Configuration management and the attic

Feb 2005
David Barrett-Lennard

Introduction

Configuration management depends on the concept of a user having a *local copy* which is able to diverge from the repository. ie the user can work off-line. The user needs to be able to perform an *update* - which is where operations and contexts are sent to the repository. This information transfer is strictly one-way. Independently of this, the user must be able to initiate a *commit*, which is where contexts and operations are downloaded from the repository. This is also strictly one way.

There are two requirements for a commit

- it must be atomic - ie it either succeeds completely or not at all
- it only succeeds if an *up-to-date check* succeeds. Up-to-date means there are no outstanding operations that haven't been sent to the user.

The atomicity and up-to-date check are over a *set of contexts*. ie we don't deal with each context independently. In practice we sometimes have contracts across context boundaries, such as a dependency between two software projects (in different contexts) that have a "using" relationship.

The "repository" will use a single linear history buffer for each context in the normal way. Each operation will represent all the delta's associated with a check in. A universal operation is ideal for this purpose, because it is very efficient at storing many edits on a collection of documents. Note that vector times are used in the normal way as well, so we see that sequence numbers are incremented for each check in.

A number of features are offered that are not available in CVS

- A local user can compress operations as he works off-line, so a commit is very fast because there is no need to calculate differences. Instead the difference is already present in the form of a universal operation and this is simply sent and applied to the repository.
- The repository itself can be distributed on a number of computers, and these can keep TCP socket connections alive and actively propagate changes. An incoming operation on one of these "repository" computers is like a "check in".
- A site can have a local repository that may be offline with respect to the "main" repository. In practice this means we could have developers in one city using a local repository for check ins, and not be affected if there is a network failure to another city where a repository is being synchronised.
- There can be arbitrary topology changes or roll backs to the distributed repositories, and it is guaranteed that all sites will converge at quiescence.

The need for atomicity of a commit is precisely associated with the need for *long running transactions*. It allows a computer to make extensive changes to a local copy - perhaps taking many days to complete, and know that this change can be applied atomically to the repository.

Sometimes a user wants to perform a long running transaction atomically. A solution may be to spawn another process specifically for this purpose.

Collaboration

A user may choose to host a collaborative session for a given partition. The partition will listen on a port for connections from clients that want to be part of the collaboration. These clients can in turn listen on a port to allow other clients to connect to them directly. Therefore a large tree of connected computers may be employed for a collaborative session.

During the collaboration, edits are sent between computers and merged using dual IT. Assuming network latency is not excessive, each computer will have a relatively small "suffix" of operations to

be transformed against incoming operations. Note therefore that in the normal case the CPU load is only a function of local network latency, local fan-out, and the global rate of operations being generated. Importantly the cost is independent of the total length of the collaboration, and the total number of (spectator) computers that are connected. Therefore the system is scalable to very large numbers of collaborating users - particularly when many are spectating at a given point in time.

If the network fails for a long period it is better to treat the session as having failed rather than try to use dual IT on a very long suffix to carry on with the session. This is important because dual IT is O(nm) where n and m are the respective number of operations that need to be merged between the two sites. It is proposed that some concept of timeout based on the length of the suffix is required.

The tail of the history buffer contains operations that are associated with the collaborative session. This tail is divided into two parts. The tail begins with a universal operation called the *Compaction Operation* (CO) that is used to accumulate the on-the-fly compaction of operations. Following the CO are the operations that have yet to be coalesced into the CO.

The host of the collaboration decides what operations belong to the CO and what don't. This information must be provided to all the other sites. The host of the collaboration will send two special messages. These are forwarded on by intermediate nodes so that all sites in the collaboration receive these messages.

AllowCompaction(vt)	The given vector time indicates the set of operations that have currently been assigned to the next CO. Sites must not coalesce operations into their CO unless this message has been received to indicate that it is allowable. The reason is that compaction is not information preserving and can't be reversed.
CommitCompactionOperation(s,t,vt)	Used to indicate the end of a compaction operation. All sites should use adjacent swap as required to divide the HB into prefix and suffix according to the given vector time. Due to causality preservation this message can only be received after all operations belonging to the CO have already been received. The receiver of this message should complete the task of coalescing operations into the CO, committing it with given (s,t) values, and starting a new CO. Note that all sites will agree on the set of operations that have been assigned to the committed CO.
EndOfCollaboration()	Used to indicate that the host no longer wants to continue the collaboration. When a site receives this message, it should discard any operations that haven't been assigned to a committed CO.

A site can determine whether an operation has already been sent to all its existing adjacent computers that are part of the collaboration. This is an additional condition that must be satisfied before an operation can be coalesced into the CO.

A computer may go down then back up again. When it connects again as part of the ongoing collaboration it may not be able to re-synchronise with its server using its existing set of operations in the HB because of the way each process compacts operations. Therefore a client will throw away the previous operations not assigned to a committed CO and roll back to the state corresponding to the end of the last committed CO. It will then download all the operations for the next CO from the server from scratch. This is not terribly onerous because of the on-the-fly compaction of operations by the server.

Latecomers can join an existing collaboration in the same way. Again this should be reasonably efficient due to the on-the fly compaction of operations.

When a process boots it must roll back (ie undo and discard) operations that belong to a collaboration but have not been assigned to a committed CO. This includes the host of the collaboration! It is permissible for the host to continue with the collaboration by downloading operations for the next, not yet committed CO from an adjacent computer.

Working off-line

A user may want to work off-line. There is still the concept of on-the-fly compaction. Furthermore, occasionally the user will want to commit the current CO, and start a new CO.

It is possible that the user will suddenly want to host a collaborative session. This could use the existing off-line work.

Branches and processes

A branch is regarded as synonymous with a process. Although it may be convenient to say that users A and B are "working on the same branch", in reality we must regard the two associated ceda processes (for these two users) as separate branches. We understand that they are following a convention of check-in that makes them look like they are working on the same "branch", but we should avoid this terminology and be more precise!

A ceda process only supports linear history. The process is associated with exactly one active branch, and there is no ambiguity about the concept of the current working version of all the objects - this is always associated with the "sticky tag" at the end of the linear history.

The attic

There is a provision for tagging of old versions, and the ability to fetch old versions. An "attic" is used to cache copies of old versions to reduce the time taken to fetch old versions (with no cache it is always necessary to rewind history).

Note therefore that we regard a ceda process as acting as both a repository for old versions, as well as a working set. This is quite a different picture to CVS which distinguishes between these two concepts.

Each object in the shared data model has an OID that uniquely identifies it across all computers. There is no need for an envelope for versions, because all old versions are stored (and indexed) independently in the attic. To cache a "snapshot" of an object in the attic, the working version of the object is first member-wise copied. This provides a new and independent object with the same state for all data members, including outgoing pointers (prefs). When it becomes reachable from the attic for the first time, the clone is assigned an OID that is different from the working version. Therefore the clone in the attic can exist in the persistent store independently from the working version. Over time, the working version changes, as operations are applied to it. By contrast the clone stored in the attic is never modified.

It is important to understand that the pref members within old versions of objects in the attic always point at the working versions of objects. ie an object in the attic will never point at another object in the attic! This is good for two reasons

- When a remote process requests an old version of an object, it is serialised down the wire using pref members that use the OIDs associated with the agreed "identity" of the referenced objects.
- When an object changes leading to the need to copy old versions to the attic, we don't get a domino effect which would happen if old versions pointed directly at old versions.

A context has-a attic. The attic in turn has the following members

`int m_nextTagId`	The next TagId to be assigned when the context is tagged
`map<OID, VersionSet> m_map`	For each OID stores the old versions of objects with that OID.

A VersionSet has the following members

`bool m_needToTag`	Indicates whether the current working version has been tagged but has not yet been copied. Initialised to false when a VersionSet for an object is first created.
`map<TagId, prefbase> m_map`	Stores pointers to the old versions of the object.

Let 'T' denote the tagging of a context, and 'M' denote the modification to an object within the context (by applying an operation). A copy of an object is added to the attic for each T to M transition. Eg the sequence TTTTTMMMMTTTMM only contains two T to M transitions. This is achieved using the following algorithm

```
int Context::TagCurrentWorkingVersions()
{
    loop (i in m_attic.m_map)
    {
        i->second.m_neetToTag = true;
    }
    return m_attic.m_nextTagId++;
}

void Context::OnBeforeApplyOperationToWorkingVersionObject(IObject* p)
{
    VersionSet& vs = m_attic.m_map[p->m_oid];
    if (vs.m_needToTag)
    {
        vs.m_needToTag = false;
        vs.m_map[m_attic.m_nextTagId] = p.ShallowCopy();
    }
}
```

This is essentially a copy-on-write algorithm.

Consider that for a given object the version set map is as follows

TagId	prefbase
23	p1
45	p2
86	p3

This should be really be interpreted as follows

TagId range	version associated with tag range

[0,23)	p1
[23,45)	p2
[45,86)	p3
[86,...)	working version

Fetching old versions
Objects in the attic are read only. As a result it is feasible to send a context to a machine without the attic!

The framework allows for an old version of a context to be fetched. This should be regarded as transient and inaccessible from the data model. It can be used for transient purposes, such as

- to send an old version to another process (in which case it can become part of the working set on that process)
- Displaying differences to the user
- Performing a query on old versions

The fetch function could be passed the OID of the object and the vector time. A more restricted approach is to only support fetch of previously tagged versions.

TODO: Actually a whole context must be fetched at a time.

Displaying differences
The user can look at differences between versions. Note that versions always lie on the same linear branch, so this is conceptually nothing more that highlighting the set of operations between the two versions.

Purging history
A process is allowed to purge history (and old versions in the attic). It may also be desirable to avoid downloading history when creating a branch. When there are hundreds of active branches, it seems silly to have hundreds of copies of the old versions. This represents significant wasted space.

Project management
It is proposed that an explicit shared data model be used for project management, and in particular to track knowledge about the currently active branches (ie processes). Being a real data model it is maintained using operations, and knowledge about branches is merged to allow all users to learn about branches and about what the other users are doing. This data model could store information about the following

- The set of tasks. Note that tasks form a tree (because a task typically has sub-tasks).
- The time estimates assigned to the tasks.
- The actual time taken by a task. This information could be generated automatically!
- The expected date/time when a task will be completed.
- Risk management
- Resource management. Eg the assignment of tasks to people
- Comments by users about a given task. Eg a user may write a comment that he thinks the task should be done in a certain way.
- The tasks that users are currently doing.
- Network status. What computers are down? What printers are working?
- Are users at their computer? When are they leaving the office? When are they contactable? When are they going on holidays?
- Any meetings that have been scheduled.
- The milestones for the project. When are they?
- The set of active branches (processes) in the system? What IP address and ports are they listening on? What collaborations are taking place?
- The users that are requesting that other users join a collaboration

- The association between the active branches (ie processes) and the tasks?
- Timesheets

In order for this information to be useful, everyone on the project must take the effort to keep it up to date. It is a shared responsibility. As tasks become redundant they should be deleted. As new tasks are found they should be added.

Note that with fine grained processes that are associated with tasks, the system will be able to keep track of time spent on a given task. This eliminates the need for manual entry of timesheets. It also allows the system to accurately log time spent on tasks for the purposes of project management.

Having fine grained processes for tasks is a good idea because it allows for tasks like "bug fixes" to be easily merged into branches very easily. A user can pull in other people's (recent) work like "shopping".

The idea to update only a range of operations (between two tags) is not supported because that could break causality preservation. It also is at odds with the idea that a vector time describes the set of operations in a history buffer. This is a good reason for users to use reasonably fine grained processes as they work on tasks.

To make this feasible, creating a local branch must be quick and simple (ie a click with the mouse!). It is best done directly from the project management GUI, or else the process from which a branch is being derived.

The following actions should generate operations on the project management data model
- Creating a new branch
- Deleting a branch
- Hosting a collaboration
- Joining a collaboration
- Updating from a remote process
- Committing to a remote process

TODO

We need to support the following (for a given context)
- A process can update from another process
- A process can update a "section" from another process (defined by two tags - one of which may be the sticky tag at the end of the branch). This is subject to causality preservation constraints.
- A process can commit to another process
- A branch can be created by creating a process and performing an update from another machine. It is important that the update can be up to some given tag. The sender may need to send a context using old versions of objects that have been fetched from the attic. This branch may then diverge as required.
- A process can display information about its own versions. In particular it can display a difference between any two versions along the linear history.

Fetching a tagged version
There is a concept of fetching a tagged version for a given context. Objects are only sent if they are reachable during this fetching process.