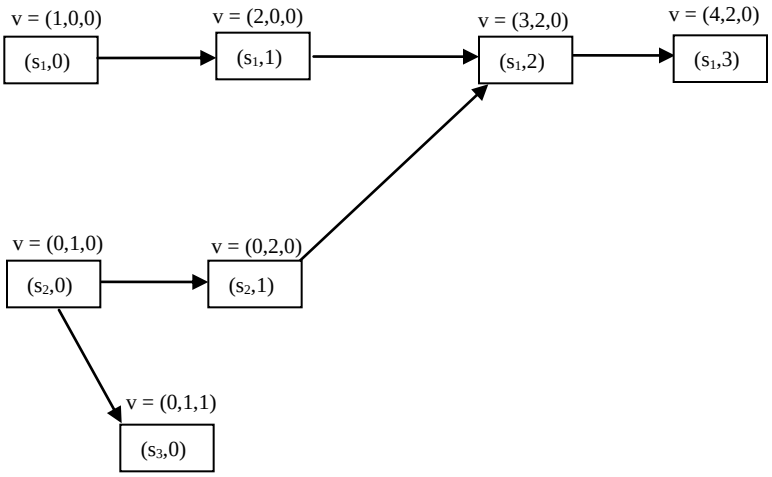
Repository Graph

Feb 2008
David Barrett-Lennard

Introduction
In this document we assume the definition of vector time, sites S, op(s,t) described in [1].

A *Repository Graph* is a DAG used to keep track of all operations that have ever been checked into a repository.

The following is an example, where there have been 4 check-ins from site s_1, 2 check-ins from site s_2 and 1 check-in from site s_3.



Each node corresponds to a check-in of a single operation (generated by a single site) uniquely identified by its (s,t). Let node(s,t) denote the check-in node associated with the check-in of op(s,t).

For node n, let $\alpha(n)$ denote the set of direct in-nodes, and $\beta(n)$ denote the set of direct out-nodes.

Alongside each node we have provided an associated *vector time*, which in this case is 3 dimensional because there are three sites. The vector time $v = (t_1,t_2,t_3)$ represents the number of operations that have been applied by the respective sites s_1,s_2,s_3 in order to land on that node, inclusive of the check-in itself.

For example the top left check-in was the first operation performed by site s_1, taking the vector time from (0,0,0) to (1,0,0). The operation itself is uniquely identified using $(s,t) = (s_1,0)$.

In our implementation each node records its associated (s,t), but not the associated vector time. Instead algorithms that recurse the graph can calculate the vector time during the recurse.

Overall vector time for the repository
The implementation records a vector time corresponding to all check-ins into the repository. In the above example this is

vr = (4,2,1)

There isn't necessarily a node corresponding to this vector time.

The check in of op(s,t) causes vr to grow by adding (s,t+1).

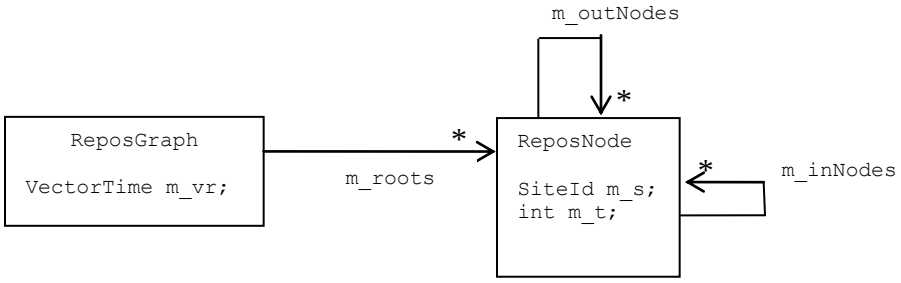
Validation
This graph can be used to validate causality in the check-ins. It can also be used to validate the two vector times passed to GetDiff() on the repository. This will help allow the repository to validate all external requests - an important step towards an implementation of a reliable server repository that is even safe against malicious clients.

Scalability
Note that storing a graph of all checkins is reasonable in many practical cases because the number of checkins will not be excessive - a few thousand per year being typical on a project with half a dozen developers.

Roots
The roots are the set of nodes that have no in-nodes

i.e. roots(G) = { $n \in G \mid \alpha(n) = \{\}$ }

URL - implementation



The precedes relation
Definition: For given check-in nodes n_1,n_2 we say n_1 precedes n_2 (written $n_1 \rightarrow n_2$) if there exists a path from n_1 towards n_2.

This relation is reflexive:

$\forall n,\ n \rightarrow n$

and transitive:

$n_1 \rightarrow n_2$ and $n_2 \rightarrow n_3 \Rightarrow n_1 \rightarrow n_3$

so it is a preorder.

It is also antisymmetric:

$n_1 \rightarrow n_2$ and $n_2 \rightarrow n_1 \Rightarrow n_1 = n_2$

so it is a partial order.

The purpose of the graph is basically to record the precedes relation amongst all check-ins. The precedes relation is transitive, so for efficiency we never store redundant edges. ie if there is an edge from n_1 to n_2 and from n_2 to n_3, we don't store the edge from n_1 to n_3.

Extent of a vector time

Definition: The *extent* of vector time v is the set of check-in nodes denoted by $\chi(v)$ satisfying

$$\chi(v) = \{ \text{node}(s,t) \mid t < v(s) \}$$

Note the following

- $\chi(v_\varnothing) = \{\}$
- $\chi(v_1 \uparrow v_2) = \chi(v_1) \cup \chi(v_2)$
- $\chi(v_1 \downarrow v_2) = \chi(v_1) \cap \chi(v_2)$
- $v_1 \leq v_2 \Leftrightarrow \chi(v_1) \subseteq \chi(v_2)$

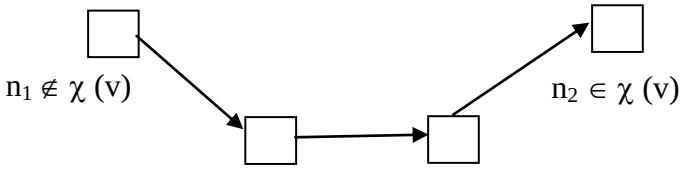
Causality violation

Definition: The set of *valid vector times* with respect to repository graph G, is denoted by

$V(G) = \{ v \mid$
$v \leq vr(G)$ and
$\forall n_1,n_2 \in G,\ n_2 \in \chi(v)$ and $n_1 \rightarrow n_2 \Rightarrow n_1 \in \chi(v) \}$

Note that $v \leq vr(G) \Leftrightarrow \chi(v) \subseteq G$

Vector times with $v \leq vr(G)$ that break the second condition are said to *break causality*. ie $\exists\ n_1, n_2 \in G$ st $n_1 \notin \chi(v)$ and $n_2 \in \chi(v)$ and $n_1 \rightarrow n_2$



Claim: $v_1 \in V(G)$ and $v_2 \in V(G) \Rightarrow (v_1 \downarrow v_2) \in V(G)$

Proof:

Let $v = v_1 \downarrow v_2$

We show $v \in V(G)$.

$v_1 \leq vr(G)$ (because $v_1 \in V(G)$)

$v \leq vr(G)$ (because $(v_1 \downarrow v_2) \leq v_1$ and $v_1 \leq vr(G)$)

Let $n_1, n_2 \in G$ with $n_2 \in \chi(v)$ and $n_1 \rightarrow n_2$

Need to show $n_1 \in \chi(v)$

$\chi(v) = \chi(v_1 \downarrow v_2) = \chi(v_1) \cap \chi(v_2)$

$n_2 \in \chi(v_1)$ (because $n_2 \in \chi(v) = \chi(v_1) \cap \chi(v_2)$)

$n_2 \in \chi(v_2)$ (because $n_2 \in \chi(v) = \chi(v_1) \cap \chi(v_2)$)

$n_1 \in \chi(v_1)$ (because $n_2 \in \chi(v_1)$ and $v_1 \in V(G)$)

$n_1 \in \chi(v_2)$ (because $n_2 \in \chi(v_2)$ and $v_2 \in V(G)$)

$n_1 \in \chi(v_1) \cap \chi(v_2)$ (because $n_1 \in \chi(v_1)$ and $n_1 \in \chi(v_2)$)

$n_1 \in \chi(v)$ (because $\chi(v) = \chi(v_1) \cap \chi(v_2)$)

Claim: $v_1 \in V(G)$ and $v_2 \in V(G) \Rightarrow (v_1 \uparrow v_2) \in V(G)$

Proof: Let $v = v_1 \uparrow v_2$

We show $v \in V(G)$.

$v_1 \leq vr(G)$ (because $v_1 \in V(G)$)

$v_2 \leq vr(G)$ (because $v_2 \in V(G)$)

$v_1 \uparrow v_2 \leq vr(G)$ (because $v_1 \leq vr(G)$ and $v_2 \leq vr(G)$)

Let $n_1, n_2 \in G$ with $n_2 \in \chi(v)$ and $n_1 \rightarrow n_2$

Need to show $n_1 \in \chi(v)$

$\chi(v) = \chi(v_1 \uparrow v_2) = \chi(v_1) \cup \chi(v_2)$

if $n_2 \in \chi(v_1)$

$n_1 \in \chi(v_1)$ (because $n_2 \in \chi(v_1)$ and $v_1 \in V(G)$)

$n_1 \in \chi(v)$ (because $\chi(v_1) \subseteq \chi(v_1) \cup \chi(v_2) = \chi(v)$)

else

$n_2 \in \chi(v_2)$ (because $n_2 \in \chi(v_1) \cup \chi(v_2)$ and $n_2 \notin \chi(v_1)$)

$n_1 \in \chi(v_2)$ (because $n_2 \in \chi(v_2)$ and $v_2 \in V(G)$)

$n_1 \in \chi(v)$ (because $\chi(v_2) \subseteq \chi(v_1) \cup \chi(v_2) = \chi(v)$)

Claim: v breaks causality $\Leftrightarrow$ $\exists n \in \chi(v)$ st $\alpha(n) \setminus \chi(v) \neq \{\}$

$\exists n_1, n_2 \in G$ st $n_1 \notin \chi(v)$ and $n_2 \in \chi(v)$ and $n_1 \rightarrow n_2$

Proof: ($\Rightarrow$) (very informal - really should be using induction)

v breaks causality so $\exists\ n_1, n_2 \in G$ st $n_1 \notin \chi(v)$ and $n_2 \in \chi(v)$ and $n_1 \rightarrow n_2$

If $n_1 \in \alpha(n_2)$ then we are done.

Otherwise

let n_2' be an in-node of n_2 that lies on a path between n_1 and n_2.

Then $n_1 \rightarrow n_2'$

Repeat this process (replacing n_2 with n_2') until $n_1 \in \alpha(n_2)$

($\Leftarrow$)

let $n_2 = n$ and $n_1 \in \alpha(n_2) \setminus \chi(v)$

$n_1 \notin \chi(v)$ (because $n_1 \in \alpha(n_2) \setminus \chi(v)$)

$n_1 \rightarrow n_2$ (because $n_1 \in \alpha(n_2)$)

This is useful because it shows that during a trace of all the nodes in $\chi(v)$, we can locally check whether all in-nodes of a given node are also in $\chi(v)$. If not then we know that the given vector time breaks causality. More importantly if it never happens then the vector time must be valid. This only represents a small overhead to the trace algorithm.

Growing the graph for a check-in

A check-in provides the following information

- (s,t) of the operation
- The base vector time v

The given base vector time doesn't necessarily correspond to an existing node in the graph.

To perform a check-in for given v, a recurse is performed starting from the roots. We recurse into a node if and only if its (s,t) is in $\chi(v)$. i.e. if (s,t) is in $\chi(v)$ then we recurse into its outnodes, otherwise we ignore the node.

We need to mark which nodes we have already visited, so if we visit a node again we avoid processing it twice. Rather than store a flag in each node the recursive function uses a set of visited nodes.

If $n \in \chi(v)$ and $\beta(n) \cap \chi(v) = \{\}$ then we add an edge from node n to the new node being added to the graph.

We can check that the number of visited nodes matches the sum over t of the vector time. This proves that there is nothing in $\chi(v)$ that we missed. However, it doesn't prove that we visited everything we should have.

Consider that a check-in for the above example is attempted with v=(4,0,0), c=(s_1,4). The repository should not allow this check-in because it breaks causality, because if s_1 has performed 4 operations we know that s_2 must have performed two operations. To detect this error, when we visit a node n we ensure that $\alpha(n) \subseteq \chi(v)$.

Validating vector times for a checkin
A check-in provides a vector time v and (s,t) for the operation.

We perform the following validation checks

Error condition	Description	Error generated
$v(s) \neq t$	This ensures that the operation was done in the context of all previous operations generated by that site	Assertion failure
$t < vr(s)$	Given operation is already present in the repository	`REPOS_OPERATION_ALREADY_PRESENT`
$t > vr(s)$	Can't check-in operation because it is not the next expected operation from the site that generated the operation.	`REPOS_OPERATION_MISSING`
$\neg(v \leq vr)$	$\chi(v)$ is not a subset of $\chi(vr)$. This ensures that the repository already contains the execution context of the operation.	`REPOS_VECTOR_TIME_NOT_A_SUBSET`
$\exists n \in \chi(v)$ st $\alpha(n) \setminus \chi(v) \neq \{\}$	v breaks causality	`REPOS_VECTOR_TIME_VIOLATES_CAUSALITY`

Note that these error conditions are not mutually exclusive. They are checked in the given order.

Fast validation of check-in vector time
We avoid the set of visited nodes. Instead we decrement t values in the vector time according to nodes that we have already visited. We can use this to ensure that we visit exactly the nodes in $\chi(v)$ - nothing more, nothing less. It is nice to avoid the need for the visited set. At the end the vector time will be empty.

Proposal for a more efficient implementation
In practice there will often be long linear branches, perhaps with many hundreds of check-ins in a linear sequence.

Proposal: A class records a linear chain of check-ins using a vector of (s,t) values. Linear chains are split as required.

```
class CheckInChain
{
public:
```

```
private:
    std::vector<CheckInChain*> m_inNodes;
    std::vector<CheckInChain*> m_outNodes;
    std::vector<Opid> m_checkIns;
};
```

References

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