A Counterexample to the Distributed Operational Transform and a Corrected Algorithm for Point-to-point Communication

Gordon V. Cormack

Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Abstract

The distributed operation transform (dOPT) is proposed by Ellis and Gibbs as a lock-free algorithm to ensure the consistency of concurrently updatable distributed objects. dOPT is shown by counterexample to be incorrect. A corrected algorithm is given for a restricted environment based on point-to-point communication. There appears to be no simple correction to dOPT for a general environment based on broadcast communication.

Keywords: Distributed computing, distributed algorithm, concurrency control, causal ordering, groupware.

Introduction

The Distributed Operational Transform Algorithm (dOPT) is proposed by Ellis and Gibbs (1989) as a concurrency control mechanism for groupware systems. dOPT is attractive beyond the domain of groupware systems because it promises to ensure consistency among distributed replicated objects without requiring serialization or even serializability among updates at different locations.

The essence of dOPT is that all update messages are delivered to each site; each site executes the messages in some order consistent with Lamport's *happened-before* relation (1978). Because *happened-before* is a partial ordering, different sites may execute the messages in different orders. An *operation-transform* function T is used to ensure that the various orders yield an identical result. The definition of T embodies the concurrent semantics of a particular replicated object.

dOPT assumes an underlying reliable but unordered broadcast-based communication system and correctly computes at each site the execution order consistent with *happened-before* that results in minimal delay. In particular, messages generated at a particular site are always executed at that site without delay, and execution at other sites is delayed only to await the arrival of in-transit messages. This component of dOPT is identical to the lightweight CBCAST implementation published later by Birman, Schiper, and Stephenson (1991).

dOPT requires that each update message be given a priority such that no pair of messages from different sites have the same priority. The operation transform function T is defined for all pairs of updates u_1 and u_2 and for all pairs of unequal priorities p_1 and p_2. $u_1' = T(u_1, u_2, p_1, p_2)$ is a new update, in some sense equivalent to u_1 that is executed after u_2 although u_2 did not happen before u_1. T must have the following consistency property: the execution of

$$u_2 \text{ followed by } T(u_1, u_2, p_1, p_2)$$

must yield an identical result to the execution of

$$u_1 \text{ followed by } T(u_2, u_1, p_2, p_1).$$

Ellis and Gibbs give a set of updates and a consistent T amenable for use in a shared text editor. Several schemes are suggested for assigning priorities, including a simple encoding of site identifiers, and a more complex encoding based on historical information. Ellis and Gibbs reject the simpler scheme in favour of the more complex, citing a 3-site counterexample.

This paper presents a 2-site counterexample in which dOPT fails for any possible priority scheme. We suggest a correction to dOPT that uses the simpler priority scheme, but is suitable only for two sites connected by a point-to-point communication channel. Using several point-to-point connections to form a tree, it is easy to derive a consistent solution for an arbitrary number of sites. There appears to be no simple and efficient correction to dOPT that maintains its suitability for broadcast-based applications. A more formal development of the point-to-point and broadcast-based algorithms appears elsewhere (Cormack 1995).

The dOPT algorithm

The dOPT algorithm maintains a copy of a shared object at each site; each copy is initially identical. Each site S runs an identical algorithm (see figure 1) which accepts messages denoting the following events:

- a local update generated at site S
- a remote update generated at another site $s \neq S$.

`gvcormac@plg.uwaterloo.ca`

University of Waterloo Technical Report CS-95-08

For each local update, the algorithm modifies its copy of the object and transmits a message denoting the update to all other sites. Each remote update message is held until all *happened-before* updates have been processed; then the update is transformed to compensate for non-*happened-before* updates already executed at *S*; finally the algorithm updates its copy of the object to reflect the transformed update.

Ellis' and Gibbs' specific definitions for a shared text editor are now described. The shared object is a variable-length array of characters, and the update operations consist of:

insert[c,i]	inserts a character c at position i in the object, first moving every character at position i and beyond to the right by one position.
delete[i]	deletes the character at position i, moving every character beyond position i to the left one position.
nop	does not change the object.

The operation transform (figure 2) shifts the position of insertion or deletion so that the same character is affected. In the event of two deletions at the same place, or two insertions of the same character at the same place, the second update is annulled by transforming it into the identity function nop.

A counterexample to dOPT

Consider a text object as defined above with an initial value of abcdefg (figure 3). Site 1 deletes the a while concurrently site 2 deletes the a and then the e. That is, site 1 takes on the value bcdefg, site 2 takes on the value bcdfg, and three messages are transmitted:

```
Remote_update(1, (0,0), delete[1], p1)
```

from site 1 to site 2, and

```
Remote_update(2, (0,0), delete[1], p2)
Remote_update(2, (0,1), delete[4], p3)
```

from site 2 to site 1. The priority values p1, p2 and p3 may be chosen arbitrarily. When the message destined for site 2 arrives, its update is transformed to nop, resulting in a final value at site 2 of bcdfg. When the first message destined for site 1 arrives, its update is also transformed to nop, leaving site 1's value at bcdefg. The second message is transformed (inappropriately) to delete[3], resulting in a final value at site 1 of bcefg. Since there are no messages in transit and the two sites are unequal, the algorithm is incorrect.

A corrected point-to-point algorithm

A correct, restricted variant of the algorithm is obtained by replacing components of the algorithm as shown in figure 4. SIT is restricted to two values, and the priority of an update is simply its site id. The major modification is that each log entry is itself transformed whenever it is used to transform an incoming update. Two optimizations are occasioned by the following observations: unused log entries with indexes less than or equal to V[s]+vS will never be used and may be deleted; remote updates will not participate in transforming later remote updates and therefore need not be logged. These optimizations make the algorithm quite efficient: the size of the log is bounded by the number of unacknowledged messages, and the transformation time for an update is proportional to the size of the log. Finally, the algorithm may easily be enhanced to compensate for an unreliable communication channel: (1) any missing message will delay the next message indefinitely, and a suitable timeout and retransmission protocol may be implemented within the

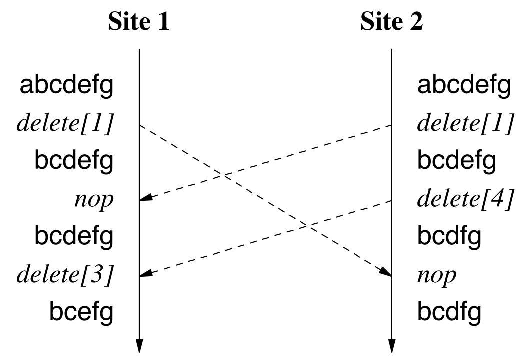
Figure 1. dOPT algorithm.

```
Type
  SIT = { the set of site ids }
  OBJ = { the type of the shared object }
  UPD = OBJ → OBJ  { an update function }
  VEC = ARRAY SIT OF INTEGER  { state vector }
  LOG = LIST OF SIT × VEC × UPD × PRI
  PRI = { priority }

Const
  S: SIT { the site id for this site }
  $X_0$:  OBJ { the initial value }
  T: UPD × UPD × PRI × PRI → UPD  { transform }
  P: SIT × UPD × ... → PRI  { priority }

Var
  X: OBJ := $X_0$  { local copy of object }
  L: LOG := $\phi$
  V: VEC := (0, 0, ... 0)

Event
  Local_update(U: UPD) =
    Broadcast Remote_update(S, V, U, P(S,X, ... ))
    L := L <S, V, U, P(S,X, ... )>
    X := U(X)
    V[S] := V[S] + 1

  Remote_update(s: SIT, v: VEC, u: UPD, p: PRI) =
    Delay until V[i] ≥ v[i] for all i: SIT
    For each <s',v',u',p'> in L
      If v'[s'] ≥ v[s'] then
        u := T(u,u',p,p')
    L := L <s, v, u, p>
    X := u(X)
    V[s] := V[s] + 1
```

Figure 2. Operation transform for text editor.

```
T(insert[c,i], insert[c',i'], p, p') =
  insert[c,i] if i < i'
  insert[c,i+1] if i > i'
  nop if i = i' and c = c'
  insert[c,i] if i = i' and p < p'
  insert[c,i+1] if i = i' and p > p'

T(delete[i], delete[i'], p, p') =
  delete[i] if i < i'
  delete[i-1] if i > i'
  nop if i = i'

T(insert[c,i], delete[i'], p, p') =
  insert[c,i] if i ≤ i'
  insert[c,i-1] if i > i'

T(delete[i], insert[c',i'], p, p') =
  delete[i] if i < i'
  delete[i+1] if i ≥ i'

T(nop, u, p, p') = nop

T(u, nop, p, p') = u
```

Figure 3. dOPT Counterexample.



delay; (2) any duplicate message will have the property that V[s] > v[s] and may be ignored.

Consistency of the point-to-point algorithm

At any instant, it is impossible to determine whether or not there are messages in transit. Nevertheless, the algorithm is shown consistent by demonstrating that at all times, *if* there are no messages in transit, the value of X at site s is equal to the value of X at the other site $\overline{s}$.

At any given time, site s will have executed, in some sequence, V[S] local updates and V[s] remote updates.

Figure 4. Point-to-point algorithm.

```
Type
  SIT = {1, 2}
  OBJ = { the type of the shared object }
  UPD = OBJ → OBJ  { an update function }
  VEC = ARRAY SIT OF INTEGER  { state vector }
  LOG = ARRAY [1 ..  ] OF UPD { local update log }

Const
  S: SIT { the site id for this site }
  X_0:  OBJ { the initial value }
  T: UPD × UPD × PRI × PRI → UPD  { transform }

Var
  X: OBJ := X_0  { local copy of object }
  V: VEC := (0, 0)  { local state vector }
  vS: INTEGER := 0  { last v[S] from other site }
  L: LOG  { equivalent update sequence }

Event
  Local_update(U: UPD) =
    V[S] := V[S] + 1
    Broadcast Remote_update(S, V, U)
    L[V[s]+V[S]] := U
    X := U(X)

  Remote_update(s: SIT, v: VEC, u: UPD) =
    Delay until v[s] = V[s] + 1
    L[V[s]+v[S]+1..V[s]+V[S]+1] :=
      L[V[s]+v[S]..V[s]+V[S]]
    L[V[s]+v[S]] := u
    For k := V[s]+v[S]+1 to V[s]+V[S]+1 do
      Let U = L[k]
      L[k] := T(U, u, S, s)
      u := T(u, U, s, S)
    vS := v[S]
    V[s] := V[s] + 1
    X := u(X)
```

Also, site $\overline{s}$ is known to have executed at least V[s] local updates and vS updates from s. Thus, the most recent V[S] - vS are not known to have been executed at $\overline{s}$. The log entries L[V[s]+vS+1..V[s]+V[S]] represent this sequence of updates. When a remote update arrives, it is known that that update was executed at $\overline{s}$ *before* this sequence, and it must be transformed over the sequence. Furthermore, the sequence must be transformed over the update.

We extend the definition of T to apply to sequences:

$$T(U, u_1 u_2 \cdots u_k, S, s) =_{def} T(T(U, u_1, S, s), u_2 \cdots u_k, S, s)$$

$$T(u_1 u_2 \cdots u_k, U, s, S) =_{def} T(u_1, U, s, S)\, T(u_2 \cdots u_k, T(U, u_1, S, s), s, S).$$

We denote by $u_1 u_2 \cdots u_k(X) =_{def} u_k(u_{k-1}(\cdots u_1(X)\cdots))$ the application of a sequence of updates. Two update sequences $u_1 \cdots u_k$ and $u'_1 \cdots u'_{k'}$ are equivalent if, for all X, $u_1 \cdots u_k(X) = u'_1 \cdots u'_{k'}(X)$. Obviously, equal sequences are equivalent, and substitution of equivalent subsequences preserves equivalence.

Theorem 1 – Extended consistency of T. For any update U, any sequence of updates $u_1 u_2 \cdots u_k$ where $k \geq 1$, and any $S \neq s$, the following two sequences are equivalent:

$$u_1 u_2 \cdots u_k\, T(U, u_1 u_2 \cdots u_k, S, s)$$
$$U\, T(u_1 u_2 \cdots u_k, U, s, S)\,.$$

Proof. For $k = 1$ the theorem holds because of the (unextended) definition of T. For $k = m > 1$, we assume the hypothesis that the theorem holds for all $1 \leq k < m$. By substitution of equivalent subsequences, the following pairs of update sequences are equivalent:

$$u_1 u_2 \cdots u_k\, \underline{T(U, u_1 u_2 \cdots u_k, S, s)}$$
$$u_1 u_2 \cdots u_k\, \underline{T(T(U, u_1, S, s), u_2 \cdots u_k, S, s)} \qquad \text{[defn. T]}$$

$$u_1 \underline{u_2 \cdots u_k\, T(T(U, u_1, S, s), u_2 \cdots u_k, S, s)}$$
$$u_1\, \underline{T(U, u_1, S, s)\, T(u_2 \cdots u_k, T(U, u_1, S, s), s, S)} \qquad \text{[hyp.]}$$

$$\underline{u_1\, T(U, u_1, S, s)}\, T(u_2 \cdots u_k, T(U, u_1, S, s), s, S)$$
$$\underline{U\, T(u_1, U, s, S)}\, T(u_2 \cdots u_k, T(U, u_1, S, s), s, S) \qquad \text{[defn. T]}$$

$$U\, \underline{T(u_1, U, s, S)\, T(u_2 \cdots u_k, T(U, u_1, S, s), s, S)}$$
$$U\, \underline{T(u_1 u_2 \cdots u_k, U, s, S)}\,. \qquad \text{[defn. T]}$$

□

Theorem 2. Consistency of algorithm. Whenever both sites have executed the same n updates their values of X are equal.

Proof. We maintain the invariant that the sequence L[1..V[s]+vS] is equivalent to the same sequence at the other site. As a consequence, if there are no messages in transit, L[1..V[s]+V[S]] must be equivalent to the same sequence at the other site. Finally, we show the invariant that X = L[1..V[s] + V[S]](X_0) and therefore the values of X are equal. □

CoEd

CoEd is a conference editor implemented by Holtz (1991) which contains an unpublished variant of dOPT. Holtz attempts to keep each log sorted according to a common order, and inserts remote updates in a position so as to maintain the order. Although CoEd was developed independently, its treatment of the log resembles the corrected algorithm presented here. However, it fails the counterexample in the same manner as dOPT.

Elfe

Vadura (1984) has used the corrected algorithm presented here and EMACS to implement Elfe, a distributed conference editor for files shared over a network. Using Elfe, a file to be edited is specified using a Uniform Resource Locator. Any number of users may view and update the file, subject to the restrictions imposed by a security system. All updates are effected immediately within the editor, and are transmitted to all other users where they are applied in a consistent manner as enforced by the algorithm. The shared file behaves as a shared bulletin board that is insensitive to network delay.

Acknowledgements

The author thanks Charlie Clarke for many useful suggestions. Thanks also to Brian Holtz for his C++ implementation of CoEd, and to Dennis Vadura for reverse-engineering the CoEd algorithm from its implementation. This research is supported by the Natural Sciences and Engineering Research Council of Canada.

References

Birman K.B., Schiper A. and Stephenson P. (1991), *Lightweight causal and atomic group multicast*, **ACM TOCS 9:3**, 272-314.

Cormack G.V. (1995), *A calculus for concurrent update*, University of Waterloo CS-95-06. http://plg.uwaterloo.ca/~gvcormac/ccu.ps

Ellis C.A. and Gibbs S.J. (1989), *Concurrency control in groupware systems*, Proc. 19 ACM SIGMOD Conference on Management of Data, in **ACM SIGMOD Record 18:2**, 399-407.

Holtz B. (1991) *CoEd – A shared text editor*, Sun Microsystems. ftp://plg.uwaterloo.ca/pub/CoEd.

Lamport L. (1978), *Time, clocks, and the ordering of events in a distributed system*, **Commun. ACM 21:7**, 558-565.

Vadura D. (1994) *Elfe – Editing Live Files Everywhere*, University of Waterloo. ftp://plg.uwaterloo.ca/pub/elfe.