Consistency Control Based On Commutative Replicated Data Type

Khin Sandar Myint, Ei Chaw Htoon
University of Computer Studies, Yangon
Khinsandarmyint.ucsy@gmail.com, htoon.eichaw@gmail.com

Abstract

Commuting operations greatly simplify consistency in distributed systems. A Commutative Replicated Data Type (CRDT) is one where all concurrent operations commute. The replicas of a CRDT converge automatically without complex concurrency control .In this paper, Treedoc; a CRDT design for cooperative text editing is described in cooperative editing system in groupware application. An essential property is that the identifiers of Treedoc atoms are selected from a dense space. Using Treedoc CRDT, we present how to control the consistency in groupware application.

1. Introduction

To share information, users located at several sites may independently update a common object, e.g., a text document. Each user operates on a separate replica (i.e. local copy) of the document. A well-studied example is cooperatively editing a shared document [2].

Cooperative editing is an important class of groupware application. User to cooperate by editing a shared document concurrently in synchronous or asynchronous mode is allowed. A replica of the document at every participating site is maintained in this system [4].

As users make modifications, replicas diverge from one another. Operations initiated on some site propagate to other sites and are integrated or replayed there. Eventually, every site executes every action [5]. Despite this, replicas might not converge, if they execute operations in different orders. In order to guarantee convergence, the designing of replicated data type such that concurrent operations commute with one another. It is called a Commutative replicated data type or CRDT. If operations replay in happened-before order, replicas of a CRDT converge automatically, without complex concurrency control [5].

As an existence proof of non-trivial, useful, practical and efficient CRDT, the design of Treedoc which implements an ordered set with insert-at-position and delete operations are described in the consistency control of cooperative editing in groupware application.

The CRDT is used in a shared document editing concurrently in synchronous (e.g. Groove, Coword and Google Docs) or asynchronous mode (e.g. CVS). The CRDT is also used in cooperative editing of Wikipedia pages. The CRDT applies the editing a shared file (e.g. text file, source code file and html file) in groupware application.

The paper is organized as follows. Section 2 provides the related work for the system. Section 3 describes the background theory of the system. Section 4 shows the implementation of the system. Section 5 presents the experimental analysis. Section 6 describes the conclusion of the system.

2. Related Work

A comparison of another approach to the problem of collaboratively editing a shared text is described in this section.

Operational transformation (OT) [1] considers collaborative editing based on non-commutative single-character operations. To this end, OT transforms the arguments of remote operations to take into account the effects of concurrent executions. OT requires two correctness conditions [1]: the transformation should enable concurrent operations to execute in either order, and furthermore, transformation functions themselves must commute.

OT attempts to make non-commuting operations commute after the fact. The CRDT is a better approach of the designing of commutative operations. This is more elegant and avoids the complexities of OT [1].

In the treedoc design, common edit operations execute optimistically. It can support arbitrary mixture of edit operations.

3. Background Theory

3.1 System Model

In an asynchronous distributed system, consisting of *N sites* (computers) are connected by a network. Communication between connected sites is reliable. A site may disconnect but eventually reconnects. An epidemic style of communication, i.e., a site connects at arbitrary intervals with arbitrary other sites, sending both local updates and those previously received from other sites. Eventually, every update reaches every site, either directly or indirectly [2].

With no loss of generality, single object

replicated at any number are consider in this system. A user accesses the object through his local replica, initiating operations at the current site. The operations execute locally and are logged. Eventually, the log is transmitted and the operations it contains are *replayed* at other sites. Eventually, all sites execute the same operations (either by local submission or by remote reply), in some sequential order, but not necessarily in the same order [2].

Two operations *o and o′* commute if, whatever the current state of the objects, such that execution of the either o and o′ immediately followed by o′ also succeeds, and leads to the same state as executing o′ immediately followed by o [2].

CRDT is a data type where all concurrent operations commute with one another. CRDT provide that every site executes every operation in an order consistent with happens-before, the final state of replicas is identical at all sites [2].

Furthermore, CRDTs support serialisable transactions with virtually no overhead. If all operations commute, so do arbitrary sets of operations. If every site executes transactions sequentially, in an order consistent with happens-before, the local orders are all equivalent. This ensures serialisability. Furthermore, transactions never abort. Hence, very little mechanism is needed [2].

3.2 The Treedoc CRDT

In Treedoc, an atom identifier is called a TID. It represents a path in a tree. If the tree is balanced, the average TID size is logarithmic in the number of atoms. The treedoc design is represented the binary version. The order "<" is infix traversal orders [3]. The design of the Treedoc is implemented by insert and delete. The treedoc design with binary structure is shown in Figure1.

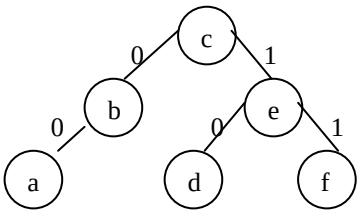

Figure 1: The TID for "b" is 0, TID for "c" is empty and TID for "d" is "10"

The update operations of the ordered-set abstraction are the following [3].

-*insert (ID, newatom),* where ID is a fresh identifier. This operation adds *newatom* to the ordered-set.

-*delete (ID),* deletes the atom identified ID from the ordered-set.

3.2.1 Identifier

Atom identifiers should have the following properties: (i) Two replicas of the same atom have the same identifier. (ii) No two atoms have the same identifier. (iii) An atom's identifier remains constant for the entire lifetime of the ordered-set. (iv) There is a total order "<" over identifiers, which defines the ordering of the atoms in the ordered-set. (v) The identifier space is dense [3].

Property (v) means that between any two identifiers P and F, P<F, we can allocate a fresh identifier N, such that P<N<F. Thus, we are able to insert a new atom between any two existing ones [3].

3.2.2 Treedoc algorithm

Procedure *contents (N) / /N: a treedoc node*
Walk subtree rooted at N infix order
Return the atoms of the non-empty nodes
Procedure *insert (A, N, S) // A: atom to insert, N:*
Require: S= the initiator site
Require: A≠ nil
Require: *Iaminitiator* => all ancestors of node N exist
Require: *Iaminitiator* => N [[S]] does not exist
Create side node n= N[siteID] with n.contains = A
Procedure *delete (N, S) //N:node to be deleted*
Require: *Iaminitiator* => N [[S]] exists and N[[S]].contains ≠ nil
Require: S = the initiator site
N [[S]].contains:= nil
Send acknowledge of delete (N, S) to all site
Procedure *stabledel (N)//Await stable delete of node N*
if acknowledgment for *delete (N,*)* receive from all sites then **return** true **else return** false
Procedure *gc (N) // N: a treedoc leaf*
Require: N.contents (=) nil
Require: *stabledel (N)*
Require: *Iaminitiator*
Remove *N*
Procedure *cleanside (N): a treedoc node*
Require: *stableinsert (N)*
Require: *Iaminitiator*
Remove redundant disambiguator
Procedure *stableinsert (N)*
If current site has received some Operation that happens-after insert (*, N,*) from every site
then return true
else return false
Procedure flatten (N) // rebalancing the tree
- Walk subtree in infix order
- Return a linear buffer containing the atoms of the non-empty nodes

3.3 Deleting

Deleting an atom simply replaces its node's content with nil. Since identification of the deleted node is unique, it is clear that the initiator and replay executions will all delete the same node.

Sometimes, during replay, the node to be deleted may not exist between may not exist because it was already previously. To solve this problem, we introduce a *gc (N)* procedure in the following. Procedure *stabledel (N)* tests for stability. Conceptually, *stabledel (N)* waits for acknowledgements from all sites that have executed *deleteN*. Procedure *gc (N)* that removes leaf *N* if it is stably deleted. A node may call *gc (N)* at any time after *N* is deleted.

3.4 Inserting

To insert newatom between $atom_P$ and $atom_F$, we must grow the tree in a way that satisfied the relation id ($atom_P$) < (newatom) <id ($atom_F$). A very simple algorithm that does not attempt to balance the tree is present in this paper. This algorithm starts by checking recursively whether there is a node between $atom_P$ and $atom_F$. When there is no node between $atom_P$ and $atom_F$, three cases may occur:

- Node uid_P is an ancestor of uid_D. In this case, uid_F has no left child, so we create a new left child of node uid_F. The new identifier is uid_F □ 0.
- Or, symmetrically, node uid_F is an ancestor of uid_P. The new identifier us uid_P □ 1.
- Or, neither is an ancestor of the other. In this case, uid_P has no right child, so we create a new right child of node identifier uid_P □ 1.

In the example of figure, for inserting atom X between c and d, a left child is created under d with identifier [100], as shown in Figure2.

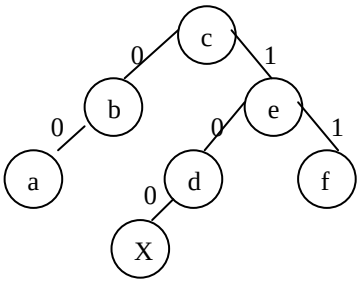

Figure 2: Identifier after inserting a new atom

3.5 Concurrent Insert

In the case of concurrent updates, a binary tree becomes insufficient because two users can concurrently insert an atom at the same position. We maintain the basic binary tree structure; we extend the basic tree structure, allowing a node to contain a number of internal nodes called mini-nodes. A node containing mini-nodes will be called a major-node.

Inside a major node, mini-nodes are distinguished by a disambiguator, which is the identifier of the site that inserted the node [3]. Disambiguators are unique. Assuming that characters X and Y were inserted with the associated disambiguator's idX and idY and are shown in Figure3.

Id (X) = [(1) (0) (0, idX)]
Id (Y) = [(1) (0) (0, idY)]

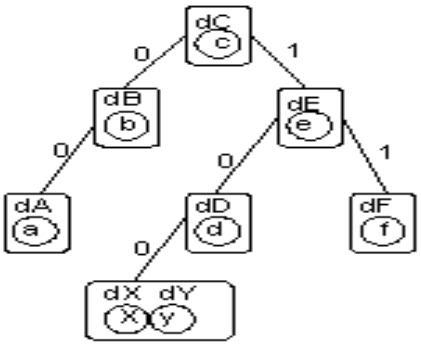

Figure 3: A treedoc node with disambiguators

3.6 Rebalancing the tree

In the approach described so far, depending on the pattern of inserts and deletes, the tree may become badly unbalanced. To alleviate this problem, internal operation *rebalance* balances the tree. Since a balanced tree is equivalent to an array, this eliminates all memory overhead. Rebalancing is a radical form of garbage collection. The flatten algorithm for the rebalancing is described in follow.

Table 1: Flatten algorithm

Procedure flatten(N) //N, root of the subtree to be flatten
Walk subtree inorder
Return a linear buffer containing the atoms of the non empty nodes.

4. System Implementation

This system is implemented for the consistency control of cooperative editing in groupware application. In which, the shared files can be edited by the users who are in the groupware application. Each user can edit the files that are needed to edit by the user's appropriate parts at any time in their own site. All files in this system are consistent by using CRDT method. Even though, the users will edit the same file at the same time, it can be consistent without conflicting. In this system, journals are written altogether many editors. Moreover, programming language and source code and text files are allowed in this system by using CRDT.

This system is implemented with the core site (server site) and nebula site (client site). The core consists of a small group of sites that are well connected. Sites that are not in the core are part of the nebula. The nebula may contain any number sites, which are connected to the network, or may freely initiate update. This system is implemented by a server site (core site) and many client sites (nebula

site) that they are user's want.

A site can leave the core at any time, simply by invoking the membership protocol. For a nebula site to send updates to the core, or in order for it to join the core, a protocol is need. This protocol is called a catch-up protocol. Every nebula site will go through the catch-up protocol, when it wants to migrate to the core site.

4.1 System Flow Diagram

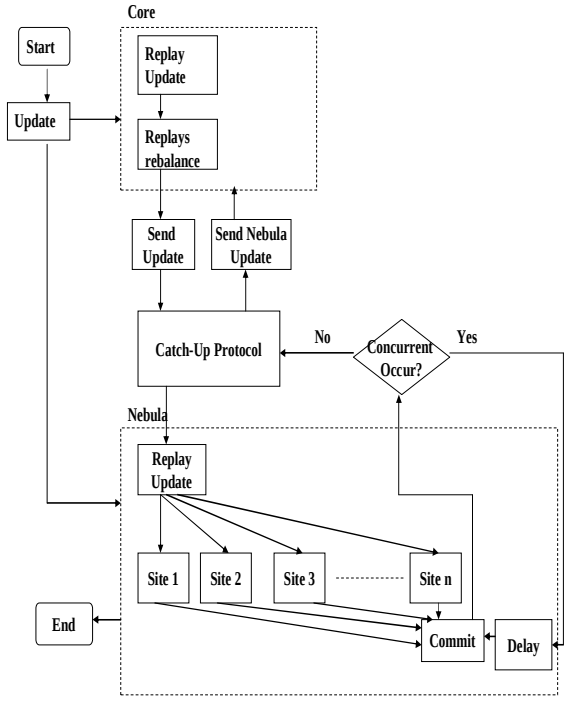

Figure 4: System flow diagram of the system

4.2 Implementation process

In the first step, the user can choose the files and these files are uploaded to the server site (core site). Each file is created into binary tree structure with their own identifier according to their liness. The html file is described as an example in the Figure5 and Table2.

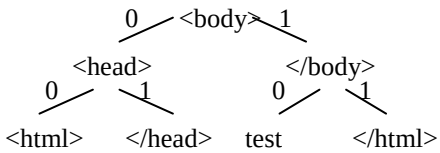

Figure 5: The html file with binary tree structure

Table 2: The html file with identifier

UID:	**00**	**<html>**
UID:	**0**	**<head>**
UID:	**01**	**</head>**
UID:		**<body>**
UID:	**10**	**test**
UID:	**1**	**</body>**
UID;	**11**	**</html>**

In the second step, a client downloads the file who wants to edit form server site (core site) to their local site (nebula site). At the same time, other clients also download that file. Each client edits that file in their own site independently one another. And then, the binary trees corresponding to that file change into new one in their local site. First client site with its tree and identifier (UID) is described in Figure 6 and Table 3. And then, another client site is also described in Figure 7 and Table 4.

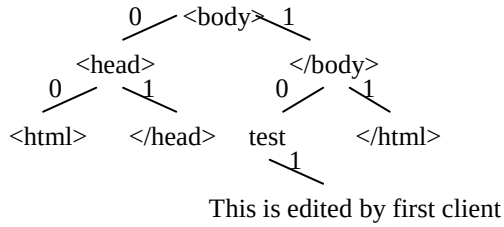

Figure 6: The tree edited by first client

Table 3: The html file with identifier in first client site

UID:	**00**	**<html>**
UID:	**0**	**<head>**
UID:	**01**	**</head>**
UID:		**<body>**
UID:	**10**	**test**
UID:	**101**	**This is edited by first client**
UID:	**1**	**</body>**
UID;	**11**	**</html>**

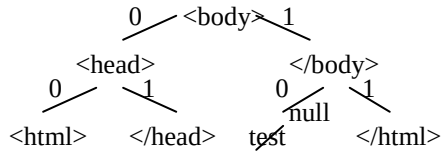

Figure 7: The tree edited by another client

Table 4: The html file with identifier in another Client site

UID:	**00**	**<html>**
UID:	**0**	**<head>**
UID:	**01**	**</head>**
UID:		**<body>**
UID:	**10**	**null**
UID:	**1**	**</body>**
UID;	**11**	**</html>**

When first client commit the edited file to the server, this file is received by server and server site return successful message to committed site and balance the tree. After balancing, server site sends this balance tree to all client sites. As the same way, another client also commits their file. Finally, all client sites and server site are consistent with new one in each site. This balance tree and identifier (UID) are described in Figure 8 and Table 5.

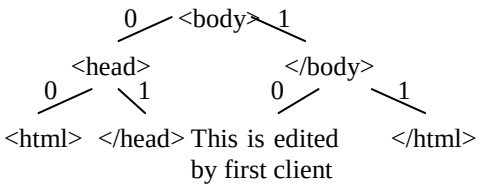

Figure 8: The consistent state in core and server sites

Table 5: The html file with identifier in core and server sites

UID:	**00**	**<html>**
UID:	**0**	**<head>**
UID:	**01**	**</head>**
UID:		**<body>**
UID:	**10**	**This is edited by first client**
UID:	**1**	**</body>**
UID:	11	</html>

5. Experimental Analysis

In this section, we execute experimentally the behavior of Treedoc. Our goal is to measure the overhead is equal to the total size of Treedoc nodes, including tombstones and this process generally improves when flattening more aggressively.

Modifying an atom is modeled as deleting the original and inserting the modified atom. This result is an unexpectedly larger number of deletes.

When flatten is not used, up to 95% of nodes are tombstones. Flattening succeeds in reducing the total number of nodes by 50%.

The time complexity for executing an insert or delete operation in a linear buffer representation is constant O (n). In treedoc, it is O (log n) for a balanced tree. In both case, auxiliary indexing structures could be used to improve performance at the cost of greater space.

6. Conclusion

According to the system implementation we discussed that commutative simplifies consistency maintenance, without concurrency control, allowing updates to execute in arbitrary orders while guaranteeing that replicas converge to the same result. We described to implement a CRDT called Treedoc that maintains an ordered set of atoms while providing insert and delete operations for Groupware Application.

A CRDT is ideal for building complex transactions out simple operations. Since, individual operations commute when concurrent and concurrent group of operations commute as well. To ensure serialisability, it is sufficient to ensure that transactions are executed sequentially, whether at the initiator site. The treedoc design of CRDT is to support arbitrary mixture of edit operations. It is consistent without conflicting in any situation. Individual operations commute, a transaction never aborts; therefore transaction support can be very cheap.

Garbage collection is requirement in practice, it is disruptive and requires consensus, but it has lower precedence the updates and it is not in the critical path of application.

7. References

[1] Chengzheng Sun and Clarence Ellis. Operational transformation in real time group editors: issue, algorithms, and achievements. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW).* Seattle WA, USA, November 1998.

[2] Marc Shapiro, Nuno Preguiça, Designing a commutative replicated data type. Rapport de recherché n· 6320- October 2007.

[3] Mihai Letia, Nuno Preguiça, Marc Shapiro. Consistency without concurrency control in large, dynamic systems. SOSP Workshop on Large Scale Distributed Systems and Middleware (LADIS) 2009, 10-11 October 2009, Big Sky, MT, USA.

[4] Nuno Preguiça, Marc Shapiro. Designing a commutative replicated data type for cooperative editing systems. Technical report 2-2008 DI-FCT-UNL.

[5] Preguiça, Marquès, Shapiro, Letia. A commutative replicated data type for cooperative editing. In *Int. Conf.* on *Distributed Comp. Sys (ICDCS)* (Montrèal, Canada, June 2009).