Lossy assignment operations

Mar 2008
David Barrett-Lennard

Introduction

This document describes a new approach to representing and transforming assignment operations for an interactive collaboration.

The proposal avoids the use of a q-position as described in [1]. It also eliminates the need for a site to record a HB suffix for each session to transform incoming operations. As such it avoids the need to implement the transpose of adjacent assignment operations.

There are significant performance advantages to this proposal.

Lossy assignment operations

A *working set object* is a client side representation of an object (as distinct from a representation of an object in a repository).

A working set object records the current value of an assignable field without any need to store additional information. Assignment operations are always generated with q = 0, so unlike vector<T> fields there is no need to record a correspondence between q-position and p-position. Therefore disabled assignment operations (ie with q > 0) have absolutely no effect on the working set objects. *They can be treated as though they never actually occurred at all*.

Before doing a check-in it is desirable to compress the log so that the check-in only records the overall changes to the working set objects. For each assignable field, we can always compress all the assignments down to a single assignment that dominates all the other assignments.

This suggests the following representation of a check-in assignment operation to a given field of type T:

```
template <class T>
struct AssignmentOp
{
    SiteId s;
    int t;
    T value;
};
```

It is implicit that it has q = 0 according to the concept of q-position in [1].

It would seem this operation can't be used in the *History Buffer* (HB) on the client because it cannot support being disabled under IT.

Proposal for how to deal with assignments in an interactive collaboration

We assume that fields on objects are uniquely identified by a *FieldId*. A FieldId combines an *Object Identifier* (OID) plus a *FieldPath*.

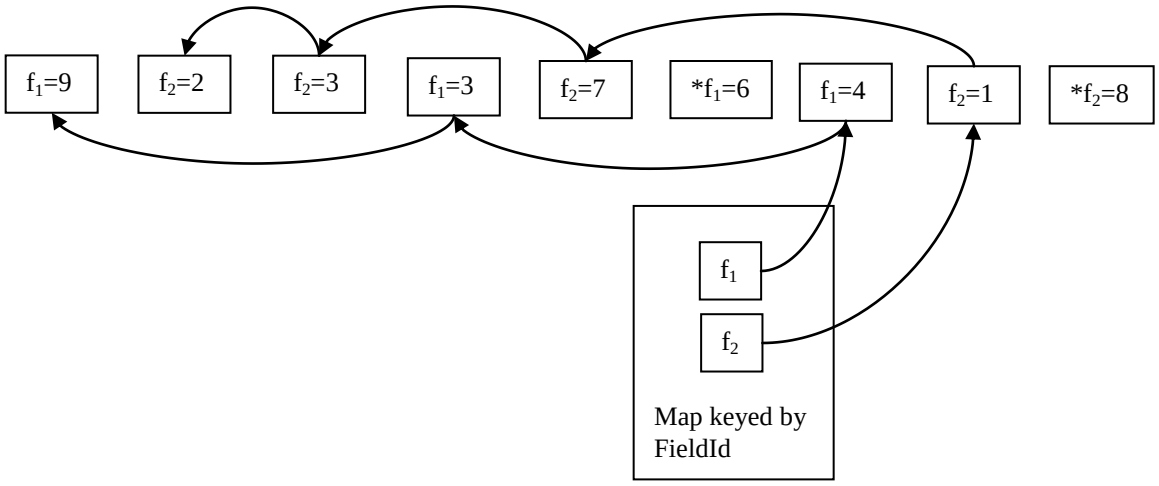
We don't record a q-position on assignment operations. Instead an *enabled flag* is recorded. An assignment is enabled if and only if q = 0 according to the algorithm in [1].

Enabled assignments always dominate earlier assignments to the same field that appear in the HB. For a given field the overall winner is always the right most enabled assignment operation on the field that appears in the HB.

Assignments in the HB that are disabled according to the algorithm in [1] (ie that under IT end up with q > 0) are almost completely redundant. The only reason they appear in the HB is to allow the system to manage vector times and causality. In fact only the (s.t) on disabled operations are relevant for this purpose.

On a given site, at a given time and for a given FieldId, there is at most one (right-most) enabled assignment that dominates all earlier assignments. The HB records a map keyed by FieldId that provides a pointer to the one and only dominating assignment operation. There is no entry in the map if there has been no assignment to the field.

In the following example there are two fields f_1,f_2. The HB consists of a sequence of assignment operations ordered left to right. An asterisk indicates a disabled operation. The map gives the right most enabled assignment operations (f_1=4 and f_2=1 respectively) on the two fields.



Furthermore for a given field the assignments are assumed to *backward chain*, meaning that each assignment stores a pointer to the previous enabled assignment to the same field that appears to its left in the HB. The backward chain is terminated with a NULL pointer.

The following illustrates how this might be implemented using C style data structs.

```
struct AssignmentOp
{
```

```
    // The (s,t) associated with the original generation of the assignment
    // operation
    SiteId s;
    int t;

    bool enabled;

    // Identifies the field assigned by the operation
    FieldId fid;

    // Value to be assigned to the field
    T value;

    // Previous enabled assignment operation in the HB that assigns to the same
    // field, or NULL if there is no such assignment.
    AssignmentOp* prev;
};

struct HB
{
    // Ordered list of operations in the HB
    vector<AssignmentOp> list;

    // X(hv) = all ops recorded in the HB
    VectorTime hv;

    // For each FieldId on which one or more assignments have been performed,
    // provides the pointer to the one and only assignment that dominates all
    // other operations.  This is always the right most enabled assignment to
    // the field in the HB.
    map<FieldId, AssignmentOp*> dm;
};
```

Local operation

When a client performs a local assignment operation it is marked as enabled and appended to the end of the HB. The map is updated to point to the new operation (which obviously dominates the previous assignment, if any). The new operation has its 'prev' member initialsed to point at the previously dominating assignment.

Note that the map makes backward chaining efficient (ie we avoid the need to scan the HB).

Finding the execution context of an operation in the HB

Given the HB we can start with vector time v = hv (ie the vector time describing the entire content of the HB) and iterate backwards through the linear list of operations and use the (s,t) recorded in each operation to assign v(s) = t. This provides the execution context (as a vector time v) for each operation during the reverse iteration.

To work correctly this depends on the recording and sending of disabled assignment operations.

```
// Get the execution context (as a vector time) of the ith operation in the HB
VectorTime HB::GetExecutionContextOfOp(int i) const
{
    VectorTime v = hv;
    for (int j = list.size()-1 ; j >= i ; --j)
    {
        v(list[j].s) = list[j].t;
    }
    return v;
}
```

Sending causally ready operations

A site can easily find the left most operation in its HB that hasn't been sent to a remote site (whether enabled or not). This provides a basis for streaming operations in an order consistent with causality preservation, and we can provide the execution context as a vector time for each operation sent over the wire.

When an operation is sent we provide the following information

```
struct SentAssignmentOp
{
    // Execution context
    VectorTime v;

    // The (s,t) associated with the original generation of the assignment
    // operation
    SiteId s;
    int t;

    bool enabled;

    // Identifies the field assigned by the operation
    FieldId fid;

    // Value to be assigned to the field
    T value;
};
```

Testing whether a remote operation dominates all existing operations

Let remote operation O_r be received with execution context v. Assuming O_r is enabled, the following procedure is used to determine whether O_r dominates all existing assignments to the same field on the local site:

```
bool HB::RemoteOpDominates(AssignmentOp Or, VectorTime v)
{
    AssignmentOp* Ox = dm.find(Or.fid);    // Ox = NULL if not found
    while(Ox && Ox->t >= v(Ox->s))
    {
        // Ox || Or
        if (Ox->s < Or.s) return false;   // Ox dominates Or
        Ox = Ox->prev;
    }
    return true;    // Or dominates all other assignments
}
```

`Or.fid` identifies the field. The local site looks up the local map `dm` using `Or.fid`. If no entry is found then O_r is a winning operation. Otherwise let `Ox = dm.find(Or.fid)` be the pointer to the local operation that currently dominates all other assignments on the field.

If $O_x.t < v(O_x.s)$ then we know that O_x is already in the execution context of O_r, hence O_r must dominate O_x (otherwise O_r would be disabled)

Otherwise $O_x \parallel O_r$.

Proof: If $O_x \rightarrow O_r$ then O_x would be in execution context of $O_r \Rightarrow$ contradiction

If $O_r \rightarrow O_x$ then O_r would already be present on the local site $\Rightarrow$ contradiction

Therefore we can compare siteids to see which assignment dominates the other. If O_r loses then we are done. Otherwise we need to test O_r against the backward chained operations. So we repeat using O_x.prev.

It can be shown that this algorithm is equivalent to the one using q-positions described in [1]. Note that most generally the HB consists of a mixture of assignments that are in X(v) and outside X(v). The conventional approach is to calculate a HB suffix to transform the incoming operation.

In the following assume $Y_i \in X(v)$ and $X_i \notin X(v)$.

HB = $[Y_1 \quad Y_2 \quad Y_3 \quad X_1 \quad Y_4 \quad Y_5 \quad X_2 \quad Y_6 \quad X_3 \quad X_4 \quad X_5]$

Disabled operations (ie with q > 0) never cause enabled operations to become disabled under IT. Therefore we tend to ignore them! So consider that all these operations have q = 0.

Conventionally we would need to transpose adjacent pairs of operations $[X_i\ Y_j]$ into $[Y_j'\ X_i']$ in order to move the X_1,X_2 above the Y_4,Y_5,Y_6. It will be found that X_1,X_2 become disabled when they IT past some of the Yj. Therefore when an enabled remote operation O_r with q = 0 transforms past $[X_1'\ X_2'\ X_3\ X_4\ X_5]$, we see that O_r must dominate X_1', X_2'. ie it is only necessary to compare O_r.s to X_4.s, X_5.s, X_6.s. The upshot is that in order to determine whether O_r dominates all the enabled concurrent operations X_i we can simply scan from right to left in the original HB and stop as soon as we reach the first enabled operation $Y_j \in X(v)$ (in this example Y_6).

Processing remote operations

The following code shows the overall algorithm used to process a remote operation Or with execution context v:

```
// Apply remote operation Or which has execution context v
void HB::ProcessRemoteOp(AssignmentOp Or, VectorTime v)
{
    if (Or.enabled)
    {
        if (RemoteOpDominates(Or,v))
        {
            Or.Do();
            Or.prev = dm[Or.fid];
            dm[Or.fid] = &Or;
        }
        else Or.enabled = false;
    }
    list.push_back(Or);
    hv(Or.s) = Or.t + 1;
}
```

Advantages

The above approach avoids the need for the HB suffix for each session. This avoids the following

- No need for the HB suffix for each session
- No need to transpose operations. No need to implement ET!

- No need to copy operations in the HB to form a suffix.
- Avoids the inefficiency of O(nm) complexity when ITing n incoming operations against m suffix operations. This instead becomes O(nk) where k is the number of assignment operations to the same field, and typically k << m.

References

[1]	*Operational transform - Assignment operations* David Barrett-Lennard. Aug 2005
[2]	*Repository - Assignable fields* David Barrett-Lennard. Feb 2005