A Calculus for Concurrent Update

Gordon V. Cormack

Department of Computer Science, University of Waterloo

1. Introduction

This paper introduces a calculus for concurrent update (CCU) that is used to specify distributed objects. The calculus permits updates to be effected immediately at each site − no central server, locking, token passing, rollback, or other form of serialization is enforced. Notice of each update at each site is transmitted to every other site, where a corresponding update is effected. Unless special provisions are taken, network transmission delay may cause corresponding updates to be effected at different sites in different orders, potentially rendering them meaningless or inconsistent. CCU avoids this eventuality: transformations are applied to corresponding updates as necessary to preserve overall meaning and consistency.

CCU derives from the Distributed Operational Transform (dOPT) algorithm proposed by Ellis and Gibbs (1989) as a concurrency control mechanism for groupware systems. dOPT introduces the notion of consistency-preserving transformations, and embeds exactly the lightweight CBCAST algorithm published later by Birman, Schiper and Stephenson (1991). However, Ellis and Gibbs merely allude to the need to preserve meaning, and present no method for reasoning about either the overall consistency or meaningfulness of the transforms as applied by dOPT. Indeed, dOPT is incorrect, as demonstrated by a counterexample in which it fails to maintain consistency (appendix A).

In 1991, the author implemented a conference editor and discovered the error in dOPT, which was corrected for the special case of two sites. It was necessary to develop CCU to prove the algorithm meaningful and consistent, to generalize it beyond two sites, and to extend its domain of application beyond simple text editors.

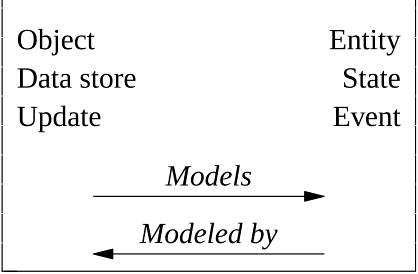
Waterloo, Ontario N2L 3G1, Canada
`gvcormack@plg.uwaterloo.ca`

Research report CS-95-06

2. Objects as Models

An *object* is a *data store* with a set of functions that models some abstract *entity*. Each value of the store models a *state* of the entity. Each *update* − the application of a function to modify the store − models an *event* that affects the state of the entity. The object models the entity if its store initially models the entity's state, and updates are performed that model all events affecting the entity (figure 1).

Figure 1. Modeling an entity.



In a sequential model, updates model events in the order that they occur. Each update modeling an event relies on the the store modeling the state before the event, and ensures that the store models the state after the event. It is reasonably well understood how to design sequential models for sequential events.

It is less well understood how to model concurrent events − events occurring from distinct sources for which it is impossible to determine a temporal order. One approach is to use a sequential model, which dictates that an artificial order must be imposed on the modeling of concurrent events. This serialization may introduce unwanted delay: the update corresponding to an event can be applied only after it is determined that there could be no unmodeled event earlier in the order.

A concurrent model provides an alternative to serializing concurrent events. In a concurrent model, the update modeling an event may not rely on the store modeling the state immediately before the event. The store may, in fact, model a state that differs by the effect of several events. It is necessary to define how updates should be applied in this situation.

This paper presents a calculus for defining concurrent models. A concurrent model is specified by a sequential model augmented with definitions for the concurrent application of all possible pairs of elementary updates. From these definitions is derived a consistent definition for all possible concurrent update sequences.

The concurrent model is implemented by a set of objects: one for each source of events. Each event is modeled at its source by an immediate update to the corresponding object. A notice of the update is transmitted to all other objects, which are updated to model the same event as defined by the calculus. Although the objects have different update orders, each is a model for the same entity.

3. Modeling Sequential Events

We wish to model some abstract entity that takes on states $\{S_i\}$ and is affected by events $\{E_i\}$. The effect of an event E_i on an entity with state S_i is to transform it to a new state S_j − this transformation is denoted $S_i < E_i > S_j$. The model is constructed from an object whose store takes on values $\{X_i\}$ manipulated by update functions $\{F_i\}$. An update, denoted $F_i\colon X_i$ is the replacement of a specific data value X_i by $F_i(X_i)$.

The sequential model maintains a homomorphic mapping from values to states, and from updates to events. This mapping is denoted M: $M(X_i) = S_i$ means that the value X_i models the state S_i; $M(F_i\colon X_i) = E_i$ means that the update $F_i\colon X_i$ models the event E_i. By definition, if $S_i < E_i > S_j$ and $M(X_i) = S_i$ and $M(F_i\colon X_i) = E_i$ it must be the case that $M(F_i(X_i)) = S_j$. Therefore, the state S_n resulting from the transformation of S_0 by the sequence of events $E_1 E_2 \cdots E_n$ on S_0, denoted

$$S_0 < E_1 E_2 \cdots E_n > S_n,$$

is modeled by

$$F_n(F_{n-1}(\cdots F_1(X_0)))$$

where $M(X_0) = S_0$ and

$$M(F_i\colon F_{i-1}(\cdots F_1(X_0))) = E_i \quad (0 < i \le n).$$

U is a *valid update sequence with respect to* X_0 if

$$U = F_1\colon X_0 \;\; F_2\colon X_1 \;\; \cdots F_n\colon X_{n-1}$$

where $X_i = F_i(X_{i-1}) \;\; (0 < i \le n)$. The result of applying U to X_0 is

$$U(X_0) = F_n(F_{n-1}(\cdots F_1(X_0))) \,.$$

U models the sequence of events $E_1 E_2 \cdots E_n$ where

$$E_i = M(F_i\colon X_{i-1}) \;\; (0 < i \le n).$$

In addition, we know that

$$M(X_0) < E_1 E_2 \cdots E_n > M(U(X_0)).$$

4. Modeling Pairs of Concurrent Events

Consider two concurrent events E_1 and E_2 affecting a common state S_0. Individually, E_1 and E_2 transform S_0 to S_1 and S_2 respectively; that is, $S_0 < E_1 > S_1$ and $S_0 < E_2 > S_2$. Since the relative order of E_1 and E_2 is unknown, the combined effect of E_1 and E_2 might be to transform S_0 to either S_{12} or S_{21} where $S_0 < E_1 E_2 > S_{12}$ and $S_0 < E_2 E_1 > S_{21}$. It is not true in general that $S_{12} = S_{21}$ − the combined meaning of concurrent events must be defined in terms of some arbitrary *event order*.

Our model for the combined effect of concurrent events is built from the sequential model for each event. Consider F_1, F_2 and X such that $E_1 = M(F_1\colon X)$ and $E_2 = M(F_2\colon X)$ and $S_0 = M(X)$. One might consider constructing the update sequence $F_1\colon X \;\; F_2\colon X$, but this sequence is not valid with respect to X because it is not true in general that $F_1(X) = X$. Another approach might be to construct the sequence $F_1\colon X \;\; F_2\colon F_1(X)$ − this sequence may be valid but does not model $E_1 E_2$ because it is not true in general that $M(F_2\colon X) = M(F_2\colon F_1(X))$.

If we find F' such that $M(F'\colon F_1(X)) = M(F_2\colon X)$, we can model $E_1 E_2$ by the update sequence $U = F_1\colon X \; F'\colon F_1(X)$. We can model S_{12} where $S_0 < E_1 E_2 > S_{12}$ by $U(X_0)$. One approach to finding a suitable F' might be to require the implementor to provide an axiomatic definition of M, and to use this definition to derive a suitable F'. We do not regard this as a viable approach, as M is an intangible mapping only tacitly defined by a sequential object.

Our approach is based on an axiomatization of the equivalence relation $=_M$ defined as follows:

$$F_i\colon X_j =_M F_k\colon X_m$$

means that

$$M(F_i\colon X_j) = M(F_k\colon X_m) \,,$$

and

$$F_1\colon X_1 \cdots F_n\colon X_n =_M G_1\colon Y_1 \cdots G_n\colon Y_n$$

means that

$$F_i\colon X_i =_M G_i Y_i \;\; (0 < i \le n) \,.$$

That is, we do not have to define the meaning of updates; we define instead equivalence classes of updates having the same meaning. Using this definition we construct a common model for concurrent events from separate sequential models for the same events.

A second equivalence relation $=_X$ relates update sequences with a common effect. We say that

$$U_1 =_X U_2$$

if U_1 and U_2 are valid update sequences with respect to the same value X and

$U_1(X) = U_2(X)$.

$=_X$ is used to construct alternative models for the state of an object.

A *canonical* model for $E_1 E_2 \cdots E_n$ is a valid update sequence

$U_1 = F_1 \colon X_0 \;\; F_2 \colon X_1 \;\; F_n \colon X_{n-1}$

where

$M(F_i \colon X_{i-1}) = E_i \;\; (0 < i \le n)$.

It follows from the definition of $=_M$ that any $U_2 =_M U_1$ is also a canonical model for the same event sequence. Any other other update sequence $U_3 =_X U_2 =_M U_1$ is a *noncanonical* model for the same event sequence, as it has exactly the same effect on the store as the canonical model U_2.

4.1. Axioms for $=_M$

A definition of the operator / (read *after*) must be specified for all pairs of update functions F_1 and F_2 such that

$F_1/F_2 \in \{F_i\}$ and for all $X \in dom(F_1) \cap dom(F_2)$

$F_1 \colon X =_M F_1/F_2 \colon F_2(X)$.

That is, $F_1/F_2 \colon F_2(X)$ is defined to model the same event as $F_1 \colon X$ after the update $F_2 \colon X$.

Consider two concurrent events E_1 and E_2 modeled separately as $F_1 \colon X$ and $F_2 \colon X$. Building a common model requires that the events be sequenced: a canonical model for $E_1 E_2$ is

$F_1 \colon X \;\; F_2/F_1 \colon F_1(X)$,

while a canonical model for $E_2 E_1$ is

$F_2 \colon X \;\; F_1/F_2 \colon F_2(X)$.

In a canonical model, updates modeling individual events are applied in the chosen event order.

4.2. Axioms for $=_X$

The concurrent model relies on being able to construct noncanonical models in which updates corresponding to events are applied in a different order. Such update sequences are specified using a dual operator \ (read *before*), defined in terms of / such that for all F_1 , F_2, and X

$F_1/F_2(F_2(X)) = F_2 \backslash F_1(F_1(X))$;

that is,

$F_1 \colon X \;\; F_2/F_1 \colon F_1(X) =_X F_2 \colon X \;\; F_1 \backslash F_2 \colon F_2(X)$.

The first update sequence models the events in canonical order; that is, updates modeling events are applied in one-to-one correspondence and in the same order as the events they model. The second sequence is noncanonical: its first update models E_2; its second update does not model E_1 − rather it transforms the data store as if E_1 had been modeled before E_2. Such noncanonical update sequences are the essence of the concurrent model - it is possible to model concurrent updates in various orders while ensuring equivalence with a canonical update sequence.

5. A Concrete Example - A Shared Text Buffer

Figure 2 gives / and \ modeling concurrent edits to a text buffer, as used in a simple text editor. The data store is a variable-length array of characters with two parameterized update functions:

Insert(*position*, *string*)
Delete(*position*, *length*)

Insert adds *string* to the array at *position*, moving the following characters to make room. Delete removes *length* characters from the array starting at *position*. While a text editor typically offers the user a large number of editing operations, each may be reduced to some combination of Insert and Delete.

The buffer

"the brown fox jumps over the dog"

may be subject to the concurrent updates

Insert(5, "quick ")
Insert(30, "lazy ")

intended to change the buffer to

"the quick brown fox jumps over the dog"

and

"the brown fox jumps over the lazy dog"

respectively. Informally, the events modeled by these updates are

insert the word quick *between* the *and* brown

and

insert the word lazy *between* the *and* dog.

The result of modeling this pair of events in either order should be

"the quick brown fox jumps over the lazy dog".

If the updates are applied in the above order without modification, the result is inappropriate:

"the quick brown fox jumps ovelazy r the dog".

To model the second event correctly, it is necessary to transform the second update:

Insert(30, "lazy ") / Insert(5,"quick ")
  = Insert(36, "lazy ") .

Events are not always commutative. Consider the pair of updates

Insert(5,"quick ")
Insert(5,"sly ")

modeling the concurrent events inserting quick and sly at the same position in the buffer.

The result of modeling these events might be

"the quick sly brown fox jumps over the dog"

Figure 2. / and \ for text buffer.

Insert(p1,s1) / Insert(p2,s2) =
  Insert(p1,s1) $(p1 < p2)$
  Insert(p1+|s2|,s1) $(p2 \le p1)$

Insert(p1,s1) \ Insert(p2,s2) =
  Insert(p1,s1) $(p1 \le p2)$
  Insert(p1+|s2|) $(p2 < p1)$

Delete(p1,l1) / Delete(p2,l2) =
  Delete(p1,l1) $(p1 + l1 \le p2)$
  Delete(p1,p2-p1)
    $(p1 \le p2 \le p1 + l1 \le p2 + l2)$
  Delete(p1,l1-l2) $(p1 \le p2 \le p2 + l2 \le p1 + l1)$
  Delete(p2,0) $(p2 \le p1 \le p1 + l1 \le p2 + l2)$
  Delete(p2,p1+l1-p2+l2)
    $(p2 \le p1 \le p2 + l2 \le p1 + l1)$
  Delete(p1-l2,l1) $(p2 + l2 \le p1)$

Delete(p1,l1) \ Delete(p2,l2) =
  Delete(p1,l1) / Delete(p2,l2)

Delete(p1,l1) / Insert(p2,s2) =
  Delete(p1,l1) $(p1 + l1 \le p2)$
  Delete(p1,l1+|s2|) $(p1 \le p2 < p1 + l1)$
  Delete(p1+|s2|,l1) $(p2 < p1)$

Insert(p1,s1) / Delete(p2,l2) =
  Insert(p1,s1) $(p1 \le p2)$
  Insert(p2,"") $(p2 < p1 < p2 + l2)$
  Insert(p1-l2,s1) $(p2 + l2 \le p1)$

Delete(p1,l1) \ Insert(p2,s2) =
  Delete(p1,l1) / Insert(p2,s2)

Insert(p1,s2) \ Delete(p2,l2) =
  Insert(p1,s2) / Delete(p2,s2)

or
  "the sly quick brown fox jumps over the dog"
depending on the arbitrary order chosen for the concurrent events. Using the definition of / in figure 2, the stated event order yields the first result, and the opposite order yields the second. The use of \ allows the update order to be reversed while yielding the same result.

The case of concurrent deletions is handled in a similar manner, except that overlapping deletions are modified to eliminate the overlap: Delete events are commutative and therefore the definitions of / and \ are identical.

Finally, deletions and insertions may be concurrent with one another. In the case that the deleted interval contains the insert position, the insertion is nullified; otherwise the position of the update is shifted as necessary. Deletions and insertions are commutative with respect to one another. For example, consider the updates
  Delete(21,13)
  Insert(30,"sly ")
modeling the events
  delete "over the dog "
  insert sly *between* the *and* dog .
Regardless of event order, the deletion extends to include the insertion of sly, resulting in
  "the brown fox jumps"

These axioms specify precisely a reasonable model for concurrent text buffer edits. Other definitions are possible for text buffers and for other objects. It is impossible to say that any particular set of axioms is right or wrong; different axioms merely model different abstractions. As in the design of sequential objects, it is the responsibility of the implementor to ensure that the model aptly represents the intended abstraction.

6. Modeling sets of events

Consider the set of events $\{E_i\}$ modeled individually by elements of $\{F_i: X_i\}$. We wish to construct a common model for all the events according to some specific event order. It is convenient to denote the event order by affixing to each update a unique timestamp T, totally ordered by the relation $<$; $T_1 < T_2$ indicates that the event modeled by $F_1: X_1: T_1$ precedes the event modeled by $F_2: X_2: T_2$ in the event order. In all other respects $F: X: T$ represents the same update as $F: X$.

Let W be a set of updates with unique timestamps. W denotes the event sequence $E_1 E_2 \cdots E_n$ where
  $E_i = M(F_i: X_i: T_i)$ $(0 < i \le n)$ and
  $T_i < T_{i+1}$ $(0 < i < n)$ and
  $W = \{F_i: X_i: T_i\}$.
A valid sequence U is a *canonical form* for W if
  $U = F'_1: X'_1: T_1 \cdots F'_n: X'_n: T_n$
where
  $X'_1 = X_1$ and
  $F'_i: X'_i: T_i =_M F_i: X_i: T_i$ $(0 < i \le n)$ and
  $T_i < T_{i+1}$ $(0 < i < n)$ and
  $W = \{F_i: X_i: T_i\}$.

U clearly models the set of events denoted by W, as does any $U' =_X U$. Unfortunately, it is not possible to find a canonical form for arbitrary W. The algorithm manipulates only W known to have a specific canonical form, denoted $[W]$ (figure 5, section 7.3).

7. The concurrent model

The concurrent model is implemented by one object per source of events. Each object models directly the events from a particular source. In addition, each object transmits its updates to all other objects. Each object merges its own updates with those of the other objects. Each object thus converges toward a common model for all events from all sources.

An implementation framework for the concurrent model involves the execution of the same algorithm for each object (figure 3). Timestamps must have two orderings: the partial ordering $\subset$ must implement exactly Lamport's (1978) *happened before* relation, and the total ordering $<$ may be chosen arbitrarily provided it is consistent with $\subset$. T_0 is the minimal timestamp; that is, $T_0 < T$ and $T_0 \subset T$ for all $T \neq T_0$.

Figure 3 . CCU Algorithm at Site s.

```
Initialization:
    $X^s \leftarrow X_0$      − data store
    $W^s \leftarrow \{\}$      − update history
    $T^s \leftarrow T_0$      − timestamp

Occurrence of local event $E$:
    determine $F$ such that $M(F\colon X^s) = E$
    compute globally unique $T$ such that
        $T^s \subset T$
        $T_i \not\subset T \ \ (F_i\colon X_i\colon T_i \notin W^s)$
    transmit $T^s$, $T$, $F$ to other sites
    $W^s \leftarrow W^s \cup \{F\colon X^s\colon T\}$
    $X^s \leftarrow F(X^s)$
    $T^s \leftarrow T$

Receipt of message $T^r$, $T$, $F$:
    delay until $T^r \subseteq T^s$
    $W^s \leftarrow W^s \cup \{F\colon X^r\colon T\}$
        where $X^r = [\{F_i\colon X_i\colon T_i \in W^s \mid T_i \subseteq T^r\}]\,(X_0)$
    $X^s \leftarrow [W^s](X_0)$
    compute $T'$ such that
        $T^s \subseteq T'$
        $T \subseteq T'$
        $T_i \not\subset T' \ \ (F_i\colon X_i\colon T_i \notin W^s)$
    $T^s \leftarrow T'$
```

Several issues must be addressed in order to transform this conceptual implementation into concrete algorithms and data structures. T must be computed so as to generate globally unique timestamps with the total order $<$ (event order) and the partial order $\subset$ (causal order). A delay mechanism must hold the processing of updates whose causal prerequisites have not yet arrived due to transmission latency. A reliable delivery mechanism must ensure that all transmitted messages are eventually received by all sites, albeit with arbitrary delay and arrival order. Finally, an algorithm must be found to compute $U(X_0)$ where U is a canonical form for W^s. As mentioned previously, it is not possible to compute a canonical form for arbitrary W, but a solution $[W^s]$ exists for the specific class of W that arise in the concurrent model outlined here.

7.1. Timestamp generation

We must be able to generate locally and without delay a timestamp meeting two ordering criteria: the total event order $<$ and the partial causal order $\subset$. The generated timestamp must succeed in event order and in causal order every element of a known set (the timestamps of W^s). The timestamp must be unique among timestamps generated at all sites, and must not succeed in causal order any generated timestamp other than those in W^s.

A suitable timestamp for n sites is of a vector of n integers. The causal order $\subset$ is correctly implemented by:

$$T_1 \subset T_2 \text{ if } T_1 \subseteq T_2 \text{ and } T_1 \neq T_2$$

where $\subseteq$ is the conjunction of element comparisons:

$$(a_0 a_1 \cdots a_m) \subseteq (b_0 b_1 \cdots b_m) \quad \text{if } a_i \leq b_i \ \ (0 \leq i \leq m)$$

A suitable total event order $<$ is implemented as simple vector comparison:

$$(a_0 a_1 \cdots a_m) < (b_0 b_1 \cdots b_m) \quad \text{if } a_0 < b_0 \quad \text{or } a_0 = b_0 \text{ and } (a_1 \cdots a_m) < (b_1 \cdots a_m)$$

The sites are statically numbered from 0 to $n-1$, and at site s a new timestamp T succeeding any timestamp in W is defined as

$$T[s] = 1 + T^s[s]$$
$$T[i] = T^s[i] \ \ (i \neq s)\ .$$

7.2. Delay and reliable message delivery

The notice of an update cannot be processed until every event that causally precedes the update has been modeled. If this is not the case, it is because the notice of a causally preceding update is delayed or lost. In this case processing the message is delayed until the preceding update arrives or is recovered. This delay is accomplished by queuing the incoming message; the processing of other updates is unaffected.

The timestamp generation strategy ensures that timestamps T_1 and T_2 generated consecutively at a given site t have consecutive integral values at position t; that is, $T_2[t] = T_1[t] + 1$. Whenever delay occurs, it must be the case that $T^s[t] < T^r[t]$ for some t, indicating that a message with $T^r[t]$ was sent by t but not yet received by s. A retransmission protocol may be invoked to recover the message from r, from t, or from any other site that has previously received the missing message. Duplicate messages arising from spurious retransmission have no effect. Network partitioning is merely another form of delay; the models in separate partitions proceed independently, and when connectivity is re-established, they are merged automatically.

Thus it is possible to meld a retransmission protocol with the concurrent model, because the model is insensitive to transmission delay and delivery order, and because the timestamp provides the necessary information to detect missing and duplicated messages. The specific details of a recovery protocol are omitted from this presentation, and the reader is referred to Birman et al (1991) for further details. Our presentation simply assumes that all messages are eventually delivered.

7.3. Finding canonical forms

The simplest set W for which a canonical form is known is the set of elements of a valid update sequence. That is, if $U = F_1\!:X_1\!:T_1 \cdots F_n\!:X_n\!:T_n$ is a canonical sequence and $W = \{F_i\!:X_i\!:T_i \;\; (1 \le i \le n)\}$, $[W] = U$.

The operator $\hat{}$, defined in figure 4, is used to append a valid sequence to another that is concurrent with it (i.e. valid with respect to the same X). The empty sequence λ is valid with respect to any X.

Theorem 1. Given U_1 and U_2 both valid with respect to X and having disjoint timestamps,

$$U_1(U_2\hat{}\,U_1) =_X U_2(U_1\hat{}\,U_2) \ .$$

Proof. If $U_1 = \lambda$ or $U_2 = \lambda$ the theorem holds trivially. For $U_1 \ne \lambda$ and $U_2 \ne \lambda$, the theorem is proved by induction on $n = |U_1| + |U_2| \ge 2$. If $n = 2$, we have $U_1 = F_1\!:X\!:T_1$ and $U_2 = F_2\!:X\!:T_2$ and either $T_1 < T_2$ or $T_2 < T_1$. In either case, the theorem holds

Figure 4. Definition of ˆ.

$$F_1\!:X\!:T_1\hat{}\,F_2\!:X\!:T_2 =$$
$$F_1/F_2\!:F_2(X)\!:T_1 \quad (T_2 < T_1)$$
$$F_1\backslash,F_2\!:F_2(X)\!:T_1 \quad (T_1 < T_2)$$

$$U\hat{}\,\lambda = U$$

$$\lambda\hat{}\,U = \lambda$$

$$U_1\hat{}\,(U_2U_3) =_X (U_1\hat{}\,U_2)\hat{}\,U_3$$

$$(U_1U_2)\hat{}\,U_3 =_X (U_1\hat{}\,U_3)(U_2\hat{}\,(U_3\hat{}\,U_1))$$

by the definitions of $\hat{}$, $/$, and $\backslash$. For $n > 2$, we assume by induction the theorem holds for all U'_1 and U'_2 where $2 \le |U'_1| + |U'_2| < n$. Since $n > 2$ it must be the case that either $U_1 = U_{1a}U_{1b}$ ($U_{1a} \ne \lambda$ and $U_{1b} \ne \lambda$) or $U_2 = U_{2a}U_{2b}$ ($U_{2a} \ne \lambda$ and $U_{2b} \ne \lambda$). In the first case we begin with

$$\underline{U_1}(U_2\hat{}\,\underline{U_1})$$

and substitute at each step the underlined sequence by an equivalent:

$$=_X U_{1a}\underline{U_{1b}(U_2\hat{}\,(U_{1a}U_{1b})} \qquad \text{[subst =]}$$
$$=_X U_{1a}\underline{(U_{1b}((U_2\hat{}\,U_{1a})\hat{}\,U_{1b}))} \qquad \text{[defn ˆ]}$$
$$=_X \underline{U_{1a}(U_2\hat{}\,U_{1a})}(U_{1b}\hat{}\,(U_2\hat{}\,U_{1a})) \qquad \text{[induction]}$$
$$=_X U_2\underline{(U_{1a}\hat{}\,U_2)(U_{1b}\hat{}\,(U_2\hat{}\,U_{1a}))} \qquad \text{[induction]}$$
$$=_X U_2(\underline{(U_{1a}U_{1b})}\hat{}\,U_2) \qquad \text{[defn ˆ]}$$
$$=_X U_2(U_1\hat{}\,U_2) \qquad \text{[subst =]}$$

In the second case we have

$$U_1(\underline{U_2}\hat{}\,U_1)$$
$$=_X U_1\underline{((U_{2a}U_{2b})\hat{}\,U_1)} \qquad \text{[subst =]}$$
$$=_X \underline{U_1(U_{2a}\hat{}\,U_1)}(U_{2b}\hat{}\,(U_1\hat{}\,U_{2a})) \qquad \text{[defn ˆ]}$$
$$=_X U_{2a}\underline{(U_1\hat{}\,U_{2a})(U_{2b}\hat{}\,(U_1\hat{}\,U_{2a}))} \qquad \text{[induction]}$$
$$=_X U_{2a}U_{2b}\underline{((U_1\hat{}\,U_{2a})\hat{}\,U_{2b})} \qquad \text{[induction]}$$
$$=_X \underline{U_{2a}U_{2b}}(U_1\hat{}\,\underline{(U_{2a}U_{2b})}) \qquad \text{[defn ˆ]}$$
$$=_X U_2(U_1\hat{}\,U_2) \qquad \text{[subst =]}$$

□

The operator $|$ (figure 5) is used to append the canonical form of one set W_1 to that of another, W_2. In the special case that W_2 is empty, $|$ computes the canonical form $[W_1]$. We say that W is valid with respect to X_0 if for every $F\!:X\!:T \in W$, $X = [W^{\subset F:X:T}](X_0)$, where $W^{\subset F:X:T}$ denotes $\{F_i\!:X_i\!:T_i \in W \mid T_i \subset T\}$. We say that W_1 and W_2 are mutually valid with respect to X if W_1, W_2 and

$W_1 \cup W_2$ are all valid with respect to X. In addition, we use the notation $W^{<F:X:T}$ to denote $\{F_i \in W : X_i : T_i \mid T_i < T\}$.

Figure 5. Definitions of $\mid$ and [ ].

$W \mid W = \lambda$

$W_1 \mid W_2 =$
$(W_1^{<u} \mid W_2)\ (u\hat{}\ (W^{<u} \mid W^{\subset u}))$ if $u \notin W_2$

$(W_1^{<u} \mid W_2^{<u})\hat{}\ (u\hat{}\ (W_2^{<u} \mid W^{\subset u}))$ if $u \in W_2$

where
$W = W_1 \cup W_2$
$u = F_u : X_u : T_u \in W$ such that
$T_i \le T_u$ for all $F_i : X_i : T_i \in W$

$[W] = W \mid \{\} = [W^{<u}]\ u\hat{}\ (W^{<u} \mid W^{\subset u})$

where
$u = F_u : X_u : T_u \in W$ such that
$T_i \le T_u$ for all $F_i : X_i : T_i \in W$

Theorem 2. Given W_1 and W_2 mutually valid with respect to X,

$$[W_2]\ (W_1 \mid W_2) =_X [W_1 \cup W_2]\ .$$

Proof. The theorem is proved by induction on $n = |W_1| + |W_2|$. For $n = 0$ we have $W_1 = W_2$ and the theorem holds trivially. For $n > 0$, we assume by induction that the theorem holds for W'_1 and W'_2 such that $0 \le |W'_1| + |W'_2| < n$.
Let $W = W_1 \cup W_2$ and $u = F_u : X_u : T_u \in W$ such that $T_i \le T_u$ for all $F_i : X_i : T_i \in W$. Either $u \in W_2$ or $u \notin W_2$. Consider the case of $u \notin W_2$.

$[W_2]\ \underline{(W_1 \mid W_2)}$

$=_X \underline{[W_2]\ (W_1^{<u} \mid W_2)}\ (u\hat{}\ (W^{<u} \mid W^{\subset u}))$ [defn $\mid$]

$=_X \underline{[W^{<u}]\ (u\hat{}\ (W^{<u} \mid W^{\subset u}))}$ [induction]

$=_X [W]$ [defn [ ]]

Now consider the case of $u \in W_2$.

$\underline{[W_2]}\ (W_1 \mid W_2)$

$=_X [W_2^{<u}]\ u\hat{}\ (W_2^{<u} \mid \underline{W_2^{\subset u}})\ (W_1 \mid W_2)$ [defn [ ]]

$=_X \underline{[W_2^{<u}]}\ u\hat{}\ (W_2^{<u} \mid W^{\subset u})\ (W_1 \mid W_2)$ [W valid]

$=_X [W^{\subset u}]\ \underline{(W_2^{<u} \mid W^{\subset u})\ u\hat{}\ (W_2^{<u} \mid W^{\subset u})}\ (W_1 \mid W_2)$ [ind.]

$=_X [W^{\subset u}]\ u\ (W_2^{<u} \mid W^{\subset u})\hat{}\ u\ \underline{(W_1 \mid W_2)}$ [theorem 1]

$=_X [W^{\subset u}]\ u\ \underline{(W_2^{<u} \mid W^{\subset u})\hat{}\ u\ (W_1^{<u} \mid W_2^{<u})\hat{}\ (u\hat{}\ (W_2^{<u} \mid W^{\subset u}))}$

[defn $\mid$]

$=_X [W^{\subset u}]\ u\ \underline{((W_2^{<u} \mid W^{\subset u})\ (W_1^{<u} \mid W_2^{<u}))}\hat{}\ u$ [defn ˆ]

By the inductive assumption and because $W^{\subset u} \subseteq W_2^{<u}$ we know that
$[W_2^{<u}] =_X [W^{\subset u}]\ (W_2^{<u} \mid W^{\subset u})$.
Also by the inductive assumption we know that
$[W^{<u}] =_X [W_2^{<u}]\ (W_1^{<u} \mid W_2^{<u})$
and
$[W^{<u}] =_X [W^{\subset u}]\ (W^{<u} \mid W^{\subset u})$.
Solving these three equations yields
$W^{<u} \mid W^{\subset u} =_X (W_2^{<u} \mid W^{\subset u})\ (W_1^{<u} \mid W_2^{<u})$.
This equation is substituted into the underlined subsequence above to show that

$[W_2]\ (W_1 \mid W_2)$

$=_X [W^{\subset u}]\ \underline{u\ (W^{<u} \mid W^{\subset u})\hat{}\ u}$

$=_X \underline{[W^{\subset u}]\ (W^{<u} \mid W^{\subset u})}\ u\hat{}\ (W^{<u} \mid W^{\subset u})$ [theorem 1]

$=_X \underline{[W^{<u}]\ u\hat{}\ (W^{<u} \mid W^{\subset u})}$ [induction]

$=_X [W]$ [defn [ ]]

□

Theorem 3. If W is valid with respect to some X, $[W]$ is a canonical form for W.

Proof. By straightforward induction on the $n = |W|$. The key observation is that if $[W_1]$ is a canonical form for W_1 valid with respect to X and $[W_2]$ is a canonical form for W_2 valid with respect to $[W_1]\ (X)$, $[W_1]\ [W_2]$ is valid with respect to X and is a canonical form for $W_1 \cup W_2$. □

Theorem 4: Convergence of CCU algorithm. The data store at each site converges to a common value that models a common entity as affected by the set of events at all sites.

Proof. Every event causes one message to be generated. The message delivery mechanism ensures that each message transmitted or received is added to W^s. Therefore as messages are delivered W^s approaches a common value for all s. Theorem 2 shows that $[W^s]$ and therefore X^s also approach a common value, and theorem 3 shows that they model a common entity. □

Corollary. When there are no messages in transit, the stores at all sites are equal.

8. Implementation Considerations

It is unnecessary to store W^s or to compute $[W^{s]}$ in its entirety. Theorem 2 permits the two statements

$W^s \leftarrow W^s \cup \{F : X^r : T\} \cdots .$
$X^s \leftarrow [W^s](X_0)$

to be replaced by

$X^s \leftarrow ((W^s \cup \{F: X^r: T)\}) \mid W^s)\ (X^s)$
$W^s \leftarrow W^s \cup \{F: X^r: T\} \cdots .$

Computing $\mid$ occasions computation only to the extent that W^s differs from W^r; if there have been have been no out-of-order messages, W^s will differ from W^r only in the number of messages transmitted by s that r had not received at the time the incoming message was sent. Each element of W^s must be maintained at site s only until it is known that every other site has processed it. Thus W^s is really a queue. The property of $\mid$ that it never references elements in common between W^s and W^r ensures that purged elements are never accessed. All set operations are performed on timestamps representing subsets of this queue.

In the 2-site case, there is a particularly simple implementation. At site S, rather than storing W^s we store $U^s = W^s | W^r$, where W^r is the last known value of W^r. When a new message T^r, T, F arrives, we must compute

$U^s \leftarrow U^s$ less any elements $F_i: X_i: T_i$ with $T_i \subseteq T^r$
$X^s \leftarrow (F: X^r: T\ \hat{}\ U^s)(X^s)$
$U^s \leftarrow U^s\ \hat{}\ F: X^r: T$

The size of U^s is bounded by the number of unacknowledged messages, and the time necessary to effect a remote update is linear in the size of U^s.

The n-site case is more problematic. It is relatively simple to store W^s as a set, but the algorithm for remote update is doubly recursive and potentially requires exponential time in the number of unacknowledged messages. In practice, it may be that only pathological combinations of transmission delay result in an exponential running time; the investigation of this phenomenon provides an interesting avenue for future research.

The simplest n-site algorithm is a star built from multiple versions of the 2-site implementation, with site 0 at the hub. It is important that site 0 be at the hub, because the canonical order over all sites must be the order that they are processed at the hub. Vadura (1994) has used this approach to build a shared text editor based on EMACS. Various tree topologies might be used; care would have to be taken to ensure a valid canonical ordering. An n-site broadcast algorithm could be built along the lines of the 2-site algorithm with a set of $n-1$ U^r at each site − i.e. one U^r for each remote site. A broadcast topology would have a number of advantages over a tree in terms network performance and robustness.

9. Conclusions

Distributed concurrency control is of interest in many disciplines including operating systems, databases, simulation, and computer supported collaborative work. Although Ellis and Gibbs propose dOPT within the context of computer supported collaborative work, it is unique and has application within the context of each of these areas. CCU advances the approach by providing a method for reasoning about concurrent update and an algorithm that is provably correct. The algorithm applies to any object provided that / and \ are defined for all pairs of update functions f and g asserting two axioms:

$f: x\ \ g/f: f(x) =_X g: x\ \ f \setminus g: g(x)$
$g/f: f(x) =_M g: x\ .$

Recent articles (Cheriton and Skeen 1993, 1994, Birman 1993, 1994, van Renesse 1994, and Cooper 1994) have debated the benefits and liabilities of causally and totally ordered communication systems. Cheriton argues that total ordering is expensive to implement, difficult to use, and causes unwanted delay. Instead, he suggests that concurrency control should be done in the application layer. Birman argues that embedding concurrency control at the application level is difficult and error-prone, and that total ordering provides a good abstraction with which to build applications. We offer CCU as a compromise in this debate: it does not require totally ordered communication, but it supports in a sense the abstraction of a total order; it allows the specification of application-level control mechanisms while providing a system-level algorithm that maintains consistency. In addition to being provided by an operating system as a tool for application support, we foresee CCU being used to implement objects within distributed operating systems. A typical example would be a file-system directory with operations to create, destroy, rename, and modify the attributes of files.

Bhargava (1987, p. 40) mentions the notion of application-dependent *reconciliation formulas* to merge transactions from a partitioned system, but describes no method for their construction. CCU is such a method. Bhargava's example is trivial: expressed in terms of CCU, the data store is a simple integer and the only update is $add(k)$ which adds a constant k to the store. For this example,

$add(k)\ /\ add(j) = add(k) \setminus add(j) = add(k)\ .$

To illustrate the use of CCU, we extend the example slightly: instead of $add(k)$, we use the update function $axb(a, b)$ which multiplies the store by a constant a and adds another constant b. A definition of / and \ that preserves these semantics is

$$axb(a_1, b_1) \,/\, axb(a_2, b_2) = axb(a_1, b_1)$$
$$axb(a_1, b_1) \setminus axb(a_2, b_2) = axb(a_1, a_2 b_1 - a_1 b_2) \ .$$

In this case, / obviously preserves the semantics as stated, and \ can be shown consistent using simple algebra. The algorithm ensures that appropriate algebraic manipulations are performed on concurrent updates to yield a result exactly as if those updates had been performed in some specific order.

Our notion of a total order is not the same as that of serializability in database systems. The issue in serializability is to examine the elementary data read and written by transactions to deduce potentially conflicting operations. From this information serializable schedules may be deduced. Either delay or rollback is used to avoid nonserializable schedules. In a sense, CCU is used to make all schedules serializable; it is conceivable that a partial implementation of / and \ could be used to relax the conditions under which transactions could proceed without delay or rollback. It is also conceivable that locks could be enforced (updates are annulled if the lock is not held), or rollback could be effected (for example, $f/g = g^{-1}$) using CCU. It is also possible to implement a lock or a circulating token as a CCU object. Various other objects like error logs, buffers, and user interfaces are amenable to definition with CCU. A topic of future research is to examine to what extent the semantics of existing control mechanisms can be expressed using CCU as a tool.

Acknowledgments

The author thanks Charlie Clarke for his insightful criticism. This research is supported by the Natural Sciences and Engineering Research Council of Canada.

References

Bhargava B.K. [ed.] (1987), *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold.

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

Birman K. (1993), *The process group approach to reliable distributed computing*, **CACM 36:12**, December, 37-53.

Birman K. (1994), *A response to Cheriton and Skeen’s criticism of causal and totally ordered communication*, **SIGOPS Review 28:1**, January, 11-20.

Cheriton D.R. and Skeen (1993), *Understanding the limitations of causally and totally ordered communication*, ACM SIGOPS Symposium on Operating Systems Principles, 44-57.

Cheriton D.R. and Skeen D. (1994), *Comments on the responses by Birman, van Renesse and Cooper*, **SIGOPS review 28:1**, January, 32.

Cooper R. (1994), *Experience with causally and totally ordered communication support − A cautionary tale*, **SIGOPS Review 28:1**, January, 28-31.

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.

van Renesse R. (1994), *Why bother with CATOCS?*, **SIGOPS Review 28:1**, January, 22-27.

Appendix A − Counterexample to dOPT

The dOPT algorithm is given in figure A1. The definition of the priority function is elided because Ellis and Gibbs give several possibilities and the counterexample demonstrates that dOPT fails regardless of the definition. The set of update operations used here (figure 2) is more general than that of Ellis and Gibbs but yields an identical result for this example. An unpublished variant of dOPT due to Holtz (1991) fails the counterexample with an identical result.

Figure A1 . dOPT Algorithm at Site *s*.

```
Initialization:
   $X^s \leftarrow X_0$            − data store
   $L^s \leftarrow \lambda$            − update log; initially empty
   $T^s \leftarrow (0, 0, \cdots 0)$       − timestamp

Occurrence of local update $u$
   transmit $< s, T^s, u, prio(s, u, \cdots) >$ to other sites
   $L^s \leftarrow L^s, < s, T^s, u, prio(s, u, \cdots) >$
   $X^s \leftarrow u(X^s)$
   $T^s[s] \leftarrow T^s[s] + 1$

Receipt of message $< r, T^r, u, p >$
   delay until $T^r[i] \le T^s[i]$ for all $i$
   for each $< s', T', u', p' >$ in $L$
      if $T^r[s'] \le T'[s']$ then
         if $p < p'$ then
            $u \leftarrow u / u'$
         else
            $u \leftarrow u \backslash u'$
   $L \leftarrow L, < r, T^r, u, p >$
   $X \leftarrow u(X)$
   $T^s[r] \leftarrow T^s[r] + 1$
```

Consider a text object with an initial value of abcdefg. Site 1 deletes the a while concurrently site 2 deletes the a and then the e (figure A2). That is, site 1 takes on the value bcdefg while site 2 takes on the value bcdfg, and three messages are transmitted:

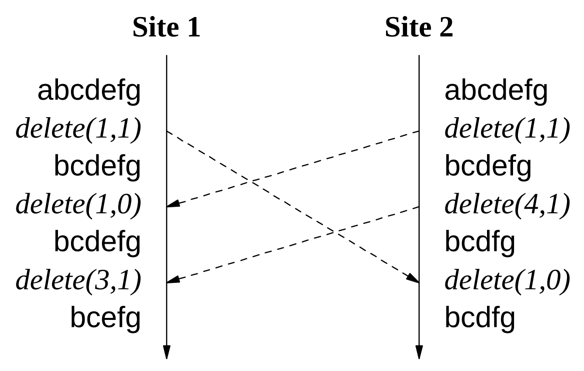
$< 1, (0, 0), delete(1, 1), p1) >$

from site 1 to site 2, and

$< 2, (0, 0), delete(1, 1), p2) >$
$< 2, (0, 1), delete(4, 1), p3) >$

from site 2 to site 1. (*p1*, *p2* and *p3* may be chosen arbitrarily.) When the message destined for site 2 arrives, its update is transformed to *delete(1,0)*, 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 *delete(1,0)*, leaving site 1's value at bcdefg. The second message is transformed (inappropriately) to *delete(3,1)*, 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.

Figure A2. dOPT Counterexample.



Discussion

In relation to the two-site implementation, it is easy to point out the flaw in dOPT. dOPT performs the following computations on the arrival of a remote message:

$U^s \leftarrow U^s$ less any elements $F_i : X_i : T_i$ with $T_i \subseteq T^r$
$X^s \leftarrow (F : X^r : T \,\hat{}\, U^s)(X^s)$

After this update, U^s is not updated, and any subsequent concurrent update will be transformed incorrectly. The algorithm is corrected (for the two-site case) by replacing the statements

$u \leftarrow u / u'$ and $u \leftarrow u \backslash u'$

by

$u, u' \leftarrow u / u', u' \backslash u$ and $u, u' \leftarrow u \backslash u', u' / u$.

There does not appear to be any simple way to correct dOPT for a general broadcast topology. It is apparent that no simple single-queue implementation can capture the exponential number of message delivery orderings that may arise. Furthermore, the use of *priority* in dOPT appears to belie that the following definition of < is consistent with the causal order $\subset$:

$T^s < T^r =_{def}$
$T^s \subset T^r$ or
$T^s \not\subset T^r$ and $T^r \not\subset T^s$ and $s < r$.

Ellis and Gibbs suggest replacing $s < r$ in this definition by a more complex priority scheme; it is unclear whether the resulting definition of < is consistent with $\subset$.