View-centric Operational Transformation for Collaborative Editing

Hyunjoon Jung Hyeonsun Song
Advanced Technology Laboratory
Future Technology Research Department
KT
Seoul, 137-792, Korea
jwithk@kt.co.kr, hssong@kt.co.kr

Abstract— **Collaborative editing enables multiple users who reside remotely to share and edit some documents at the same time. It is fundamentally based on operational transformation which adjusts the position of operation according to the transformed execution order. For a last decade, many researches have been performed in this area and the correctness and possibility of operational transformation have been proved.**

Even though operational transformation gives us the possibility of collaborative editing, it has a limitation with a viewpoint of usability and efficiency. In other words, the existing operational transformation is devised without considering the properties of collaborative editing such as the frequency of operational transformation and human-centric viewpoint.

In this paper, we would like to introduce *view-centric operational transformation* which considers the priority of transforming operation according to user's viewpoint. Using this way, we have tried to improve the existing operational transformation and provide more useful and efficient collaborative editing.

Key Words: Operational Transformation, Collaborative Editing, Distributed Computing System, CSCW

I. Introduction

Collaborative editing system is which enables multiple users who reside remotely to share and edit some text / graphical / multimedia documents. This is an application area of Computer-supported cooperative work and it has been known for enhancing the efficiency and productivity of jobs and solving the physical limitation with virtualization [16], [17].

For collaborative editing, the system should ensure the consistency of the shared documents and supports the rapid processing of documents editing requests. In other words, it should be the same as one user edits some documents alone. To satisfying these kinds of real limitation, various novel algorithms were proposed such as locking, operational transformation.

In the locking approach writing documents at the specific part is allowed only for one user at one moment and reading documents is allowed for any user. Therefore, it cannot support real collaborative environment which can provide multiple users to edit some documents at the same time. Even though some locking algorithms try to support collaborative editing with using small-part locking, it cannot support totally real-time collaboration.

As for operational transformation, it ensures that the effect of executing a group of concurrent operations is the same as if the operation were executed in the same total order at all sites. By transform the order of operation effectively, it ensures the consistency of the shared documents. Besides, it ensures the short response time for multiple users request because operational transformation is based on the replica-based approach [8], [9].

In spite of the novelty of operational transformation, there are some blind sides. It is not proposed without any consideration of human-centric viewpoint and practical usage of collaborative editor. As for technical viewpoint, as the size of document and the number of users grows, the number of invoked operation increase. Therefore, the complexity of operation transformation is bigger and the scalability is limited due to the huge size of operational transformation. In the case of human-centric viewpoint, the most important part of the shared document is at current caret. However, to achieve the consistency of replica-based shared document system with the existing operational transformation, it is enforced to process all invoked operations from other users but unrelated to current caret.

Through the deliberate experience, we are capable of figuring out this problem. We would like to solve this problem with borrowing the famous and great concept such as priority queue and cache in computer system. In other words, multiple users have their own history buffer for operational transformation and accept instantly the invoked operation related to their current caret in the shared document and postpone other operation from low-priority buffer. It is a kind of classified history buffer management system which conform the priority of operation for transformation.

In the rest of paper, we would like to describe this algorithm and architecture in detail. First, in order to describe operational transformation, the motivation of this research would be introduced. Next, section 3 describes VOT algorithm. Section 4 explains how to implement the architecture and algorithm of collaborative editor. And then, we simulated VOT algorithm and analyze the performance quantitatively in section 6. This paper concludes with a discussion of the implications of this work to the development of collaborative computing, and proposals for future work.

1-4244-0429-0/06/$20.00 ©2006 IEEE

II. Motivation

In practical, shared document size is very large, which consists of more than hundred pages. And the number of users is not of small number in collaborative editing.

We assume one large document which contains many users at the same moment. They might write and read the same part of document or not. It means there is possibility that each users viewpoint can be ***well-distributed*** in a total document or ***located at a specific part*** of the document. For the consistency of shared documents, it does not need any action in the former case, but it does in the latter case. To cope with this situation efficiently and effectively, we can exploit the existing operational transformation. However, it remains the possibility that the complexity and scalability might be troubled factor with growing the size of document and the number of simultaneous users.

The existing operational transformation ([4],[5],[6]) assumes that every local replica has its own history buffer for managing the invoked operation. Once the invoked operation arrives at local replica, it needs to look into every operation in history buffer and then to transform the accepted operation appropriately with operational transformation technique. In this way, as the size of history buffer grows the computational cost also increases. If n users have their own history buffer and m operations are invoked at every site, each local replica broadcast their own operation to neighbors. After every site accepts the sent operation, it transforms each operation with the existing operations of its own history buffer. In that case, it should be compared and transformed $m*(n-1)*m$ times at each node in the very short moment. Totally, $m*m*(n-1)*n$ times of comparisons and operational transformations would be processed.

With experiencing this situation, we have tried to devise efficient and effective history buffer management with the existing operational transformation. Hence we had looked into the usage pattern of collaborative editor carefully. Then it is derived that the classified buffer management should be proposed for both ***conflict-occurred*** case which users viewpoints are located at specific part of documents and ***conflict-free*** case that users viewpoint are well distributed in a total document. It is so called view-centric operational transformation.

As for conflict-occurred case, view-centric operational transformation performs the modified operational transformation algorithm similar to the existing one. In the case of conflict-free case, view-centric operational transformation ensures the efficient buffer management mechanism similar to priority queue.

In this way, users in the context of collaboration do not need to transform all operations instantly unrelated to their current viewpoint from other users. To translate this idea into reality, the classified buffer management should be implemented. In detail, we would like to describe next chapter.

III. VOT algorithm

A. Assumptions

Collaborative editor could be possible effectively in the presence of operational transformation. With an algorithmic viewpoint, a few assumptions should be premised as following.

- **Assumption 1** *Optimistic Content Delivery* - The invoked operations are ultimately delivered at each site even though its arrival may be delayed.
- **Assumption 2** *Operation Generation Model* - Every operation is processed as following five steps.
  - Generated at one site
  - Broadcasted to other sites
  - Rejected by other sites and recorded into executed buffer
  - Executed on other sites
  - Recorded to history buffer at each site
- **Assumption 3** *State vector timestamping* - Every operation contains its own state vector which is recorded at the moment of generation.
- **Assumption 4** *Total ordering relation* - Given two operations O_1 and O_2, generated at sites i and j and timestamped by SV_{O_1} and SV_{O_2}, respectively, then $O_1 \Rightarrow O_2$, iff it conforms $Definition4$.
- **Assumption 5** *Operation Composition* - Every operation is composed as follows.
  $Op = (OP_{type}, OP_{position}, OP_{timestate-vector}, OP_{site-index})$

Assumption 1 describes that no operation missing happens in VOT. In other words, network environment in VOT supports atomic reliable broadcasting and the loss of operation delivery could not possible. In the case of assumption 2, it is a life-cycle model of operation in VOT. This model is assumed to discuss operational transformation effectively. More description would be in next section 5.

Both of assumption 3 and 4 is devised for causality preservation. As we mentioned before, time-stamping vector may be a nice choice for devise an algorithm and implement a program. Finally, assumption 5 states the composition of operation which is used in VOT. OP_{type} is a operation like *insert,delete*. And $OP_{position}$ is a position of operation. $OP_{timestate-vector}$ is a time-stamp vector which contains the originating site's state vector. $OP_{site-index}$ is a site's identifier of operation.

B. Integration Algorithm

Before the description of integration algorithm, an input operation is assumed as causally-ready operation which has timestamp state vector. And this operation conforms the model of operation assumption 2 in III-A and it is passed into the three kinds of buffers which is defined as execution buffer, history buffer, and delayed buffer according to the case.

Figure 1 describes integration algorithm part 1. EB is execution buffer that contains operations which are involved in the current viewpoint of user. In other words, when operation arrives at one site, VOTDeamon algorithm part checks where the current viewpoint of user includes the position of this operation. If operation is related to the viewpoint, this operation is passed and saved into the execution buffer. Otherwise, the operation is passed into the delayed buffer DB. HB is history buffer that has operations which are already transformed and executed. The operations

in HB would be exploited to transform upon the arrival of new operation.

```
VOT Deamon :
While(true){
    op1 = getOperationFromNetworkBuffer();
    if(includeViewpoint(op1) == true){
        EB.add(op1);
    }else{
        DB.add(op1);
    }
    increaseNumberOfOperation();
}
```

Fig. 1. VOT Deamon Algorithm

Integration algorithm is responsible for transforming operation with OT function as a figure 2. In case 1, there are only dependent operations in history buffer. Therefore, it does not need to apply operational transformation to an input operation. Input operation may be executed instantly without any transformation.

In the case 2 and 3, it is a situation that at least one operation in HB is an independent relationship with an input operation. Independent operation may be generated in the same state or not. If an operation was generated in the same state, then it only needs to be applied with inclusion transformation as case 2. Otherwise, exclusion transformation should be applied to an input operation for correct transformation.

For easy understanding about the case 2 and 3, we would like to show an example. At first, HB is composed of five operations as $[OP_1, OP_2, OP_3, OP_4, OP_5]$.

- Case 2 : Independent case
  The relationship between input operation and operations in HB is as follows.
  $OP_1 \rightarrow OP_{new}$, $OP_2 \rightarrow OP_{new}$, $OP_3 \parallel OP_{NEW}$, $OP_4 \parallel OP_{new}$, $OP_5 \parallel OP_{new}$
  As described in figure 2, scanning HB is performed from left to right in order to find the first independent operation against input operation. In case 2, the first independent operation against input operation is OP_3. And the rest of operation after OP_3 are independent against OP_{new}. Hence it just needs to apply inclusion transformation with OP_{new} and $HB[3,5]$.

- Case 3 : Mixed case The relationship between input operation and operations in HB is as follows.
  $OP_1 \rightarrow OP_{new}$, $OP_2 \parallel OP_{new}$, $OP_3 \rightarrow OP_{NEW}$, $OP_4 \parallel OP_{new}$, $OP_5 \rightarrow OP_{new}$
  By using the same way of case 2, the first indenpedent operation is found as OP_2. However, the rest operations after OP_2 are not independent operation. In other words, dependent operation is located among the independent operations. To transform operations effectively and correctly in this case, it should be considered exclusion operational transformation. Therefore, integration algorithm constructs temporary buffer TB which would contain dependent operations from the rest operations. And then operations in TB is transformed by using *transformation function T* and *list transformation function LT*. In this way, OP_2 and OP_3 are mutually transformed to OP_3' and OP_2'. And then OP_2', OP_4 and OP_5 are also transformed to OP_5', OP_2'' and OP_4' by applying $LTranspose(3,5)$. Finally, the operations of list are like as follows.

$$[OP_1, OP_3', OP_5', OP_2'', OP_4']$$

  At last, these operations are ready to apply inclusion transformation. Hence, it performs to transforming between operations in HB and input operation.

```
Transpose(op1,op2)
{
    op2' = ET(op2,op1);
    op1' = IT(op1,op2');
    return (op2',op1');
}
```

```
LTranspose(List L)
{
    for(i=List.size;i>1;i--){
        (L[i-1],L[i]) = Transpose(L[i-1],L[i]);
    }
}
```

Fig. 3. Transpose Function T and List Transpose Function LT

Figure 3 describes transpose function T and list transpose function LT. T is used to change the position of operation and apply transformation between two operation. This function is devised to achieve the same context of execution operation. List transpose function LT has a role that reverses an input list and transforms with contiguous operations in an input list.

```
IsOperationIndependent(op1,op2)
{
    TimestampVector op1TV = op1.getTV();
    TimestampVector op2TV = op2.getTV();
    TimestampVector tempTV;
    int judgePlus,judgeMinus = 0;

    for(int i=0;i<op1TV.size(),i++){
        tempTV[i] = op1TV.getTime(i) - op2TV.getTime(i) ;
    }

    for(int j=0;j<tempTV.size();j++){
        if (tempTV[i] > 0 ) judgePlus ++;
        else if(tempTV[i]) < 0) judgeMinus --;
    }

    if(judgePlus >0 && judgeMinus <0) return true;
    else return false;
}
```

Fig. 4. Operation Dependency Checking algorithm

Surely, many functions are exploited for constructing an integration algorithm I. Among those functions, *isOperationIndependent()* has a role to verify whether the relationship between two input operation is dependent or not. To check the dependency, timestamp state vector of each operation is used as a judgment factor.

```
Integration Algorithm :
While(true){
    op1 = EB.getTopOperation();
    int index=0;
    while( (op2 = HB.getOperation(index)) != null){
        if(isOperationIndependent(op1,op2) == false){
            return eo = op1;  // case 1
        }else if(isOperationIndependent(op1,op2) == true){
            if( (TB = scanDependentOperations(op2.index+1, HB.size) )== null ){
                return eo = inclusionTransformation(op1,HB(op2.index+1,HB.size);
                // case 2
            }else {
                TB = scanDependentOperations(op2.index+1, HB.size);
                for(int i=0;i<TB.size;i++){
                    Ltranspose(HB.getSubList(index+i-1,TB.getOperation(i).position);
                }
                return eo = inclusionTransformation(op1,HB.getSubList(op2.index+TB.size, HB.size));
                // case 3
            }
        }
    }
}
```

Fig. 2. Integration Algorithm

C. Transformation Function

In this section, we would like to introduce an inclusion transformation function IT which has a role of merging two concurrent operations defined on the same state. And then, the description of exclusion transformation ET would be followed in order to talk about how to effectively exclude one operation from another operation which was already executed. ET is crucial to achieve the intention-preservation in collaborative editing.

```
InclusionTransformation(Operation Op, List L)
{
    if(Op == null) Op' = Op;
    else Op' = InclusionTransformation(applyIT(Op, L.getFirstOperation()),L.getRestOperation());
    return Op';
}
```

Fig. 5. Inclusion Transformation algorithm

1) Inclusion Transformation: Figure 5 describes *inclusion transformation Function IT*. This function accepts two input parameters - operation from execution buffer and a part of list from history buffer. And then input operation OP and one operation which is extract from the input list as $L.getFirstOperation()$ is passed into $applyIT()$ function. And then, the types of two operations are checked and primitive transformation betweens two operations would be applied according to each operation's type. To effectively implement inclusion transformation, a recursive function call is used until operational transformation is terminated.

$ApplyIT$ has a role of checking the types of input operations and calling proper function as circumstance requires. For instance, if OP_1 is a *insert* operation and OP_2 is a *delete*, then $inclusionTransInsDel()$ should be invoked. In this way, every case would be processed by invoking primitive four inclusion transformation functions.

```
ApplyIT(Operation Op1, Operation Op2)
{
    if(Op1.getType() == INSERT && Op2.getType() == INSERT) InclusionTransInsIns(Op1,Op2);
    else if(Op1.getType() == INSERT && Op2.getType() == DELETE) InclusionTransInsDel(Op1,Op2);
    else if(Op1.getType() == DELETE && Op2.getType() == INSERT) InclusionTransDelIns(Op1,Op2);
    else InclusionTransDelDel(Op1,Op2);
}
```

Fig. 6. Inclusion Transformation algorithm - applyIT()

Figure 7 describes four primitive transformation functions which transform OP_1 against OP_2 and returns transformed operation. Prior to this transformation, inclusion transformation is based on the ground that OP_1 and OP_2 was invoked at the same document state. Consider $OP_1 = Insert("X", 4)$ and $OP_2 = Delete(1, 3)$, independently generated from the document state like "$A1234$". After applying the inclusion transformation to O_1 against O_2, O_1 will become $O_1' = Insert("X", 3)$. If O_2 was executed on the original document state, the changed document state would be "$A134$". Then, the execution of transformed operation O_1' on the new document stae would make the document state as "$A1X34$".

Apparently, the execution of O_1' on the new document state achieves the same effect as the execution of O_1 on the original document state since O_1 and O_1' take effects to the document state and the effect of O_2 is correctly preserved in the document state.

```
InclusionTranInsIns(Operation Op1, Operation Op2)
{
    If( Op1.getPosition() < Op2.getPosition()) return Op1;
    else {
        Op1.setPosition(Op1.getPosition()+1);
        return Op1;
    }
}

InclusionTransInsDel(Operation Op1, Operation Op2)
{
    If(Op1.getPosition() > Op2.getPosition()){
        Op1.setPosition(Op1.getPosition()-1);
        return Op1;
    }
    else returun Op1;
}

InclusionTransDelIns(Operation Op1, Operation Op2)
{
    if(Op1.getPosition() < Op2.getPosition()) return Op1;
    else {
        Op1.setPosition(Op1.getPosition()+1);
        return Op1;
    }
}

InclusionTransDelDel(Operation Op1, Operation Op2)
{
    if(Op1.getPosition() > Op2.getPosition()){
        Op.setPosition(Op1.getPosition() -1);
        return Op1;
    }else if(Op1.getPosition() == Op2.getPosition()){
        return nullOp;
    }else return Op1;
}
```

Fig. 7. Inclusion Transformation algorithm - main part

2) Exclusion Transformation: Exclusion Transformation between two operations is for ensuring the same document state for inclusion transformation. In other words, intention-preservation properties results in exclusion transformation that transforms one operation against another operation by changing parameter effectively.

```
ExclusionTransformation(Operation Op, List L)
{
    if(Op == null) Op' = Op;
    else Op' = ExclusionTransformation(applyET(Op, L.getFirstOperation()),L.getRestOperation());
    return Op';
}
```

Fig. 8. Exclusion Transformation algorithm

Figure 8 is similar to the case of inclusion transformation. Recursively, one input operation is transformed against the others operations in the sub-list which was made from history buffer. In section III-B, the example case 3 represents the exclusion transformation. To ensuring the same state, OP_2 and OP_4 are effectively excluded from the document state and transformed to the OP_2'' and OP_4'.

Exclusion Transformation is composed of 3 steps

```
ApplyET(Operation Op1, Operation Op2)
{
    if(Op1.getType() == INSERT && Op2.getType() == INSERT) ExclusionTransInsIns(Op1,Op2);
    else if(Op1.getType() == INSERT && Op2.getType() == DELETE) ExclusionTransInsDel(Op1,Op2);
    else if(Op1.getType() == DELETE && Op2.getType() == INSERT) ExclusionTransDelIns(Op1,Op2);
    else ExclusionTransDelDel(Op1,Op2);
}
```

Fig. 9. Inclusion Transformation algorithm - applyET()

the same as inclusion transformation. At first, $exclusionTransformation()$ function calls $applyET$ recursively with new input operation and operations in input list. And then $applyET()$ function checks the type of operations. According to the type, primitive exclusion transformation function is invoked. For instance, if OP_1 is *insert* and OP_2 is *delete*, then it is called as $ExclusionTransInsDel(OP_1, OP_2)$.

```
ExclusionTranInsIns(Operation Op1, Operation Op2)
{
    If( Op1.getPosition() < Op2.getPosition()) return Op1;
    else {
        Op1.setPosition(Op1.getPosition()-1);
        return Op1;
    }
}

ExclusionTransInsDel(Operation Op1, Operation Op2)
{
    If(Op1.getPosition() >= Op2.getPosition()){
        Op1.setPosition(Op1.getPosition()+1);
        return Op1;
    }
    else returun Op1;
}

ExclusionTransDelIns(Operation Op1, Operation Op2)
{
    if(Op1.getPosition() < Op2.getPosition())    return Op1;
    else {
        Op1.setPosition(Op1.getPosition()-1);
        return Op1;
    }
}

ExclusionTransDelDel(Operation Op1, Operation Op2)
{
    if(Op1.getPosition() > Op2.getPosition()){
        Op.setPosition(Op1.getPosition() +1);
        return Op1;
    }else if(Op1.getPosition() == Op2.getPosition()){
        return nullOp;
    }else return Op1;
}
```

Fig. 10. Exclusion Transformation algorithm - main part

Figure 10 describes 4 primitive functions which transforms OP_1 against OP_2. The mechanism of these four functions is reverse to inclusion transformation primitive functions. Namely, in order to exclude the effect of OP_1, it needs to change the document state into the original state of which OP_1' was not executed.

Suppose the initial document state is $JH123$, and two operation OP_1 and OP_2 are invoked as follows. First, operation $OP_2 = Insert("x", 1)$ was generated from the above document state. After the execution of OP_2, the document state is changed into "$xJH123$". Then, operation $OP_1 = Delete(1, 4)$ was generated form the above document state. Apparently, OP_2 happend before OP_1. If OP_1 is applied with the exclusion transformation against OP_2, the transformed operation OP_1 would be $OP_1^{'} = Delete(1, 3)$. In this way, $OP_1^{'}$'s intention which tries to delete "2" could be preserved and the result in the original document state is the same as the changed document state "$xJH123$" which applies operation OP_2.

D. Buffer Management

VOT mainly exploits three kinds of buffers as we mentioned above. Execution buffer(EB) and history buffer(HB) have a role that holds operations which are invoked within the user's viewpoint. Then how and when to transform and execute operations in delayed buffer(DB) remains a troublesome problem. To resolve this problem, the operations which are saved in delayed buffer should be applied effectively via transformation at some moment. There exist two considerations - when it should be performed and how it should be done.

For the former, VOT considers user's interaction. In other words, users *savedocument* request is regarded as the timing of transformation of operations in DB. If user doen not request *save*, then VOT enforces this process automatically. However, the origin of VOT is that removes a unrelated operational transformation in real-time and reduce the overhead from the operational transformation. Therefore, the operations in DB don't have to applied frequently.

The transformation of operations in DB is the same as operations in EB. When one operation is extracted from DB, it is transformed against operations in HB sequentially. And, IT and ET is effectively applied to the operations from DB the same as that in EB. By using this mechanism, delayed operations should be effectively applied to result document. The difference with the existing operational transformation is that VOT only deals the interesting operations in real-time and others are delayed and transformed at a breath.

As figure 11, delayed buffer algorithm is similar to that of execution buffer. One more things to be considered in delayed buffer, $thenumberofdelayedoperation$ should be maintained for correctly applying transformation delayed operations. For instance, when 120 delayed operations are saved in DB, the position information of operations might be created at its generation time. Hence, the position of each operation should be changed according to the number of operations. Consider that of OP_1 is "47". And assume that 119 operations were accepted at VOT_Deamon and they were saved in DB after OP_1, the type of 70 operations are $Insert$ and the others are $Delete$ among 119 operations. In this case, the position of OP_1 should be changed into 68(47+21) according to the number of operations since position change can be calculated as 21 from 70(insert) - 49(delete).

```
DelayedQueueIntegration Algorithm :
While( (op1 = DB.getOperation(i++)) != null){
        int index=0;
    while( (op2 = HB.getOperation(index)) != null){
        if(isOperationIndependent(op1,op2) == false){
            return eo = op1;  // case 1
        }else if(isOperationIndependent(op1,op2) == true){
            if( (TB = scanDependentOperations(op2.index+1, HB.size) )== null ){
                return eo = inclusionTransformation(op1,HB(op2.index+1,HB.size);
                // case 2
            }else {
                TB = scanDependentOperations(op2.index+1, HB.size);
                for(int i=0;i<TB.size;i++){
                    Ltranspose(HB.getSubList(index+i-1,TB.getOperation(i).position);
                }
                return eo = inclusionTransformation(op1,HB.getSubList(op2.index+TB.size, HB.size));
                // case 3
            }
        }
    }
}
```

Fig. 11. Delayed Buffer Integration Algorithm

In this way, delayed operation can be executed via delayed transformation. This mechanism removes the burden which should transforms all operations from the neighbor in real-time. In future work, we would like to improve the transforming delayed operations by aggregation. Generally, user in editor has a tendency to edit a character sequentially. Hence, these operations can be dealt as compound operation which reduces the number of operational transformation.

IV. VOT Architecture

A. Architecture

Based on the algorithm which was described and defined at the above chapter, we have been developed VOT application prototype as figure 12.

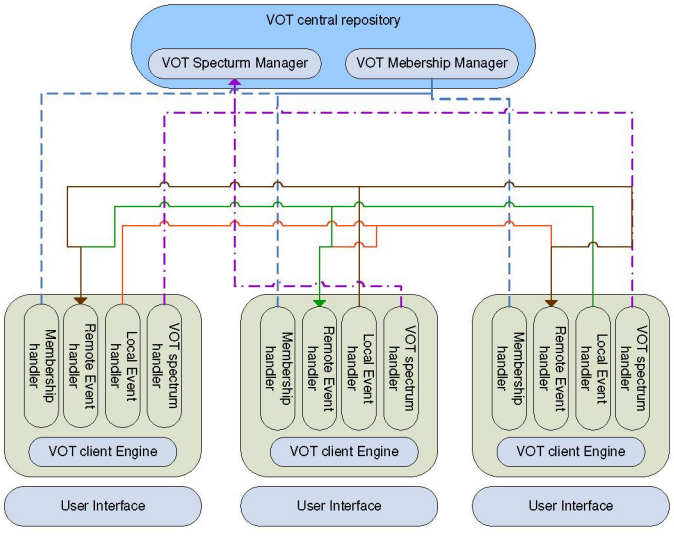

Fig. 12. VOT Architecture

First of all, *VOT central repository* is composed of the membership management and the *spectrum* management. *VOT spectrum* is a structure which contains the viewpoint information of each site. During the running time, VOT spectrum accepts the viewpoint information, which consists of

$site_id, current_viewpoint$, from each site. And then to support the correctness of a delayed operational transformation, VOT spectrum has a global viewpoint variable. For instance, consider that the viewpoint of site 1 is a (4,34) and that of site 2 is a (15,45). If 4 insert operations are invoked, then global viewpoint variable might be increased by 4. And if 2 delete operation is invoked at site 2, then global viewpoint variable also might be decreased by 2. In this way, the global viewpoint variable is changed dynamically, and it can be referenced to each site for a correct operational transformation. *Membership management* is a typical membership management service which can handle the members' action such as join, leave. In VOT architecture, it is a simple service for supporting the communication among VOT members.

Secondly, each site has 5 components for VOT service. In current computing environment, a client-side computer has a full performance to perform VOT. Hence, we have devised that main components are located in client-side such as P2P computing. *Membership handler* is a thread which supports membership service such as $join$ and $leave$. And it normally communicates with the *membership management service* in VOT central repository. *Remote event handler* and *Local event handler* are threads which have a role to handle predefined event such as $insert$ and $delete$. The only difference in two handlers is that events are divided into $local$ and $remote$. For short response time, local events are handled by *local event handler*. Otherwise, remote events are handled independently and they are passed into *VOT engine*.

VOT spectrum handler has a role to manage the current viewpoint of user. If the current viewpoint may be changed, *VOT specturm handler* sends a message to *VOT spectrum* in VOT central repository. And upon receipt of message from client, it updates global viewpoint variable dynamically.

In VOT, the most important part is *VOT client engine*. It performs operational transformation based on VOT algorithm.*VOT client engine* is composed of integration part, operational transformation part and delayed OT part, same as in algorithm section.

B. Operational Flow

In this section, we would like to describe the operational flow in VOT.

Figure 13 describes how to process user's event in VOT. When operation is invoked at local site, local event handler catches this event and passes that into VOT engine. And then, VOT engine checks whether this operation is relevant to the current viewpoint of user. In this case, operation is saved into *execution buffer* since this operation is local event and surely relevant to user's viewpoint. If operation is not included in user's viewpoint, then it saved into *delayed buffer*. Meanwhile, VOT engine picks out the operation from the top of *execution buffer*. And then it performs operational transformation by using the operations in history buffer. And when a transformed operation is created as a result of operational transformation, it is reflected in document of user interface.

Apart from this processing in local site, each local operation is sent to the other members by local event handler at the moment which that event is passed into operation checking mode in VOT engine.

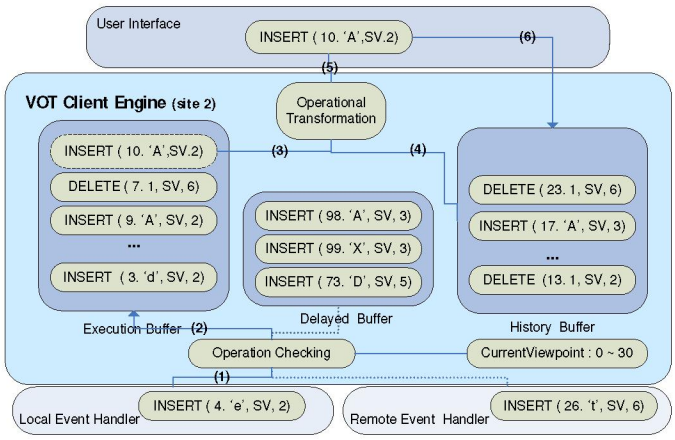

Fig. 13. VOT Operational Flow

In the case of remote operation, the mechanism of operational transformation is the same as local events. The only difference is that remote operations don't have to be sent to the other members since it is already received from the other members.

The last important thing in VOT is *VOT specturm handler*. Namely, it propagates message to *VOT specturm* in *central repository* for updating global viewpoint variable. If the type of operation is $'Insert'$,it propagates messagesuch as $'increase'$. According to the type of received message, *central repository* also updates global viewpoint by increasing 1. At the same time, when the other members accepts $Insert$ operation, it updates global viewpoint of each site. In this way, global viewpoint can be synchronized and operational transformation could be performed correctly.

V. VOT SIMULATION AND ANALYSIS

Due to the various use case of collaborative editing at running time, analysis on operational transformation has never been performed yet. Therefore, we have formulized a usage pattern of collaborative editing to some extent and simulated the running status of it.

The purpose of simulation is that verify the performance difference between the existing operational transformation and VOT. Among the existing operational transformation, we choose *General Operational Transformation Optimization*[1], [2] for comparing the overhead of operational transformation. The things that are simulated in this section is as follows.

- The relation between the number of operations and the execution time of operational transformation
- The cost of a delayed operational transformation

For simulations, we consider that all assumptions and consistency model should be observed in this simulation. And then we have developed a simple $OTTester$ which generates operations periodically and transforms operations

by OT and VOT. To obtain an approximate result, this tester generates operations periodically per 300 ms such that human user generally invokes operations as 100 - 200 typings per minute. And such operations are processed by operational transformation the same as that of collaborative editor.

Simulation was performed at the java virtual machine 1.5.0 in Windows XP operating system. The simulation machine is composed of Intel Pentium 3.0 CPU, 1GB RAM. And this tester program and OT algorithm was implemented via java language.

A. Analysis

The most important issue of operational transformation is an overhead which results from the increase in number of operation. GOTO and VOT have their own way to deal with this problem. As for GOTO, garbage collection mechanism is used for removing needless operations for operational transformation. Namely, GOTO finds a first operation from history buffer which is independent from input operation and then remove all causally preceding operations against input operation.

In the case of VOT, it fundamentally uses a categorized and prioritized queue mechanism such as execution buffer and delayed buffer. Therefore, VOT ensures at least static cost of operational transformation since it transforms operations relevant to viewpoint area. However, GOTO shows variable cost of operational transformation according to circumstances as shown in figure 14.

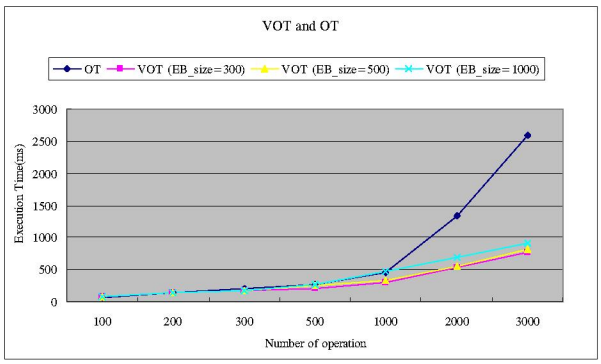

Fig. 14. Performance comparison of VOT and OT

In figure 14,as the number of operations grows, the performance of OT and VOT shows a different result. To 500 operations, all of methods consume relatively similar execution time. However, after 500 operations, the gap of OT and VOT grows increasingly. In the case of OT, there is no garbage collection mechanism to deal with accumulated operations in HB. Therefore, the performance of OT is worse than the other cases. If garbage collection mechanism is applied in OT, the execution time may be reduced or the same. Since collaborative editor is used in WAN environment and the latency may be variable, the effectiveness of garbage collection cannot be ensured. In other words, the existing operational transformation cannot support stable performance due to its variable property from network latency. On the contrary, VOT is capable of providing stable and efficient performance since it is fundamentally based on the different approach.

The next thing we consider is the overhead of delayed operational transformation. If there is no consideration of delayed operational transformation, then the performance of VOT would be worse than that of the existing operational transformation. For solution, we use threshold value of delayed operational transformation for efficiency. However, we don't know how to tune threshold and how often we perform a delayed operational transformation. Therefore we evaluated the cost of a delayed operational transformation.

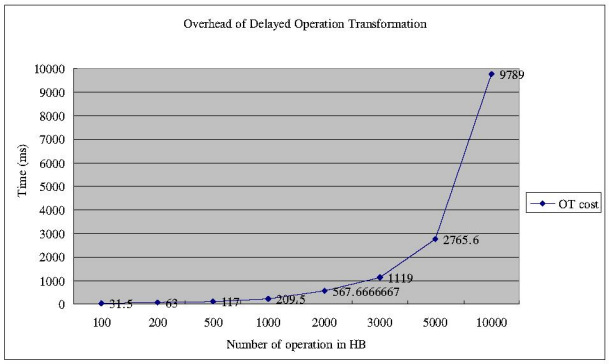

Fig. 15. Overhead of delayed operational transformation

As shown in figure 15, the execution time of the delayed operational transformation grows rapidly when the number of operation in DB is about 3000. It means that the threshold value of delayed buffer management should be tuned about 2500 operations. Because the execution time of operational transformation is less than one second, users may not feel any trouble in collaborative editing at running time.

VI. Futureworks

View-centric operational transformation have been devised and implemented but it has not been fully developed as a level of commercial level. It means that it has not been approved by any user or tools. Since this kind of work is closed with user's response, we have a plan to verify the adaptability and usability of VOT application.

Through this research, we have been only focused on the document field of collaborative editing. As a future research target, graphical editor may be the interesting field which can be applied with view-centric operational transformation. The object in graphical editor is different from that of document editor. Namely, it needs to consider the collaborative element of graphical editor with a different viewpoint. Therefore, we would like to look into the behavior and usability of graphical editor.

VII. Conclusion

View-centric operational transformation is fundamentally based on the novel operational transformation. In this research,

we have found the limitation of the existing operational transformation and tried to enhance and improve the collaborative editor with an unique aspect. Namely, VOT is created from the careful consideration about the property of collaborative editor. Generally collaborative editor assumes that many users and large documents. Therefore, the case which applies operational transformation may not be conflict in normal case. Therefore, we divide operations into two categories such as execution operation and delayed operation. And according to priority, each operation is processed appropriately. The existing operational transformation only considers the possibility of collaboration. The motivation and effort of VOT is to improve of the usability and efficiency of operational transformation.

In this research, we have experienced with the unique properties of collaborative computing such as group work and large shared objects. As a concluding remark, the unique properties of collaborative works have to be considered more carefully for supporting comfortable collaborative computing environment.

REFERENCES

[1] C. Sun, X. Jia, Y, Zhang, Y, Yang, and D.Chen, "Achieving Convergence, Causality, Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems", ACM Transactions on Computer-Human Interaction, volume 5, 1998, March

[2] Chengzheng Sun, and Clarence Ellis, "Operational Transformation in Real-Time Group Editors : Issues, Algorithms, and Achievements", Proceedings of ACM Conference on Computer Supported Cooperative Work, 1998

[3] C. Sun, D. Chen, and Xiaohua Jia, "Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems", Proceedings of the 21st Australian Computer Science Conference, 1998

[4] C. Sun, Y. Yang, Y. Zhang, and D. Chen, "A consistency model and supporting schemes for real-time cooperative editing systems", Proceedings of the 19th Australian Computer Science Conference, January, 1996

[5] C. Sun, Y. Yang, Y. Zhang, and D. Chen, "Distributed concurrency control in real-time cooperative editing systems. Proceedings of the Asian Computing Science Conference, 1996

[6] C. Sun, X. Jia, Y. Yang, and Y. Zhang, "REDUCE : a prototypical cooperative editing system", Proceedings of the 7th ACM International Conference on Human Computer Interaction, August, 1997

[7] C. Sun, X. Jia, Y. Zhang, and Y.Yang, "A generic operational transformation scheme for consistency maintenance in real-time cooperative editing systems", Proceedings of ACM International Conference on Supporting Group Work (Group'97), November, 1997

[8] Steven Xia, D. Sun, C, Sun, D. Chen, and Haifeng Shen, "Leveraging Single-user Applications for Multi-user Collaboration : the CoWord Approach", Proceedings of ACM Conference on Computer Supported Cooperative Work, November, 2004

[9] David Sun, Steven Xia, Chengzheng Sun, and David Chen, "Operational Transformation for Collaborative Word Processing", Proceedings of International Conference on Computer Supported Cooperative Work, November, 2004

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", Communications of the ACM 21, Vol 21, July, 1978

[11] C.A. Ellis and S.J. Gibbs, "Concurrency control in groupware systems", Proceeding of ACM SIGMOD conference on Management of Data, June, 1989

[12] M. Knister and A. Prakash, "Issues in the design of a toolkit for supporting multiple group editors", Journal of the Usenix Association, page 135-166, 1993

[13] M. Ressel, D. Nitsche-Ruhland and R. Gunzenbauser, "An integrating, transformation-oriented approach to concurrency control and undo in group editors", Proceeding of ACM Computer Supported Group Work, page 288-297, 1996

[14] Claudia Ignat, and Moira Norrie, "Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems", Proceeding of ACM Computer Supported Cooperative Work, 2002,

[15] A. Imine, P. Molli, G. Oster and M. Rusinowitch, "Development of transformation functions assisted by a theorem prover", Proceeding of the Fourth International Workshop on Collaborative Editing

[16] Shailesh N. Humbad, "Freetext : A Consistency Algorithm For Group Text Editors", Proceeding of ACM Computer Supported Cooperative Work, 2004

[17] A. Klimetschek, "DocSynch - Collaborative Editing for Text Editors", http://docsynch.sourceforge.net