Operational transform
Control Algorithm

Aug 2005
David Barrett-Lennard

Abstract
This paper discusses an efficient control algorithm that uses operational transform to support real-time collaboration amongst a set of sites.

Introduction
[1] and [2] demonstrate that it is possible to define operations together with simple and efficient implementations of IT, ET that support properties TP1, TP2.

All changes to the replicated document state are represented by operations. An operation can be regarded as an efficient delta between document states.

Each site stores its own copy of the document state, and a single linear history buffer of operations.

Locally generated operations are executed immediately allowing the system to be as responsive as a conventional single user application. ie it allows a user to work without being exposed to network latency delays.

Remote operations are processed in the background. They undergo transformation before being executed and appended to end of the local history buffer. Operations may appear in different orders at different sites. Despite this, the system is able to ensure that all sites converge at quiescence.

We support arbitrary changes to the connection topology. Any pair of sites can connect and successfully exchange operations. Operations are not multi-cast. Instead pairs of sites use a connection based communication channel to exchange operations. This channel is a bi-directional message queue that can be implemented easily on top of TCP.

The sender is responsible for ensuring causality preservation instead of the receiver. An operation is only sent if it is causally ready on the receiver.

Each site listens on a port for connections from other sites. New sites can be added to the system simply by connecting to an existing site. Typically sites will form an acyclic connection graph that mimics the underlying network topology.

A *delta vector time* (rather than a full vector time) is sent with every operation.

An operation that is transformed and appended to a history buffer can subsequently be sent to another site (in its transformed form). This means the precedes relation doesn't relate to the context of a received operation. We distinguish between the vector time corresponding to when the operation was generated and the vector time corresponding to its execution context.

Benefits of connection-based approach
The following are the benefits of the connection-based approach

- When the number of sites is large (eg a few hundred) sending a vector time with every operation is expensive. Given that pairs of sites exchange operations with message queues, we can send a delta vector time with every operation, providing a saving in network bandwidth.

- There is no need for multi-cast, which is difficult to implement efficiently. Multi-cast needs to address the following issues
    - Avoiding constraints on the size of operations
    - Flow control
    - Resending lost packets
    - Utilising ideal packet size when network bandwidth becomes the system bottleneck
    - Resending operations when sites go down and come back up again
    - Avoiding ack/nack implosion problems etc
    - keeping track of the members of the group
    - Accounting for the underlying network topology by fanning out packets as they propagate to the group members.
    - Deploying such a system on existing networks
- We support a dynamic system where sites may repeatedly go online and offline, or new sites are added.
- As an operation propagates through the system, it undergoes transformation, reducing subsequent CPU load when it arrives on a site - because incoming operations have a greater tendency to arrive in the right order. This is a substantial saving that should allow the system to scale far better than if all sites need to *independently* transform operations in their history buffers.
- There is no need for a site to buffer (ie delay) incoming operations because an operation is only sent when it is causally ready on the receiver.
- When computers reconnect they work out where they are up to by exchanging vector times that relate to the state of their history buffers.

Consistency model

Definition: Let O_b be an operation originally generated on site B. We write $O_a \rightarrow O_b$ if O_a was executed on B before O_b was generated on B.

Note that we don't take the transitive closure of this relation, because it simplifies the development of the theory. We are only interested in systems that preserve causality, and in that case this relation must be transitive anyway.

Definition: O_a and O_b are *concurrent* (written $O_a \parallel O_b$) if neither $O_a \rightarrow O_b$ nor $O_b \rightarrow O_a$

Definition: The *consistency model* has three requirements

Convergence	The copies of the shared documents are identical at quiescence
Causality preservation	$O_a \rightarrow O_b \quad \Rightarrow \quad O_a$ is executed before O_b on all sites
Intention preservation	For any operation O (a) At every site, the execution effect of O equals its intention (b) $O_x \parallel O \quad \Rightarrow \quad O_x$ doesn't interfere with execution effect of O

Claim: Causality preservation $\Rightarrow$ $\rightarrow$ is a transitive relation, and therefore defines a partial ordering

Proof: Suppose $O_a \rightarrow O_b$ and $O_b \rightarrow O_c$. Let O_c be generated on site C. $O_b \rightarrow O_c$ so O_b was executed on site C before O_c was generated. By causality preservation and $O_a \rightarrow O_b$, we know that O_a is executed before O_b at all sites, and in particular at site C. Therefore O_a was executed at site C before O_c was generated at site C, so we deduce $O_a \rightarrow O_c$.

Operations

For state S and operation O, S' = S+O denotes the state obtained after executing O on state S. Operation O may only be executed on state S. Therefore we define $state_{in}(O) = S$.

Let id(O) denote the site identifier of the site on which O was originally generated. It is assumed there is a total ordering on site identifiers.

Two operations O_1, O_2 are *equivalent* (written $O_1 \sim O_2$) if $state_{in}(O_1) = state_{in}(O_2) = S$ and $S+O_1 = S+O_2$. Note that this doesn't imply that the operations are equal. For example, it is possible that $id(O_1) \neq id(O_2)$.

$[O_1 ... O_n]$ denotes the list of operations $O_1,...,O_n$ assumed to be contextually serialised - ie intended to be performed in the given order on some initial state S. $S + [O_1...O_n]$ denotes the state $(((S+O_1)+O_2)+...+O_n)$.

Execution context of an operation
For operation O, the execution context ec(O) is defined to be the set of operations that have been executed before O. Operational transform is essentially about transforming an operation O into alternative forms with different execution contexts.

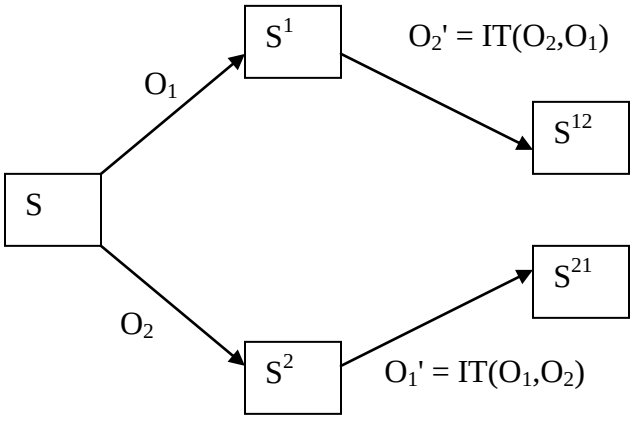
Definition of IT, ET
Let $O_1 \parallel O_2$ and $state_{in}(O_1) = state_{in}(O_2) = S$. Then we define $O_1' = IT(O_1,O_2)$ as being a transformed version of O_1 that maintains the original intention of O_1, whilst being executed in the document state following execution of O_2. ie $state_{in}(O_1') = S+O_2$. Therefore $[O_2\ IT(O_1,O_2)]$ is contextually serialised.

ET is defined to be the inverse of IT. ie whenever $O_1' = IT(O_1,O_2)$ is defined, we define $ET(O_1', O_2) = O_1$. Note that ET is only defined on contextually serialised concurrent operations

It has been shown in [3] that if IT satisfies conditions TP1, TP2 (defined below) then sites are able to execute operations in different orders yet achieve convergence at quiescence.

TP1
$\forall\ O_1, O_2$ where $O_1 \parallel O_2$ and $state_{in}(O_1) = state_{in}(O_2) = S$, $S + [O_1\ IT(O_2, O_1)] = S + [\ O_2\ IT(O_1, O_2)\ ]$

In the following diagram we require $S^{12} = S^{21}$.



TP2
$\forall\ O_1, O_2, O_3,\quad IT(\ IT(\ O_3, O_1),\ IT(O_2, O_1)) = IT(\ IT(\ O_3, O_2),\ IT(O_1, O_2))$

Causality preservation

Let the system be regarded as a Finite State Machine (FSM) consisting of sites - and over time sites can be added or removed. Let each site store a single linear history buffer (HB). Let the initial FSM contain no sites.

Let the following transitions on the FSM be defined

T1. A new site with an empty HB is added to the system

T2. A site is removed from the system

T3. A site generates a local operation, executes it and appends it to the end of its HB

T4. An operation is received from another site. *This must be the operation with the lowest HB index in the sender's HB that hasn't yet been incorporated into the receiver's HB*. The receiver transforms the operations, executes it and appends the operation to the end of its HB.

Claim : The FSM ensures Causality Preservation (CP). ie $O_a \rightarrow O_b \Rightarrow O_a$ is executed before O_b at all sites.

Proof : (by induction)

The initial state of the FSM contains no sites (and no operations) so CP is trivially true

We now show that each of the possible transitions of the FSM maintains the CP condition.

T1. Adding a site trivially maintains CP.

T2. Removing a site trivially maintains CP.

T3. Consider the FSM transition T3 of a locally generated operation O_b at site B. Let O_a be given such that $O_a \rightarrow O_b$. By definition of the precedes relation O_a was executed on B before O_b was generated on B. So O_a was executed before O_b was executed on site B. There are no other sites containing O_b, so therefore CP is maintained.

T4. Consider the FSM transition T4 of an operation O_b originally generated on site B that is sent from site X to Y. Let O_a be given such that $O_a \rightarrow O_b$. Then O_a was executed on B before O_b was generated on B. The CP condition on the previous FSM state implies that O_a was executed before O_b on all sites - and in particular on X. O_a must appear before O_b in X's HB. Therefore O_a must be sent to Y before O_b, hence O_a must be executed before O_b on Y. Therefore CP is maintained

GSNs

Each site maintains its own *Generating Sequence Number* (GSN). It is initialised to zero and is incremented each time an operation is locally generated.

An operation stores the following fields

s	The generating site identifier
t	GSN assigned to the operation when it was generated

The first operation generated by a site is assigned t=0 for its GSN.

These fields are initialised when the operation is first generated, and are sent (unchanged) with the operation as it propagates through all other sites in the system.

If a site fails it may on recovery roll back to an earlier consistent state. This could allow it to inject new operations with previous values of (s,t). This must never happen, therefore it is assumed that a site can detect roll back to an earlier state and in that case it is assigned a fresh site identifier and the GSN is reset to zero.

There is at most one operation in the system that has given (s,t) values. Let this be denoted by op(s,t).

Vector time

Usually a vector time is defined with respect to a fixed number of sites in the system. Instead, we use a formulation that naturally allows for sites to appear and disappear over time.

Definition: A *vector time* v is a set of (s,t) values where s is a site identifier and t is a positive integer, and the s values are never repeated. We write sites(v) for the set of s values in v. We write v(s) to retrieve t for given s. v(s) is defined to be zero for $s \notin sites(v)$

Definition: For vector time v, the *extent* X(v) is defined by $X(v) = \{ op(s,t) \mid t < v(s) \}$

This definition relates to the whole purpose for vector times - for a given operation O, we describe its execution context using the vector time v satisfying ec(O) = X(v). This is a summary of the complete set of operations that have been executed prior to O.

Note that for any given vector times v_1, v_2 $X(v_2) \setminus X(v_1) = \{ op(s,t) \mid v_1(s) \le t < v_2(s) \}$

Definition: For given vector times v_1, v_2 we write $v_1 \le v_2$ if $X(v_1) \subseteq X(v_2)$. This relation is reflexive, anti-symmetric and transitive and therefore defines a partial ordering on vector time. It is easy to show that

$$v_1 \le v_2 \iff \forall\, s \in sites(v_1),\ v_1(s) \le v_2(s)$$

Delta vector times

Definition: A *delta vector time* $\Delta v = v_2 - v_1$ where v_1 and v_2 are vector times is defined as follows

$$\Delta v = \{ (s,t) \in v_2 \mid t \ne v_1(s) \} \cup \{ (s,0) \mid s \in sites(v_1) \setminus sites(v_2) \}$$

We define $v_1 + \Delta v$ as follows.

$$v_1 + \Delta v = \{ (s,t) \in v_1 \mid s \notin sites(\Delta v) \} \cup \{ (s,t) \in \Delta v \mid t > 0 \}$$

Claim: $\Delta v = v_2 - v_1 \Rightarrow v_2 = v_1 + \Delta v$

We can define addition of delta vector times $\Delta v = \Delta v_1 + \Delta v_2$, satisfying (for any given vector time v)

$$(v + \Delta v_1) + \Delta v_2 = v + (\Delta v_1 + \Delta v_2)$$

Note that addition of delta vector times doesn't commute.

It can be show that

$$\Delta v_1 + \Delta v_2 = \Delta v_2 \cup \{ (s,t) \in \Delta v_1 \mid s \notin sites(\Delta v_2) \}$$

Vector time implementation

To facilitate fast look up of a vector time, a suitable implementation is a red-black tree. However a delta vector time has no need for fast look up, and therefore may be more efficiently stored using a variable size array of (s,t) pairs.

We can define an add method on a vector time in pseudo code as follows

```
v.add(s,t)
{
    if (v.hasentry(s))
    {
        v.remove(s);
    }
    if (t > 0)
    {
        v.insert(s,t);
    }
}
```

Applying a delta to a vector time simply involves calling add(s,t) for each (s,t) in the delta.

We can define an add method on a delta vector time in pseudo code as follows

```
Δv.add(s,t)
{
    if (Δv.hasentry(s))
    {
        Δv.remove(s);
    }
    Δv.insert(s,t);
}
```

This may lead to a delta that has more (s,t) entries that it needs - because it doesn't check for redundancy w.r.t the original vector time to which the delta applies.

History buffer

A *History Buffer* (HB) is a linear array h of operations. Let h[i] be the i^{th} operation in the HB, indexed from zero onwards, and |h| be the size of the HB. h[i].s is the generating site of the i^{th} operation, and h[i].t is the GSN assigned to the i^{th} operation.

For each site, we can consider all the operations in our HB generated by that site. It is a requirement that these operations are in order of assigned GSN and none are repeated or skipped. This actually makes the h[i].t values redundant, because

$$h[i].t = |\{ j < i \mid h[j].s = h[i].s \}| \qquad \text{where } |A| \text{ denotes the cardinality of set A}$$

Definition: For i = 0,...,|h| we define a vector time vt(h,i) recursively as follows

- $vt(h,0) = \phi$
- for i = 0,..., |h|-1 $vt(h,i+1) = vt(h,i) + \Delta v$ where $\Delta v = \{ (h[i].s, h[i].t + 1) \}$

This definition is for theoretical purposes only. These vector times are not explicitly stored. The following is an example with sites A,B and a history buffer with |h| = 7.

i	0	1	2	3	4	5	6	
h[i].s	A	A	B	B	A	B	A	
h[i].t	0	1	0	1	2	2	3	
vt(h,i)		(A,1)	(A,2)	(A,2) (B,1)	(A,2) (B,2)	(A,3) (B,2)	(A,3) (B,3)	(A,4) (B,3)

Claim:

1. $X(vt(h,i)) = \{ h[j] \mid j < i \} = ec(h[i])$
2. $i \le j \Rightarrow vt(h,i) \le vt(h,j)$

3. $vt(h,i) = vt(h,i+1) + \Delta v$ where $\Delta v = \{ (h[i].s, h[i].t) \}$

Definition: The *HB Vector Time* is defined as $hv = vt(h,|h|)$. This summarises all the operations stored in the HB, and for performance is explicitly stored in a practical implementation. By the third claim above we have a means to iterate *backwards* through the HB, efficiently calculating vt(h,i) as we go.

An operation O_a can't be inserted into the HB unless $hv(O_a.s) = O_a.t$. If $O_a.t > 0$, the insertion position must come somewhere *after* the existing operation h[i] in the HB satisfying $h[i].s = O_a.s$ and $h[i].t = O_a.t - 1$. In this case hv changes to $hv + \{ (O_a.s, O_a.t + 1) \}$.

Sessions

When two sites connect they establish a *session*. This uses a bi-directional message queue to send operations between the two sites. The message queue guarantees delivery of operations in the same order that they were sent, without repeats or missing operations.

An operation is only sent when it is causally ready on the receiver site. At a given point in time there may be a number of operations in the HB that need to be sent interspersed with operations that don't need to be sent). At all times the next operation to be sent (or at least processed by the receiving site) must be the operation with the lowest index position in the sender's HB.

The following state is stored in a session

rhv	The Remote HB Vector Time - an (under) estimate of the remote hv
ui	The Upload Index - an index into the local HB for the next operation to be uploaded
uv	The Upload Vector Time. uv = vt(h,ui).
U	The suffix of operations used for transforming incoming operations.

This is transient information and doesn't need to persist.

When two sites first connect they exchange hv's. An incoming hv is used to initialise the *Remote HB Vector Time* (rhv). Site A maintains $A.rhv_B$ for each adjacent site B that it exchanges operations with.

Over time, a rhv may underestimate (but never overestimate) the remote hv. ie $A.rhv_B \leq B.hv$. The rhv tells us what we *don't* need to send. $A.rhv_B \leq B.hv \Rightarrow X(A.rhv_B) \subseteq X(B.hv)$. In other words all operations in the extent of $A.rhv_B$ already reside on B.

Note that the rhv may not neatly divide the HB into two sections - ie operations on the left satisfying $vt(h,i) \leq rhv$, and operations on the right that don't.

Scanning for the ui after a connection

Consider the previous example of a history buffer

i	0	1	2	3	4	5	6	
h[i].s	A	A	B	B	A	B	A	
h[i].t	0	1	0	1	2	2	3	
vt(h,i)		(A,1)	(A,2)	(A,2) (B,1)	(A,2) (B,2)	(A,3) (B,2)	(A,3) (B,3)	(A,4) (B,3)

Let rhv = { (A,3), (B,1) }. Then the next operation to send is h[3].

Definition: Let the *Upload Index* for given rhv be defined as follows

$$ui = \max \{ i \mid 0 \le i \le |h| \text{ and } vt(h,i) \le rhv \}$$

Note that this is well defined because the set is not empty (it must contain i=0). ui is the index position of the next operation to be uploaded, or ui = |h| if no operations need to be uploaded.

The operations Ls that need to be sent are

$$Ls = X(hv) \setminus X(rhv) = \{ op(s,t) \mid rhv(s) \le t < hv(s) \}.$$

Let $P = \{ op(s,rhv(s)) \mid rhv(s) < hv(s) \}$. If $P = \phi$, then ui = |h|. Otherwise we scan backward from the end of the HB, "checking off" operations in P, until all are accounted for. The last operation that we "check off" is the first operation in Ls that needs to be sent - and therefore gives us ui.

During this scan we need to compute the upload vector time uv = vt(h,ui). The following pseudo-code shows how this can be achieved

```
OnConnect()
{
    // Calculate set of (s,t) in p
    p.clear();
    for each (s,t) in hv
    {
        tr = rhv(s);
        if (tr < t) p.add(s,tr);
    }

    // Scan backwards through HB for the upload index.  Compute the upload vector time
    ui = h.size();
    uv = hv;
    while(!p.empty())
    {
        --ui;
        s = h[ui].s;
        t = h[ui].t;
        uv.add(s,t);
        if (p.contains(s,t)) p.remove(s);
    }
}
```

Receiving an improved estimate of the rhv

Consider that a site B receives an operation O_a from site A that suggests that A has a poor estimate of B's hv. Two things should happen

1. B should use a special message to send one or more (s,t) pairs from its hv to A to improve A's poor estimate of B's hv.
2. B should update its rhv for dealing with A to make sure it doesn't send its own copy of O_a back to A.

Apart from that the incoming O_a can be disregarded.

In both cases we need to deal with an improved estimate of the rhv. This may allow the upload index to shift to the right. The following pseudo-code shows how this can be achieved.

```
ScanUploadIndexForwards()
{
    // Scan for next ui.  Update the uv
    while(ui < h.size)
    {
        s = h[ui].s;
```

```
        t = h[ui].t;
        tr = rhv(s);
        if (t >= tr) break;
        uv.add(s,t+1);
        ++ui;
    }
}
```

Updating the rhv after sending an operation

It is a good idea for the rhv to represent a reliable estimate of the set of operations on the remote HB, to avoid unnecessary sending of operations.

When site A sends an operation O_a to B, A should set A.rhv = A.rhv + { (O_a.s, O_a.t + 1) }. This will shift the ui one or more positions to the right. The following pseudo-code shows how this is achieved

```
SendOperation(Oa)
{
    < Send Oa, and delta uv >

    // Update rhv to account for sending operation Oa
    rhv.add(Oa.s, Oa.t + 1);

    ScanUploadIndexForwards();
}
```

Updating rhv when operations are received from that site

To avoid site B reflecting an incoming operation O_a from A back to A, it is necessary to update B.rhv to account for operations received from A. This is achieved as follows

```
OnReceiveOperation(Oa)
{
    if (rhv(Oa.s) ≤ Oa.t) rhv.add(Oa.s, Oa.t + 1);
}
```

Use of the suffix U

A session uses a suffix U to transform operations received by the session from the remote site.

The suffix is not calculated until the first operation is received. Therefore a Session stores a boolean flag initU to indicate whether the suffix has been initialised.

When the operation o with vector time v is received, it is necessary to calculate the suffix U for the session, then dual IT o and U. Operation o can then be executed and appended to the local history buffer. It is also necessary to notify all other sessions that an operation has been appended to the HB. This allows the other sessions to append the operation to their suffix U.

```
bool Session::ProcessNextReceivedOp(Operation& o, VectorTime& v)
{
    if (!ExtentContainsOp(site.hv,o))
    {
        if (initU)
        {
            gfnCalculateSuffix(v,U);
        }
        else
        {
            site.hb.CalculateSuffix(v, U);
            initU = true;
        }
        gfnDualIT(o,U);
        site.ExecuteAndAppendOp(this, o);
    }
}
```

A session is notified each time an operation is appended to the local HB. The notification is provided with the identity of the session that caused an operation to be appended, or NULL to indicate that it was a local operation.

```
void Session::OnAppendOp(Session* session, const Operation& o)
{
    // Local operations have session = NULL
    // We treat incoming operations from a different session like local operations. These needs to
    // be appended to U.  It is important to clone the operation or else transformations on U will
    // affect the operations in the HB.
    if (initU && session != this)
    {
        U.append(o.Clone());
    }
}
```

Avoiding the need to send a vector time with every operation

When an operation O_a = A.h[i] is sent from site A to another site B, it is necessary for B to understand the context of the operation. This is described by vt(A.h,i).

Given the use of message queues between a pair of sites, it is possible for A to only send delta vector times on each operation that is sent. B will be able to compute the corresponding vector time by applying the delta to the previous vector time. Let $B.session_A.av$ be the vector time stored by B for its session with A, for the purpose of accumulating deltas. When B first connects to A and receives A.hv, $B.session_A.av$ can be initialised to A.hv.

In practice it is expected that vector times of sent operations will vary slowly - ie a delta vector time will contain a small number of entries, so this approach will provide a dramatic saving in network bandwidth when the total number of sites in the system is large.

References

[1]	David Barrett-Lennard. Operational transform - Single character insert and delete operations. Jul 2005
[2]	David Barrett-Lennard. Operational transform - Assignment operations. Aug 2005