to objects can be represented by operations. A sequence of operations can be compressed in linear time into a single operation that is provably as concise as it can be.
- Operations allows for inclusion transform and exclusion transform. If this is performed after compression then it can be done in linear time. It allows for branching, merging, synchronisation and real time collaboration
- There is a layer for a repository that stores a map indexed by (OID, fieldId). The entry is a graph of operets. Each operet stores (s,t) values.
- An incoming operation can be added to the repository by breaking it down by OID and fieldId. The operet stored incoming edges from operets according to which operets are in the extent of the vector time of the operation.
- Given a vector time, it is possible to calculate the value of the field from all the operets.

IDEA: 19 Sep 2005

Consider that we store the effects document as a sequence of intervals, where for each interval we store the (s,t) associated with when it was inserted, and the set of (s,t) associated with when it was deleted. Note that in the effects document a character can only be inserted once. However it can be "deleted" more than once along different branches.

For a given vector time v, we can build the document state for v by iterating through the intervals and test whether each each interval is present. The interval is present if the insertion (s,t) is in X(v), and none of the deletion (s,t) are in X(v). This test is quite fast, so it should be easy to recreate any document state very quickly. Note furthermore than even a very large effects document can be linearly scanned from disk, so efficient use of the hard-disk is easy to achieve.

Given two vector times v_1, v_2, we can also linearly scan the intervals in the effects document to calculate a difference - corresponding to $X(v_2) - X(v_1)$. For each interval we determine whether changes occurred between these vector times. If so then the interval is part of the difference. Note that a difference may indicate that an interval is being added or removed.

When a check-in is performed, we need to translate q-positions. This is basically an inclusion transform problem. The repository merges all check-ins as it goes! Note that p-positions are irrelevant.

A check-in is associated with a particular vector time v. We can process the effects document according to this vector time - recreating the context for the check-in. As we do this we can find the shift in q-position. This is used to find the insertion position for the check-in.

Conclusion: A check-in also involves a linear scan through the effects document. Therefore it is

TODO: See whether this approach is useful for collaboration.

Insert then delete multi-char operations

An operation that stores insertions then deletions is able to delete its own characters, making it suitable for compressing the log in a way that is compatible with IT/ET. Inserts are in post-insertion coords, and deletes are in pre-deletion coords. Therefore insertions and deletions are expressed in the same intermediate document state! This suggests that the form [I X] is the most declarative possible because all coords are expressed in the same document state. It even seems possible to store insertions and deletions in a single list ordered by q-position, and the dual IT of two operations only requires a single scan through the intervals.

[I X] may be better for move operations because we can perform an in-place memcpy. There is no need to create a temporary buffer and copy the bytes twice. This is quite a useful saving.

References

